A simple proof of the necessity of the failure detector Σ to implement a register in asynchronous message-passing systems

François Bonnet, Michel Raynal

Publications Internes de l'IRISA
ISSN : en cours
PI 1932 – juin 2009

A simple proof of the necessity of the failure detector Σ to implement a register in asynchronous message-passing systems

François Bonnet* , Michel Raynal**
francois.bonnet@irisa.fr, raynal@irisa.fr

Abstract: This paper presents a simple proof that the quorum failure detector class (denoted Σ) is the weakest failure detector class required to implement an atomic read/write register in an asynchronous message-passing system prone to an arbitrary number of process crashes.

Key-words: Asynchronous message-passing system, Atomic register, Necessity proof, Process crash, Weakest failure detector.

Une preuve simple de la nécessité du détecteur de fautes Σ pour implémenter un registre dans un système asynchrone à passage de messages

Résumé : *Ce rapport propose une preuve simple de la nécessité du détecteur de fautes Σ pour implémenter un registre dans un système asynchrone à passage de messages.*

Mots clés : *Asynchronisme, Défaillances des processus, Détecteur de fautes, Preuve de ncessité, Registre atomique.*

---

* Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes
** Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes

1 Introduction

Atomic register Among the objects that allow concurrent processes to exchange information and cooperate to a common goal, the atomic register is certainly the most fundamental. Such an object (let us denote it REG) provides the processes with two operations $REG.\mathsf{read}()$ and $REG.\mathsf{write}(v)$. The read operation provides the invoking process with the value of the object, while the write operation associates a new value v with the object.

Atomicity [10, 11] means that the read and write operations issued on a register appear as if they have been executed sequentially, and this "witness sequence" is (1) legal (a read returns the value written by the closest write that precedes it in this sequence) and (2) respects the real time occurrence order on the operations (if the operation $op1$ terminates before an operation $op2$ starts, $op1$ appears before $op2$ in the witness sequence).

Simulating a register in an asynchronous system In an asynchronous message-passing system, the processes communicate by sending and receiving message through channels, and there are assumptions neither on the speed of processes, nor on message transmission delays.

If the system is reliable, it is easy to build an atomic register on top of an asynchronous message-passing system. This is no longer the case if processes can crash. Let n be the number of processes that compose the system and t be a model parameter that defines an upper bound on the number of processes that may crash. Algorithms that build an atomic register object despite asynchrony and up to $t < n/2$ process crashes are described in [1, 2].

An important result is also shown in [2], namely, there is no algorithm implementing an atomic register in asynchronous message-passing systems where $t \geq n/2$. The intuition that underlies this impossibility is that, due to asynchrony and the fact that $t \geq n/2$, the system can appear as being partitioned, in such a way that each partition considers that the processes in the other partition have crashed (while they actually have not).

The failure detector approach to circumvent the "$t \geq n/2$" impossibility The failure detector approach [4] has been introduced to circumvent impossibility results. It consists in enriching each process of an unreliable asynchronous system with an additional device (sometimes called "oracle") that provides it with hints on process failures. According to the type and the quality of these hints, several classes of failure detectors can been defined.

The class of *quorum* failure detectors, denoted Σ, has been introduced by Delporte-Gallet, Fauconnier and Guerraoui in [5]. It is shown in [5, 6] that Σ is the weakest class of failure detectors that allows building an atomic register object in asynchronous message passing systems despite any number of process crashes (i.e., in systems where $t = n-1$). "Weakest" means that Σ captures the minimal information on failures that has to be known by the processes in order to implement a register. The definition of Σ is given below. (A quorum is a set of processes. Quorums have first been introduced by Gifford [9].)

Content of the paper Showing that Σ is the weakest class of failure detectors to build a register amounts to show two things, namely, that Σ is sufficient and that it is necessary.

On the "sufficiency" part, designing a Σ-based algorithm that builds a register is relatively easy. It consists in replacing the "majority of non-faulty processes" assumption used in algorithms such as the ones described in [2, 3, 12] by a Σ quorum. From an operational point of view, this amounts to replace the statement "wait for messages from a majority of processes" by the statement "wait for messages from a quorum". Such Σ-based algorithms are described in [5, 8].

The difficult part is the "necessity" part. Let D be any failure detector that allows building a register in an asynchronous message-passing system despite any number of process crashes, and A be any D-based algorithm that builds a register. The proof of the "necessity" part consists in showing that, given any D-based algorithm A, it is possible to build a failure detector of the class Σ (we say that it is possible to "extract" Σ from A). The first such extraction algorithm appeared in [5].

We present in this paper a new proof of the "necessity" part, that is particularly simple. This proof is based on a technique totally different from the one used in [5]. Interestingly and in addition to its simplicity, the proposed extraction algorithm does not use sequence numbers and requires only a bounded local memory at each process.

2 Computation model and definitions

2.1 Asynchronous message-passing Systems

As already indicated, the computation model consists of n asynchronous processes (denoted p_1, ..., p_n) that communicate by exchanging messages through point-to-point reliable asynchronous channels. The integer i is the identity of p_i. Let $\Pi = \{1, \ldots, n\}$. Up to $t = n-1$ processes can crash. A crash is a premature halting: the process stops executing. Until it crashes a process executes correctly the code of its algorithm. Given a run, a process that crashes is said to be *faulty* in that run. Otherwise, it is correct. In the following $\mathcal{C}$ denote the set of correct processes.

The underlying time model is the set of integers. This time notion is not accessible to the processes. It can only be used from an external observer point of view to state or prove properties. Time instants are denoted by τ,τ', etc.

2.2 The failure detector class Σ

As indicated previously, the *quorum failure detector* class has been introduced and investigated in [5]. Each process p_i is provided with a local variable (denoted Σ_i) that it can only read. At any time, such a variable contains a set of process identities (quorum). Let Σ_i^τ be the value of Σ_i at time τ. The class Σ contains all the failure detectors that satisfy the following properties ($\mathcal{C}$ denotes the set of identities of the processes that are correct in the considered run):

- Intersection property. $\forall\, i,j \in \{1,\ldots,n\}$: $\forall\, \tau,\tau'$: $\Sigma_i^\tau \cap \Sigma_j^{\tau'} \neq \emptyset$.
- Liveness property. $\exists \tau$: $\forall\, \tau' \geq \tau$: $\forall i \in \mathcal{C}$: $\Sigma_i^{\tau'} \subseteq \mathcal{C}$.

The first property states that the values of any two quorums taken at any times do intersect. This property prevent partitioning and is consequently used to maintain the consistency of the atomic register. The second property states that a quorum cannot block the process that uses it. Because two majorities always intersect, it is easy to see that Σ can be implemented in systems where $t < n/2$. Differently, it cannot be implemented in pure asynchronous systems when $t \geq n/2$.

3 Σ is necessary to build a register

3.1 Principle

Aim The aim is to design an algorithm that emulates the output of Σ at each process p_i. This algorithm uses as a subroutine any algorithm A and failure detector D such that A is a D-based algorithm that implements an atomic register in an asynchronous message-passing system prone to any number of process crashes.

A simple task Q being any non-empty set of processes, let us consider an array of n atomic registers $REG_Q[1..n]$, initialized to $[\bot,\ldots,\bot]$, and the task denoted WR_Q where each process p_i such that $i \in Q$ executes the following algorithm (where $reg_i[1..n]$ is an array local to p_i):

algorithm WR:
$REG_Q[i]$.write($\top$); **for each** $x \in \{1,...,n\}$ **do** $reg_i[x] \leftarrow REG_Q[x]$.read() **end for**.

The process p_i first writes the value $\top$ in its entry of the array REG_Q, and then reads asynchronously all its entries. The $REG_Q[i]$.write($\top$) and $REG_Q[x]$.read() operations are provided to the processes by the previous algorithm A. (Let us notice that the value obtained by a read is irrelevant. As we will see, what is important is the fact that $REG[x]$ has been written or not.) A corresponding run of WR_Q is denoted E_Q. In that run, no process outside Q sends or receives messages related to the task WR_Q.[1]

Let us observe that, as the underlying failure detector-based algorithm A that builds a register is correct, if the set Q contains all the correct processes (i.e., $\mathcal{C} \subseteq Q$), E_Q is such that every correct process terminates the task WR_Q. In the other cases, i.e., for the tasks WR_Q such that $\neg(\mathcal{C} \subseteq Q)$, E_Q is such that a process of Q either terminates WR_Q, or blocks forever, or crashes (this depends on the actual failure pattern, the outputs of the underlying failure detector D used by the algorithm A, and the code of A).

Running concurrently 2^n-1 tasks The extraction algorithm considers the 2^n-1 distinct tasks WR_Q where Q is a non-empty set of 2^Π. This means that each process p_i manages 2^{n-1} threads, one for each subset Q such that $i \in Q$. Let us notice that the crash of a process p_i entails the crash of all its threads.

Let us finally recall that each register of an array $REG_Q[1..n]$ is implemented by the algorithm A executed by $|Q|$ threads associated with the processes of Q. Due to the correctness of A, each $REG_Q[x]$ is an atomic register.

3.2 The extraction algorithm

The algorithm that extracts Σ is described in Figure 1. Let us recall that the aim is to provide each process p_i with a local variable Σ_i such that the $(\Sigma_x)_{1\leq x\leq n}$ variables satisfy the intersection and liveness properties defined in Section 2.2.

To that end, each process p_i manages two local variables: a set of sets of process identities, denoted $quorum\_sets_i$, and a queue denoted $queue_i$. The aim of $quorum\_sets_i$ is to contain all the sets Q such that p_i terminates WR_Q (task $T1$), while $queue_i$ is managed

[1] When we consider the underlying failure detector-based algorithm A that implements the registers $REG_Q[1..n]$, as the processes that are not in Q do not participate in WR_Q, the messages sent by the processes of Q to these processes are never received, or are delayed for an arbitrarily long period. Alternatively (as in [7]), we could say that, in WR_Q, the processes of Q "omit" sending messages to the processes that are not in Q.

in such a way that eventually the correct processes appear in it before the faulty processes (tasks $T2$ and $T3$).

The idea is to select a set of $quorum\_sets_i$ as the current output of Σ_i. As we will see in the proof, given any pair of processes p_i and p_j, any quorum in $quorum\_sets_i$ has a non-empty intersection with any quorum in $quorum\_sets_j$, thereby supplying the required intersection property.

The main issue is to ensure the liveness property of Σ_i (eventually Σ_i has to contain only correct processes) while preserving the intersection property. This is realized with the help of the local variable $queue_i$ as follows: the current output of Σ_i is the set (quorum) of $quorum\_sets_i$ that appears as being the "first" in $queue_i$. The formal definition of "first set of $quorum\_sets_i$ with respect to $queue_i$" is stated in the task $T4$. To make it easy to understand, let us consider the following example. Let $quorum\_sets_i = \{\{3,4,9\},\{2,3,8\},\{4,7\}\}$, and $queue_i =< 4,8,3,2,7,5,9,\cdots >$. The set $S = \{2,3,8\}$ is the first set of $quorum\_sets_i$ with respect to $queue_i$ because each of the other sets $\{3,4,9\}$ and $\{4,7\}$ includes an element (9 and 7, respectively) that appears in $queue_i$ after the elements of S. (In case several sets are "first", any of them can be selected).

Init: $quorum\_sets_i \leftarrow \{\{1\ldots,n\}\}$; $queue_i \leftarrow < 1,\ldots,n >$;
  for each $Q \in (2^{\Pi} \setminus \{\emptyset,\{1,\ldots,n\}\})$ **do**
    if $(i \in Q)$ **then** launch a thread associated with the task WR_Q **end if end for**.
  % Each process p_i participates concurrently in all the tasks WR_Q such that $i \in Q$ %

Task $T1$: **when** p_i terminates in the task WR_Q: $quorum\_sets_i \leftarrow quorum\_sets_i \cup \{Q\}$.

Task $T2$: **repeat periodically** broadcast ALIVE(i) **end_repeat**.

Task $T3$: **when** ALIVE (j) **is received**: suppress j from $queue_i$; enqueue j at the head of $queue_i$.

Task $T4$: **when** p_i reads Σ_i:
  let $m = \min_{Q \in quorum\_sets_i}(\max_{x \in Q}(rank[x]))$ where $rank[x]$ denotes the rank of x in $queue_i$;
  return (a set Q such that $\max_{x \in Q}(rank[x]) = m$).

Figure 1: Extracting Σ from a failure detector-based algorithm A that implements a register (code for p_i)

Remark Initially $quorum\_sets_i$ contains the set $\{1,\ldots,n\}$. As no set of processes is ever withdrawn from $quorum\_sets_i$ (task $T1$), $quorum\_sets_i$ is never empty. Moreover, it is not necessary to launch the task $WR_{\{1,\ldots,n\}}$ in which all the processes participate. This is because, as the underlying failure detector-based algorithm A (that implements a register) is correct, it follows that all the correct processes decide in the task $WR_{\{1,\ldots,n\}}$. This case is directly taken into account in the initialization of $quorum\_sets_i$ (thereby saving the execution of the task $WR_{\{1,\ldots,n\}}$).

3.3 The necessity theorem

Theorem 1 *Let A be any failure detector-based algorithm that implements an atomic register in an asynchronous message-passing system prone to any number of process crashes. Given A, the algorithm described in Figure 1 is a bounded construction of a failure detector of the class Σ.*

Proof

Proof of the intersection property The proof is by contradiction. Let us first observe that the set Σ_i returned to a process p_i is a set of $quorum\_set_i$ (that contains the set $\{1,\ldots,n\}$ -initial value- plus all the sets Q such that p_i terminates WR_Q). Let us assume that there are two sets Q_1 and Q_2 such that (1) $Q_1, Q_2 \in \bigcup_{1 \le j \le n}(quorum\_set_j)$, and (2) $Q_1 \cap Q_2 = \emptyset$. The first item means that Q_1 and Q_2 can be returned to some processes as their local value for Σ.

Let p_i be a process that terminates WR_{Q_1} and p_j a process that terminates WR_{Q_2} (due to the "contradiction" assumption, such processes do exist). Using the fact that the message-passing system is asynchronous, let us construct the runs E_{Q_1} and E_{Q_2} associated with WR_{Q_1} and WR_{Q_2} as follows. If any (see footnote 1), the messages sent by the processes of Q_1 to the processes of Q_2, when they execute A to implement each register of the array REG_{Q_1}, are delayed for an arbitrarily long period (until p_i has added Q_1 to $quorum\_set_i$ and p_j has added Q_2 to $quorum\_set_j$). And similarly for the messages sent by the processes of Q_2 to the processes of Q_1 when they execute A for each register of the array REG_{Q_2}.

Let us observe that, in the concurrent runs E_{Q_1} and E_{Q_2}, the algorithm A that is executed only by (1) the processes of Q_1 in E_{Q_1} to build the registers $REG_{Q_1}[1..n]$, and (2) only the processes of Q_2 in E_{Q_2} to build the registers $REG_{Q_2}[1..n]$, is fed with the same outputs of the underlying failure detector D. Due to the fact that (if any) the messages from Q_1 to Q_2 and from Q_2 to Q_1 are delayed, we have that p_i reads $\perp$ from $REG_{Q_1}[j]$ in E_{Q_1}, and p_j reads $\perp$ from $REG_{Q_2}[i]$ in E_{Q_2}.

Let us construct a run $E_{Q_{12}}$, where $Q_{12} = Q_1 \cup Q_2$, that is a simple merge of E_{Q_1} and E_{Q_2} defined as follows. In this run, the algorithm A (that involves only the processes in Q_{12} and implements the array of registers $REG_{Q_{12}}[1..n]$) is fed with the same failure detector outputs as the ones supplied to the concurrent runs E_{Q_1} and E_{Q_2}. Moreover, the messages from Q_1 to Q_2 and from Q_2 to Q_1 are delayed as in E_{Q_1} and E_{Q_2}. So, p_i (resp., p_j) receives the same messages and the same outputs from the underlying failure detector in $E_{Q_{12}}$ and E_{Q_1} (resp., E_{Q_2}).

- On the one side, we have the following. As the process p_i receives the same messages and the same failure detector outputs in $E_{Q_{12}}$ as in E_{Q_1}, the arrays $REG_{Q_1}[1..n]$ and $REG_{Q_{12}}[1..n]$ contains the same values. Consequently, p_i reads $\perp$ from $REG_{Q_{12}}[j]$. Similarly, p_j reads $\perp$ from $REG_{Q_{12}}[i]$.
- On the other side we have the following. In $E_{Q_{12}}$, the process p_i writes $\top$ into $REG_{Q_{12}}[i]$ and the process p_j writes $\top$ into $REG_{Q_{12}}[j]$. Moreover, one of these operations terminates before the other. Without loss of generality, let us assume that the write by p_i terminates before the write by p_j. Consequently, p_j reads $REG_{Q_{12}}[i]$ after it has been written. Due to the atomicity of that register, it follows that p_j obtains the value $\top$ when it reads $REG_{Q_{12}}[i]$.

The second item contradicts the first one. It follows that the initial assumption (existence of a failure detector-based algorithm A that builds a register, $Q_1, Q_2 \in \bigcup_{1\leq j\leq n}(quorum\_set_j)$ and $Q_1 \cap Q_2 = \emptyset$) is false, from which we conclude that at least one of the assertions $Q_1, Q_2 \in \bigcup_{1\leq j\leq n}(quorum\_set_j)$ and $Q_1 \cap Q_2 = \emptyset$ is false, which completes the proof of the intersection property (Corollary 1 -stated below- is an immediate consequence of that property).

Proof of the liveness property As far as the liveness property is concerned, let us consider the task $WR_{\mathcal{C}}$ (recall that $\mathcal{C}$ is the set of correct processes). As the underlying failure detector-based algorithm A that implements the registers $REG_{\mathcal{C}}[1..n]$ is correct (assumption), each correct process p_i terminates its $REG_{\mathcal{C}}[i].\mathsf{write}(\top)$ and $REG_{\mathcal{C}}[x].\mathsf{read}()$ operations in $E_{\mathcal{C}}$. Consequently, in the extraction algorithm, the variable $quorum\_set_i$ of each correct process p_i eventually contains the set $\mathcal{C}$.

Moreover, after some finite time, each correct process p_i receives $\text{ALIVE}(j)$ messages only from correct processes. This means that, at each correct process p_i, all the correct processes eventually precede the faulty processes in $queue_i$. Due to the definition of "first set of $quorum\_set_i$ with respect to $queue_i$" stated in the task $T4$, it follows that, from the time $\mathcal{C}$ has been added to $quorum\_set_i$, the quorum Q selected by the task $T4$ is always such that $Q \subseteq \mathcal{C}$, which proves the liveness property of Σ_k.

The construction is bounded A simple examination of the extraction algorithm shows that (1) both the variables $queue_i$ and $quorum\_sets_i$ are bounded, and (2) messages carry bounded values, from which it follows that the construction is bounded. $\square_{Theorem\ 1}$

The proof of intersection property shows that it is not possible to have two sets Q_1 and Q_2 such that $Q_1 \cap Q_2 = \emptyset$ and at least one process of Q_1 terminates WR_{Q_1} and at least one process of Q_2terminates WR_{Q_2}. Hence the following corollary.

Corollary 1 *Let two sets Q_1 and Q_2 such that $Q_1 \cap Q_2 = \emptyset$. Then, no process of Q_1 terminates WR_{Q_1} or no process of Q_2 terminates WR_{Q_2} (or both).*

References

[1] Attiya H., Efficient and Robust Sharing of Memory in Message-passing Systems. *Journal of Algorithms*, 34(1):109-127, 2000.

[2] Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, 42(1):121-132, 1995.

[3] Attiya H. and Welch J., Distributed Computing: Fundamentals, Simulations and Advanced Topics, (2d Edition), *Wiley-Interscience*, 414 pages, 2004.

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

[5] Delporte-Gallet C., Fauconnier H. and Guerraoui R., Shared memory vs Message Passing. *Tech Report* IC/2003/77, EPFL, Lausanne, December 2003.

[6] Delporte-Gallet C., Fauconnier H., Guerraoui R., Hadzilacos V., Kouznetsov P. and Toueg S., The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 338-346, 2004.

[7] Delporte-Gallet C., Fauconnier H., Guerraoui R. and Tielmann A., The Weakest Failure Detector for Message Passing Set-Agreement. *Proc. 22th Int'l Symposium on Distributed Computing (DISC'08)*, Springer-Verlag LNCS #5218, pp. 109-120, 2008.

[8] Friedman R., Mostefaoui A. and Raynal M., Asynchronous Bounded Lifetime Failure Detectors. *Information Processing Letters*, 94(2):85-91, 2005.

[9] Gifford D.K., Weighted Voting for Replicated Data. *Proc. 7th ACM Symposium on Operating System Principles (SOSP'79)*, ACM Press, pp. 150-172, 1979.

[10] Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[11] Lamport L., On Interprocess communication. Part I: Formalism. Part II: Algorithms. *Distributed Computing*, 1-2(2):87-103, 1986.

[12] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.