Computing Global Functions in Asynchronous Distributed Systems with Process Crashes

Jean-Michel Hélary, Michel Hurfin, Achour Mostefaoui, Michel Raynal, Frédéric Tronel

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Computing Global Functions in Asynchronous Distributed Systems with Process Crashes

Jean-Michel Hélary, Michel Hurfin, Achour Mostefaoui, Michel Raynal et Frédéric Tronel

N ° 3656

Avril 1999

THÈME 1

ISSN 0249-6399

RENNES

Computing Global Functions in Asynchronous Distributed Systems with Process Crashes

Jean-Michel Hélary, Michel Hurfin, Achour Mostefaoui, Michel Raynal et Frédéric Tronel

Thème 1 — Réseaux et systèmes
Projet ADP

Rapport de recherche n???? — Avril 1999 — 20 pages

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. To be consistent, the global data must contain (at least) all the values provided by the processes that do not crash. To solve this problem called the *Global Data Computation* (GDC) 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 non-crashed 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 let 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 paper also shows that, with respect to the possibility to solve a problem in an asynchronous system with process crashes, the GDC problem and the contruction of a perfect failure detector are equivalent problems: any solution to one of these problems can be transformed into a solution for the other one. Finally, two distributed computing problems (Atomic Commitment and Distributed Termination Detection) are solved as instances of the GDC problem.

Keywords: Asynchronous Distributed Computation, Global Data, Global Function Computation, Perfect Failure Detector, Problem Reduction, Process Crash.

(Résumé : tsvp)

Unit´e de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
T´el´ephone : 02 99 84 71 00 - International : +33 2 99 84 71 00
T´el´ecopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Calcul de fonctions globales dans des systèmes répartis asynchrones non fiables

Résumé : Une *donnée globale* est un vecteur comportant une entrée par processus. Chaque entrée contient une valeur appropriée qui aura été fournie par le processus correspondant. Plusieurs problèmes répartis peuvent se ramener au calcul d'une fonction sur une donnée globale. Cet article propose un protocole pour résoudre de tels problèmes dans les systèmes répartis asynchrones où les processus peuvent connaître des pannes franches. Pour être cohérente, la donnée globale doit contenir (au moins) toutes les valeurs fournies par les processus qui n'ont pas été défaillants. Pour calculer une donnée globale (Global Data Computation – GDC), les processus exécutent une séquence de tours asynchrones durant laquelle ils déterminent (de manière décentralisée) la valeur de la donnée globale. Au bout d'un temps fini, chaque processus correct en obtient une copie. Pour prendre en compte les défaillances, le protocole utilise un détecteur de défaillances parfait.

Le protocole proposé a été conçu pour être efficace en temps. Il permet des décisions au plus tôt. Soit t le nombre maximal de pannes pouvant se produire ($t < n$, n étant le nombre total de processus dans le système) et soit f le nombre de pannes effectives ($f \leq t$). Dans le pire des cas, le protocole termine en $\min(2f + 2, t + 1)$ tours. De plus, le protocole ne nécessite pas de s'échanger des informations sur la perception locale des pannes.

Cet article montre également que, du point de vue de sa solubilité dans un environnement asynchrone non fiable, le problème GDC et la construction d'un détecteur de défaillances parfait sont des problèmes équivalents: toute solution à l'un de ces problèmes peut être adaptée pour résoudre l'autre. Enfin, deux problèmes particuliers (diffusion atomique et detection de la terminaison) sont résolus via un calcul de fonction globale.

Mot-Clés: Calcul réparti asynchrone, Donnée globale, Calcul de fonctions globales, Détecteur de défaillances parfait, Réduction de problème, Panne franche.

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 [9, 13] constitutes a key component for solving many distributed computing problems. It consists in: (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, 10]. 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 (the same for all processes), each process commits or invalidates the local computation it has previously performed . More generally, some problems require repeated computations of a global function [9]. The *Distributed Termination Detection* problem is an example of a problem that can be solved by two successive global function computations [13, 17, 21].

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 2 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.

This paper investigates the distributed approach to design a general protocol that allows processes to compute global functions in presence of process crashes. One of the main problems created by process crashes concerns the fact that all non-crashed 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, while 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)? The proposed protocol 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 to get a global data containing as much meaningful values as possible.

To cope with process crashes, the proposed protocol 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 timeout values, but those are implementation mechanisms that are not made visible outside the failure detector module. By hiding all time-dependent aspects in a black box (the failure detector), this approach allows to design *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 non-crashed process with a copy of it, is identified as the *Global Data Computation* problem. To solve it, processes can execute a sequence of asynchronous rounds during which they build (in a decentralized way) the value of the global data. The proposed protocol, based on perfect failure detectors, has been designed to be time-efficient. It allows early decision. Let t be the maximum number of processes that may crash ($t < n$) and let f be the number of actual crashes ($f \leq t$). If $t = 0$ (*i.e.*, when the system is reliable) the protocol requires a single round. When $t > 0$, in the best case it terminates in two rounds. In the worst case, it terminates in $\min(2f + 2, t + 1)$ rounds. Furthermore, the protocol 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 seven 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 4.3 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, 8, 12]. Then, Section 6 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 7 concludes the paper.

2 Distributed System Model

2.1 Asynchronous Distributed System

The system consists of a finite set of n processes, $\Pi = \{p_1, \ldots, p_n\}$. 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 (no spurious messages, no loss, no corruption) but 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.*

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,

we consider a Crash/no Recovery model. A process that does not crash is called a *correct* process. Otherwise, it is a *faulty* process. Recall that t ($< n$) denotes the maximum number of processes that may crash, and f ($\leq t$) denotes 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]. We consider here perfect failure detector modules: no guess is erroneous. More precisely, these failure detectors are defined by the two following 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 detector modules. Some networks have a notion of privileged, high priority channels satisfying strong timing assumptions [4, 19, 22] (*e.g.*, field buses such as FIP or CAN [20]). 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 a perfect failure detector* [5, 15]. 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

It is important to remark that, when we consider round-based protocols[1], 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 round $r+1$.
- In an asynchronous distributed system it is possible that p_i is 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), or
  - 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

[1] During round r of a round-based protocol, each process executes sequentially the following steps: sending of a round r message to all processes, waiting for a round r message from each process, and then execution of a local computation.

- 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 its round r message, or before its round r message, or even before its round $r-1$ message (second uncertainty).

So, for round-based protocols, an 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[2].

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 Figure 1. The function Global_data returns the value of GD that is locally saved in 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 $\mathcal{F}(GD)$. (To simplify the statement of the specification given in Section 3.2, the variables $term_i$ and GD_i are assumed to survive process crashes.)

```
Function Global_function

begin
        ... % local determination of v_i (≠ ⊥)%
        GD_i ← Global_data(v_i);
        term_i ← true;
        ... % local computation of F(GD_i) %
end
```

Figure 1: Computing a Global Function

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

[2] When there are no process crashes, it is well known how to to simulate synchronous systems on top of asynchronous systems: *synchronizers* [1, 13] perform such simulations.

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[3]:

- Termination. The boolean flag $term_i$ of every correct process p_i eventually becomes true.
- Validity. $\forall i: \quad (term_i \Rightarrow (\forall j: \quad GD_i[j] \in \{v_j, \perp\}))$.
- Agreement. $\forall i, j: \quad ((term_i \wedge term_j) \Rightarrow \quad (\forall k: (GD_i[k] = GD_j[k])))$.
- Obligation. $\forall i: \quad (term_i \Rightarrow (GD_i[i] = v_i))$.

The first property is a Non-Blocking (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, 7]. 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: a value "proposed" by a correct process cannot 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[4]. 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 [15], we are interested in a protocol that does not require processes to exchange lists of suspects. A message is allowed to carry only its identity[5] and an estimate of the value of the global data. Allowing messages to carry lists of suspects would give rise to a 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 Figure 2. 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 $T2$.

[3]The GDC problem is similar to the *Interactive Consistency* problem that has been defined in the context of synchronous systems with Byzantine process failures [18]. Concerning this similarity, see also the open problem (related to optimality [6]) stated at the end of the conclusion.

[4]When the failure detector is unreliable, the "process crash" notion can at best be approximate. In that case, a weakest problem called *the Global View Computation* problem [14] can be defined and solved in an asynchronous system equipped with a failure detector belonging to the class $\diamond\mathcal{S}$.

[5]The identity of an *estimate* message is the pair defined by the identity of its sender plus its sequence number (namely, a round number).

Task $T1$ This is the main task: it constructs 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 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. During round r, they exchange their current estimates of the value of the global data in order to obtain a more significant estimate value. The local variable r_i (initialized to 0) defines the current round number of p_i.

Let $\mathsf{part}(i, j, r)$ 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, $\forall r \leq 0,\ \forall i,\ \forall j,\ \mathsf{part}(i, j, r) = true$. 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 $r \geq 1$:
- $prev\_expected_i = \{j | \mathsf{part}(i, j, r-2)\}$. 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 round $r-1$.
- $cur\_expected_i = \{j | \mathsf{part}(i, j, r-1)\}$. 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.
- $next\_expected_i = \{j | \mathsf{part}(i, j, r)\}$. This set contains the processes from which p_i has received a round r message during its execution of round 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. 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 below) 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 the processes it does not suspect[6] (message $decide(i, gd_i)$ sent at line 14), and returns gd_i as the result of the call to $\mathsf{Global\_data}(v_i)$ (**return** statement, line 14).

[6]Note that some process it does not suspect can actually have crashed.

Task $T2$ This task is associated with the processing of a $decide(k, rec\_gd_k)$ message. If, while the execution of Global_data(v_i) is not yet locally terminated, 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* [12] that disseminate the termination decision. As it is possible that processes terminate at distinct rounds, this ensures that the "last" processes to decide will not block before deciding.

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 estimates of the other processes*". When this occurs, the estimates of all non-crashed processes are equal, and consequently p_i's current estimate can no more be improved; this value defines then the global data. Moreover, as the processes have then the same estimate value, if a process p_j terminates during a round $> r$, 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 $C_{2.2}$).

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 [12]. Such a protocol ensures that after $t + 1$ rounds, every initial value is known either by every non-crashed process, or by none of them[7]. This observation permits to conclude that, after $t + 1$ rounds, the non-crashed processes agree on the global data, and thus can terminate.

Condition C_2: $(prev\_expected_i = next\_expected_i) \wedge (\forall j \in next\_expected_i : \ (gd_i = rec\_gd_j))$
This condition allows early termination. When this conjunction of predicates, evaluated by a process p_i, holds at the end of a round r, all processes that have completed $r - 1$ have the same estimate value. This condition is actually a predicate that detects "a stability" property over the last two rounds.

- **Sub-condition $C_{2.1}$:** $prev\_expected_i = next\_expected_i$
  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

[7] 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.

$r-1$ have the same global data estimate[8]. But, due to the asynchrony of crash suspicions (see the discussion of Section 2.3), a process that crashes while executing r can be suspected by other processes while those are executing $r-1$. Consequently, $C_{2.1}$ alone cannot allow p_i to safely terminate.

- **Sub-condition $C_{2.2}$**: $\forall j \in next\_expected_i : (gd_i = rec\_gd_j)$
  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 of crash suspicions, there may exist 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 sub-condition 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-*$\perp$ *values than the other estimates*".

In the worst case, C_2 holds during the round $2f+2$. Combined with C_1, we get $\min(t+1, 2f+2)$ as an upper bound for the number of rounds.

```
Function Global_data(v_i)

  cobegin
(1) task T1:  gd_i ← [⊥, ..., v_i, ..., ⊥];
(2)           r_i ← 0; cur_expected_i ← Π; next_expected_i ← (Π − suspected_i);
(3)           loop
(4)             r_i ← r_i + 1; prev_expected_i ← cur_expected_i; cur_expected_i ← next_expected_i;
(5)             ∀p_j ∈ cur_expected_i do send estimate(i, r_i, gd_i) to p_j enddo;
(6)             wait until (∀p_j ∈ cur_expected_i : ((estimate(j, r_i, rec_gd_j) is received) ∨ (p_j ∈ 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 ∈ next_expected_i do
(9)               forall p_k such that p_k ∈ Π do
(10)                if ((gd_i[k] = ⊥) ∧ (rec_gd_j[k] ≠ ⊥)) then gd_i[k] ← rec_gd_j[k] endif
(11)              enddo
(12)            enddo;
(13)            if (r_i = t + 1) ∨ ((prev_expected_i = next_expected_i) ∧ (∀j ∈ next_expected_i : (gd_i = rec_gd_j)))
(14)              then ∀p_j ∈ (Π − (suspected_i ∪ {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)          ∀p_j ∈ (Π − (suspected_i ∪ {i, k})) do send decide(i, rec_gd_k) to p_j enddo; return(rec_gd_k)
  coend
```

Figure 2: The Global Data Computation Protocol

[8]This would follow from the fact that the non-crashed processes have received an estimate from each other, and from the fact there is no suspicion during $r-1$.

4.3 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.

Synchronization

Lemma 1 *When a process completes the execution of the* **wait** *statement of a round r, any non-crashed 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 has 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 non-crashed process has previously completed the execution of the round $r-1$. $\square_{Lemma\ 1}$

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

Agreement Property

Lemma 2 *If all the non-crashed processes complete a round r with the same global data value then any process p_i that completes a round $r' \geq r$, has also 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 $r'+1$ has also 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 it is no more 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 can not be different from gd. $\square_{Lemma\ 2}$

Lemma 3 *Let r be a round during which some processes learn new initial values. Let p_i be such a process, and v_k (the initial value of process p_k) be one of the values learnt 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. $\forall\ 1 < l \leq r \quad (gd^l_{\sigma(l)}[k] = v_k) \wedge (gd^{l-1}_{\sigma(l)}[k] = \perp)$,
2. *the first $r-1$ processes are faulty and have crashed 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 because $p_{\sigma(1)} = p_i$ has learnt the value v_k from $p_{\sigma(0)} = p_k$ during the first round. 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: $\mathsf{part}(\sigma(r), \sigma(r-1), r) \wedge gd^{r-1}_{\sigma(r-1)}[k] = v_k$. We now show (by contradiction) that $p_{\sigma(r-1)}$ has learnt the value v_k during round $r-1$. If it was not the case, one would have: (i) $gd^{r-2}_{\sigma(r-1)}[k] = v_k$. Since $\mathsf{part}(\sigma(r), \sigma(r-1), r)$, we deduce that (ii) $\mathsf{part}(\sigma(r), \sigma(r-1), r-1)$. (i) and (ii) imply that: $gd^{r-1}_{\sigma(r)}[k] = v_k$, a contradiction. Thus, $p_{\sigma(r-1)}$ satisfies the assumptions of the lemma during round $r-1$, from which we conclude that the first part of the lemma holds 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 $gd^{l-1}_{\sigma(l)} = \perp$, we have $\neg\mathsf{part}(\sigma(l), \sigma(l-2), l-1)$, and thus $p_{\sigma(l)}$ has suspected $p_{\sigma(l-2)}$ before round l. Thus, $p_{\sigma(l-2)}$ has crashed before round $l+1$ (by Lemma 1). Indeed, it 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 $r+1$.

$\square_{Lemma\ 3}$

Lemma 4 *The processes that terminate 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 $(p_{\sigma(0)}, \ldots, p_{\sigma(r)})$, 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 $(p_{\sigma(0)}, \ldots, p_{\sigma(r)} = (p_{\sigma(0)}, \ldots, p_{\sigma(n-2)}, p_{\sigma(n-1)})$. From Lemma 3, only $p_{\sigma(n-2)}$ and $p_{\sigma(n-1)}$ can be non-crashed during round $n = t+1$. In that case, both known the value v_k by the end of round $n-1 = t$. It follows that every non crashed process has learnt v_k by the end of round $t+1$.

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 learnt 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 $t+1$.

$\square_{Lemma\ 4}$

Lemma 5 *Let r be the first round during which the condition C_2 is satisfied for a process (p_i). Then, all the processes that have completed round $r-1$ belong to the set $prev\_expected^r_i$.*

Proof The proof is a case analysis.

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

– $r = 2$
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, $p_k \in prev\_expected_i^2$.

– $r > 2$
As process p_i has reached round r, it has obviously received a message sent by p_k during round $r-2$. In other words, $\mathsf{part}(i, k, r-2)$ and thus $p_k \in next\_expected_i^{r-2} = cur\_expected_i^{r-1} = prev\_expected_i^r$ (line 4).

$\square_{Lemma\ 5}$

Theorem 1 *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 ([9]). 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:

- Either $C_1(r = t+1)$ is satisfied. Due to Lemma 4, all processes share the same global data at the end of round r.
- Or C_2 is satisfied. From Lemma 5, we conclude that $X = prev\_expected_i^r = next\_expected_i^r$ contains all the processes that had not crashed at the end of $r-1$. The condition $C_{2.2}$ guarantees that all these processes had 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.

$\square_{Theorem\ 1}$

Obligation Property

Theorem 2 *If a process p_i completes the execution of* Global_data, *then the value of the returned global data includes 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 1. 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.

[9] These processes possibly decide at different rounds.

- 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 (this is 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 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 $[\bot, \ldots, v_i, \ldots, \bot]$ 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.

$\Box_{Theorem\ 2}$

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 $\bot$.*

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). $\Box_{Theorem\ 3}$

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 first 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, due to the completeness property of the underlying failure detector, they will eventually 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 at 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. $\Box_{Theorem\ 4}$

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 $\mathrm{C}_1, \mathrm{C}_{2.1}$ and $\mathrm{C}_{2.2}$ to be satisfied, respectively. Since the termination condition is $\mathrm{C}_1 \vee (\mathrm{C}_{2.1} \wedge \mathrm{C}_{2.2})$, 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 end of round $f+1$ (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 is not satisfied as long as exactly one new suspicion is learnt 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. $\square_{Theorem\ 5}$

5 Hardness of the *GDC* Problem

5.1 Problem Reduction

This section considers the *hardness* of a problem with respect to the difficulty to solve it in presence of process crashes: a problem is *easy* if it can be solved despite arbitrarily process crashes; it is *hard* if it requires additional assumptions to be solved in presence of process crashes [8, 12]. To address this issue, Chandra, Hadzilacos and Toueg [5, 12] 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 $\mathcal{A}_{P1 \to P2}$ 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 (may be more) required to solve $P2$ are necessary to solve $P1$ (the situation can even be worst, it is possible that $P2$ can be solved while $P1$ cannot). If $P1 \geq P2$ and $P2 \geq P1$, then $P1$ and $P2$ are said *equivalent*. This is denoted $P1 \simeq P2$.

5.2 A New Reduction

$P \geq GDC$
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 GDC. Consequently, this protocol is a *reduction* transforming any protocol solving P into a protocol solving GDC. Hence, we have: $P \geq GDC$.

$GDC \geq P$
We give here a simple protocol that, given a solution to the GDC problem, solves P. This protocol is described in Figure 3. For each process p_i, the protocol is made of two tasks.

$T2$ 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.

$T1$ 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 had crashed before this global data computation (note that $suspected_i$ can also include processes that have crashed during this computation).

```
suspected_i ← ∅;
cobegin
task T1: while true do
                GD_i ← Global_data(v_i); % v_i ≠ ⊥ %
                suspected_i ← {j | GD_i[j] = ⊥}
         enddo

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

Figure 3: $GDC \geq P$

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 k^{th} instance of the GDC problem. If p_i has crashed before the k^{th} 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 *Non-Blocking Atomic Commitment* problem [3], and TRB, the *Terminating Reliable Broadcast* problem[12][10]. Combined with previous results on the classification of problems in asynchronous systems prone to process crashes [5, 8, 11, 12], we have the following problem equivalence:

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

So, according to the problem classes defined in [8], GDC belongs to the class (called NFC) which contains the hardest problems to solve in presence of process crashes.

6 Examples of Global Function Computation

The solution to several distributed computing problems amounts to compute a global function [10, 13]. This section shows how the previous framework allows to solve those problems in the context of asynchronous distributed systems with process crashes. To this end, two problems are briefly examined.

Non-Blocking Atomic Commitment
This well-known problem is mainly encountered in data management systems [3]. It has been sketched in the Introduction.

[10]The TRB problem is a variant of the Byzantine generals problem. See [12] for more details.

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, this depends 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 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*" [11].

Distributed Termination Detection

This problem is a well-known paradigm of reliable distributed computing. We assume the reader 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, as any message reactivates its destination process). A lot of protocols have been proposed to detect distributed termination [13, 17, 21].

In our context, a distributed computation is *terminated* if it has entered a state from which each of its non-crashed 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 non-crashed. Its value is obtained as the result of a global function computation (*i.e.*, all the processes have the same value for NC). Initially, $NC = \Pi$.

We extend here a variant of a protocol (denoted M) proposed by Mattern [17] 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 to 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 respectively $GD1$ and $GD2$. 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. $GD2$)[11] 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 $GD2$.

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

- If $NC1 = NC2$, then process p_i computes:
  - $R1 = \sum_{j,k \in NC1, rec_j \in GD1} rec_j[k]$. This value represents the number of messages received by processes in $NC1$ from processes in $NC1$, 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 non-crashed, 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 to cope with process crashes. It is possible to show that if the application terminates, then the protocol will eventually claim it (liveness of the detection), and 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. (2) If a process p_i crashes during $GD2$, it does belong to $NC1$ (hence, to $NC2$). So, messages sent or received by such a process p_i appear in R1 and S2.

7 Conclusion

This paper has addressed the computation of a global data function in 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 non-crashed 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. We have shown that GDC belongs to the class of problems that are the *hardest* to solve (with respect to the assumptions they require) in 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 2 rounds (when $t > 1$). Moreover, processes never exchange information on crashes. The message size depends only on the round number size 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, then round numbers can be implemented $mod\ 2$. 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 ([6], Section 5, page 740), similar to the Global Data Computation problem, that can be solved in a synchronous distributed system in $\min(f + 2, t + 1)$ rounds (which has been shown to be a lower bound [6] in such systems). So, an interesting open question is the following one: "*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.

References

[1] Awerbuch B., Complexity of Network Synchronization. *Journal of the ACM*, 32(4):802-823, 1985.

[2] Bermond J.-Cl., Konig J.-Cl. and Raynal M., General and Efficient Decentralized Consensus Protocols. *Proc. 2d Int. Workshop on Distributed Algorithms (WDAG'87)*, Springer-Verlag LNCS 312 (J. Van Leeuwen Ed.), pp. 41-56, Amsterdam, 1987.

[3] P.A. Bernstein, V. Hadzilacos and N. Goodman., *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 370 pages, 1987.

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

[5] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, March 1996.

[6] Dolev D., Reischuk R. and Strong R., Early Stopping in Byzantine Agreement. *Journal of the ACM*, 37(4):720-741, April 1990.

[7] 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.

[8] Fromentin E., Raynal M. and Tronel F., On Classes of Problems in Asynchronous Distributed Systems with Process Crashes. *Proc. 19th IEEE Int. Conf. on Distributed Computing Computing (ICDCS'99)*, Austin (TX), May 1999.

[9] Garg V.K. and Ghosh J., Repeated Computation of Global Functions in a Distributed Environment. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):823-834, 1994.

[10] Garg V.K., *Principles of Distributed Systems* (Chapter 10). Kluwer Academic Pub., 254 pages, 1996.

[11] Guerraoui R., Revisiting the Relationship between Non-Blocking Atomic Commitment and Consensus. *Proc. 9th Int. Workshop on Distributed Algorithms (WDAG'95)*, Springer-Verlag LNCS 972 (J.M. Hélary and M. Raynal Eds), pp. 87-100, 1995.

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

[13] Hélary J.-M. and Raynal M., *Synchronization and Control of Distributed Systems and Programs*, John Wiley & Sons, 124 pages, 1990.

[14] Hurfin M., Raynal M., and Tronel F. A Practical Building Block for Solving Agreement Problems in Asynchronous Distributed Systems. *Proc. 16th IEEE Int. Performance, Computing and Communications Conference (IPCCC'98)*, pp.25–31, Phoenix, AZ, February 1998.

[15] Lynch N., *Distributed Algorithms* (Chapter 21, Section 21.4). Morgan Kaufmann, 872 pages, 1996.

[16] Matocha J. and Camp T., A Taxonomy of Distributed Termination Detection Algorithms. *Journal of Systems and Software*, 43:207-221, 1998.

[17] Mattern F., Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161-175, 1987.

[18] Pease L., Shostak R. and Lamport L., Reaching Agreement in Presence of Faults. *Journal of the ACM*, 27(2):228-234, 1980.

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

[20] Rufino J., Veríssimo P., Arroz P., Almeida C. and Rodrigues L., Fault-Tolerant Broadcast in CAN. *Proc. 28th Int. Symposium on Faut-Tolerant Computing (FTCS'28)*, Munich, pp. 150-159, 1998.

[21] Tel G., *Introduction to Distributed Algorithms* (Chapter 8). Cambridge University Press, 534 pages, 1994.

[22] Veríssimo P., *Real-Time Communication* (Chapter 17 in *Distributed Systems, 2nd Edition)*. Addison-Wesley and ACM Press, (S. Mullender Ed.), pp. 447-490, 1993.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399