



A methodology to design arbitrary failure detectors for distributed protocols[☆]

Roberto Baldoni^{a,*}, Jean-Michel H elary^b, Sara Tucci Piergiovanni^a

^a *Dipartimento di Informatica e Sistemistica "Antonio Ruberti", Universit  di Roma La Sapienza, Via Ariosto 25, Roma, Italy*

^b *IRISA, Campus de Beaulieu, 35042 Rennes-Cedex, France*

Received 10 February 2007; received in revised form 25 October 2007; accepted 11 November 2007

Abstract

Nowadays, there are many protocols able to cope with process crashes, but, unfortunately, a process crash represents only a particular faulty behavior. Handling tougher failures (e.g. sending omission failures, receive omission failures, arbitrary failures) is a real practical challenge due to malicious attacks or unexpected software errors. This is usually achieved either by changing, in an ad hoc manner, the code of a crash resilient protocol or by devising a new protocol from scratch. This paper proposes an alternative methodology to detect processes experiencing arbitrary failures. On this basis, it introduces the notions of liveness failure detector and safety failure detector as two independent software components. With this approach, the nature of failures experienced by processes becomes transparent to the protocol using the components. This methodology brings a few advantages: it makes possible to increase the resilience of a protocol designed in a crash failure context without changing its code by concentrating only on the design of a few well-specified components, and second, it clearly separates the task of designing the protocol from the task of detecting faulty processes, a methodological improvement. Finally, the feasibility of this approach is shown, by providing an implementation of liveness failure detectors and of safety failure detectors for two protocols: one solving the consensus, and the second solving the problem of global data computation.

  2007 Elsevier B.V. All rights reserved.

Keywords: Failure detectors; Arbitrary failures; Adaptive fault tolerance; Consensus; Global data computation problem; Distributed algorithms

1. Introduction

Before seminal paper from Chandra and Toueg [6], distributed protocols running on crash-prone system models merged the aspect related to fulfill their goal and the aspect related to the detection of crashes. Chandra and Toueg were the first to propose an approach that encapsulates the task of detecting process crashes in a component, external to the process, called failure detector. A crash failure detector is a distributed oracle that can be consulted by a process to have hints about the state of another process.

From an operational viewpoint, a crash failure detector undertakes the burden of dealing with the bad behavior (i.e., asynchrony and failures) of the underlying system, letting the protocol designer concentrate on the essential part of the development without worrying about the underlying problems. The interest of their approach lies in the fact that it clearly separates concerns between the task of solving some distributed computing problems and the task of implementing the failure detector [22].

The notion of failure detectors have been firstly introduced as a way to solve the consensus problem by circumventing the FLP impossibility result [12]. This result states that consensus cannot be solved in a pure asynchronous distributed system when also a single process might crash. A solution to the consensus problem consists in designing a deterministic protocol in which all the processes that do

[☆] The research has been partially supported by the Network of Excellence "RESIST".

* Corresponding author.

E-mail address: baldoni@dis.uniroma1.it (R. Baldoni).

not crash reach a common decision based on their initial opinions. Failure detectors actually add to the underlying model the synchrony necessary to solve consensus.¹

Unfortunately, failures can become more subtle than crashes. If we consider consensus protocols resilient to arbitrary failures and based on failure detectors (e.g. [1,9,16,17,19]) share a common factor: they include the task of failure detection, but process crashes, into the code of the protocol. This has two main disadvantages. These protocols, firstly, loose the main strength of Chandra–Toueg’s approach as failures more subtle than crashes have to be handled by the code developed by the protocol designer. Secondly, changing the protocol code means does not leverage from the protocol designed in the crash-prone environment.

Concerning the latter point, in the context of consensus handling muteness failures,² Doudou et al. in [10] pointed out that a protocol designed in a crash-stop model can be reused, *modulo a few change in its code*, in a muteness failure model, just by replacing the crash failure detector by a muteness failure detector.

The focus of this paper is to propose a methodology to handle arbitrary failures that aims to get a separation of concerns between solving the conceptual problem (e.g., agreement-like problem) and the failure handling one by encapsulating the task of detecting such failures in an arbitrary failure detector. This has a noteworthy consequence, it allows to reuse the code of a crash resilient protocol, in a system model with arbitrary failures just by replacing the crash failure detector with an arbitrary one without impacting the protocol’s correctness.

More specifically, the paper proposes a systematized sequence of steps (i.e., the methodology) that takes as input a protocol \mathcal{A} correct with respect to a system model prone to crash failures and returns an arbitrary failure detector specifically designed for the protocol \mathcal{A} . This failure detector can be used then by \mathcal{A} to resist to arbitrary failures. The arbitrary failure detector is formed by two components: a liveness process failure detector and a safety process failure detector. A liveness failure detector associated with protocol \mathcal{A} is a distributed oracle which detects any process that does not make progress with respect to specification of \mathcal{A} (i.e., this process is no longer live w.r.t. \mathcal{A}). A safety failure detector associated with protocol \mathcal{A} is a distributed oracle which detects any process that does not execute statements according to specification of \mathcal{A} (i.e., this process is not safe w.r.t. \mathcal{A}).

To show the feasibility of the approach, we apply the methodology to two case studies: a protocol solving the

consensus problem [15], and a protocol solving the Global Data Computation problem [8].

Let us finally remark only *crash* failure detectors can be designed independently of the protocol that will use them [9,10]. Consequently, both liveness and safety failure detectors components need to make reference to properties of the protocol they are associated with. So, their design is necessarily ad hoc, and this raises the following question: does it make sense to adopt this ad hoc approach, rather than solutions adopting crash failure detector and changing the protocol code to catch the non-crash failures? We advocate that answer is yes, for two reasons. Firstly, changing the crash-resilient protocol code does not allow a more generic approach than our external components approach. Secondly, in our approach, the nature of failures experienced by processes becomes transparent to the protocol using these components. Thus, from an operational point of view, it makes possible to increase the resilience of a protocol designed in a crash failure context without changing its code, and, from a methodological point of view, it clearly separates the task of designing the protocol from the task of designing faulty processes.

The paper is made of six sections. Section 2 presents the computation models and the notions of process faults and failures. Section 3 introduces the concepts of liveness failures and of safety failures. Section 4 presents the principles underlying the design of liveness failure detectors and of safety failure detectors and the way a protocol might use them. Finally, Sections 5 and 6 present the two case studies.

2. The model of computation

2.1. Protocol

A protocol is composed of n sequential programs. Each program involves two kinds of statements: internal and communication. A protocol is specified by one or several properties. A protocol is designed with respect to a *system model*. A system model is a description of the environment able to support the executions of this protocol. A protocol is *correctly designed w.r.t. a system model* if any execution of this protocol in an environment satisfying the system model assumptions satisfies the specification of the protocol.

2.2. System models

System models considered in this paper share the following characteristics:

- The execution of a sequential program P_i is a process p_i , that produces a (possibly infinite) sequence of events.
- Each process has its own local environment (local memory, input–output buffers, etc.) and runs on a processor.
- Processes communicate together by exchanging messages through channels connecting an output buffer of the sender to an input buffer of the receiver.

¹ Consensus in an asynchronous system can be solved through the use of random source observable by all participants (e.g., [3,4,21]). However in this paper we focus on solution of consensus based on failure detectors.

² A process “A” suffers a mute failure with respect to process “B” if “B” does not receive application messages from “A” while the “B”’s crash failure detector says that A is alive.

Different system models are obtained according to different additional assumptions. These assumptions concern in particular:

- *The time* (synchronism/asynchronism): *synchronous* models are characterized by the three following *timing assumptions* ([13]):
 - (1) There is a known upper bound on the time required by any process to execute an action.
 - (2) Every process has a local clock with known bounded rate of drift with respect to real time.
 - (3) There is a known upper bound on the time taken to send, transport and receive a message over any channel.
 On the contrary, in completely *asynchronous* models, none of these three timing assumptions hold. Thus, asynchrony concerns processes as well as channels. Intermediate models, where some of these timing assumptions or weaker timing assumptions hold, can be defined. They are referred to as *partially synchronous* models.
- *The reliability*: responsibility for faulty behavior is assigned to the system's components (i.e. communication channels and processes). Therefore, reliable models assume reliability properties for both channels and processes. Unreliable models include models where some of those channels/process reliability requirements are not assumed.

For example, an *asynchronous* distributed system prone to *process crash* failures, is a distributed system where no time assumption is made (asynchronous), channels are assumed reliable, and processes fail only by crashing.

2.3. Process fault and process failures

During the execution of a protocol \mathcal{A} , a process *fails* if it deviates from its specification. When a process fails, we say that a *process failure* occurs. A process that does not fail is *correct*. Process failures are the consequence of underlying *faults* [20]. Examples of faults are mistakes in protocol designing, software bugs, failures of hardware components used by the process, congestion of underlying network, etc. Examples of process failures are crashes, permanent or transient sending (or receiving) message omissions, corruption of the field of a message, etc. Note that sequences of events produced by a correct process are consistent with the protocol specification, but the converse is not true: a process could indefinitely delay its activity after having produced a finite prefix of a consistent sequence of events.

It is well known that some faults can be hidden in some runs, i.e., they do not produce a failure in this run. But another fact worth to be noticed is that some failures can remain local, i.e., they do not impact the behavior of other processes. Suppose for example that a process p crashes at a particular point of its execution. Consider all the consistent

sequences of events that could be produced by p after this point, if p had not crashed. If none of these sequences contain the sending of a message to a process q , the latter will not be directly affected by the crash of p . Another example is as follows: if a process corrupts the value of a local variable, and if this value has no influence on the decision to send messages nor on the content of sent messages, this failure does not impact other processes. In the rest of this paper, the expression *process failure* will be restricted to failures that have some impact on the behavior of other processes.

During the execution of a protocol \mathcal{A} , the manifestation of failures of a process p passes through the analysis of messages sent by p while running \mathcal{A} . A *crash failure* of p means that p stops taking any action, and this implies that expected messages never arrive to the intended destination. A *muteness failure* of p w.r.t. another process q means that p stops sending protocol messages to q (while continuing to take other actions, e.g. sending expected protocol messages to processes except to q). In a *transient message omission sending* failure, process p skips the sending of one or more message during some time. In a *message corruption failure*, a process p corrupts the value of a field of the message.

3. Liveness failures and safety failures

In this section, a classification of process failures between process *liveness failures* and process *safety failures* is proposed. Before going further on, it is worth to emphasize that this classification is not directly linked to liveness and safety protocol specifications. For example, the following specification is a liveness specification:

- *Eventually, every correct process decides a colour.*

Note that this specification involves only correct processes, i.e., processes that do not suffer any failure. In other words, process liveness failures (like crash failures) do not necessarily impact the liveness properties of the protocol.

The situation would obviously be different with other specifications, such as, for example:

- *Eventually, every process decides a colour.*

3.1. Process liveness failures

In the past, some types of process failures have been specifically considered, e.g., crash failures, or muteness failures. They constitute examples of *liveness* failures, in the sense that, if a process p suffers from such a failure, this process stops to show some progress w.r.t. at least one another process q (independently of the detection of this fact by the impacted process q).

More formally, let consider any two processes p and q running a protocol \mathcal{A} . In any execution of \mathcal{A} , the history of p includes a sequence of relevant events, namely $step(p, q)_1, \dots, step(p, q)_\ell, \dots$, where $<_\ell$ is the relation of local

precedence on events on process p and where each event is either the sending of a message to q , or an internal event such that there is the sending of a message to q not after the next relevant event. This sequence possibly includes an event $stop(p, q)$, denoting the termination with success of the code run by p , i.e., after $stop(p, q)$, no send event of a message from p to q exists.

Definition 1. A process p is *stalled* w.r.t. a process q (in a run of a protocol \mathcal{A}) if there exists k such that $step(p, q)_k$ occurred and $step(p, q)_{k+1}$ or $stop(p, q)$ will never occur.

Definition 2. A process p suffers a *liveness failure* (in a run of a protocol \mathcal{A}) if there exists at least one process q such that p is stalled w.r.t. q .

The following example shows that process liveness failures are not limited to crash failures or to muteness failures. Suppose that the protocol \mathcal{A} governing p includes a code like the one shown in Fig. 1 where C is a condition becoming true only after the receipt of some messages. If p fails by permanently omitting to receive messages (it suffers a permanent receive omission failure) enabling C to become true, then the event $step(p, q)_{k+1}$ will never occur and thus p will be stalled w.r.t. q (i.e., p suffers a liveness failure). However, p will continuously perform the sending of $m(k)$ to q and thus, p will not be mute to q .

Let us remark, however, that if p suffers only transient omission fault, then after a while p may execute `statement k+1` (i.e., the statement producing $step(p, q)_{k+1}$). In that case, p is not stalled w.r.t. \mathcal{A} and q .

Let us also remark the importance of the event $stop(p, q)$. If the execution of p produces this event, then p will never be stalled w.r.t. q .

3.2. Process safety failures

Let us consider any two processes p and q running a protocol \mathcal{A} . In any execution of \mathcal{A} , the history of p includes a sequence of events $s_k = send(m, q)$, where, for each s_k , the content of the protocol message m is a function of the events locally preceding s_k on process p .

Definition 3. A process p is *unsafe* w.r.t. a process q (in a run of a protocol \mathcal{A}) if there exists k such that the protocol message involved in s_k is not consistent with the specification of \mathcal{A} .

```

...
statement k % relevant event step(p, q)_k %
while not C do
    send m to q
...
endwhile
statement k+1 % relevant event step(p, q)_{k+1} %
...

```

Fig. 1. Liveness process failures: an example.

```

k:=0
repeat
    k:=k+1
    upon receipt of m'(a) from q'
        x:=f(a)
    upon receipt of m" from q"
        send m(x) to q % relevant event step(p, q)_k %
    ...
until some condition

```

Fig. 2. Safety process failures: an example.

Definition 4. A process p suffers a *safety failure* (in a run of a protocol \mathcal{A}) if there exists at least one process q such that p is unsafe w.r.t. q .

The following example shows that a process p could suffer a safety failure without being stalled, e.g., if it suffers *transient omission failures*. Let the code of the protocol governing p be shown in Fig. 2. Suppose that p temporarily omits to receive some messages $m'(a)$ from q' , but otherwise is correct. It means that some updates of local variable x do not occur, and thus the content x of some messages $m(x)$ are wrong. Thus, p is unsafe w.r.t. q , but is not stalled w.r.t. q (as events $step(p, q)_k$ occur).

4. Designing liveness and safety failure detectors

4.1. Handling liveness process failures

(1) *Specifications of a liveness failure detector.* A liveness failure detector is a distributed oracle aiming at detecting stalled processes. It is composed of *local* modules, one per process. A local module can observe the state of the process p to which it is associated, and its output is the set (*suspected_liveness_p*) of processes it suspects to be stalled w.r.t. p . To be more precise, we adopt the model patterned after the one in [6]. A liveness failure detector can make mistakes by not suspecting a stalled process or by suspecting a live one. It is thus specified with two properties: completeness (a property on the actual detection of stalled processes) and accuracy (a property that restricts the mistakes on erroneous suspicious). Note that in these specifications, the abbreviation *lf_{dm}* stands for “liveness failure detector module”. Thus, we get the following classification.³

Eventual completeness. Eventually, every process that is stalled w.r.t. a correct process p is *permanently* suspected by p 's *lf_{dm}*.

Eventual weak accuracy. Eventually, there is *at least one* live process that will never be suspected by any correct process' *lf_{dm}*.

³ It is possible, as in [6], to present a more formal specification based on the notion of failure pattern. Although this presentation is not adopted here, it would not be difficult to obtain.

Eventual strong accuracy. Eventually, every live process will never be suspected by other correct processes' *lfdm*.

Weak accuracy. There is at least one live process that will never be suspected by any correct process' *lfdm*.

Strong accuracy. Any live process will never be suspected by a correct process' *lfdm*.

Similarly with the notations introduced in [6] and widely used in the case of crash failure or muteness failure, we will denote by $\diamond \mathcal{LTP}_{\mathcal{A}}$ the class of liveness failure detectors satisfying eventual completeness and eventual weak accuracy for a protocol \mathcal{A} (eventually strong liveness failure detector). We will denote by $\diamond \mathcal{LTP}_{\mathcal{A}}$ the class of liveness failure detectors satisfying eventual completeness and eventual strong accuracy. And we will denote by $\mathcal{LTP}_{\mathcal{A}}$ the class of liveness failure detectors satisfying eventual completeness and strong accuracy (Perfect liveness failure detector). The suffix \mathcal{A} will be omitted when no confusion is possible.

4.2. Hints for designing liveness failure detectors

Implementations of crash failure detectors were mainly based on the notion of “I-am-alive” messages (heartbeats) exchanged between the instances of crash failure detector associated with each process. If a failure detector of a process q stops receiving heartbeats from the failure detector of process p then the failure detector of q suspects p to be crashed. There is then a sharp separation between the messages exchanged by the protocol and the messages exchanged by the failure detectors. This makes crash process failure detector independent from the underlying protocol.

It has been shown in Refs. [9,10] that designing muteness failure detectors cannot be independent from the protocol run by processes. In fact, the receipt of heartbeats is no longer a guarantee that p is correct: p could indeed stop sending protocol messages, but continue to send heartbeat messages. So, a muteness failure detector must be able to detect a process that is not crashed, but stops sending protocol messages. Consequently, the authors pointed out that a necessary condition to design such a muteness failure detector is that each process has to know the set of messages exchanged by a protocol \mathcal{A} .

When designing a liveness failure detector previous condition does not suffice to ensure detection of stalled processes. As shown in Section 3, p could continue to send protocol messages to q without doing any progress with respect to the protocol \mathcal{A} .

Therefore, a liveness failure detector has to be able to capture

- the progress of a process p with respect to \mathcal{A} , and
- the termination with success of the code of p with respect to \mathcal{A} .

(1) *Requirements imposed to \mathcal{A} .* It results from the previous section that a protocol \mathcal{A} has to embed mechanisms that allow a liveness failure detector to capture its progress

in its runs. In particular, the execution of each process must be patterned according to the model presented in Section 3.1, and, moreover, messages involved in send events occurring between consecutive $step(p, q)$ events must transmit the value of a local p 's variable k that is increased at each $step(p, q)$ event. Therefore, the code of each process has to be modelled according to the following one.

So, if the liveness failure detector associated with a process p receives protocol messages from a process q while the variable k remains unchanged, then, in this run of \mathcal{A} , it can suspect q to be stalled with respect to p .

4.3. Handling safety process failures

(1) *Specifications of a safety failure detector.* The discussion and the classification obtained in the case of liveness failure detectors can be applied to the case of safety failure detectors as well, where the word “stalled” becomes “unsafe”, “live” become “safe”, and the abbreviation *sfdm* stands for “safety failure detector module”. In particular, the output of the local module associated with p is the set (*suspected_safety_p*) of processes it suspects to be unsafe w.r.t. p . Thus, we get the following classification.

Eventual completeness. Eventually, every process that is unsafe w.r.t. a correct process p is permanently suspected by p 's *sfdm*.

Eventual weak accuracy. Eventually, there is at least one safe process that will never be suspected by any correct process' *sfdm*.

Eventual strong accuracy. Eventually, every safe process will never be suspected by other correct processes' *sfdm*.

Weak accuracy. There is at least one safe process that will never be suspected by any correct process' *sfdm*.

Strong accuracy. Any safe process will never be suspected by a correct process' *sfdm*.

However, even if formally the classification is similar to the case of liveness failure detectors, there is a difference between them. In fact, detecting safety failures rests on mechanisms (see the next section) that do not rely on “time”, but on the very structure of the protocol. This explains why, practically, when the structure of the protocol allows to apply these mechanisms, safety failure detectors are perfect (eventual completeness and strong accuracy).

(2) *Hints for the design of safety failure detectors.* As explained in Section 2.3, detection of failures is closely related to the receipt of protocol messages. Therefore, when one has to cope with detection of safety failure, the key idea is: each process has to check whether the right message has been sent by the right process at the right time with the right arguments. This leads to identify two kinds of “externally” visible behaviors:

- (1) Wrong messages (i.e., right time, but wrong message or wrong content). This case includes messages sent after an alternative statement has been misevaluated

(substituted messages), or messages whose content is syntactically or semantically incorrect.

- (2) Unexpected messages (i.e., wrong time). This corresponds to an “out_of_order” message, revealing either a *transient sending omission* or a *sending duplication*. This case includes in particular the case of messages that are not generated during fail-free executions of the protocol.

Detection of wrong or unexpected messages is based, on the one hand, on certification mechanisms, and, on the other hand, on state machines built from the text of the protocol (both tools are explained in detail below). Certificates can be analyzed (at the recipient side) by a state machine to detect wrong messages. As the state machine is built from the text of the protocol, this machine can also detect unexpected messages. It results from this discussion that the task of designing safety failure detectors essentially consists in the design of appropriate certificates and in modelling the protocol with a state machine.

Let us now present in detail each of these tools and the structure of a safety failure detector local module, attached to a process.

(a) *Certificates*. A certificate is a piece of redundant information, appended to a message in order to detect wrong expected messages. Its aim is to “witness” (i) the content of the message and (ii) the fact that the decision to send the message has properly been taken by the sender. A certificate includes a part of the process history. This history includes internal, send and receipt events. A certificate can be appended to a message upon its sending, and is used by the receiver to check if the content of the message is consistent with the senders history (no semantically incorrect messages). It also allows the receiver to check that the decision to send this message (and not another one, in case of choice) is the correct one (no substituted messages).

Consider a message m sent by p , containing a value v . This value has been updated by p according to its own history. Similarly, the sending event of m is a consequence of the receipt of other messages, and is enabled by a set of conditions involving local variables of p . The certificate appended to m must contain proper information able to witness: the value v , the fact that the required receipt events have been correctly taken into account, and the values of p 's local variables involved in the enabling condition.

Let us remark that we have to assume that certificates themselves cannot be corrupted, since a corrupted certifying information could be consistent with a corrupted information to certify. The concept of *reliable certification component* encapsulates this assumption. Technically, this assumption can be enforced by the very structure of certificates: they are composed of a set of *signed messages*, e.g. messages whose receipt is the cause of the sending of m , or whose content has influenced the update of a local variable whose value is involved in m . Reliability results from the fact that *no process can falsify the content of a signed message without being detected as faulty by a correct receiver*

ver [23], and, if necessary, from the cardinality of the set of signed messages allowing to perform majority tests. The correction of a certificate can thus be verified at the recipient side, by a *certificate analyzer*.⁴ The consequences of this assumption will be further analyzed in Section 5.4.

Definition 5. A certificate attached to a message m is *well-formed with respect to a value v* if it has been analyzed as non-corrupted and if the receiver can extract information consistent with the value of v and with the action to send m .

Notation. Let m be a message sent by a process p , and certified with a certificate $cert$. The pair $(m, cert)$, signed with the unforgeable signature of p , will be denoted by $\langle m, cert \rangle_p$. It means, in particular, that no process can falsify the information contained between \langle and \rangle without being detected as faulty.

The design of certificates depends on the protocol to be transformed. The previous principles constitute a “guide-line” for this design. If the protocol has been proved correct in a failure model involving only liveness failures (e.g., in the crash model), it remains only to prove that certificates are well-formed with respect to (1) values carried by messages and (2) decisions enabling their send event.

(b) *State machines*. Let us consider a state machine modelling the behavior of process p with respect to q . In this state machine, transitions are triggered when p receives a message from q . In every state, a set of receipt events are enabled. *Unexpected messages* are those whose receipt events are not enabled. *Syntactically incorrect messages* are those whose receipt event is enabled, but whose syntactic composition is not consistent with the one of the corresponding expected message. *Semantically incorrect and substituted messages* are those whose receipt event is enabled, but whose certificate is not well formed with respect to either its arguments or the action to send that particular message. When such events occur, they trigger a transition to a particular terminal state, called *faulty state*. The actual design of a particular state machine has to be done in the particular context of the protocol (just like the design of particular syntactic analyzers has to be done in the context of each grammar).

(c) *Structure of safety failure detection local modules*. The safety failure detector module associated with a process p (hereafter *sfdm*) is composed of three sub-modules (called also modules, for simplicity): (i) a *signature* module, (ii) a *verification* module, and (iii) a *certification* module. More precisely, the structure of an *sfdm* is given in Figs. 3, 4. The same figure also shows the path followed by a message m (resp. m') received (resp. sent) by p .

Signature module. Each signed message arriving at p is first processed by this module which verifies the signature

⁴ Note that there are many system models that assume byzantine processes interacting with “reliable” components. The most known is probably the *Wormhole model* [7,11]. The wormhole is a hardware/software system that cannot be corrupted. On top of it, there are processes that can be byzantine.

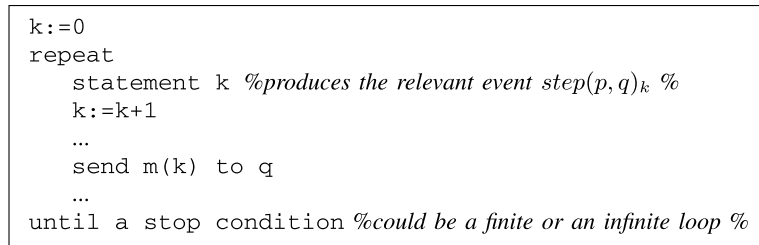
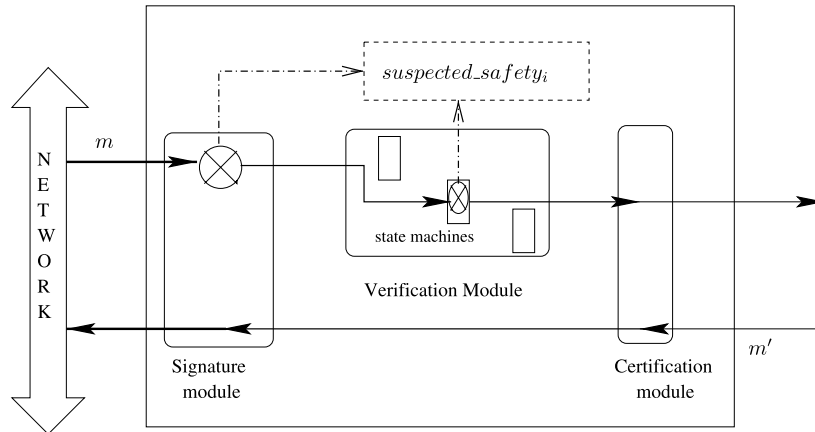
Fig. 3. Structure of the code of a process in \mathcal{A} .

Fig. 4. Structure of a local safety failure detector.

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 and its sender identity (known thanks to the unforgeable signature), is added to the local output $\text{suspected_safety}_p$. Otherwise, the signed message is passed to the verification module. Also, each message sent by p is signed by the signature module just before going in the network. This module is generic, in the sense that it can be implemented independently of the protocols using it [23].

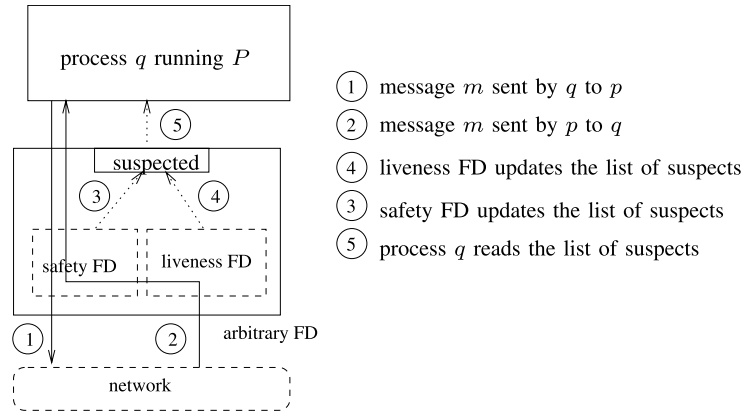
Verification module. This module receives (certified and signed) messages from the signature module. It implements the *certificate analyzer* mentioned in the previous section. For each message m , it first checks whether m is properly formed (syntax) and if its certificate is well-formed w.r.t. values carried by m (semantics). Then, it checks whether the receipt of m follows the program specification of the sender. To this aim, the verification module is composed of a set of state machines, one for each possible sender. If the checks are positive, it passes the (certified and signed) message to the certification module. Otherwise, it appends the identity of the sender of m to the set $\text{suspected_safety}_p$.

It is important to note that, if the certificates are correctly designed and the messages are signed, then this module is reliable, i.e., if p is correct and $q \in \text{suspected_safety}_p$, then q has experienced an incorrect behavior detected by the verification module of p . This is enforced by the fact that, if the content of the signed message, and in particular

the included certificate, had been corrupted, this would be detected by the signature module in the previous stage. Thus, the verification module can safely rely on the values contained in a certified message to verify that the content of the message and the decision to send this message is consistent with its certificate (e.g., by “replaying” the code of the sender with the data contained in the certificate).

Certification module. This module is responsible, upon the receipt of a (certified and signed) message from the verification module, for updating the corresponding certificate local variable. In particular, it does not play any direct role in the detection of safety failures of message senders. It is also in charge of appending properly formed certificates to the messages that are sent by p (as described in Section 4.3(2a)).

(3) **Requirements imposed to \mathcal{A} .** It results from the previous section that the design of a safety failure detector (related to a given protocol \mathcal{A}) is possible if one is able to design a finite-state machine modelling the behavior of each process. Stating formal requirements on the structure of protocols for which such designs are possible remains an open problem and is out of the scope of this paper. However, for some regular protocol structures such as, e.g., round-based protocols, such a design is possible. In a round-based protocol, each process sequentially executes the following steps. (1) It sends the same round message to each process. (2) It waits for a round message from each other process (or from a given number of processes). (3) It executes local computations.

Fig. 5. General failure detection mechanism for protocol \mathcal{A} .

Fortunately, the case studies (Sections 5 and 6) meet these requirements: both are *round-based* protocols, exchanging a predefined and well-structured flow of messages during each round.

4.4. Using failure detectors

A protocol \mathcal{A} can use liveness and safety failure detectors according to the mechanism depicted in Fig. 5. To get protocol transparency, each detector provides each process p with a list of suspects, $suspected_liveness_p$ and $suspected_safety_p$, respectively, whose union is written onto a variable $suspected_p$.

5. A first case study: the consensus problem

In this section, we show how the component-based architecture can be applied to the consensus problem. One of the representative protocols \mathcal{A} that solves that problem in a crash-prone model is selected, and then its liveness and safety failure detector components are designed.

Consensus is a fundamental paradigm for fault-tolerant 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). There are several protocols that solve the problem in the distributed asynchronous with crash failures model, augmented with a crash failure detector of the class $\diamond\mathcal{S}$ [6,15,18,24]. When t denotes the maximum number of processes that can crash, and n the total number of processes, all require $t \leq \lfloor \frac{n-1}{2} \rfloor$. We selected a version of Hurfin–Raynal’s protocol [15] that assumes FIFO channels, because it appears to be particularly simple (however, what follows should also apply to other consensus protocols modulo some changes in the liveness and in the safety failure detectors module caused by the intrinsic protocol differences).

5.1. Hurfin–Raynal’s consensus protocol

This consensus protocols proceeds in successive asynchronous rounds, using 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).

(a) *Automaton states*. During each round, the behavior of each process p_i is determined by a finite state automaton. This automaton is composed of 3 states. The local variable $state_i$ will denote the automaton state in which p_i currently is. During a round, the states of the automaton have the following meaning:

- $state_i = q_0$: 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*.

(b) *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 , $state_i = q_0$. Then, during r , the transitions are:

- *Transition $q_0 \rightarrow q_1$ (p_i first votes current)*. This transition occurs when p_i receives a *current* vote (line 8), and is in the initial state q_0 (line 11). This means that p_i is the round coordinator, or has not previously suspected the round coordinator. Moreover, when p_i moves to q_1 and is not the current coordinator, it broadcasts a *current* vote (line 12).
- *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 15). 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 16).
- *Transition $q_1 \rightarrow q_2$ (p_i changes its mind)*. This transition (executed by statements at line 20) 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

```

function consensus( $t, v_i$ )
(1)  $r_i \leftarrow 0$ ;  $est_i \leftarrow v_i$ ;
    cobegin
(2)   || upon receipt of  $decide(p_k, est_k)$ 
(3)     send  $decide(p_i, est_k)$  to  $\Pi$ ; return( $est_k$ )

(4)   || loop % on a sequence of asynchronous rounds %
(5)      $c \leftarrow (r_i \bmod n) + 1$ ;  $r_i \leftarrow r_i + 1$ ;  $state_i \leftarrow q_0$ ;  $rec\_from_i \leftarrow \emptyset$ ;  $nb\_next_i \leftarrow 0$ ;  $nb\_current_i \leftarrow 0$ ;
(6)     if ( $i = c$ ) then send  $current(p_i, r_i, est_i)$  to  $\Pi$  endif;

(7)     while ( $nb\_next_i \leq (n - t)$ ) do % wait until a branch can be selected, and then execute it %
(8)       upon receipt of  $current(p_k, r_i, est_k)$ 
(9)          $nb\_current_i \leftarrow nb\_current_i + 1$ ;  $rec\_from_i \leftarrow rec\_from_i \cup \{p_k\}$ ;
(10)        if ( $nb\_current_i = 1$ ) then  $est_i \leftarrow est_k$  endif;
(11)        if ( $state_i = q_0$ ) then  $state_i \leftarrow q_1$ ;
(12)          if  $i \neq c$  then send  $current(p_i, r_i, est_i)$  to  $\Pi$  endif;
(13)        endif;
(14)        if ( $nb\_current_i > (n - t)$ ) then send  $decide(p_i, est_i)$  to  $\Pi$ ; return( $est_i$ ) endif

(15)      upon ( $p_c \in suspected_i$ )
(16)        if ( $state_i = q_0$ ) then  $state_i \leftarrow q_2$ ; send  $next(p_i, r_i)$  to  $\Pi$  endif

(17)      upon receipt of  $next(p_k, r_i)$ 
(18)         $nb\_next_i \leftarrow nb\_next_i + 1$ ;  $rec\_from_i \leftarrow rec\_from_i \cup \{p_k\}$ 

(19)      upon ( $change\_mind$ )
(20)         $state_i \leftarrow q_2$ ; send  $next(p_i, r_i)$  to  $\Pi$ 
(21)      endwhile

(22)    if ( $state_i \neq q_2$ ) then  $state_i \leftarrow q_2$ ; send  $next(p_i, r_i)$  to  $\Pi$  endif
(23)  endloop
coend

```

Fig. 6. Hurfin–Raynal’s $\diamond_{\mathcal{S}}$ -based consensus protocol (adapted to FIFO channels).

(CURRENT or *next*) vote from a majority of processes 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 20). In the text, the abbreviation *change_mind* means $(state_i = q_1) \wedge (|rec_from_i| > (n - t))$.

(c) *Protocol description*. 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.
- $nb_current_i$ (resp. nb_next_i) counts the number of *current* (resp. *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, $suspected_i$ is a set managed by the associated failure detector module; p_i can only read this set.

Function `consensus()` is parameterized with the maximum number of failures t that can be tolerated. It consists of two concurrent tasks. The first task handles the receipt of a *decide* message (lines 2-3); it ensures that if a process p_i decides (line 3 or line 14), then all correct processes will also receive a *decide* message. The second task (lines 4–23) describes a round: it consists of a loop that constitutes the core of the protocol. Each (*current* or *next*) vote is labelled with its round number.⁵

- 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 rec_from_i (lines 9 and 18).
- 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).

⁵ 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 $r_i = r_k$.

- 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 14).
- When a process progresses from round r to round $r + 1$ it issues a *next* (line 22) 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).

The complete analysis and proof of this protocol can be found in [15].

5.2. Implementing a liveness process failure detector

This section proposes an implementation of an eventual strong liveness failure detector of class $\diamond\mathcal{S}\mathcal{T}\mathcal{S}$, relying on two assumptions, like in [9,10]: (1) a *partial synchrony* assumption, namely the existence of bounds on communication delays and relative speeds of processes, these bounds being unknown and required to hold only eventually, and (2) time assumptions on the Consensus protocol.

The liveness failure detector module associated with a process p_i , denoted $lfdm(i)$, manages a set of timers, one for each interacting process, and updates the variable *suspected_liveness_i*. To update such a variable, $lfdm(i)$ has to know, for each $p_j \in \Pi$, the following information concerning the consensus protocol (as specified in Section 4):

- (1) the set of messages (*next*, *current* and *decide*) exchanged between p_j (sender) and p_i (receiver) in the context of the consensus protocol;
- (2) the field attached to each protocol message exchanged between p_j and p_i (i.e., the round number r) that manifests the progress of p_j with respect to the consensus protocol;
- (3) the event $stop(p_j, p_i)$ in the consensus protocol corresponds to the receipt by p_i of a *decide* message from process p_j ;

The protocol described in Fig. 7 shows an implementation of $lfdm(i)$.

As a first action, $lfdm(i)$ starts a timer Δ_i for each process (line 1) to an initial value $init_\Delta$ and sets to zero the variable $r_i[j]$ showing the progress of a process p_j . Then it enters in an infinite loop. Upon the receipt of a consensus protocol message from a process p_j , $lfdm(i)$ first checks whether this message corresponds to $stop(p_j, p_i)$ event (line 4).

- In the affirmative, $lfdm(i)$ removes p_j from *suspected_liveness_i* (line 5) without starting a successive timer. In fact, from $lfdm(i)$'s point of view, process p_j terminated with success w.r.t p_i . and thus it has to be permanently removed from *suspected_liveness_i*.
- In the negative, it checks if there was a progress from the previous receipt from the same process (line 6). In the affirmative, it sets $r_i[j]$ to the new value and if p_j was

suspected to have suffered a liveness failure, it is removed from that list (line 8). Finally it sets the timer value to a new value $\Delta_i[j]$ defined according to several indices dependent from the protocol and from the network such as the structure of the protocol, the experienced network delays, etc. This is abstracted by the function `update()` (line 9). Then the timer $\Delta_i[j]$ is started.

5.3. Implementing a safety process failure detector

As explained in Section 4.3(2), a safety failure detector is composed of modules, one for each process, and the design of each module amounts to design a certification module and a verification module.

(1) *Designing the certification module.* Three types of messages are exchanged in the protocol, namely, *current*, *next* and *decide*. These messages carry the identity of their sender, the round number where they have been sent, and (except *next*), the current estimation of the sender.

Identity of the sender. This value is certified by the signature of the message, as explained in Section 4.3(2c).

Estimate value. This value, denoted est_i in the protocol, will be certified by the certificate est_cert_i . The initial value, $est_i = v_i$, cannot be certified by parts of processes histories. However, in a run, p_i could have to broadcast this initial value when it acts as coordinator (e.g., when $i = 1$ at the beginning of the first round). In such a case, it could happen that p_i , being unsafe, sends different est_i values to different processes, and the receivers would have no mean to detect this failure. So, in order to allow the detector to detect this particular failure, it is mandatory that the initial values are certified. We will suppose that their certificates are built by an external mechanism (e.g., by using tools provided at the Operating System level, or during a pre-processing phase as in [24]). We will say that the initial certificate est_cert_i is well-formed w.r.t est_i if it contains a piece of information from which the value v_i can be retrieved. The variable est_i can then take successive values: it can be updated at most once per round due to the delivery of the first *current* message received during this round. When this occurs, the certificate est_cert_i is also updated, with the certificate of the *current* message. Since this message is properly formed, its certificate $cert_k$ contains a correct certificate est_cert_k (i.e., a certificate well-formed with respect to the value est_k contained in the *current* message). During a round, est_cert_i is said to be “well-formed with respect to est_i ” if the value est_i is the value included in the messages contained in est_cert_k . Otherwise, it means that process p_i has either omitted to execute the update of est_i , or corrupted its value.

Similarly, a process decides a value est_c at round r either when it has received $(n - t)$ valid *current* messages (line 14) or when it receives a valid *decide* message from another process (lines 2-3). In the first case the process authenticates its decision by using a certificate $current_cert_i$ made

liveness_failure_detector(*suspected_liveness_i*)

```

(1) for all  $j \in \Pi$  do  $r_i[j] \leftarrow 0$ ;  $\Delta_i[j] \leftarrow \text{init}_\Delta$ ; start( $\Delta_i[j]$ ); enddo
(2) loop
(3) upon receipt of a protocol message TYPE( $r, \dots$ ) from  $p_j$ 
(4)   if TYPE = decide % event stop( $p_j, p_i$ ) %
(5)     then if  $p_j \in \text{suspected\_liveness}_i$  then  $\text{suspected\_liveness}_i \leftarrow \text{suspected\_liveness}_i - \{p_j\}$ ;
(6)     else if  $r > r_i[j]$ 
(7)       then  $r_i[j] \leftarrow r$ ; %  $p_j$  is not stalled w.r.t.  $p_i$  in the run of the consensus protocol %
(8)       if  $p_j \in \text{suspected\_liveness}_i$  then  $\text{suspected\_liveness}_i \leftarrow \text{suspected\_liveness}_i - \{p_j\}$ ;
(9)        $\Delta_i[j] \leftarrow \text{update}(\Delta_i[j])$ ; start( $\Delta_i[j]$ );
(10)    endif
(11)  endif

(12) upon the expiration of TIMER( $\Delta_i[j]$ )
(13)    $\text{suspected\_liveness}_i \leftarrow \text{suspected\_liveness}_i \cup \{p_j\}$ ;
(14)endloop

```

Fig. 7. An implementation of $lfdm(i)$ for the consensus protocol.

of the set of the $(n - t)$ signed *current* messages (line 14) received during the current round; this certificate is reset to the empty set at the beginning of each round. In the second case the message *decide* (with the same certificate) is relayed to the other processes (line 3).

We say that *current_cert_i* is well formed with respect to r and *est_vect* if the following conditions are satisfied:

- $|\text{current_cert}_i| = (n - t)$ (otherwise process p_i misevaluated the condition of line 14).
- the certificate of each message in *current_cert_i* contains a *est_cert_k* well formed with respect to *est_k*.
- the certificate of each message in *current_cert_i* contains a *next_cert* well formed with respect to r (see below the meaning of *next_cert*).

Round number. A process progress from round $r - 1$ to round r ($r > 1$) when it has received enough *next* messages. So, the new value of r (resulting from the assignment $r \leftarrow r + 1$ performed at the beginning of the new round) is certified by the signed messages *next* whose reception has triggered the start of the new round. This set of signed messages constitute the certificate *next_cert_i*. This certificate is reset to the empty set at the beginning of each round. If p_i is the coordinator, this certificate must be append to the *current* message sent line 6. So, it is saved in *old_next_cert_i* before being reset. At the end of a round $r - 1$ ($r > 1$) (resp. when a new round r starts), we say that *next_cert_i* (resp. *old_next_cert_i*) is well-formed with respect to $r - 1$ if the following two conditions are satisfied:

- $|\text{next_cert}_i| = (n - t)$ (resp. $|\text{old_next_cert}_i| = (n - t)$). Otherwise process p_i has misevaluated the condition of line 7.
- The value $r - 1$ is consistent with respect to the information in the $(n - t)$ *next* messages contained in *next_cert_i* (resp. *old_next_cert_i*), i.e., all messages refer to round

$r - 1$. Otherwise process p_i has corrupted the value of r at the beginning of the round (line 5). As far as round 1 is concerned, this value cannot be certified by *next* messages. Since *old_next_cert_i* is initialized to the empty set, we will say that *old_next_cert_i* is well-formed (with respect to round 0) if *old_next_cert_i* = \emptyset . Otherwise either p_i has corrupted r or c at the beginning of the round 1 (line 7) or has misevaluated the condition $i \neq c$ (line 12).

Sending events of messages current.

- At the beginning of a round r , the current coordinator p_c proposes its estimate *est_c* to become the decision value by broadcasting a *current* message carrying this value (line 6). This message is certified by *est_cert_c* \cup *old_next_cert_c*. *est_cert_c* is used to certify the value proposed by the coordinator *est_c*, that is, *est_cert_c* must be well formed with respect to *est_c*. *old_next_cert_c* is used to certify the value of the current round r , i.e., *old_next_cert_c* must be well formed with respect to $r - 1$.
- When p_i receives the first *current* valid message while in state q_0 (line 10), it relays a *current* message (line 12) by using the signed valid message *current* just received as a certificate. This certificate contains *est_cert_c* \cup *old_next_cert_c* used to certify r and *est_c* (as above).

Sending events of messages next.

- If, while it is in the initial state q_0 , p_i suspects the current coordinator, it broadcasts a *next* message (line 16) and moves to q_2 . This message is certified by *est_cert_i* \cup *current_cert_i* \cup *next_cert_i*. Those certificates (*current_cert_i* and *next_cert_i*) will be used by the verification module of the receiver to decide whether p_i has misevaluated or not the sending condition *state_i* = q_0 . In fact, *state_i* is not certified (it is not sent in the messages). So, the receiver can only use information contained in the *next*

messages to verify the condition. But, $state_i = q_0$ holds if and only if, in the current round, no *current* message has been received by p_i and p_i has not sent a *next* message, that is, $(|current_cert_i| = 0) \wedge \langle next(p_i, r_i), cert \rangle_i \notin next_cert_i$.

- When the predicate *change_mind* becomes true (line 19), in order to avoid a deadlock, process p_i broadcasts a *next* message to favor the transition to the next round (line 20). This message is certified by $current_cert_i \cup next_cert_i$. $current_cert_i$ and $next_cert_i$ are used to certify the non-misevaluation of the predicate $change_mind \equiv (state_i = q_1) \wedge (|rec_from_i| > (n - t))$. In fact, for the same reason as above, $state_i$ and rec_from_i are not certified. But, we have $state_i = q_1$ if and only if a *current* message has been received by p_i and p_i has not sent a *next* message, i.e., $(|current_cert_i| \geq 1) \wedge \langle next(p_i, r_i), cert \rangle_i \notin next_cert_i$. Also, $|rec_from_i|$ is null at the beginning of each round, and is incremented each time a message *current* or *next* is received. Thus, $|rec_from_i| = |\{p_\ell | \langle next(p_\ell, r_\ell), cert \rangle_\ell \in next_cert_i \vee \langle current(p_\ell, est, vect_\ell, r_\ell), cert \rangle_\ell \in current_cert_i\}|$.
- When a process progresses from round r to round $r + 1$ it issues a *next* message if it did not do so in the *while* loop. These *next* messages are used to prevent other processes from remaining blocked in round r (line 22). This message is certified by $next_cert_i$ which will allow a receiver to check the correct evaluation of the condition at line 7 by verifying if $next_cert_i$ is well formed with respect to r .

Messages decide. We have discussed above the use of certificates $current_cert$ to authenticate the decision to send those messages.

Text of the certification module. The text of the certification module attached to p_i is described in the Fig. 8.

(2) *The verification module.* The verification module of process p_i is composed of a set of finite state automata, one for each process. The finite state automaton monitoring p_k , denoted $VM(i, k)$, is depicted in Fig. 9. It represents the view p_i has, during the current round r_i , on the behavior of p_k with respect to the Consensus protocol. 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. $PF_{a,b}(type_of_message_k)$ returns true if the message is properly formed.⁶ The predicate $type_of_message_k$ is true if a syntactically correct message of that type has been received from p_k .⁷

Automaton states. The automaton $VM(i, k)$ is composed of five 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 final state (*final*) and one is the state declaring p_k is faulty (*faulty*). Note that the predicate $p_k \in suspected_safety_i$ is true if the automaton related to p_k of process p_i is in state *faulty*. It is false otherwise.

Automaton Transitions.

- *Transition $q_0 \rightarrow q_1$.* Upon the arrival of $\langle current(p_k, r_i, est_k), cert_k \rangle_k$ at $VM(i, k)$ there are two cases:
 - Case 1.** p_k is the coordinator (the message was sent by p_k from line 6). That is, $cert_k = est_cert_k \cup old_next_cert_k$. The predicate $PF_{0,1}(current_k)$ returns true if:
 - (1) est_cert_k is well-formed with respect to a value est_k , and
 - (2) $old_next_cert_k$ is well-formed with respect to the round number $r_i - 1$.
 - Case 2.** p_k is not the coordinator (the message was sent by p_k from line 12). That is, $cert_k = current_cert_k$. According to the protocol, the certificate must include the message $\langle current(p_c, est_c, r_i), cert \rangle_c$ in order to be properly formed. Then, once extracted this message from the certificate, all tests of case 1 can be executed on $est_cert_c \cup old_next_cert_c$.
- *Transition $q_0 \rightarrow q_2$.* Upon the arrival of $\langle next(p_k, r_i), cert_k \rangle_k$ at $VM(i, k)$ there are two cases:
 - Case 1.** If $cert_k$ does not contain any *current* message, it was sent by p_k from line 22 (with $cert_k = next_cert_k$). 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 7.
 - Case 2.** If $cert_k$ contains at least one *current* message, the *next* message was sent by p_k from line 16 (with $cert_k = current_cert_k \cup next_cert_k \cup est_cert_k$). This sending is guarded by the condition on the same line. $PF_{0,2}(next)$ is true if, by using the information contained in $cert_k$, est_cert_k is well-formed with respect est_k and if the condition $state_i = q_0$ (i.e., $(|current_cert_k| = 0) \wedge \langle next(p_k, r_i), cert \rangle_k \notin next_cert_k$) can be evaluated to true.
- Remark:** Note that even if $PF_{0,2}(next)$ is true, the process p_k could be faulty as it could misevaluate the predicate $p_c \in suspected_i$ used at line 15 and no information about that fault can be found in the certificate. But, as at most t processes are faulty, even though all of them misevaluate that predicate and generate false *next* messages, this does not prevent other processes to get a decision due to conditions of line 7.
- *Transition $q_1 \rightarrow q_2$.* Upon the arrival of $\langle next(p_k, r_i), cert \rangle_k$ at $VM(i, k)$, it executes the following tests on the certificate $cert_k$. The *next* message was sent by p_k from line 20 (with $cert_k = current_cert_k \cup next_cert_k$). That sending is guarded by the condition line 19, expressed in terms of certificates $cert_correct$ and $cert_next$, as explained page 20. $PF_{1,2}(next)$ is true if, using the information contained in $cert_k$, the predicate *change_mind* can be evaluated to true.
- *Transition $q_1 \rightarrow final$ and transition $q_2 \rightarrow final$.* These transitions occur upon the arrival of a message

⁶ The implementation of the set of predicates PF corresponds to the certificate analyzer (see Section 4.3(2a)). It allows to detect substitute and semantically incorrect messages.

⁷ The receipt of a syntactically incorrect message move the automaton from any state to *faulty* state. So, for a better reading, these edges are omitted from Fig. 9 and from the sequel of the text.

certification module

```

when  $r_i = 0$ 
   $est\_cert_i$  is an initial certificate;  $current\_cert_i \leftarrow \emptyset$ ;  $next\_cert_i \leftarrow \emptyset$ ;

when  $r_i$  changes its value
   $current\_cert_i \leftarrow \emptyset$ ;  $old\_next\_cert_i \leftarrow next\_cert_i$ ;  $next\_cert_i \leftarrow \emptyset$ ;

when  $current(p_i, r_i, est_i)$  is sent
  if  $i = (r_i \bmod n + 1) \% coordinator$ 
    then append  $old\_next\_cert_i \cup est\_cert_i$  to  $current(p_i, r_i, est_i)$ 
    else append  $current\_cert_i$  to  $current(p_i, r_i, est_i)$ ;
  pass the certified message to the signature module

when  $next(p_i, r_i)$  is sent
  if  $(|current\_cert_i| = 0) \wedge \langle next(p_i, r_i), cert_i \rangle_i \notin next\_cert_i$ 
    % i.e.  $state_i = q_0$  %
    then append  $current\_cert_i \cup next\_cert_i \cup est\_cert_i$  to  $next(p_i, r_i)$ 
    elseif  $change\_mind$ 
      %  $change\_mind$  is expressed in terms of certificates, as explained above page 20%
      then append  $current\_cert_i \cup next\_cert_i$  to  $next(p_i, r_i)$ 
      else append  $next\_cert_i$  to  $next(p_i, r_i)$ 
      % corresponds to line 22 in the protocol %
    endif ;
  pass the certified message to the signature module

when  $decide(p_i, est_i)$  is sent
  if  $decided_i$ 
    then append  $est\_cert_i$  to  $decide(p_i, est_i)$ ;
    else append  $current\_cert_i \cup est\_cert_i$  to  $decide(p_i, est_i)$ 
  endif ;
  pass the certified message to the signature module

when  $\langle current(p_k, r_i, est_k), cert \rangle_k$  is received
   $current\_certif_i \leftarrow current\_certif_i \cup \langle current(p_k, r_i, est_k), cert \rangle_k$ ;
  if  $|current\_certif_i| = 1$ 
    then  $est\_cert_i \leftarrow cert$ 
  endif;
  pass  $current(p_k, r_i, est_k)$  to  $sfdm$  module

when  $\langle next(p_k, r_i), cert \rangle_k$  is received
   $next\_cert_i \leftarrow next\_cert_i \cup \langle next(p_k, r_i), cert \rangle_k$ ;
  pass  $next(p_k, r_i)$  to  $sfdm$  module

when  $\langle decide(p_k, est_k), cert \rangle_j$  is received
  if not  $decided_i$ 
    then  $decided_i \leftarrow true$ ;  $est\_cert_i \leftarrow cert$ ;
  endif;
  pass  $decide(p_k, est_k)$  to  $sfdm$  module

```

Fig. 8. Certification module for the consensus protocol.

$\langle decide(p_k, est_k), cert \rangle_k$ at $VM(i, k)$. $PF_{1,2}(decide)$ (resp. $PF_{1,final}(decide)$) is true if $cert_k$ is well formed with respect to est_k and r_i .

- *Transition* $q_0 \rightarrow faulty$. Upon the arrival of a message, $VM(i, k)$ declares p_k faulty, if one of the following three conditions is true: (i) $PF_{0,1}(current)$ is false (in the case of the receipt of a *current* message); or (ii) $PF_{0,2}(next)$ is false (in the case of the receipt of a *next*

message); or (iii) the message is of type *decide*. In the case (iii), as channels are FIFO, a *next* or *decide* message has been omitted by p_k .

- *Transition* $q_1 \rightarrow faulty$. Upon the receipt of a message, $VM(i, k)$ declares p_k faulty, if one of the following three conditions is true: (i) $PF_{1,2}(next_k)$ is false (in the case of the receipt of a *next* message); or (ii) $PF_{1,final}(decide_k)$ is false (in the case of the receipt of a *decide* message); or

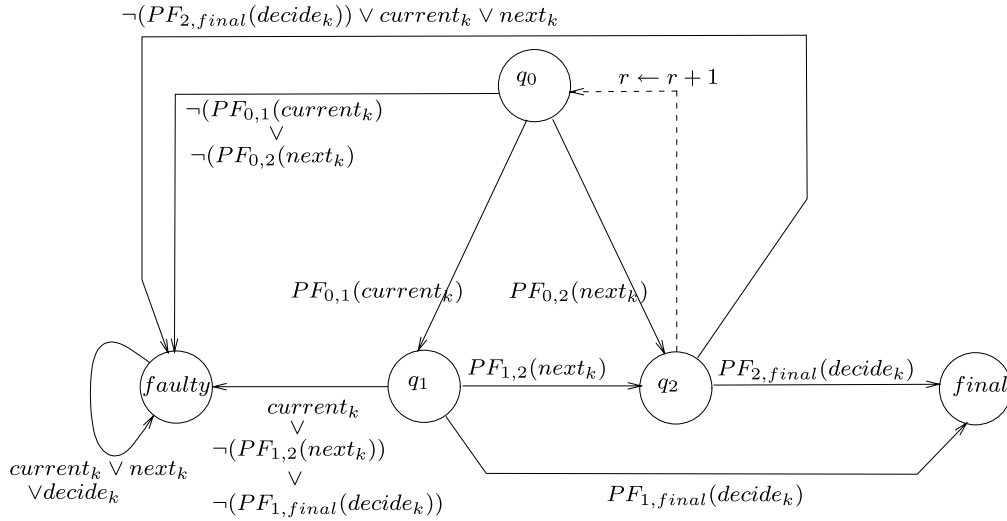


Fig. 9. The automaton of verification module of p_i w.r.t. p_k .

(iii) the message is of type *current*. In the latter case, as channels are FIFO, a *next* or *decide* message should be received by p_i from p_k , then p_k is faulty.

- *Transition $q_2 \rightarrow$ faulty*. Upon the receipt of a message, $VM(i, k)$ declares p_k faulty, if one of the following three conditions is true: (i) $PF_{2,final}(decide_k)$ is false (in the case of the receipt of a *decide* message); or (ii) the message is of type *current*; or (iii) the message is of type *next*. In the latter two cases, as channels are FIFO only a *decide* message should be received by p_i from p_k , if a message of type *current* or *next* is received from p_k , then p_k is faulty.
- *Transition faulty \rightarrow faulty*. Once p_k is declared *faulty* it remains in the state *faulty* whatever type of message is received.
- *Transition $q_2 \rightarrow q_0$ (denoted by a dotted line)*. This transition occurs when $VM(i, k)$ observes that p_i starts a new round (r_i changes its value). Note that it occurs “simultaneously” on all the $VM(i, k)$ ($k \in \Pi$). The fact that p_k moved to q_0 could also be detected if the successive message received by the process from p_k is either a *next* or a *current* message related to round $r + 1$.

5.4. The value of t

Let us finally remark that the number of arbitrary failures that can be tolerated by the Consensus protocol in Fig. 6 augmented with the safety and liveness failure detector is $t = \lfloor \frac{n-1}{2} \rfloor$ as in the case of the crash failures.

Many other Byzantine Agreement Protocols can tolerate a number of arbitrary failures $t < n/3$ (e.g., [2,4,5]). This difference is explained by the assumption that by zantine and correct processes run on the top of a reliable certification infrastructure formed by a certification and verification module, one for each process (Section 4.3). With reliable certification a process p is actually allowed to look into the history of another process q and, therefore p can

decide, without waiting messages from any other process, if q suffered an arbitrary failure. Previous protocols do not endow this assumption, therefore, a process is required to take such a decision based only on the content of the messages it receives from other processes (it is not allowed to look into the history of a process). This is why a process needs to receive a certain number of messages from other processes in order to come to the decision if some process suffered an arbitrary failure or not.

6. A second case study: the global data computation problem

The global data computation problem (*GDC*) can be defined as follows. Let $GD[1, \dots, n]$ be a vector of data with one entry per process (the i th entry being associated with p_i) and let v_i denote the value provided by p_i to fill its entry of the global data. *GDC* consists in building *GD* and providing each process with a copy of it. Let GD_i denote the local variable of p_i intended to contain the local copy of *GD*. The problem is formally specified by the following set of four properties (\perp denotes a default value that will be used instead of a proposed value when the corresponding process is not correct.)

Termination. Eventually, every correct process p_i decides a local vector GD_i .

Validity. No spurious initial value: $\forall i$: if p_i decides GD_i then $(\forall j : GD_i[j] \in \{v_j, \perp\})$.

Agreement. No two processes decide different global data: $\forall i, j$: if p_i decides GD_i and p_j decides GD_j then $(\forall k : (GD_i[k] = GD_j[k]))$.

Obligation. If a process decides, its initial value belongs to the global data: $\forall i$: if p_i decides GD_i then $(GD_i[i] = v_i)$.

In an asynchronous distributed system prone to process crashes, the *GDC* problem has no deterministic solution. This is an immediate consequence of the well known FLP

```

Task T1
(1)  $r_i := 0$ ;  $GD_i = [\perp, \dots, v_i, \dots, \perp]$ ;  $LP_i(0) := \emptyset$ ;  $GD\_Full_i := \infty$ ;
(2) loop % Sequence of asynchronous rounds %
(3)  $r_i := r_i + 1$ ;  $LP_i(r_i) := \emptyset$ ;
(4) send estimate( $GD_i, LP_i(r_i - 1), i, r_i$ ) to all;
(5) wait until forall  $j$ : % Waiting phase %
(6) (estimate( $GD_j, LP_j(r_i - 1), j, r_i$ ) received from  $j$ :  $rec_i[j] := \text{true}$ 
(7)  $\forall j \in suspected_i$ :  $rec_i[j] := \text{false}$ )
% Processing phase %
(8) forall  $j$  s.t.  $rec_i[j] \wedge ((\forall k : rec_i[k] \Rightarrow j \in LP_k(r_i - 1)) \vee (r_i = 1))$  do
% No process suspected  $p_j$  during the previous round %
% Update  $LP_i$  and consider the contribution of  $p_j$  %
(9)  $LP_i(r_i) := LP_i(r_i) \cup \{j\}$  % Update  $LP_i$ :  $p_i$  “considers”  $p_j$  %
(10) forall  $k$  s.t.  $GD_j[k] \neq \perp$  do  $GD_i[k] := GD_j[k]$  endforall
(11) endforall;
(12) if forall  $j$  :  $GD_i[j] \neq \perp$  then % All values are known %
(13)  $GD\_Full_i := \min(r_i, GD\_Full_i)$  endif;
(14) if ( $C1 \vee C2 \vee C3 \vee C4$ ) then % Send the decision, decide and stop %
(15) send decide( $GD_i$ ) to all; return  $GD_i$  endif
(16) endloop

Task T2 % Upon the receipt of a decision: propagate it, decide and stop %
(17) wait until decide( $GD$ ) is received: send decide( $GD$ ) to all ; return  $GD$ 

```

Fig. 10. Early deciding global data computation protocol.

impossibility result related to consensus [12]. Hence, the system has to be enriched with additional properties in order that the problem becomes solvable in a deterministic way. It has been shown that, when the system is equipped with a failure detector that outputs lists of processes suspected to have crashed [6], the *GDC* problem requires a *perfect* crash failure detector [14], i.e., a crash failure detector satisfying *eventual completeness* and *strong accuracy*. In particular, this problem is strictly *harder* than Consensus, since it is not possible to obtain a solution to *GDC* from a solution to Consensus (however, the converse is obviously true).

In the literature, a few solutions have been proposed in asynchronous distributed system prone to process crashes, augmented with a perfect crash failure detector [13,14,8]. All these solutions rest on round-based protocols. If n denotes the number of processes, t the maximum number of processes that can crash and f the number of actual crashes, the solution proposed in [8] decides in at most $\min(n, t + 1, f + 2)$ rounds, a result proved to be optimal in the number of rounds. The case study presented here consists in designing its liveness and safety failure detectors.

6.1. The Delporte–Fauconnier–Helary–Raynal protocol (DFHR)

This protocol proceeds in asynchronous rounds: each process proceeds in a sequence of rounds, and terminates as soon as it can decide by meeting a *decision condition* at the end of a round, or by receiving a decision message from another process having decided. There is no restriction on

the number of processes that can fail. During each round, each process (1) sends to each other an *estimate* message, piggy-backing the data *GD* and *LP* (see below) (2) waits to have received an *estimate* message from each process which it does not suspect, and (3) performs some local computation updating its local variables. Each process decides at the end of a round as soon as it meets any of the four conditions denoted by (*C1*), (*C2*), (*C3*), (*C4*) (lines 14, 15). In that case, it decides its vector GD_i . Moreover, it can decide earlier, if it receives a message *decide* sent by a process that has already decided (line 17). In that case, it decides the vector attached to the message.

The precise definition of the underlying computation model (asynchronous system + process crashes + rounds), the protocol principles and its proof are described in [8].

(a) *Data structures*. Each process p_i manages the following data structures:

- r_i : p_i 's round number. Initialized to 0 (line 1) it is incremented at the beginning of each round (line 3).
- GD_i : vector that contains p_i 's current estimate of the global data. Initially, with v_i denoting the value provided by p_i to fill its entry of the global data, $GD_i = [\perp, \dots, v_i, \dots]$ (line 1). The protocol ensures that, at any time, $\forall k : GD_i[k] = v_k$ or $GD_i[k] = \perp$. The GD_i vector is updated after the waiting phase according to the vectors GD_j received from the other processes during this round (line 10), and appended to the *estimate* messages sent by p_i at the next round (line 4).
- $LP_i(r)$: set containing the processes that p_i “considers” in round r . At the beginning of each round, this set is reset to empty (lines 1, 3). It is updated after the waiting phase

by including all the processes that p_i “takes into consideration” (line 9). Those are the processes (1) from which p_i received a message during the current round, and (2) that have been taken into consideration in the previous round by all the processes from which p_i has received an *estimate* message in this round (line 8). To maintain this information, $LP_i(r-1)$ is appended to the messages *estimate* sent by p_i at round r (line 4).

- rec_i : boolean vector such that $rec_i[j]$ is true iff p_i has received a message from p_j in the current round. This array, set by p_i during each waiting phase (lines 6 and 7) is then used to update $LP_i(r)$ (test of line 8).
- $suspected_i$: set of processes currently suspected by p_i (perfect failure detector).
- GD_Full_i : number of the first round (if any) where p_i has got all values. Initially, its value is $+\infty$. *Stop conditions.*
- (C1): $r_i = \min(t+1, n)$.
- (C2): $LP_i(r_i-3) = LP_i(r_i-2)$.
- (C3): $(LP_i(r_i-2) = LP_i(r_i-1)) \wedge \forall j \in LP_i(r_i) : LP_j(r_i-1) = LP_i(r_i-1)$.
- (C4): $(GD_Full_i \leq r_i-1) \wedge (\forall j \in LP_i(r_i) : GD_j = GD_i)$.

6.2. Implementing a liveness process failure detector

This section proposes the design of a perfect liveness failure detector of class $\mathcal{S}\mathcal{T}\mathcal{P}_{DFHR}$.

In the case of crash failures, a perfect detector can be implemented in a synchronous distributed system, for which there exists a known bound δ on every communication. Under the same assumption⁸ (hereafter the *synchrony* assumption), a perfect liveness failure detector for the DFHR protocol can be implemented. The idea (that will be formally proved below) is the following: if p_i is correct, it has completed any round r by its local time $\delta * r$. Thus, if a process p_j is not stalled w.r.t p_i , then p_i should have received the message *estimate*(\cdot, \cdot, j, r) by this time. The implementation of the perfect liveness failure detector is based on these properties.

The program of the detector module for the process p_i is shown Fig. 11. This module manages the variables Δ_i , ρ_i and arr_i , with the following signification:

- Δ_i is a local timer, reset to 0 every δ unit of local times,
- ρ_i is an integer measuring the number of times where Δ_i has been reset to 0,
- arr_i is an array of integer sets, such that $r \in arr_i[j]$ means that p_i has received a message *estimate*(\cdot, \cdot, j, r).

(b) *Proof of eventual completeness.*

Theorem 1. *The liveness failure detector implemented Fig. 11 satisfies Eventual Completeness.*

Proof. Let p_i be a correct process, and p_j stalled w.r.t p_i . Let r_j be the greatest integer such that *estimate*(\cdot, \cdot, j, r_j) is received by p_i . Such an integer exists because p_j is stalled w.r.t p_i . When p_i takes the values r_j+1 , either $j \in suspected_i$ or else, as *estimate*(\cdot, \cdot, j, r_j+1) has not been received by p_i , we have $r_j+1 \notin arr_i[j]$. Thus, from line 7, j is included in $suspected_i$. \square

(c) *Proof of strong accuracy.*

Lemma 1. *Under the synchrony assumption, each correct process p_i completes any of its round r by its local time $r * \delta$.*

Proof. The proof is by induction on r . Let p_i be a correct process. *Base case.* When p_i completes its round 1, for each p_j it has either received a message *estimate*($\cdot, \cdot, j, 1$) or $j \in suspected_i$. If the first event occurs, it is not later than δ . Otherwise, j is included in $suspected_i$ at time δ (line 7).

Induction. Suppose the property is true up to round $r-1$. Any correct process p_j starts its round r not later than $\delta * (r-1)$. Thus, for every j that does not belong to $suspected_i$ at the beginning of round r , either p_i receives the message *estimate*(\cdot, \cdot, j, r) before time $\delta * r$, or j is included in $suspected_i$ at time $\delta * (r-1) + \delta = \delta * r$. \square

Theorem 2. *Under the synchrony assumption, the liveness failure detector implemented Fig. 11 satisfies Strong Accuracy.*

Proof. Let p_i be a correct process and p_j be a process not stalled w.r.t p_i . $\forall r \geq 1$, p_j sends its message *estimate*(\cdot, \cdot, j, r) not later than $\delta * (r-1)$ (Lemma 1). By the synchrony assumption, this message arrives at p_i not later than p_i 's local time $\delta * r$, and thus, when $\rho_i = r$, j is not included in $suspected_i$. \square

The previous implementation can be improved, if we take into account the actions of the safety failure detector described thereafter. In fact, this detector will filter out the wrong *estimate* messages received by a process. In particular, it will not allow a process p_i to receive two messages *estimate* from a same process p_j with the same round number r . So, the test of line 10 will be useless. Moreover, the analysis of the protocol shows that, while a process p_i executes its round r_i , it can receive *estimate*(\cdot, \cdot, \cdot, r) with $r = r_i$ or $r = r_i + 1$. So, the size of the sets $arr_i[j]$ can be bounded to two.

6.3. Implementing a safety process failure detector

(1) *The certification module.* Protocol messages are of two types: *estimate* and *decide*. The fields GD and LP of a message *estimate* sent by p_i at round r ($r \geq 2$) are the values GD_i and LP_i at the end of round $r-1$. These values have been updated from the values contained in messages *estimate* received by p_i in round $r-1$. So, they are certified by the signed messages received by p_i during round $r-1$. Similarly, the field GD of a message *decide* sent by p_i at

⁸ in fact the assumption can be limited, here, to the *estimate* messages.

```

perfect_liveness_failure_detector(suspected_liveness)

(1)  $\Delta_i \leftarrow 0$ ;  $\rho_i \leftarrow 0$ ;
for all  $j \in \Pi$  do  $arr_i[j] \leftarrow \emptyset$  enddo

(2) loop
(3) until a message decide is sent or received
(4)   when  $\Delta_i$  clicks  $\delta$  do
(5)      $\rho_i \leftarrow \rho_i + 1$ ;  $\Delta_i \leftarrow 0$ ;
(6)     for each  $j$  such that  $j \notin suspected_i \wedge \rho_i \notin arr_i[j]$ 
(7)       do  $suspected_i \leftarrow suspected_i \cup \{j\}$  enddo
(8)   enddo
(9)   upon receipt of estimate( $\dots, j, r$ ) do
(10)    if  $r \notin arr_i[j]$ 
(11)    then  $arr_i[j] \leftarrow arr_i[j] \cup \{j\}$ 
(12)    endif
(13)  enddo
(14) endloop

```

Fig. 11. A perfect liveness failure detector for the DFHR protocol.

round r ($r \geq 2$) are either the values GD_i and LP_i at the end of round r , or the value contained in the message *decide* just received by p_i . So, in the first case, this value is certified by the signed messages *estimate* received by p_i during round r , in the second case by the signed message *decide* just received by p_i . Also, in the first case, the decision to send a message *decide* is based on the validity of one of the stop conditions. This validity is certified by the certificate of the message *decide*. In the first round, the values GD and LP sent by p_i are known to all processes, except for the *initial* value v_i proposed by p_i . Clearly, as each process is free to propose an arbitrary value, these initial values cannot (and have not to) be certified. The initial certificate of GD is thus empty.

Finally, as the verification module accepts, during a round r , *estimate* messages sent during round r or $r + 1$, the certification module stores the “early” messages in a buffer in order to process them in the next round.

The text of the certification module attached to p_i is described in the Fig. 12.

(2) *The verification module.* The automaton of process p_i related to a process p_j , hereafter denoted $VM(i, j)$, monitors the messages received by p_i from p_j , after being filtered out by the signature module. From the analysis of the protocol [8], the only *estimate* messages that a process can receive during its round r are those sent during the round r or $r + 1$ of their sender. Both messages are verified by $VM(i, j)$ and, if they are correct, are passed to the certification module. So, during a given round r_i , the valid sequences of *estimate* messages received by $VM(i, j)$ are (round numbers fields) $[r_i]$, $[r_i \cdot r_{i+1}]$, $[r_{i+1} \cdot r_i]$ and $[r_{i+1}]$. The latter case means that p_j has failed to send the *estimate*(\dots, j, r_i) message, but this will be detected by the liveness failure detector.

The finite state automaton $VM(i, j)$ is described Fig. 13. It is composed of six states:

- $VM(i, j)$ is in state q_0 when p_i starts a new round.
- $VM(i, j)$ is in state q_1 when, during the current round r_i , exactly one *estimate*(\dots, j, r_i) message is arrived at $VM(i, j)$.
- $VM(i, j)$ is in state q_2 when, during the current round r_i , exactly one *estimate*($\dots, j, r_i + 1$) message is arrived at $VM(i, j)$.
- $VM(i, j)$ is in state q_3 when, during the current round r_i , two *estimate*(\dots, j, r) messages with $r = r_i$ and $r = r_i + 1$ have arrived at $VM(i, j)$.
- $VM(i, j)$ is in state *final* if a message *decide* has arrived at $VM(i, j)$. Note that this state is reached in particular when p_i decides because it meets one of the stop conditions (line 15 in Fig. 10) since, in that case, p_i sends a *decide* message to itself.
- $VM(i, j)$ is in state *faulty* as soon as a transition predicate has been evaluated to *false*.

The transitions and the associated predicates are the following:

- *Transition* $q_0 \rightarrow q_1$. It occurs when an $\langle estimate(\dots, j, r), cert \rangle_j$ message arrives (passed by the signature module). The predicate $PF_{0,1}(estimate_j)$ returns *true* if:
 1. *cert* is well-formed w.r.t r , and certifies that $r = r_i$, and
 2. *cert* is well-formed w.r.t GD , and
 3. *cert* is well-formed w.r.t LP
- *Transition* $q_0 \rightarrow q_2$. It occurs when an $\langle estimate(\dots, j, r), cert \rangle_j$ message arrives (passed by the signature module). The predicate $PF_{0,2}(estimate_j)$ returns *true* if:
 1. *cert* is well-formed w.r.t r , and certifies that $r = r_{i+1}$, and
 2. *cert* is well-formed w.r.t GD , and
 3. *cert* is well-formed w.r.t LP .

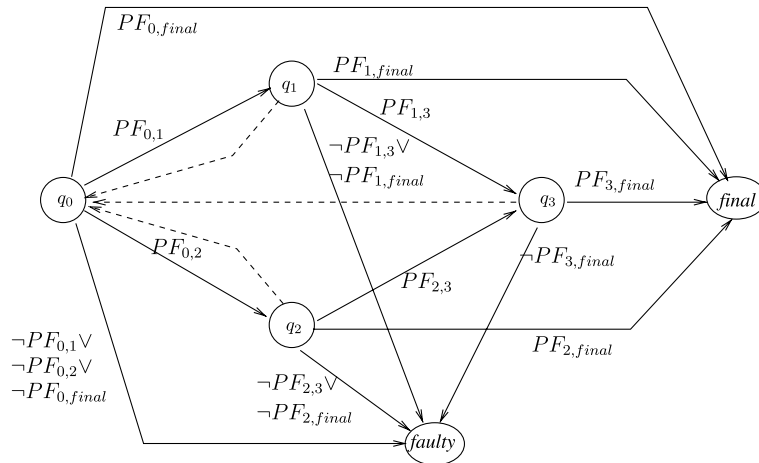
```

certification module

decidedi ← false ;
certifi ← ∅ ;
when ri changes its value
  previous_certifi ← certifi ;
  certifi ← ∅ ;
  deliver messages stored in the buffer
when estimate(GD, LP, i, ri) is sent
  append the certificate previous_certifi to estimate(GD, LP, i, ri);
  pass (estimate(GD, LP, i, ri), previous_certifi) to the signature module
when decide(GD) is sent
  if decidedi
    then append cert_decidei to decide(GD);
    pass (decide(GD), cert_decidei) to the signature module
  else append certifi to decide(GD);
    pass (decide(GD), certifi) to the signature module
  endif
when < estimate(GD, LP, j, r), cert >j is received
  % from the verification module or from the buffer
  if r = ri
    then certifi ← certifi ∪ < estimate(GD, LP, j, r), cert >j;
    pass estimate(GD, LP, j, r) to LFD module
  else store < estimate(GD, LP, j, r), cert >j in the buffer
  endif
when < decide(GD), cert >j is received
  if not decidedi
    then decidedi ← true;
    cert_decide ← cert;
  endif;
pass decide(GD) to LFD module

```

Fig. 12. Certification module for the DFHR protocol.

Fig. 13. The automaton of verification module of p_i w.r.t. p_j .

- **Transition $q_1 \rightarrow q_3$.** It occurs when an $\langle estimate(\cdot, \cdot, j, r), cert \rangle_j$ message arrives (passed by the signature module). The predicate $PF_{1,3}(estimate_j)$ is the same as $PF_{0,2}$.
- **Transition $q_2 \rightarrow q_3$.** It occurs when an $\langle estimate(\cdot, \cdot, j, r), cert \rangle_j$ message arrives (passed by the signature module). The predicate $PF_{2,3}(estimate_j)$ is the same as $PF_{0,1}$.
- **Transitions $q_i \rightarrow final$ ($i = 0, 1, 2, 3$).** These transitions occur as soon as a $\langle decide(GD), cert \rangle_j$ arrive (passed by the signature module). The predicated $PF_{i,final}(decide_j)$ ($i = 0, 1, 2, 3$) return *true* if *cert* is well-formed w.r.t *GD*.
- **Transitions $q_i \rightarrow faulty$ ($i = 0, 1, 2, 3$).** These transitions occur if one of the corresponding *PF* is found to be *false*.
- **Transitions $q_i \rightarrow q_0$ ($i = 1, 2, 3$).** They occur when *VM*(*i, j*) observes that p_i starts a new round. *VM*(*i, j*) increments r_i .

7. Conclusion

In this paper, we have proposed a methodology to design arbitrary failure detectors. This methodology lies on the assumption of reliable certification component. The methodology allows to completely reuse the code of a crash-failure resilient protocol while adapting its degree of fault tolerance by composing itself with well-designed components built through the application of the methodology. Compared to ad-hoc solutions that merge the task of detecting failures weaker than crashes with the code of the protocol, the methodology presented in this paper has the advantage that, starting from a protocol resilient to crash failures, it could automatically create components needed for making the protocol resilient to arbitrary failures.

This methodology opens some interesting scenario such as the realization of the assumption of a reliable certification infrastructure and the design, the implementation and the composition of distributed components that dynamically adapt the fault-tolerance of such protocols. The case studies included here show the feasibility of this approach.

References

- [1] R. Baldoni, J.-M. Helary, M. Raynal, From crash fault tolerance to arbitrary fault tolerance: towards a modular approach, in: Proceedings of the International Conference on Dependable Systems and Networks (DSN 00), New York, June 2000, pp. 283–292.
- [2] G. Bracha, An asynchronous $[(n - 1)/3]$ -resilient consensus protocol, in: Proceedings of the ACM Principles of Distributed Computing (PODC 84), 1984, pp. 154–162.
- [3] M. Ben-Or, Another advantage of free choice: completely asynchronous agreement protocols (extended abstract), in: Proceedings of the ACM Principles of Distributed Computing (PODC 83), 1983, pp. 27–30.
- [4] C. Cachin, K. Kursawe, K. Shoup, Random oracles in constantinople: practical asynchronous byzantine agreement using cryptography, *J. Cryptol.* 18 (3) (2005) 219–246.
- [5] R. Canetti, T. Rabin, Fast asynchronous Byzantine agreement with optimal resilience, in: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, 1993, pp. 42–51.
- [6] T. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, *J. ACM* 34 (1) (1996) 225–267.
- [7] M. Correja, N. Ferreira Neves, P. Verissimo, How to tolerate half less one byzantine nodes in practical distributed systems, in: Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS 04), October, 2004, pp. 174–183.
- [8] C. Delporte-Gallet, H. Fauconnier, J.-M. Hélary, M. Raynal, Early stopping in global data computation, *IEEE TPDS* 14 (9) (2003) 909–921.
- [9] A. Doudou, A. Schiper, Muteness failure detectors for consensus with Byzantine processes, Brief announcement, in: Proceedings of the 17th ACM PODC, 1998, p. 315.
- [10] A. Doudou, B. Garbinato, R. Guerraoui, A. Schiper, Muteness failure detectors: specification and implementation, in: Proceedings of the EDCC'99, LNCS, vol. 1667, Springer-Verlag, 1999, pp. 71–87.
- [11] N. Ferreira Neves, M. Correja, P. Verssimo, Solving vector consensus with a wormhole, *IEEE Trans. Parallel Distributed Comput.* 16 (12) (2005) 1120–1131.
- [12] M.J. Fischer, N. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, *J. ACM* 32 (2) (1985) 374–382.
- [13] V. Hadzilacos, S. Toueg, Reliable broadcast and related problems, in: S. Mullender (Ed.), *Distributed Systems*, ACM Press, New York, 1993, pp. 97–145.
- [14] J.M. Hélary, M. Hurfin, A. Mostefaoui, M. Raynal, F. Tronel, Computing global functions in asynchronous distributed systems with perfect failure detectors, *IEEE Trans. Parallel Distributed Comput.* 11 (9) (2000) 897–909.
- [15] M. Hurfin, M. Raynal, A simple and fast asynchronous consensus protocol based on a weak failure detector, *Distributed Comput.* 12 (4) (1999) 209–233.
- [16] K.P. Kihlstrom, L.E. Moser, P.M. Melliar-Smith, Solving consensus in a byzantine environment using an unreliable fault detector, in: Hermes (Ed.), *Proceedings of the First International Symposium on Principles of Distributed Systems (OPODIS'97)*, Chantilly, France, December 1997, pp. 61–76.
- [17] D. Malkhi, M. Reiter, Unreliable intrusion detection in distributed computations, in: Proceedings of the 10th IEEE Computer Security Foundations Workshop, Rockport, MA, June 1997, pp. 116–124.
- [18] A. Mostefaoui, M. Raynal, Solving consensus using Chandra–Toueg's unreliable failure detectors: a general quorum-based approach, in: Proceedings of the International Symposium on Distributed Computing, September 1999, pp. 49–63.
- [19] G. Neiger, S. Toueg, Automatically increasing the fault-tolerance of distributed algorithms, *J. Algorithm* 11 (3) (1990) 374–419.
- [20] D. Powell, Failure mode assumptions and assumption coverage, in: Proceedings of the FTCS-22, IEEE Computer Society Press, Boston, MA, USA, 1992, pp. 386–395.
- [21] M.O. Rabin, Randomized Byzantine generals, in: Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS 83), Tucson, Arizona, November 1983, pp. 403–409.
- [22] M. Raynal, A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News, Distrib. Comput. Column* 36 (1) (2005) 53–70.
- [23] R.L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Commun. ACM* 21 (2) (1978) 120–126.
- [24] A. Schiper, Early consensus in an asynchronous system with a weak failure detector, *Distrib. Comput.* 10 (1997) 149–157.