A Component-Based Methodology to Design Arbitrary Failure Detectors for Distributed Protocols

Roberto Baldoni[†], Jean-Michel Hélary[⋆], Sara Tucci Piergiovanni[†]
[†] Dip. di Informatica e Sistemistica "Antonio Ruberti", Universitá di Roma *La Sapienza*, Italy
[⋆]University of Rennes, Campus de Beaulieu, 35042 Rennes-cedex, France
baldoni,tucci@dis.uniroma1.it, helary@irisa.fr

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 paper proposes a component-based methodology allowing to take a protocol $\mathcal{A}$ resilient to crash failures and to add software components in order to adapt *the protocol $\mathcal{A}$ to be resilient to more general failures than crash, without changing the code of $\mathcal{A}$. On this basis, it introduces the notions of liveness failure detector and safety failure detector, two independent software components to be used by a protocol to increases its resilience respectively to liveness and safety failures of processes running the protocol. Then, the feasibility of this approach is shown, by providing an implementation of liveness failure detectors and of safety failure detectors for a protocol solving the problem of global data computation.*

Keywords: Composable systems, Asynchronous Systems, Arbitrary Failures, Liveness Failure Detector, Safety Failure Detector, Adaptive Fault Tolerance, Global Data Computation Problem.

1 Introduction

A common technique used to make distributed protocols resilient to process failures consists in the detection of such failures during run time, in order to eliminate the faulty processes. In the past, distributed protocols running on crash-prone system models merged the aspect related to fulfill its goal and the aspect related to the detection of crashes. Chandra and Toueg [4] 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 failure detector undertakes the burden of dealing with the asynchrony of the underlying system, letting the protocol designer concentrate on the essential part of the development without worrying about the underlying problems.

Formally, a failure detector component is defined by two properties: completeness (a property on the actual detection of process crashes), and accuracy (a property that restricts the mistakes on erroneous suspicions) which constraint the number of admissible mistakes done by a failure detector. Chandra and Toueg have also shown that the usage of failure detectors allows to solve the consensus problem in asynchronous crash-prone systems, thus circumnavigating the FLP impossibility result [10] stating that consensus problem has no deterministic solutions in such a system.

Designing solutions for distributed protocols in an environment where processes can exhibit arbitrary behavior (*e.g.*, omit to execute a statement or corrupt the value of a local variable) is notably more difficult than in a crash context [3]. As a consequence solutions used in the crash model are inadequate as a malicious process can exhibit failures more subtle than crashes and these failures can lead to the violation of the cor-

rectness criteria of the protocol. In the literature there have been examples of protocols resilient to crash failures which have been transformed into protocols resilient to arbitrary failures [1, 8, 14, 15, 16]. However, all these examples share a common factor: they change the original code of the protocol. Doudou et al. pointed out for the first time in [9] that, in the context of consensus handling muteness failures, a protocol designed in a crash-stop model can be reused, *modulo a few change in its code*, in the presence of a weaker failure semantic (i.e., muteness failures) simply replacing the failure detector $\diamond S$ with the one used by that semantic.

This paper makes a further advance in that direction by proposing a methodology based on software components which allows a crash-resilient protocol to adapt to weaker failure assumptions of the underlying system model[1]. This adaptation is done by composing the protocol with well-specified software components while completely reusing the code of the crash-resilient protocol. These software components can be systematically designed from the crash-resilient protocol code and the current system model assumptions.

More precisely, two main components are considered, dealing with liveness and safety issues respectively: the *liveness failure detector*, and the *safety failure detector*. This component-based structure is based on a classification of arbitrary process failures into liveness process failures and safety process failures, proposed in this paper. Previous specifications of failure detectors considered only some kinds of liveness failures (crashes [4], quietness [15], muteness [9, 14], omissions [7], to cite a few), and did not consider safety failures. Intuitively, given a protocol $\mathcal{A}$, a process p suffers a liveness failure if one or more processes detect that p does not progress w.r.t. specification of $\mathcal{A}$, and p suffers a safety process failure if one or more processes detect that p does not follow specification of $\mathcal{A}$.

Each of these failure detector components impose restrictions on the protocol. To detect liveness failures, $\mathcal{A}$ has to include in its specification some mechanism that allows a given process to show its progress to other processes running $\mathcal{A}$. The detection of safety failures, based on tools like signature, certification and state machine modelling of processes, requires $\mathcal{A}$ to have some regularity in its structure (as, e.g., to be round-based). Finally, as both components need to make reference to properties of a given protocol $\mathcal{A}$, contrarily to crash failure detector, the design of these components cannot be performed independently of the protocol that will use them [9].

removeFinally, these two components allow to reuse the code of a protocol $\mathcal{A}$, which is correct with respect to a system model prone to crash failures, in a system model with tougher failures, without impacting $\mathcal{A}$'s correctness.

The proposed methodology is feasible, as shown by its application to two case studies, namely a protocol solving the Consensus problem [13], and a protocol solving the Global Data Computation problem [6]. Due to space limitations, only the second case study is included here. The interested reader will find the first one in the full paper [2].

The paper is made of five sections. Section 2 presents the computation model and introduces the concepts of liveness failures and of safety failures. The principles underlying the design of liveness failure detectors and of safety failure detectors are addressed in Sections 3 and 4 respectively. Finally, Section 5 presents the case study.

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 *safety* and *liveness* properties (hereafter correctness criteria). A protocol is designed with respect to a *system model*. A system model describes the assumptions on 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 correctness criteria of the protocol.

2.2 System Model

System models considered in this paper share the following characteristics:

[1] The implementation of a system is based on resources: processors, memories, buffers, network, etc., constituting the "physical" environment. Such an environment can change its properties. For example, the reliable network can start lose messages; a buffer can start to be unavailable (due to an overflow problem or to physical crash) etc.

• 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.

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* ([11]):

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

In the rest of this paper, the following terminology [17] will be used.

Process faults are the underlying causes of process failures. Examples of process faults are mistakes in process designing, software bugs, statement omissions, corruptions, hardware component failures used by the process etc. A *process failure* is the external and visible manifestation of one or more process faults (we denote a generic process failure as *arbitrary process failure*). During the execution of a protocol $\mathcal{A}$, the manifestation of a failure of a process p passes through the analysis of messages sent by p while running $\mathcal{A}$. A crash failure of p will imply that expected messages never arrive to the intended destination while a corruption of a field of a message sent by p will imply a message that does not follow the program specification defined in $\mathcal{A}$. Note that some faults can be undetectable, as they do not exhibit an external manifestation (i.e., do not produce a failure). For example, if a process corrupts the value of a local variable, this might not produce a failure.

We classify arbitrary process failures in a run of a protocol $\mathcal{A}$ into two categories: liveness failures and safety failures. A process p suffers a *liveness failure* in a run of a protocol $\mathcal{A}$, if there exists a process q which detects that p does not progress w.r.t. evolution of $\mathcal{A}$. Crash process failures and muteness process failures (see next subsection) are example of liveness process failures.

Definition 1 *A process p is* live *(in a run of a protocol $\mathcal{A}$) if p never suffers any liveness failure during this run.*

A process p suffers a *safety process failure* in a run of a protocol $\mathcal{A}$, if there exists a process q which detects that p does not follow specification of $\mathcal{A}$. Corruption, transient omission, multiple statement execution, predicate misevaluation are examples of faults that can lead to safety failures.

Definition 2 *A process p is* safe *(in a run of a protocol $\mathcal{A}$) if p never suffers any safety failure during this run.*

Finally, note that a process fault could produce either (i) no process failure, (ii) one or more liveness process failures, (iii) one or more safety process failures, or, (iv) liveness and safety process failures.

Definition 3 *A process p is* correct *(in a run of a protocol $\mathcal{A}$) if p is live and safe in this run.*

2.4 The Nature of Process Liveness Failures

Liveness failures of a process p running a protocol $\mathcal{A}$ include the situation where process p crashes, or, more generally, is mute to another process q (i.e., p stops sending messages to q) with respect to $\mathcal{A}$. But liveness failures are not limited to these situations. In fact, p could be perceived by q as non mute, e.g., q continues to receive from p messages of the protocol $\mathcal{A}$, but the content of received messages indicates that p will not progress beyond an expected "point" in its execution. In this case we say that p is stalled with respect to process q (in this run of $\mathcal{A}$).

The following example shows a case where p is stalled with respect to q without being mute to q. Suppose that the protocol $\mathcal{A}$ governing p includes a code like the one shown in Figure 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 it will never reach statement $k+1$, and thus 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.

Note, from Definition 1, that, in a given run of $\mathcal{A}$, a process p is not live if and only if there exists at least one process q such that p is stalled w.r.t. q.

```
...
statement k.  % relevant event step(p,q)_k w.r.t.  q %
while not C do
     send m(k) to q ;
     do something else ;
endwhile ;
statement k+1 % relevant event step(p,q)_{k+1} w.r.t.  q %
...
```

Figure 1. Liveness process failures: an example

2.5 The Nature of Process Safety Failures

Safety failures of a process p can only be revealed by another processes q looking at the syntax and the semantic of the messages sent by p and received by q. If no message is sent by p in the protocol, no safety failure of p can be detected by other processes (in this case, however, safety failures experienced by p have no impact on the behavior of other processes running the protocol).

Let us explain the nature of the safety failures by showing that a process p could suffer, with respect to q, a safety failure without being stalled. Suppose p and q exchange messages over a bidirectional, reliable asynchronous FIFO channel. q sends, in a non-blocking way, a sequence (possibly infinite) of messages with an argument a. p echoes a and includes a local sequence number. So the code of the protocol governing p could be the one shown in Figure 2.

If p temporarily omits to receive some messages, those messages do not echo their argument to q as previous code will not be executed by p. q can detect this by comparing locally its expected echoed argument with the one contained in the incoming message. If this comparison is negative, q concludes that p omitted to execute at least the send statement of the previous code, thus, suffering a safety failure. Note that, as p increases its local sequence number i at each receipt, it actually shows a progress to q even in the presence of temporary receive omissions and, thus, p is not stalled with respect to q.

```
...
when a message m(a) arrives at p from q
    i:= i+1;
    send m(a,i) to q ;
endwhen
...
```

Figure 2. Safety process failures: an example

3 Handling Liveness Process Failures

3.1 Specifications of a Liveness Failure Detector

Intuitively, a liveness failure detector[2] is a distributed oracle aiming at detecting stalled processes. It is composed of *local* modules, one per process. The local module of each process p maintains the list of processes q that p suspects to be stalled w.r.t p. To be more precise, we adopt the model patterned after

[2] In the rest of the paper, a *failure detector component* will be more simply called *failure detector*.

the one in [4]. 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). These specifications are adapted in order to take into account the type of failures considered, namely: a correct process means a process that suffers neither liveness nor safety failures and, as the role of a liveness failure detector is to detect only stalled (i.e., non live) processes, suspected processes are restricted to *stalled* processes. With this informal discussion in mind, 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.

Eventual Weak Accuracy Eventually, there is *at least one* live process that will never be suspected by any correct process.

Eventual Strong Accuracy Eventually, *every* live process will never be suspected by other correct processes.

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

Strong Accuracy Any live process q will never be suspected by a correct process p.

Similarly with the notations introduced in [4] and widely used in the case of crash failure or muteness failure, we will denote by $\diamond\mathcal{STS}_{\mathcal{A}}$ 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 $\diamond\mathcal{STP}_{\mathcal{A}}$ the class of liveness failure detectors satisfying Eventual Completeness and Eventual Strong Accuracy. And we will denote by $\mathcal{STP}_{\mathcal{A}}$ 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.

[3]It is possible, as in [4], 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.

3.2 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 [9] 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 2.4, 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 $\mathcal{A}$.

Thus, to design a liveness failure detector for protocol $\mathcal{A}$ associated with process p it is necessary to recognize for each cooperating process q:

1. messages exchanged between q (sender) and p(receiver) within runs of the protocol $\mathcal{A}$;
2. a variable f, attached to each protocol message exchanged between q and p, that manifests the progress of q with respect to runs of $\mathcal{A}$;
3. the event $stop(q, p)$, denoting the termination with success of the the code run by p, with respect to q .

So if the liveness failure detector associated with a

process p receives protocol messages from a process q while the variable f remains unchanged, then, in this run of $\mathcal{A}$, it can suspect q to be stalled with respect to p.

3.3 Requirements imposed on $\mathcal{A}$

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. To this aim, let us consider each process "passes" over a sequence of "points", such that passing over a point is attested by a change in the value of a variable, transmitted by messages. If, during a run, a process q receives a sequence of messages from p, all with the same value of that variable, this might indicate to q that p fails to pass beyond the next updating point, i.e., that p is stalled with respect to q in this run.

More specifically, let consider any two processes p and q running $\mathcal{A}$. Any execution of p includes a sequence of relevant events, namely $step(p,q)_1, \ldots, step(p,q)_\ell, \ldots$, with respect to q and, possibly the event $stop(p,q)$, such that:

$$step(p,q)_1 <_l step(p,q)_2 <_l \ldots <_l step(p,q)_\ell <_l step(p,q)_{\ell+1} \ldots <_l stop(p,q)$$

where $<_l$ is the relation of local precedence on events on process p (note that the set of relevant events of p is a subset of the history of p). Between two consecutive $step(p,q)$ events, or between the last $step(p,q)$ event preceding the $stop(p,q)$ and the $stop(p,q)$ event, there is at least one send event of a message from p to q. After $stop(p,q)$, no send event of a message from p to q exists.

If $\mathcal{A}$ imbeds such a structure, then p is stalled w.r.t. q if there exists k such that $step(p,q)_k$ occurred and $step(p,q)_{k+1}$ or $stop(p,q)$ will never occur.

As an example, let us consider the code of Figure 1 when considering that the executions of statement k and of statement $k+1$ produce two successive relevant events with respect to q. In such a case, if p suffers a permanent receive omission fault then p will be stalled with respect to q [4]. 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. If q suspected p to be stalled, then this suspicion was wrong and q has to repent about it.

Let us also remark the importance of the event $stop(p,q)$. If the execution of p produces this event and if q becomes aware of it, then q will never more suspect process p to be stalled w.r.t. q, as p terminated correctly to run $\mathcal{A}$ w.r.t. q.

[4] In this particular case, the receive omission fault of p is perceived by an external process as a liveness failure.

4 Handling Safety Process Failures

4.1 Specifications of a Safety Failure Detector

The discussion presented 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.

However, contrarily to the case of liveness failure detectors, safety failure detectors are always perfect (they do not do mistakes). In fact, detecting safety failures rest on mechanisms (see the next section) that do not rely on "time", but on the very structure of the protocol. Perfect safety failure detectors enjoy the following properties

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

Strong Accuracy. Any safe process will never be suspected by a correct process' $sfdm$.

4.2 Hints for the Design of Safety Failure Detectors

As explained in Section 2.5, 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 mes-

sages 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. 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 of a state machine that models the protocol.

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

4.2.1 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_j to p_i, containing a value v. This value has been updated by p_j 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_j. 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_j'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 module* 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* [18], 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*.

Definition 4 *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_i, and certified with a certificate $cert$. The pair $(m, cert)$, signed with the unforgeable signature of p_i, will be denoted by $\langle m, cert \rangle_i$. 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 transform. The previous principles constitute a "guideline" 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.

4.2.2 State machines

Let us consider a state machine modelling the behavior of process p_i with respect to p_j. In this state machine, transitions are triggered when p_i receives a message from p_j. 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 to strengthen (just like the design of particular syntactic analyzers has to be done in the context of each grammar).

4.2.3 Structure of Safety Failure Detection Local Modules

A safety failure detector module associated with a process p_i (hereafter $SFDM_i$) is composed of three submodules (called also modules, for simplicity) : (i) a *signature* module, (ii) a *verification* module, and (iii) a *certification* module. More precisely, the structure of $SFDM_i$ is given in Figure 3. The same figure also shows the path followed by a message m (resp. m') received (resp. sent) by p_i.

A safety failure detector module can observe the state of the process p_i to which it is associated. In particular, it can read its variables.

The output of such a module is a set ($suspected\_safety_i$) of processes it detected to have sent a wrong or an unexpected message.

Signature module. Each signed message arriving at p_i is first processed by this module which verifies the signature of the sender (by using its public key). If the signature of the message is inconsistent with the identity field contained in the message, the message is discarded and its sender identity (known thanks to the unforgeable signature), is added to the local output $suspected\_safety_i$. Otherwise, the signed message is passed to the verification module. Also, each message sent by p_i 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 [18].

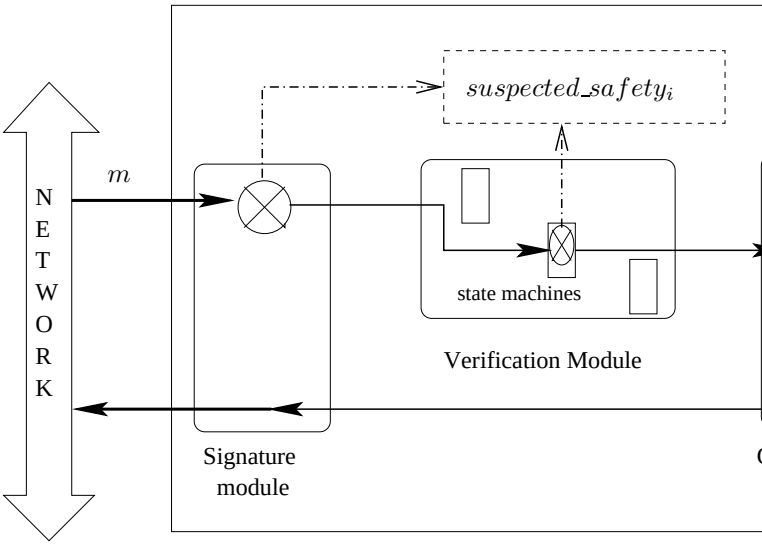

Figure 3. Structure of a local safety failure detector

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 $suspected\_safety_i$.

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_i is correct and $p_j \in suspected\_safety_i$, then p_j has experienced an incorrect behavior detected by the verification module of p_i. 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_i (as described in Section 4.2.1).

4.3 Requirements imposed on $\mathcal{A}$

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-sate machine that models 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 protocols, such a design is possible. In a round protocol, each process p 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.

Fortunately, the case study (Section 5) meet these requirements: it is a round-based protocol, exchanging a predefined and well-structured flow of messages during each round.

5 Case Study: the Global Data Computation Problem

The Global Data Computation Problem (GDC) can be defined as follows. Let $GD[1..n]$ 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 impossibility result related to consensus [10]. 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 [4], the GDC problem requires a *perfect* crash failure detector [12], 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 [11, 12, 6]. 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 [6] 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 making this protocol resilient to arbitrary failures, by designing ad-hoc liveness and safety failure detectors, without changing its original code. The variable $suspected$, read by the protocol, is updated by the appropriate failure detectors: in the present case, this variable is the union of the two variables $suspected\_liveness$ and $suspected\_safety$ updated respectively by the liveness failure detector and the safety failure detector components.

5.1 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, piggybacking 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 [9]), the protocol principles and its proof are described in [6].

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, $GD_i = [\bot, \ldots, v_i, \ldots]$ (line 1). The protocol ensures that, at any time, $\forall k : GD_i[k] = v_k$ or $GD_i[k] = \bot$. 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

• $(C1)$: $r_i = min(t+1, n)$.

• $(C2)$: $LP_i(r_i - 3) = LP_i(r_i - 2)$.

• $(C3)$: $\big((LP_i(r_i - 2) = LP_i(r_i - 1)) \wedge (\forall j \in LP_i(r_i) : LP_j(r_i - 1) = LP_i(r_i - 1)\big)$.

• $(C4)$: $(GD\_Full_i \leq r_i - 1) \wedge (\forall j \in LP_i(r_i) : GD_j = GD_i)$.

```
Task T1
(1) r_i := 0; GD_i = [⊥,...,v_i,...,⊥]; LP_i(0) := ∅; GD_Full_i := ∞;
(2) loop    % Sequence of asynchronous rounds %
(3)   r_i := r_i + 1; LP_i(r_i) := ∅;
(4)   send estimate(GD_i, LP_i(r_i − 1), i, r_i) to all;
(5)   waituntil forall j: % Waiting phase %
(6)       ( estimate(GD_j, LP_j(r_i − 1), j, r_i) received from j: rec_i[j] := true
(7)       ∨j ∈ suspected_i:                                        rec_i[j] := false)
      % Processing phase %
(8)   forall j s.t. rec_i[j] ∧ ((∀k : rec_i[k] ⇒ j ∈ LP_k(r_i − 1)) ∨ (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) ∪ {j} % Update LP_i: p_i "considers" p_j %
(10)      forallk s.t. GD_j[k] ≠ ⊥ do GD_i[k] := GD_j[k] endforall
(11)  endforall;
(12)  if forall j : GD_i[j] ≠ ⊥) then % All values are known %
(13)      GD_Full_i := min(r_i, GD_Full_i) endif;
(14)  if (C1 ∨ C2 ∨ C3 ∨ 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 untildecide(GD) is received: send decide(GD) to all ; return GD
```

Figure 4. Early Deciding Global Data Computation Protocol

5.2 Implementing a Liveness Process Failure Detector

In order to make this protocol resilient to liveness failures, a perfect liveness failure detector of class $\mathcal{STP}_{DFHR}$ will be used.

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 assumption[5] (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 $estimate(.,.,j,r)$ 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 Figure 5. 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 $estimate(.,.,j,r)$.

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** Δ_i clicks δ **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** $estimate(.,.,j,r)$ **do**
(10) **if** $r \notin arr_i[j]$
(11) **then** $arr_i[j] \leftarrow arr_i[j] \cup \{r\}$
(12) **endif**
(13) **enddo**
(14) **endloop**

Figure 5. A Perfect Liveness failure detector for the DFHR Protocol

Proof of Eventual Completeness

Theorem 1 *The liveness failure detector implemented Figure 5 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 $estimate(.,.,j,r_j)$ is received by p_i. Such an integer exists because p_j is stalled w.r.t p_i. When ρ_i takes the values $r_j + 1$, either $j \in suspected_i$ or else, as $estimate(.,.,j,r_j + 1)$ has not been received by p_i, we have $r_j + 1 \notin arr_i[j]$. Thus, from line 7, j is included in $suspected_i$. □

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 $estimate(.,.,j,1)$ 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 $estimate(.,.,j,r)$ before time $\delta * r$, or j is included in $suspected_i$ at time $\delta * (r-1) + \delta = \delta * r$. □

Theorem 2 *Under the synchrony assumption, the liveness failure detector implemented Figure 5 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 $estimate(.,.,j,r)$ 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 $suspected_i$. □

[5] in fact the assumption can be limited, here, to the $estimate$ messages.

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 $estimate(.,.,.,r)$ with $r = r_i$ or $r = r_i + 1$. So, the size of the sets $arr_i[j]$ can be bounded to two.

5.3 Implementing a Safety Process Failure Detector

5.3.1 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 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 if 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 Figure 6.

certification module

$decided_i \leftarrow$ *false* ;
$certif_i \leftarrow \emptyset$;
when r_i changes its value
  $previous\_certif_i \leftarrow certif_i$;
  $certif_i \leftarrow \emptyset$;
  deliver messages stored in the buffer
when $estimate(GD, LP, i, r_i)$ is sent
  append the certificate $previous\_certif_i$ **to** $estimate(GD, LP, i, r_i)$;
  pass ($estimate(GD, LP, i, r_i)$,$previous\_certif_i$) **to** the signature module
when $decide(GD)$ is sent
  if $decided_i$
    then append $cert\_decide_i$ **to** $decide(GD)$;
      pass ($decide(GD)$,$cert\_decide_i$) **to** the signature module
    else append $certif_i$ **to** $decide(GD)$;
      pass ($decide(GD)$, $certif_i$) **to** the signature module
  endif
when $< estimate(GD, LP, j, r), cert >_j$ is received
% from the verification module or from the buffer
  if $r = r_i$
    then $certif_i \leftarrow certif_i \cup < estimate(GD, LP, j, r), cert >_j$;
      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 $decided_i$
    then $decided_i \leftarrow true$;
      $cert\_decide_i \leftarrow cert$;
  endif;
  pass $decide(GD)$ **to** LFD module

Figure 6. Certification module for the DFHR Protocol

5.3.2 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 [6], 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 $estimate(.,.,j,r_i)$ message, but this will be detected by the liveness failure detector.

The finite state automaton $VM(i,j)$ is described Figure 7. It is composed of six states:

- $VM(i,j)$ is in sate q_0 when p_i starts a new round.

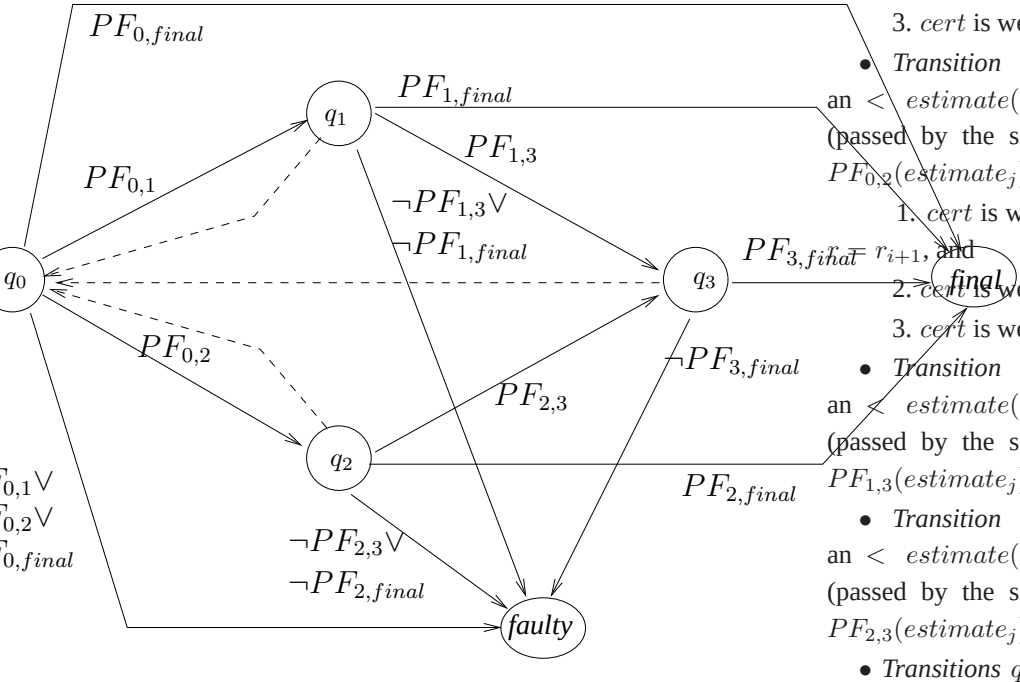

Figure 7. The verification module of p_i w.r.t. p_j

• $VM(i,j)$ is in state q_1 when, during the current round r_i, exactly one $estimate(.,.,j,r_i)$ message is arrived at $VM(i,j)$.

• $VM(i,j)$ is in state q_2 when, during the current round r_i, exactly one $estimate(.,.,j,r_i+1)$ message is arrived at $VM(i,j)$.

• $VM(i,j)$ is in state q_3 when, during the current round r_i, two $estimate(.,.,j,r)$ 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 Figure 4) 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 $< estimate(.,.,j,r), cert >_j$ 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 $< estimate(.,.,j,r), cert >_j$ 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.

• *Transition* $q_1 \rightarrow q_3$. It occurs when an $< estimate(.,.,j,r), cert >_j$ 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 $< estimate(.,.,j,r), cert >_j$ 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 $< decide(GD), cert >_j$ 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.

6 Conclusion

A few solutions have been proposed in the literature to increase the fault-tolerance of a protocol initially designed to be resilient to crash failures. All these solutions are based on the detection of faulty processes to eliminate them. But detection tools change the original code of the protocol. In this paper, we have proposed a component-based methodology which 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 software components, namely liveness and safety failure detectors. These components are designed from the system model we want to cope with and the original protocol resilient to crash-failures. The paper has presented one case study (another one is presented in the full paper [2]) that shows the feasibility of this approach.

This approach also raises several interesting open problems, e.g. :

• Detecting changes in the system model at run time in order to dynamically adapt the fault tolerance resiliency of a protocol.

• Studying the impact of the external components on the performance of the protocols (e.g., in the case study, the impact on the maximal number or rounds).

• Studying efficient (i.e., low complexity and fast) certification mechanisms.

Beyond these questions, that deserve attention, we believe that the ideas presented in this paper constitute an important step towards the construction of composable systems which are able to adapt their fault tolerance resilience to the environment.

References

[1] Baldoni R., Helary J-M and Raynal M., From Crash Fault Tolerance to Arbitrary Fault Tolerance: Towards a Modular Approach. *Proc. DSN'2000*, New York, pp.283-292, June 2000.

[2] Baldoni R., Helary J.M. and Tucci Piergiovanni S. A Component-Based Methodology to Design Arbitrary Failure Detectors for Distributed Protocols (long version), MIDLAB Technical Report 10/06 . http://www.dis.uniroma1.it/ midlab/publications.php, January 2006.

[3] Castro M. and Liskov B., Practical Byzantine Fault Tolerance. *Proc. 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans (LO), pp. 173-186, February 1999.

[4] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 34(1):225–267, March 1996.

[5] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[6] Delporte-Gallet C., Fauconnier H., Hélary J.-M. and Raynal M., Early Stopping in Global Data Computation, IEEE TPDS, 14(9):909–921, September 2003.

[7] Dolev D., Friedman R., Keidar I. and Malkhi D., Failure Detectors in Omission Failure Environments, *Brief announcement in Proc. 16th ACM PODC*, p.286, 1997.

[8] Doudou A. and Schiper A., Muteness Failure Detectors for Consensus with Byzantine Processes, *Brief announcement in Proc. 17th ACM PODC*, p. 315, 1998.

[9] Doudou A., Garbinato B., Guerraoui R. and Schiper A., Muteness Failure Detectors: Specification and Implementation. *Proc. EDCC'99*, Springer-Verlag LNCS 1667, pp. 71-87. September 1999.

[10] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[11] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

[12] Hélary J.M., Hurfin M., Mostefaoui A., Raynal M. and Tronel F., Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors. *IEEE TPDS*, 11(9):897-909, 2000.

[13] Hurfin M. and Raynal M., A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector, *Distributed Computing*, 12(4):209-233, 1999.

[14] Kihlstrom K.P., Moser L.E. and Melliar-Smith P.M., Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector. *Proc. First Int. Symposium on Principles of Distributed Systems (OPODIS'97)*, Hermes Ed., Chantilly (France), pp. 61-76, December 1997

[15] Malkhi D. and Reiter M., Unreliable Intrusion Detection in Distributed Computations. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pp. 116–124, Rockport (MA), June 1997.

[16] Neiger, .G and Toueg, S., Automatically Increasing the Fault-Tolerance of Distributed Algorithms. *Journal of Algorithms*, 11(3):374-419, 1990.

[17] Powell D., Failure Mode Assumptions and Assumption Coverage, in *Proc. FTCS-22*, Boston, MA, USA, pp.386-95, IEEE Computer Society Press, 1992.

[18] Rivest, R. L., Shamir, A. and Adleman, L., A Method for Obtaining Digital Signatures and Public-key Cryptosystems, *Communications of the ACM*, 21(2):120-126, February 1978.