Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors

J.-M. Hélary, M. Hurfin, A. Mostefaoui, M. Raynal, and F. Tronel

Abstract—A *Global Data* is a vector with one entry per process. Each entry must be filled with an appropriate value provided by the corresponding process. Several distributed computing problems amount to compute a function on a global data. This paper proposes a protocol to solve such problems in the context of asynchronous distributed systems where processes may fail by crashing. The main problem that has to be solved lies in computing the global data and in providing each noncrashed process with a copy of it, despite the possible crash of some processes. To be consistent, the global data must contain, at least, all the values provided by the processes that do not crash. This defines the *Global Data Computation (GDC)* problem. To solve this problem, processes execute a sequence of asynchronous rounds during which they construct, in a decentralized way, the value of the global data and eventually each process gets a copy of it. To cope with process crashes, the protocol uses a perfect failure detector. The proposed protocol has been designed to be time efficient: it allows early decision. Let t be the maximum number of processes that may crash, $t < n$ where n is the total number of processes, and f be the actual number of process crashes ($f \leq t$). In the worst case, the protocol terminates in $\min(2f+2, t+1)$ rounds. Moreover, the protocol does not require processes to exchange information on their perception of crashes. The message size depends only on the size of the global data.

Index Terms—Asynchronous distributed computation, global data, global function computation, perfect failure detector, problem reduction, process crash.

1 Introduction

In a distributed computation, a *Global Data* is a vector with one entry per process, each entry being filled with an appropriate value provided by the corresponding process. *Global Function* computation [11], [15] constitutes a key component for solving many distributed computing problems. It consists of: 1) requiring processes to define a global data; 2) computing a deterministic function of this data; and 3) providing each process with the corresponding result [2], [12]. The *Atomic Commitment* problem [3] constitutes a relevant example of a global function computation. According to its local computation, each process votes YES or NO. The set of all votes constitutes the global data. The result of the function is COMMIT if all votes are YES, otherwise the result is ABORT. Finally, according to the result (same for all processes), each process commits or invalidates the local computation it has previously done. More generally, some problems require repeated computations of a global function [11]. The *Distributed Termination Detection* problem is an example of problem that can be solved by two successive global function computations [15], [20], [27].

Computing a global function is relatively easy in reliable asynchronous distributed systems. In this context, two main approaches have been investigated. The centralized approach, also named "asymmetric" approach, consists in the following message exchange pattern: First, a predetermined process p gathers all the local data and computes the appropriate function of this global data. Then, the process p disseminates the result to each process. Assuming there is a channel connecting each pair of processes, this approach basically requires the exchange of $2(n-1)$ messages where n is the number of processes and costs two time units, assuming each message transfer costs one time unit and processing times are negligible. The second approach that has been studied is the distributed, or "symmetric," approach. Each process sends its data to all processes and, consequently, each process can build a copy of the global data. Then, each process computes the same deterministic function on the same global data and, thus, obtains the same result. With the previous assumptions, this approach costs $n(n-1)$ messages and only one time unit.

In this paper, we are interested in investigating the decentralized approach to compute global functions in distributed systems where processes can crash. Let t be the maximum number of processes that may crash ($t < n$) and f be the number of actual crashes ($f \leq t$). When the distributed system is *synchronous*, it is relatively easy to compute a global function in a decentralized way by requiring processes to execute a sequence of rounds. In that context, it has been shown that $\min(t+1, f+2)$ is a lowest bound on the number of rounds [8]. *Asynchronous* distributed systems are characterized by the fact that it is impossible to distinguish a very slow process from a crashed process. This makes the computation of a global

• *The authors are with IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France. E-mail: {helary, hurfin, mostefao, raynal, ftronel}@irisa.fr*

Manuscript received 23 Nov. 1999; accepted 25 May 2000.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 110988.

function more difficult to solve. One of the main problems created by process crashes concerns the fact that all noncrashed processes must get identical copies of the global data. To illustrate this issue let us consider the following scenario. It involves three processes p_1, p_2, and p_3. Processes p_1 and p_2 do not crash and broadcast their initial data and process p_3 sends its data to p_1 and crashes before sending it to p_2. Eventually, process p_2 detects the crash of p_3 and considers a global data without an entry from p_3. On the other hand, process p_1 has received values from p_2 and p_3 and considers global data with an entry from p_3. Thus, p_1 and p_2 do not have the same view of the global data. So, a crucial issue is to ensure that processes eventually get the same value for each entry of the global data. Let p be one of the processes. If p does not crash, its entry of the global data must be its initial data. If p has crashed before starting the protocol, its initial data is unavailable and, consequently, its entry will contain a default value denoted $\perp$. But if p crashes *during* the execution of the protocol, as p_3 in the previous example, what will be the value of its entry in the global data, the initial value of p or the default value?

To cope with process crashes, this paper follows the approach advocated by Chandra and Toueg [5]. Each process p is equipped with a failure detector module. The module associated with p can be seen as an oracle that provides p with the list of processes it suspects to have crashed. Formally, a failure detector module is defined by two properties, a completeness property and an accuracy property. Of course, the implementation of a failure detector is based on the use of timers and time-out values, but those are implementation mechanisms that are not made visible outside the failure detector module. Hiding all time-dependent aspects in a black box (the failure detector), allows the design of *time-free* protocols, i.e., protocols in which no statement involves physical time. As a consequence, a protocol based on a failure detector can be used without modification in any system where the assumed failure detector can be implemented. This not only makes easier the portability of the protocol, but also makes its proof independent on any particular timing mechanism.

The problem of computing a global data and providing each noncrashed process with a copy of it, is identified as the *Global Data Computation (GDC)* problem.[1] In asynchronous distributed systems subject to crash failures and equipped with perfect failure detectors, it is possible to solve the Global Data Computation Problem by executing n parallel instances (one for each process) of the Terminating Reliable Broadcast problem [14]. For each instance of the Terminating Reliable Broadcast problem (an instance is defined with respect to a specific sender process), a value v is delivered to each process. If $v = \perp$, then the sender has necessarily crashed. If $v \neq \perp$, then v is the initial value of the sender process. Solving an instance of the Terminating Reliable Broadcast problem requires $(t+1)$ rounds [14]. So, this approach for solving the Global Data Computation problem provides a solution requiring at least $(t+1)$ rounds. In this paper, we are interested in the design of a protocol that allows *early decision*. This means that the resulting protocol must involve a number of rounds that depend on f (the actual number of crashes), this number never being greater than $(t+1)$. Based on perfect failure detectors, the proposed protocol achieves the following time performances. If $t = 0$, i.e., when the system is reliable, the protocol requires a single round. When $t > 0$, it terminates in two rounds, in the best case. In the worst case, it terminates in $\min(2f+2, t+1)$ rounds. Given that one round of a synchronous system can be simulated by two rounds of an asynchronous system (see the discussion in Section 2.3), this time performance is likely to be optimal. However, the proof of this optimality issue remains an open question. So, to our knowledge, this is the first *early decision* protocol solving the Global Data Computation problem in asynchronous distributed systems prone to process crash failures. Other interesting features of the protocol are the following:

1. It adopts a "best effort" strategy, doing its best to fill each entry of the global data with the initial value of the corresponding process. This allows for a global data containing as much meaningful values as possible.
2. It does not require processes to exchange information on their perception of crashes. In addition to its identity, a message has only to carry an estimate of the global data value.

The paper is composed of eight sections. Section 2 presents the distributed system model. Section 3 specifies the *Global Data Computation* problem. Section 4 presents the protocol that solves this problem: Sections 4.1 and 4.2 describe the protocol and Section 4.4 proves the protocol correction and determines an upper bound for the number of rounds. Section 5 shows a simple, but interesting theoretical result, namely, the problem of building a perfect failure detector, problem *P*, and the *GDC* problem are "equivalent" in the sense that any solution to one of them can be transformed to solve the other [5], [10], [14]. Section 6 investigates a consensus-based approach to solve the *GDC* problem and compares it with the proposed approach. Then, Section 7 uses the protocol to solve two problems in asynchronous distributed systems with process crashes, namely, the *Atomic Commit* problem and the *Distributed Termination Detection* problem. Finally, Section 8 concludes the paper.

2 Distributed System Model

2.1 Asynchronous Distributed System

The system consists of a finite set of n processes, $\Pi = \{p_1, \dots, p_n\}$. Processes cooperate and synchronize by exchanging messages through a communication network, there is no shared memory. There is a communication channel connecting each pair of processes. Channels are reliable, i.e., no spurious messages, no loss, and no corruption. They are not required to be FIFO. Moreover, process speeds and communication delays are arbitrary. So, the system model perceived by the upper layer applications is the classical *time-free asynchronous distributed system model*.

1. The *GDC* problem is actually similar to the *Interactive Consistency* problem that has been defined in the context of synchronous systems prone to Byzantine process failures [22]. Concerning this similarity, see also the open problem (related to optimality [8]) stated at the end of the conclusion.

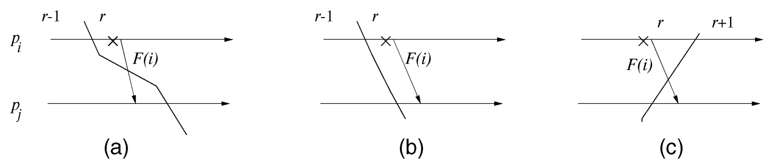


Fig. 1. First uncertainty.

The communication primitives will be denoted in the following way: When executed by p_i, the primitive "send m to p_j" entails the sending of the message m by p_i to p_j. When p_i executes "receive(m)", it is blocked until a message sent to it has arrived; then, the message is deposited in m and made available to p_i which continues its execution.

2.2 Process Failure Model and Failure Detection

Process Failure Model. A process may fail by crashing. It behaves correctly, i.e., it executes its program text, until it possibly crashes. When a process crashes, it definitively stops its activity. So, the failure model we consider is the Crash/no Recovery model. A process that does not crash is called a *correct* process. Otherwise, it is a *faulty* process. Recall that $t\ (< n)$ denote the maximum number of processes that may crash and $f\ (\leq t)$ denote the actual number of process crashes.

Process Crash Detection. Each process p_i is equipped with a failure detector module FD_i. This module provides p_i with a set variable ($suspected_i$) that contains the identities of the processes that FD_i guesses to have crashed. The process p_i can only read this variable, which is continuously updated by FD_i. According to the quality of guesses made by failure detector modules, several classes of failure detectors can be defined [5]. Here we consider perfect failure detector modules: no guess is mistaken. More precisely, these failure detectors are defined by the following two properties (when $p_j \in suspected_i$, we say that "p_i suspects p_j") [5]:

- *Completeness*: Eventually, every process that crashes is suspected by every correct process.
- *Accuracy*: No process is suspected before it crashes.

An important issue is the implementation of such failure detectors modules. Some networks have a notion of privileged, high priority channels satisfying strong timing assumptions [4], [23], [28] (e.g., field buses such as FIP or CAN [25]). Such channels can be dedicated to the implementation of perfect failure detector modules. So, at the application level, the computation model we actually consider is the *time-free asynchronous distributed system model augmented with perfect failure detectors* [5], [18]. This means that privileged, high priority channels are hidden to system applications. These channels are visible and known only by the underlying layer which implements the failure detector modules. This layer uses "I am alive" messages sent on privileged channels and timeout values to detect process crashes.

2.3 Asynchronous Systems with Perfect Failure Detectors vs. Synchronous Systems

Most agreement protocols that have been designed so far obey a regular communication pattern, based on the notion of *round* [1], [5], [8], [9], [15], [18], [27]. More precisely, each correct process p_i owns a variable r_i whose values are natural integers. As soon as $r_i = r$, we say that *process reaches round r*. Then, until $r_i = r + 1$, process p_i is said to be *in round r* [7]. While in its round r, each process executes sequentially the following steps: 1) It sends a round r message to the other processes, 2) it waits for a round r message from each process,[2] and 3) it executes local computations.

It is important to remark that, when we consider round-based protocols, synchronous distributed systems are more constrained than asynchronous distributed systems equipped with perfect failure detectors. This is due to the following observation: Let us consider two processes, p_i and p_j. Assume that, during round r, p_i has crashed after sending its round r message, namely m, to p_j.

- In a synchronous distributed system, the synchrony assumption ensures that all messages are received in the round they have been sent. So, p_j receives m before ending its round r. Consequently, p_j cannot detect the crash of p_i before executing in round $r + 1$.
- In an asynchronous distributed system, it is possible that p_i has started executing round r while p_j is executing round $r - 1$, r, or $r + 1$. So, if p_i crashes during r, p_j learns it, due to the underlying perfect failure detector,
  - while it is in round $r + 1$ (in that case the last message received by p_j from p_i is its round r message),
  - while it is in round r (in that case, the last message received by p_j from p_i is its round $r - 1$ message or its round r message), or
  - while it is executing round $r - 1$ (in that case, the last message received by p_j from p_i is its round $r - 2$ message or its round $r - 1$ message).

  Consequently, in such a system, the crash of p_i during r, is known by p_j while it is at round $r - 1$, r, or $r + 1$ (first uncertainty). Moreover, the crash of p_i can be known by p_j after or before having received p_i's round r message, or even before having received p_i's round $r - 1$ message (second uncertainty).

The three situations that give rise to the uncertainty from p_j's point of view, on the round number at which p_i crashed, are described in the Fig. 1 ($F(i)$ denotes the notification of the failure of p_i). The situations due to the second uncertainty (has p_i sent a round r or $r - 1$ message?) are described in Fig. 2. So, for round-based protocols, an

2. Let us note that in other round-based computation models, a process waits for messages from only a majority of processes.

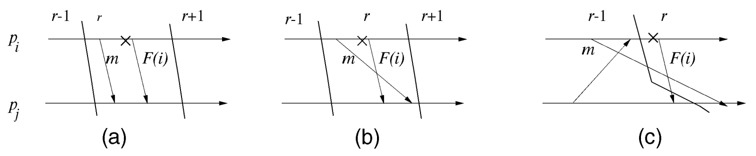


Fig. 2. Second uncertainty.

asynchronous distributed system, equipped with a perfect failure detector, is not equivalent to a synchronous distributed system.

It follows that round-based protocols are more difficult to design in asynchronous distributed systems equipped with a perfect failure detector than in synchronous systems.[3]

3 The GDC Problem: Constructing Local Copies of a Global Data

3.1 Distributed Computation of a Global Function

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 $\mathcal{F}$ be a deterministic function of GD. Moreover, let v_i denote the value provided by p_i to fill its entry of the global data.

The computation of the global function is described in Fig. 3. The function Global_data returns the value of GD that is locally saved by p_i in its local variable GD_i. The local variable $term_i$ is a Boolean (initialized to $false$) that takes the value $true$ when and only when, p_i has got its copy of GD. Then, each process can locally compute the value of $\mathcal{F}\ (GD)$.

3.2 The Global Data Computation Problem

As indicated in the Introduction, the crucial issue consists in building GD and providing a copy of it to each process. As defined previously, GD_i is the local variable of p_i intended to contain the local copy of GD. The problem of providing the same copy of GD to each process is formally specified by a set of four properties. These properties have to be satisfied by any protocol that claims to solve the problem. Let $\perp$ be a default value that will be used instead of the value v_j when the corresponding process p_j crashes prematurely. These properties are:

- Termination. The Boolean flag $term_i$ of every correct process p_i eventually becomes true: p_i correct $\Rightarrow \diamond term_i$.
- Validity. No spurious initial value: $\forall i: \ (term_i \Rightarrow (\forall j: \ GD_i[j] \in \{v_j, \perp\}))$.
- Agreement. No two processes decide different Global Data: $\forall i,j: \ ((term_i \wedge term_j) \Rightarrow (\forall k: (GD_i[k] = GD_j[k])))$.
- Obligation. If a process terminates, its initial value belongs to the Global Data: $\forall i: \ (term_i \Rightarrow (GD_i[i] = v_i))$.

3. When there are no process crashes, it is well-known how to to simulate synchronous systems on top of asynchronous systems: *synchronizers* [1], [15] perform such simulations.

The Termination property is a Nonblocking property (i.e., a Liveness property). It states that at least all correct processes must terminate despite the crash of other processes. The next three properties are Safety properties. The Validity property defines the value domain of each global data entry. The Agreement property indicates that all the processes that terminate get the same copy of the global data. Finally, the Obligation property states that if p_i terminates, then its entry of the global data cannot be the default value.

It is important to note that the Global Data Computation problem is harder than the Consensus problem [5], [9]. In the Consensus problem, processes propose values, and all correct processes have to agree on one of the proposed values. Here, the values "proposed" by, at least, all correct processes have to be pieced together to define the global data: The value "proposed" by a correct process can not be "missed." This informally explains why perfect failure detectors are needed: A correct process cannot be mistakenly suspected, as this could entail the absence of its value in the global data. Section 5 will provide a more formal treatment of this issue.

4 A Protocol That Builds Consistent Copies of a Global Data

As indicated in the introduction, our aim is the design of a distributed round-based protocol solving the GDC problem, that allows early decision. Moreover, differently from other protocols based on perfect failure detectors [18], we are interested in a protocol that does not require processes to exchange lists of suspects. Messages are allowed to carry only a control data (the identity of the message[4]) and an estimate of the value of the global data. Allowing messages to carry lists of suspects would give rise to more costly protocol.

4.1 The Protocol

The protocol that computes the global data and provides processes with the same copy of it is described in Fig. 4. Each process p_i calls the function Global_data(v_i) which returns a consistent copy of the global data. This function is made of two concurrent tasks $T1$ and $T2$.

Task $T1$. This is the main task: Its aim is to construct consistent local copies of the global data. To attain this goal, each process p_i has a local variable gd_i that contains its current estimate of the global data. Initially, gd_i contains

4. The identity of an *estimate* message is the pair defined by the identity of its sender plus its sequence number, namely, a round number.

```
Function Global_function

begin
        % local determination of $v_i$ ($\neq \perp$)%
        $GD_i \leftarrow$ Global_data($v_i$);
        $term_i \leftarrow true$;
        % local computation of $\mathcal{F}(GD_i)$ %
end
```

Fig. 3. Computing a global function.

only v_i, the value provided by p_i to fill its entry of the global data (line 1). Processes execute a sequence of asynchronous rounds to construct the global data. During round r they exchange their current estimates of the value of the global data, in order to improve their estimate value. The local variable r_i, initialized to zero, defines the current round number of p_i.

Let $\mathbf{part}(i, j, r)$ denote the predicate: "during its execution of round r, p_i has received and taken into account a round r estimate from p_j." In other words, from p_i's point of view, p_j has participated in round r. By convention, $\forall r \leq 0,\ \forall i,\ \forall j,\ \mathbf{part}(i, j, r) = true$. The local set variables $prev\_expected_i$, $cur\_expected_i$, and $next\_expected_i$ are used by p_i to keep track of round participants. They have the following meaning at the end of any round $r \geq 1$:

$prev\_expected_i = \{j | \mathbf{part}(i, j, r-2)\}$. This set contains the processes from which p_i has received a round $r-2$ message during its execution of round $r-2$. Hence, it is the set of processes that were expected by p_i to participate in $r-1$.

$cur\_expected_i = \{j | \mathbf{part}(i, j, r-1)\}$. This set contains the processes from which p_i has received a round $r-1$ message during its execution of round $r-1$. Hence, it is the set of processes that are expected by p_i to participate in the current round r.

$next\_expected_i = \{j | \mathbf{part}(i, j, r)\}$. This set contains the processes from which p_i has received a a round r message during its execution of r. Hence, it is the set of processes that are expected by p_i to participate in the next round $r+1$. Let x_i^r be the value of the set variable x_i at the end of round r. From these definitions, we have:

$$prev\_expected_i^r \supseteq cur\_expected_i^r \supseteq next\_expected_i^r,$$

and:

$$(prev\_expected_i^{r+1} = cur\_expected_i^r)$$
$$\wedge\ (cur\_expected_i^{r+1} = next\_expected_i^r).$$

A round r is made of two parts: a communication phase followed by a computation phase. More precisely, a process p_i does the following actions:

- Estimate exchange:
  - First, p_i starts its participation in the round by incrementing r_i, updating $prev\_expected_i$ and $cur\_expected_i$ (line 4). It also sends its current estimate of the global data (gd_i) to the processes of $cur\_expected_i$ (line 5). (Note that p_i always belongs to $cur\_expected_i$.)
  - Then, p_i waits until, for each $p_j \in cur\_expected_i$, either it receives the round r estimate of p_j (rec_gd_j), or it suspects p_j to have crashed (line 6).
- Local computation:
  - According to the set of estimates it has received, p_i computes $next\_expected_i$ (line 7) and updates its current estimate gd_i (lines 8-12).
  - Finally, p_i tests line 13, a Termination Condition (see next paragraph), to know whether it has got the final value of the global data. There are two cases. If the answer is negative, p_i proceeds to the next round (line 3). If the answer is positive, then p_i sends its current value of the global data, namely gd_i, to all the nonsuspected processes[5] (message $decide(i, gd_i)$ sent at line 14) and returns gd_i as the result of the call to Global_data(v_i) (**return** statement, line 14).

Task $T2$. This task is associated with the processing of a $decide(k, rec\_gd_k)$ message. If, during the execution of Global_data(v_i), such a message is received, p_i also forwards the final value of the global data, namely rec_gd_k, and locally returns it as the result of the call to Global_data(v_i) (return statement, line 18). The $decide$ messages actually implement a *Reliable Broadcast* [14] that disseminates the termination.

4.2 The Termination Condition

The Termination Condition (line 13) constitutes the core of the protocol. During a round, no local variable is updated after line 12. So, as far as the values of variables are concerned, the end of a round occurs at line 13.

A process p_i stops executing rounds when it knows that "its current estimate gd_i contains *no more and no less* non$\perp$ values than the other estimates." When this occurs, the estimates of all noncrashed processes are equal and, consequently, p_i's current estimate can no longer be improved; this value, then, defines the global data. Moreover, as the processes have then the same estimate value, if a process p_j terminates during a subsequent round, it will necessarily return the same value of the global data.

The termination condition is the disjunction of two conditions, denoted C_1 and C_2. The first one, namely C_1, is on the number of rounds that have been executed: It actually defines an upper bound on the maximal number of rounds to be executed before reaching agreement. The second, namely C_2, allows early termination by detecting a "stable state" reached by the set of processes. This second condition is a conjunction of two predicates; $C_{2.1}$ and $C_{2.2}$.

5. Note that some nonsuspected processes can actually be crashed.

```
Function Global_data($v_i$)

  cobegin
  (1) task $T1$: $gd_i \leftarrow [\bot, \cdots, v_i, \cdots, \bot]$;
  (2)            $r_i \leftarrow 0$; $cur\_expected_i \leftarrow \Pi$; $next\_expected_i \leftarrow (\Pi - suspected_i)$;
  (3)            loop
  (4)              $r_i \leftarrow r_i + 1$; $prev\_expected_i \leftarrow cur\_expected_i$; $cur\_expected_i \leftarrow next\_expected_i$;
  (5)              $\forall p_j \in cur\_expected_i$ do send $estimate(i, r_i, gd_i)$ to $p_j$ enddo;
  (6)              wait until $(\forall p_j \in cur\_expected_i : ((estimate(j, r_i, rec\_gd_j)$ is received$) \vee (p_j \in suspected_i)))$;
  (7)              let $next\_expected_i = \{j$ such that $estimate(j, r_i, rec\_gd_j)$ has been received during the wait);
  (8)              forall $j \in next\_expected_i$ do
  (9)                  forall $p_k$ such that $p_k \in \Pi$ do
  (10)                     if $((gd_i[k] = \bot) \wedge (rec\_gd_j[k] \neq \bot))$ then $gd_i[k] \leftarrow rec\_gd_j[k]$ endif
  (11)                 enddo
  (12)             enddo;
  (13)             if $(r_i = t + 1) \vee ((prev\_expected_i = next\_expected_i) \wedge (\forall j \in next\_expected_i : (gd_i = rec\_gd_j)))$
  (14)                 then $\forall p_j \in (\Pi - (suspected_i \cup \{i\}))$ do send $decide(i, gd_i)$ to $p_j$ enddo; return$(gd_i)$
  (15)             endif
  (16)           endloop

  (17) task $T2$: wait until receive $decide(k, rec\_gd_k)$:
  (18)            $\forall p_j \in (\Pi - (suspected_i \cup \{i, k\}))$ do send $decide(i, rec\_gd_k)$ to $p_j$ enddo; return$(rec\_gd_k)$
  coend
```

Fig. 4. The Global Data Computation protocol.

4.2.1 Condition C_1: $r = t + 1$

At the beginning of each round, every process broadcasts the initial values it has received during the previous rounds (line 5). This behavior is similar to the one of a reliable broadcast protocol [14]. Such a protocol ensures that after $t + 1$ rounds, every initial value is known either by every noncrashed process, or by none of them.[6] This observation will be used to conclude that, after $t + 1$ rounds, the set of noncrashed processes agrees on the global data, and thus can terminate.

4.2.2 Condition C_2 : $(prev\_expected_i = next\_expected_i) \wedge (\forall j \in next\_expected_i : (gd_i = rec\_gd_j))$.

The aim of this condition is to allow early termination. When this conjunction of predicates, evaluated by a process p_i, holds at the end of a round r, it guarantees that all processes that have completed round $r - 1$ have the same estimate value. This condition is actually a predicate that detects "a stability property" on the last two rounds.

- **Subcondition** $C_{2.1}$: $prev\_expected_i = next\_expected_i$ This condition can be restated as: $prev\_expected_i = cur\_expected_i = next\_expected_i$. When it is satisfied at the end of r, p_i knows that all the processes it assumed to participate in round $r - 1$, processes of $prev\_expected_i$, have actually participated in rounds $r - 1$ and r. In other words, p_i knows that each process that started round $r - 1$ has completed this round. If the system was synchronous, we could conclude that all the processes that have terminated round $r - 1$ have the same global data estimate.[7] But, due to the asynchrony in crash suspicion (see the discussion of Section 2.3), there can be some processes that crash while executing r and that are suspected by other processes while those are executing $r - 1$. Because of these suspicions, $C_{2.1}$ alone cannot allow p_i to safely terminate.
- **Subcondition** $C_{2.2}$: $\forall j \in next\_expected_i : (gd_i = rec\_gd_j)$. When this condition is true, all the processes that, to p_i's knowledge, have terminated $r - 1$ have the same estimate, which is equal to its current estimate, namely gd_i. But, as before, due to the asynchrony in crash suspicion, there may be processes that have completed $r - 1$ and, having crashed while executing r, do not belong to $next\_expected_i$. Hence, this condition alone does not allow p_i to conclude that all processes that have completed $r - 1$ have the same estimate.

While each subcondition is not sufficient to allow p_i to safely terminate, it appears (as shown by the proof) that their conjunction, $C_{2.1} \wedge C_{2.2}$, allows p_i to safely terminate: When they are satisfied at the end of r, all the estimates were equal at the end of $r - 1$ and, consequently, "gd_i `contains` *`no more and no less`* `non`$\bot$ `values than the other estimates.`"

In the worst case, C_2 holds during the round $2f + 2$. Combined with C_1, it will allow to conclude that $\min(t + 1, 2f + 2)$ is an upper bound for the number of rounds.

Remark. The protocol presented in Fig. 4 has some similarities with the $\mathcal{S}-$based consensus protocol introduced in [5] (let $\mathrm{CT}_{\mathcal{S}}$ be this protocol). $\mathcal{S}$ is a class of failure detectors weaker than the class $\mathcal{P}$ of perfect failure detectors. In both the proposed protocol and $\mathrm{CT}_{\mathcal{S}}$, each process computes an array; the protocols ensure that eventually these arrays are equal. The main difference between the proposed protocol and $\mathrm{CT}_{\mathcal{S}}$ lies in the introduction of the sets $prev\_expected_i$, $cur\_expected_i$, and $next\_expected_i$, and in the associated

6. Indeed, the worst case occurs when an initial value is known by only one process at the beginning of each of the t first rounds. During a given round, the process which is aware of the value transmits it to a single process before crashing.

7. This would follow from the fact that the noncrashed processes have received an estimate from each other, and from the fact there is no suspicion during $r - 1$.

early termination predicate. It is important to note that, even when executed with a perfect failure detector, $\mathrm{CT}_{\mathcal{S}}$: 1) requires $t+1$ rounds and 2) does not implement a best effort strategy as the proposed protocol does: When there is a conflict between $\perp$ and v_k for determining the final value of $gd_i[k]$, the second phase of $\mathrm{CT}_{\mathcal{S}}$ always imposes the value $\perp$ (let us note that this second phase of $\mathrm{CT}_{\mathcal{S}}$ is necessary for the processes to have the same final array). Differently, the proposed protocol does its best to have as many entries of gd_i different from $\perp$ as possible (see line 10). More details on relations between consensus and GDC problems are given in Section 6.

4.3 Improvements

The test $(gd_i[k] = \perp) \wedge (rec\_gd_j[k] \neq \perp)$ (line 10) can be simplified into $rec\_gd_j[k] \neq \perp$. We have chosen to consider the extended test to clearly show the situation where p_i "learns" new data.

Moreover, the sending of *decide* messages at line 14 is not necessary in the particular case where $r_i = t+1$. In the general case ($r_i < t+1$), this statement is required to get the termination property. At the end of a round, the C_2 condition, which allows early termination, can be true for a correct process p_j, while it is false for another correct process p_i. If this occurs, p_j stops executing the protocol, while p_i proceeds to $r+1$. As p_i does not suspect p_j and p_j does not participate in $r+1$, it follows that p_i will remain blocked forever at line 6. The *decide* messages prevent such a blocking.

The two above improvements are not considered in the proof that follows.

4.4 Correctness Proof

Recall that a round is an execution of the loop (lines 3-16). The first two lines of the protocol correspond to a fictitious round $r = 0$. In the rest of the proof, we denote, by x_i^r, the value of a variable x at the end of round r for the process p_i.

4.4.1 Synchronization

Lemma 1. *When a process completes the execution of the* **wait** *statement of a round r, any noncrashed process has already completed the execution of the round $r-1$.*

Proof. Let us consider a process p_i which completes the execution of the **wait** statement of a round r. Due to the Accuracy property of the failure detector, p_i cannot suspect a process that is not crashed. Thus, p_i has necessarily received an *estimate* message timestamped with the round number r from all processes that have not crashed. Those messages have been sent at the beginning of the round r. Consequently, any noncrashed process previously has completed the execution of the round $r-1$. □

Corollary 1. *At any time, any two noncrashed processes execute either the same round or consecutive rounds.*

4.4.2 Agreement Property

Lemma 2. *If all the noncrashed processes complete a round r with the same global data value, then any process p_i that completes a round $\geq r$ also has the same global data value.*

Proof. Obviously, Lemma 2 is satisfied when $r = r'$. Let us assume that Lemma 2 is satisfied for a round $r' \geq r$. In other words, all the processes which complete the round r' have a global data equal to gd at the end of r'. We now demonstrate, by contradiction, that a process p_i, which ends the round $r'+1$, also has a global data equal to gd at the end of this round. At the beginning of round r', the value of gd_i is equal to gd by assumption. Thus, if this is not the case at the end of the round, it means that p_i has received a message from a process p_j which contained a global data rec_gd_j not equal to gd (the test of line 10 has hold at least once.). Yet, as the value rec_gd_j has also been computed at the end of the round r', this value cannot be different from gd. □

Lemma 3. *Let r be a round such that some processes learn new initial values during this round. In particular, let p_i be such a process and v_k, the initial value of process p_k, be one of the values learned by p_i. Then, there exists a sequence of $r+1$ processes $(p_{\sigma(0)} = p_k, \ldots, p_{\sigma(r)} = p_i)$, such that:*

1. $\forall l > 1 \quad (gd^{l}_{\sigma(l)} = v_k) \wedge (gd^{l-1}_{\sigma(l)} = \perp)$,
2. *The first $r-1$ processes are faulty and crash before round $r+1$.*

Proof. The proof is by induction on the number of rounds.

1. Base case $r = 1$. The first part of the lemma holds with $p_{\sigma(0)} = p_k$ and $p_{\sigma(1)} = p_i$. The second part of the lemma trivially holds.
2. Induction. Suppose that the lemma holds for $r-1$ $(r \geq 2)$.
    a. First part. Let $p_i = p_{\sigma(r)}$ be a process learning v_k during round r, i.e., $gd^{r}_{\sigma(r)}[k] = v_k \wedge gd^{r-1}_{\sigma(r)}[k] = \perp$. From line 10, there exists at least one process, let us denote it $p_{\sigma(r-1)}$, such that: $\mathbf{part}(\sigma(r), \sigma(r-1), r) \wedge gd^{r-1}_{\sigma(r-1)}[k] = v_k$. We now show (by contradiction) that $p_{\sigma(r-1)}$ has learned the value v_k during round $r-1$. If it was not case, one would have: 1) $gd^{r-2}_{\sigma(r-1)}[k] = v_k$. Since $\mathbf{part}(\sigma(r), \sigma(r-1), r)$, then $p_{\sigma(r-1)}$ was not crashed at the beginning of round $r-1$. Thus, 2) $\mathbf{part}(\sigma(r), \sigma(r-1)), r-1)$. 1) and 2) implies that: $gd^{r-1}_{\sigma(r)}[k] = v_k$, a contradiction. Thus, $p_{\sigma(r-1)}$ satisfies the assumptions of the lemma for $r-1$, from which we conclude the first part of the result for round r.
    b. Second part. Let us consider three consecutive processes in this sequence, namely $p_{\sigma(l-2)}, p_{\sigma(l-1)}$ and $p_{\sigma(l)}$. From $gd^{l-2}_{\sigma(l-2)} = v_k$ and $gd^{l-1}_{\sigma(l)} = \perp$, one concludes $\neg\mathbf{part}(\sigma(l), \sigma(l-2), l-1)$ and, thus, $p_{\sigma(l)}$ has suspected $p_{\sigma(l-2)}$ before round l. Thus, by Lemma 1, $p_{\sigma(l-2)}$ has crashed during round $l-1$, or round l. Consequently, the first $r-1$ processes of the

sequence are faulty and have crashed before round $r+1$. □

Lemma 4. *The processes that terminate the round $t+1$ share the same global data.*

Proof. Let r be a round during which a process learns a new initial value v_k. We show that $r \leq t+1$. From Lemma 3, there exists a sequence of $r+1$ processes $(p_{\sigma(0)}, \dots, p_{\sigma(r)})$, such that the first $r-1$ processes are faulty and crash before round $r+1$. Two cases have to be considered.

1. $t = n-1$. In the worst case, all the processes of the system are included in the sequence. Hence, $r+1 = n$. This means that $r+1 = n = t+1$, i.e., $r = t$. The sequence becomes $(p_{\sigma(0)}, \dots, p_{\sigma(r)} = (p_{\sigma(0)}, \dots, p_{\sigma(n-2)}, p_{\sigma(n-1)})$. From Lemma 3, only $p_{\sigma(n-2)}$ and $p_{\sigma(n-1)}$ are noncrashed during round $n = r+1$ and both have learned v_k by the end of round $n-1 = t$. It follows that every noncrashed process has learned v_k by the end of round t.
2. $t < n-1$. Since there are at most t crashes, we have $r-1 \leq t$, i.e., $r \leq t+1$. In the worst case, $r = t+1$ and the first t processes are faulty. Thus, $p_{\sigma(t)}$, which is necessarily a correct process, has learned v_k during round t and has broadcasted it during round $t+1$. It follows that every correct processes has the same global data at the end of round $t+1$. □

Lemma 5. *Let r be the first round, during which at least one process (p_i) decides at line 14. Then, all the processes that have completed the round $r-1$ belong to the set $prev\_expected_i^r$.*

Proof. The proof is a case analysis.

- $r = 1$. $prev\_expected_i^1 = cur\_expected_i^0 = \Pi$ (line 2)
- $r \geq 2$. Any process p_k that terminates the round $r-1$ has obviously completed the **wait** statement of round $r-1$. At this time denoted t, due to Lemma 1, any noncrashed process, in particular p_i, has also completed the round $r-2$. Moreover, at time t, p_k is not crashed and, thus, no process can have suspected p_k before t. Consequently, p_i has not suspected p_k during a round $\leq r-2$.

  1. $r = 2$. One has $prev\_expected_i^2 = cur\_expected_i^1 = next\_expected_i^0 = \Pi - suspected_i^0$. As p_i has not suspected p_k during round 0, $p_k \notin prev\_expected_i^2$.
  2. $r > 2$. As process p_i has reached round r, it has obviously received a message send by p_k during round $r-2$. In other words **part**$(i, k, r-2)$ and, thus,

$$p_k \in next\_expected_i^{r-2} = cur\_expected_i^{r-1} = prev\_expected_i^r \text{ (line 4).}$$ □

Lemma 6. *Let p_i and p_j be any two processes that have terminated the execution of* Global_data. *We have $GD_i = GD_j$, (where GD_i-resp. GD_j- is the last value of gd_i-resp., gd_j).*

Proof. If a process p_i has returned from the execution of Global_data, it has either executed line 14 or line 18. In the first case, the decision value is the value of the local global data gd_i. In the last case, it decides on the value carried by a *decide* message. This message has been sent by a process, either at line 14 or at line 18, to forward a decision value (this value, launched by a process deciding at line 14, has been possibly relayed by several processes before arriving at p_i.). Thus, if processes decide on different values for the global data, it means that at least two processes have decided different values at line 14.[8] We prove that this scenario is impossible. Let r be the first round during which a process decides at line 14 and let p_i be a process that decides during this round. At the end of r, the termination condition is satisfied for p_i. Two cases have to be considered:

- $C_1 (r = t+1)$ is satisfied. Due to Lemma 4, all processes share the same global data at the end of round r.
- C_2 is satisfied. In that case, let $X = prev\_expected_i^r = next\_expected_i^r$. During r, p_i has received a message from each process $p_j \in X$. From Lemma 1 and Lemma 5, we conclude that X contains all the processes that were not crashed at the end of $r-1$. The second part of the test guarantees that all these processes have the same global data value at the end of $r-1$. Due to Lemma 2, any process that completes a round $\geq r$ has also the same global data value. In particular, this global data value is decided by any process which executes line 14 during any round $\geq r$. This ensures there is a single decided global data value. □

4.4.3 Obligation Property

Theorem 2. *If a process p_i completes the execution of* Global_data, *then the value of the returned global data contains its own initial value (i.e., $gd_i[i] = v_i$).*

Proof. Due to the initialization (line 1), $gd_i[i]$ is equal to v_i at the beginning of round one. Moreover, $gd_i[i]$ is never subsequently updated, because the test of line 10 never holds ($gd_i[i] \neq \perp$). Let us consider the two following cases:

- p_i decides at line 14. Its global data necessarily contains its own value.
- p_i decides at line 18. As already indicated, in the proof of Theorem 1, the global data decided by p_i has been previously decided by a process p_j at line 14. Let us assume that p_j started the execution of line 14 at time t. Obviously, p_i was not crashed at that time, because p_i has later received the global data value forwarded by p_j. Moreover, p_j has executed at least one round; so, it has completed, at some time $t' \leq t$, the wait statement

8. These processes possibly decide at different rounds.

of its first round. Hence, it is impossible for p_j to have suspected p_i before t'. Consequently, p_j has received the value $[\perp, \cdots, v_i, \cdots, \perp]$ sent by p_i during the first round and it has updated $gd_j[i]$ accordingly during its first round (line 10). This update has occurred before t. As the value of $gd_j[i]$ remains equal to v_i (later, the test concerning $gd_j[i]$ at line 10 never holds), it follows that the global data value decided by p_j, and later by p_i, contains the value v_i. □

4.4.4 Validity Property

Theorem 3. *If a process p_i completes the execution of the* Global_data *function, then $\forall j$, $gd_i[j]$ contains v_j or $\perp$.*

Proof. The proof follows directly from the initialization (line 1), the update of the $gd_i[j]$ (line 10), and the channel reliability (no message alteration, no spurious message). □

4.4.5 Termination Property

Theorem 4. *All correct processes decide.*

Proof. We first show that at least one correct process decides. Indeed, let us assume that no correct process decides. Due to the termination condition, this occurs if no correct process ever reaches the end of round $t+1$. The proof is by contradiction. Let $r < t+1$ be the smallest round in which some correct process p_i remains blocked forever at the **wait** statement. As far as correct processes are concerned, by assumption, none of them can remain blocked in a previous round, thus, each of them will broadcast a message to p_i at the beginning of round r. As channels are reliable, p_i will receive all these messages. As far as faulty processes are concerned, there exists a time t after which every faulty process has actually crashed and due to the completeness property of the underlying failure detector, they will be suspected by p_i. This shows that p_i eventually completes round r, a contradiction. Moreover, if at least one process decides (at line 14 or line 18), it must have sent a *decide* message to all processes. Due to the channel reliability assumption and to task $T2$, the theorem follows. □

4.4.6 Upper Bound on the Number of Rounds

Theorem 5. *In the worst case, the protocol converges in* $\min(2f+2, t+1)$ *rounds.*

Proof. Let m_1, $m_{2.1}$, and $m_{2.2}$ be the maximal number of rounds for conditions $C_1, C_{2.1}$, and $C_{2.2}$ to be satisfied, respectively. Since the termination condition is $C_1 \vee (C_{2.1} \wedge C_{2.2})$, one knows that the protocol converges in $\min(m_1, \max(m_{2.1}, m_{2.2}))$. Trivially, $m_1 = t+1$.

Let us now consider C_2. If at least one process learns a new initial value during round r, Lemma 3 tells us that at least $r-1$ crashes actually occur before $r+1$. Since there are exactly f crashes, one knows that, in the worst case, the round $f+1$ is the last round during which a process can learn a new initial value. It follows that all the processes will have the same global data value at the round of $f+1$, even if they are not aware of this fact. Due to Lemma 2, all values exchanged in rounds $> r$ are equal. Consequently, the condition $C_{2.2}$ holds at the end of any round $r \geq f+2$. Hence, $m_{2.2} = f+2$.

The condition $C_{2.1}$ is a local test, for each process p_i, about the absence of new suspicions over two consecutive rounds. In the worst case, this property can not be satisfied as long as exactly one new suspicion is learned by p_i during every pair of consecutive rounds. Since there are at most f crashes, this situation cannot last more than $2f$ rounds. Thus, the condition $C_{2.1}$ holds after at most $2f+2$ rounds.

It follows that the protocol terminates after, at most, $\min(t+1, \max(2f+2, f+2))$ rounds. □

```
$suspected_i \leftarrow \emptyset$;
cobegin
task $T1$: while $true$ do
              $GD_i \leftarrow$ Global_data($v_i$); % $v_i \neq \perp$ %
              $suspected_i \leftarrow \{j \mid GD_i[j] = \perp\}$
           enddo

task $T2$: when QUERY_P: return($suspected_i$)
coend
```

Fig. 5. $GDC \geq P$.

5 Hardness of the GDC Problem

5.1 Problem Reduction

This section considers the *hardness* of a problem with respect to the difficulty of solving it in the presence of process crashes: A problem is *easy* if it can be solved despite many arbitrary process crashes, it is *hard* if it requires additional assumptions to be solved in the presence of process crashes [10], [14]. To address this issue, Chandra, Hadzilacos, and Toueg [5], [14] have extended the notion of *problem reduction* to asynchronous distributed systems with process crashes.

A problem $P2$ *reduces* to a problem $P1$ (denoted $P1 \geq P2$) if there exists a protocol $\mathcal{A}_{P1 \rightarrow P2}$ that transforms any protocol solving $P1$ into a protocol solving $P2$. $P1 \geq P2$ means that $P1$ is at least as hard to solve as $P2$: All the assumptions, and maybe more, required to solve $P2$ are necessary to solve $P1$; the situation can even be worse—it is possible that $P2$ can be solved while $P1$ cannot. If $P1 \geq P2$ and $P2 \geq P1$, then $P1$ and $P2$ are *equivalent*. This is denoted $P1 \simeq P2$.

5.2 A New Reduction

$\mathbf{P} \geq \mathbf{GDC}$. Let P be the problem of constructing a perfect failure detector (i.e., a failure detector satisfying the Completeness and Accuracy properties defined in Section 2.2).

Assuming a solution to P, the protocol designed in Section 4.1 solves GCD. Consequently, this protocol is a *reduction* transforming any protocol solving P into a protocol solving GCD. Hence, we have $P \geq GDC$.

$\mathbf{GCD} \geq \mathbf{P}$. We give here a simple protocol that, given a solution to the GDC problem, solves P. This protocol is described in Fig. 5. For each process p_i, the protocol is made of two tasks.

```
Function Global_data($v_i$)
(1)    $\forall p_j \in \Pi$: send $val(v_i)$ to $p_j$;
(2)    wait until ($\forall p_j \in \Pi$: (($val(v_j)$ is received from $p_j$) $\vee$ ($p_j \in suspected_i$)) );
(3)    $\forall p_j \in \Pi$: if $v_j$ received then $gd_i[j] \leftarrow v_j$ else $gd_i[j] \leftarrow \perp$ endif;
(4)    return($Consensus(gd_i)$)
```

Fig. 6. A consensus-based GDC protocol.

The task $T2$ realizes the interface with the upper layer. The variable $suspected_i$ contains the set of processes currently suspected by p_i. So, when the upper layer calls QUERY_P, the task $T2$ returns the current value of this set.

The task $T1$ manages the variable $suspected_i$. It is made of an infinite loop. Each iteration is a call to the function Global_data(v_i), where the value provided by p_i is distinct from $\perp$. Then, when $T1$ has got the result of the current call to Global_data, it computes the new value of $suspected_i$. This value includes all processes p_j that were crashed before this global data computation (note that $suspected_i$ can also include processes that have crashed during this computation).

It is relatively easy to show that this protocol constructs a perfect failure detector. Here we sketch such a proof. First of all, a correct process p_i participates in all instances of the GDC problem (successive calls to Global_data) and never provides $v_i = \perp$. Due to the properties of the GDC problem, it follows that, $\forall j$ and for all instances of the GDC problem, p_j always gets $GD_j[i] \neq \perp$. So, if p_i is correct, it is never suspected. More generally, if p_i, correct or not, has not crashed when the result of the k^{th} instance of the GDC problem is delivered to p_j, then $GD_j[i] = v_i \neq \perp$ and p_i is not added to $suspected_i$ during the kth instance of the GDC problem. If p_i has crashed before the kth instance of the GDC problem, then it cannot provide a value v_i and, consequently, p_j will get $GD_j[i] = \perp$ and will add p_i to $suspected_i$. So, crashed processes are eventually suspected (Completeness) and no process is suspected before it crashes (Accuracy property).

5.3 GDC is a Hard Problem

Let us consider the following two distributed computing problems: $NBAC$, the *Nonblocking Atomic Commitment* problem [3] and, TRB, the *Terminating Reliable Broadcast* problem [14].[9] Combined with previous results on the classification of problems in asynchronous systems prone to process crashes [5], [10], [13], [14], we have the following problem equivalence:

$$P \simeq NBAC \simeq TRB \simeq GDC.$$

So, according to the problem classes defined in [10], GDC belongs to the class, called NFC, which contains the hardest problems to solve in presence of process crashes.

6 A Consensus-Based Approach

This section first presents a consensus-based approach to solve the GDC problem. Then, it compares this approach with the approach developed in Section 4.

9. The TRB problem is a variant of the Byzantine general problem. See [14] for more details.

6.1 A Consensus-Based GDC Protocol

The consensus problem can be defined informally in the following way. Each noncrashed process proposes a value and all correct processes have to decide a value (termination property) such that: 1) no two correct processes decide different value (agreement property) and 2) the decided value is a proposed value (validity property). (See [5], [9], [10] for a formal introduction to the consensus problem).

The protocol described in Fig. 6 provides a consensus-based solution to the GDC problem. It assumes a perfect failure detector and works as follows: First, each process sends its initial data to the other processes (line 1). Then, each process p_i constructs a local view of the global data (lines 2-3). Finally, the processes use the consensus routine to impose one of these local views as the decided global data (line 4). Due to the consensus termination property, every correct process decides. Due to the consensus agreement, all correct processes decide the same array and due to the validity property, they decide a global data proposed by a process. Moreover, due to the use of a perfect failure detector at line 2, the view built by a process at line 3 includes the initial value of all the correct processes.

6.2 Discussion

The previous protocol uses a perfect failure detector (line 2) in order that the local view of a process does not miss the initial values of correct processes. Its cost is one communication round, to construct the local views, plus the cost of the underlying consensus routine.

Any consensus protocol that uses a failure detector belonging to a failure detector class defined in [5] works when the underlying failure detector is perfect, but the consensus problem does not require a perfect failure detector. It can be solved with failure detectors of weaker classes [5], [6]. When equipped with a perfect failure detector, the most efficient in terms of communication steps, failure detector-based consensus protocols currently known [5], [16], [21], [26] require at least $2(f+1)$ communication steps. Hence, the consensus-based protocol described in Fig. 6 is less efficient than the protocol described in Section 4.

This is not counterintuitive. It has been shown in Section 5 that $P \simeq NBAC \simeq TRB \simeq GDC$. Let $CONS$ denote the consensus problem. It has been shown [14] that $TRB > CONS$ in asynchronous distributed systems. Hence, we have $GDC > CONS$. This means that, although the consensus problem can be solved with a perfect failure detector, the consensus alone is not sufficient to solve the GDC problem. Intuitively, this means that the lines 1-2 of Fig. 6 cannot be suppressed: Whatever the underlying consensus protocol, a perfect failure detector is required to construct consistent local views of the global data, thereby ensuring that the decided view satisfies the Obligation property defined in Section 3.2.

7 Examples of Global Function Computation

The solution to several distributed computing problems amounts to compute a global function [12], [15]. This section shows how the previous framework allows solving those problems in the context of asynchronous distributed systems with process crashes. To this end, two problems are briefly examined.

7.1 Nonblocking Atomic Commitment

This well-known problem is mainly encountered in data management systems [3]. It has been sketched in the introduction.

The local variable v_i contains the YES or NO vote of p_i. After its computation, the global data contains the votes of, at least, all correct processes. Some of its entries can contain the $\perp$ value: If $GD_i[j] = \perp$, then p_j has crashed before the end of the global data computation, but an entry associated with a process that has crashed can contain $\perp$ or its vote, depending on the crash pattern and on the estimate exchange pattern. Finally, the function $\mathcal{F}$ delivers the result COMMIT if and only if all the entries of the global data contain a YES vote, this is the all-or-nothing Atomic Commitment problem. We can see that a $\perp$ value in the global data is implicitly interpreted as a NO vote.

It is important to note that the the assumption used by the protocol is consistent with Guerraoui's theoretical result, namely, "The Atomic Commitment problem can be solved in asynchronous distributed systems prone to process crashes, only if the underlying failure detector modules are perfect" [13]. It is also important to note that the NBAC protocols described in [13], [17], [24] solve a *weak* version of the NBAC problem. In this weaker version, the "process crash" notion is replaced by a weaker notion, namely the "process crash suspicion" notion. See [13], [24] for more details.

7.2 Distributed Termination Detection

This problem is a well-known paradigm of reliable distributed computing. We assume the reader is familiar with it. An application is *terminated* when it has entered a state from which its processes will remain passive forever (this requires there is no message in transit between processes). A lot of protocols have been proposed to detect distributed termination [15], [20], [27].

In our context a distributed computation is $terminated$ if it has entered a state from which each of its noncrashed processes remains passive forever or crashes. A way to cope with arbitrary transfer delays of messages sent by crashed processes, they sent them before crashing, consists in allowing a process p_i to stop receiving messages from processes it perceives crashed. This can be done by providing processes with a variable NC containing the set of processes perceived as noncrashed. Its value is obtained as the result of a global function computation (i.e., all the processes have the same value for NC). Initially, $NC = \Pi$.

Here we extend a variant of a protocol, denoted M, proposed by Mattern [20] for crash-free systems. M requires each process p_i to maintain the following arrays of control variables: $sent_i[k]$ which counts the number of messages that p_i has sent to p_k and $rec_i[k]$ which counts the number of messages that p_i has received from p_k. As in M, the proposed extension requires a process be passive in order to participate in a global data computation; so, if it is active, it waits until it becomes passive. It also requires two *consecutive* global data computations, called $GD1$ and $GD2$, respectively. They are built from the following initial values:

- For $GD1$: $v_i = (rec_i, suspected_i)$.
- For $GD2$: $v_i = (sent_i, suspected_i)$.

So, when a global data $GD1$ (resp. $GD2$)[10] has been computed, some of its entries contain a counter array plus a set of processes, while the others contain the default value $\perp$. Note, that if a process has crashed before the computation of $GD1$ (resp. $GD2$), then its corresponding entry in $GD1$ (resp. $GD2$) is equal to $\perp$. After having obtained the values of $GD1$ (resp. $GD2$), a process p_i computes $NC1 = \Pi - crashed1$ (resp. $NC2 = \Pi - crashed2$), where $crashed1$ (resp. $crashed2$) is the union of the sets $suspected_j$ that are in $GD1$ (resp. $GD2$). Moreover, NC is updated to $NC1$ (resp. $NC2$). Then, p_i compares $NC1$ and $NC2$ (note that $NC1 \supseteq NC2$).

- If $NC1 \neq NC2$, p_i cannot claim termination. It starts new successive global data computations to get two new global data $GD1$ and $GD2$.
- If $NC1 = NC2$, then process p_i computes:
  - $R1 = \sum_{j,k \in NC2, rec_j \in GD1} rec_j[k]$. This value represents the number of messages received by processes in $NC2$ from processes in $NC2$, as perceived by the first computed global data $GD1$.
  - $S2 = \sum_{j,k \in NC2, sent_k \in GD2} sent_k[j]$. This value represents the number of messages sent by processes in $NC2$ to processes in $NC2$, as perceived by the second computed global data $GD2$.

  If $R1 = S2$, then, with respect to the set of processes perceived as noncrashed, the number of messages counted as received by $GD1$ is equal to the number of messages counted as sent by $GD2$. If this is true, p_i claims termination. Otherwise it starts a new computation to get two new global data $GD1$ and $GD2$.

This extended protocol allows process crashes to be coped with. It is possible to show that if the application terminates, then the protocol will eventually claim it (liveness of the detection); if the protocol claims termination, then the application has actually terminated (safety of the detection). Informally, from $NC1 = NC2$ we can conclude the following two points: 1) Any process p_i that is perceived crashed by $GD1$ cannot belong to $NC2$. Consequently, messages sent or received by this process p_i are counted neither in R1 nor in S2, and 2) If a process p_i crashes during $GD2$, it does belong to $NC1$ and, hence, to

10. In the reliable distributed computing terminology, the computation of such a global data is usually called a *wave*.

$NC2$. So, messages sent or received by such a process p_i appear in R1 and S2.

8 Conclusion

This paper has addressed the computation of a global data function in asynchronous distributed systems where processes may fail by crashing. A *Global Data* is an array of values with one entry per process. A particular global function is defined by the type of the value each process has to provide and by the function that has to be computed. This paper has proposed a protocol to solve such problems in the context of asynchronous distributed systems where processes may fail by crashing.

The main problem that has to be solved lies in computing the global data and providing each noncrashed process with a copy of it, despite the possible crash of some processes. To be consistent, the global data must contain, at least, all the values provided by the processes that do not crash. This is the *Global Data Computation* (GDC) problem. It has been shown that GDC belongs to the class of problems that are the *hardest* to solve, with respect to the assumptions they require, in the presence of process crashes (GDC is actually equivalent to the problem of building a perfect failure detector). To solve the GDC problem, the proposed protocol requires processes to execute a sequence of asynchronous rounds during which they construct, in a decentralized way, the value of the global data, in such a way that eventually each process gets a copy of it.

The proposed protocol was designed to allow early decision. In the best case, it terminates in two rounds, when $t > 1$. Moreover, processes never exchange information on crashes. The message size depends only on the round number and on the size of the global data: In addition to its identity, a message carries only an estimate of the global data value. Let us also note that, when communication channels are FIFO, round numbers can be implemented $mod2$. In that case, messages carry only bounded values.

In the worst case, the proposed protocol requires $\min(2f+2, t+1)$ rounds. There is a problem ([8], Section 5, p. 740), similar to the Global Data Computation problem, that can be solved in a synchronous distributed system in $\min(2f+2, t+1)$ rounds, which has been shown to be lower bound [8] in such systems. So, an interesting open question is the following: "In asynchronous distributed systems equipped with perfect failure detectors, is $\min(2f+2, t+1)$ a lower bound for the maximal number of rounds of any protocol, in which processes do not exchange lists of suspects, solving the Global Data Computation problem?" Another interesting open problem is to design a *reliable synchronizer* able to interpret synchronous protocols on top of an asynchronous system, where processes can crash, equipped with a perfect failure detector.

Acknowledgments

The authors are grateful to the anonymous referees whose comments and suggestions helped improve the presentation of the paper. An extended abstract of this paper appeared in the *Proceedings of the 20th ICDCS* (Taipei, Taiwan, April 2000), where it got the Best Paper Award.

References

[1] B. Awerbuch, "Complexity of Network Synchronization," *J. ACM,* vol. 32, no. 4, pp. 802–823, 1985.

[2] J.-Cl. Bermond, J.-Cl. Konig, and M. Raynal, "General and Efficient Decentralized Consensus Protocols," *Proc. Second Int'l. Workshop Distributed Algorithms (WDAG' 87),* J. Van Leeuwen ed., pp. 41–56, 1987.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Reading, Mass.: Addison-Wesley, 1987.

[4] R. Brand, "Iso-Ethernet: Bridging the Gap from WAN to LAN," *Data Comm.,* 1995.

[5] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM,* vol. 43, no. 2, pp. 225–267, Mar. 1996.

[6] T. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *J. ACM,* vol. 43, no. 4, pp. 685–722, July 1996.

[7] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, "Muteness Failure Detectors: Specification and Implementation," *Proc. Third European Dependable Computing Conf. (EDCC '99),* pp. 71–87, Sept. 1999.

[8] D. Dolev, R. Reischuk, and R. Strong, "Early Stopping in Byzantine Agreement," *J. ACM,* vol. 37, no. 4, pp. 720–741, Apr. 1990.

[9] M.J. Fischer, N. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374–382, Apr. 1985.

[10] E. Fromentin, M. Raynal, and F. Tronel, "On Classes of Problems in Asynchronous Distributed Systems with Process Crashes," *Proc. 19th IEEE Int. Conf. Distributed Computing Systems (ICDCS '99),* pp. 470–477, May 1999.

[11] V.K. Garg and J. Ghosh, "Repeated Computation of Global Functions in a Distributed Environment," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 8, pp. 823–834, 1994.

[12] V.K. Garg, *Principles of Distributed Systems.* Kluwer Academic, 1996.

[13] R. Guerraoui, "Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus," *Proc. Ninth Int'l Workshop Distributed Algorithms (WDAG '95),* J.M. Hélary and M. Raynal eds., pp. 87–100, 1995.

[14] V. Hadzilacos and S. Toueg, "Reliable Broadcast and Related Problems," *Distributed Systems,* S. Mullender ed., pp. 97–145, 1993.

[15] J.-M. Hélary and M. Raynal, *Synchronization and Control of Distributed Systems and Programs.* John Wiley & Sons, 1990.

[16] M. Hurfin and M. Raynal, "A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector," *Distributed Computing,* vol. 12, no. 4, pp. 209–223, 1999.

[17] M. Hurfin and F. Tronel, "A Solution to Atomic Commitment Based on an Extended Consensus Protocol," *Proc. Sixth IEEE Workshop Future Trends of Distributed Computing Systems (FTDCS '97),* pp. 98–103, 1997.

[18] N. Lynch, *Distributed Algorithms.* Morgan Kaufmann, 1996

[19] J. Matocha and T. Camp, "A Taxonomy of Distributed Termination Detection Algorithms," *J. Systems and Software,* vol. 43, pp. 207–221, 1998.

[20] F. Mattern, "Algorithms for Distributed Termination Detection," *Distributed Computing,* vol. 2, pp. 161–175, 1987.

[21] A. Mostefaoui and M. Raynal, "Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: A Generic Quorum-Based Approach," *Proc. 13th Int'l Symp. Distributed Computing (DISC '99),* pp. 49–63, 1999.

[22] L. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in Presence of Faults," *J. ACM,* vol. 27, no. 2, pp. 228–234, 1980.

[23] M. de Prycker, *Asynchronous Transfer Mode: Solution for Broadband ISDN.* Prentice Hall, 1995.

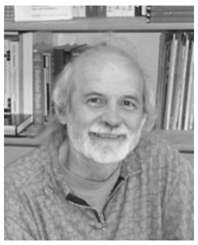
[24] M. Raynal, "Non-Blocking Atomic Commitment in Distributed Systems: A Tutorial Based on a Generic Protocol," *J. Computer Systems Science and Eng.,* vol. 15, no. 2, pp. 77–86, 2000.

[25] J. Rufino, P. Veríssimo, P. Arroz, C. Almeida, and L. Rodrigues, "Fault-Tolerant Broadcast in CAN," *Proc. 28th Int'l Symp. Fault-Tolerant Computing (FTCS '28),* pp. 150–159, 1998.

[26] A. Schiper, "Early Consensus in an Asynchronous System with a Weak Failure Detector," *Distributed Computing,* vol. 10, pp. 149–157, 1997.

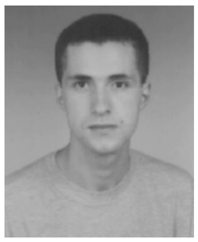
[27] G. Tel, *Intro. to Distributed Algorithms.* Cambridge Univ. Press, 1994.

[28] P. Veríssimo, *Distributed Systems*, Second ed., S. Mullender ed., Addison-Wesley, pp. 447–490, 1993.



Jean-Michel Hélary received his Docteur troisième cycle degree in applied mathematics (Numerical analysis of partial differential equations) from the University of Paris, in 1968. Then his Habilitation Doctor degree from the University of Rennes in 1988. He is currently a Professor of computer science at the University of Rennes 1, France. His research interests are distributed algorithms, protocols, programming languages, graph algorithms, and parallelism. He is a member of the INRIA research team Algorithmes Distribués et Protocoles, led by Michel Raynal. Professor Hélary has published more than 40 scientific publications in journals, reviews, and conferences. He is co-author, with Michel Raynal, of a book devoted to the Synchronization of Distributed algorithms and Systems (John Wiley, 1990). He has been a member of the program commitee and cochairman of the Distributed Algorithms track of the 14th IEEE International Conference on Distributed Computing Systems (Poznan, 1994), cochairman of the Ninth International Workshop on Distributed Algorithms (Le Mont Saint-Michel, France, 1995), and local arrangement cochairman of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (May 1999).

Michel Hurfin received the PhD degree in computer science from the University of Rennes, France, in 1993. His dissertation topic addressed execution replay and property detection in distributed applications. In 1994, he spent one postdoctoral year at Kansas State University, Manhattan, in the research group of Professor M. Mizuno. Dr. Hurfin is currently a researcher at the INRIA unit in Rennes. His research interests include distributed systems, software engineering, and middleware for distributed operating systems. Recently, he has initiated research on distributed fault-tolerant middleware.



Achour Mostefaoui received his Engineer Degree in computer science in 1990 from the University of Algiers (USTHB), and a PhD in Computer Science in 1994 from the University of Rennes, France. He is currently an assistant professor at the Computer Science Department of the University of Rennes, France. His research interests include fault-tolerance in distributed systems, group communication, consistency in DSM systems, and checkpointing distributed computations. Dr. Mostefaoui has published more than 30 scientific publications and served as a reviewer for more than 20 major journals and conferences.

Michel Raynal has been a professor of Computer Science at the University of Rennes, France, since 1984. At IRISA (CNRS-INRIA-University joint computing laboratory located in Rennes) he is the leader of the ADP (Distributed Algorithms and Protocols) research group that he created in 1986. He has served as program cochair of WDAG (now DISC, the Symposium on Distributed Computing) in 1989 and 1995. He has also served as vice-chair for the "Distributed Algorithms" track of the IEEE International Conference on Distributed Computing Systems in 1994 and 1999. Furthermore, he has served as a PC member in several international conferences. Michel Raynal has written seven books. He has published more than 50 papers in journals and 100 in conferences. Together with other european leaders, he is currently a member of the ESPRIT Basic Research Network of excellence in Distributed Computing Architectures (CABERNET) currently headed by B. Randell. Michel Raynal's research interests include distributed algorithms, distributed systems, distributed computing, and fault-tolerance. His main interest lies in the fundamental concepts, principles, and mechanisms that underly the design and the construction of distributed systems. Among them, he is currently interested in the study of the Causality concept and in Agreement problems. On the practical side, Michel Raynal is involved in the implementation of reliable communication primitives, the consistency of distributed data, the design and the use of checkpointing protocols, and the set of problems that can be solved on top of a consensus building block.

Frédéric Tronel received his master degree from the University of Rennes, France, in 1997. His doctoral dissertation, at IRISA Rennes, is about fault tolerant asynchronous distributed systems and the use of the consensus problem as a building block to master the intrinsic complexity of such systems. He is a former student of the École Normale Supèrieure of Cachan. His research interests include distributed systems, programming languages, and free software.