

## The weakest failure detector to implement a register in asynchronous systems with hybrid communication

Damien Imbs<sup>\*</sup>, Michel Raynal<sup>\*\*</sup>

**Abstract:** This paper introduces an asynchronous crash-prone hybrid system model. The system is hybrid in the way the processes can communicate. On the one side, a process can send messages to any other process. On another side, the processes are partitioned into clusters and each cluster has its own read/write shared memory. In addition to the model, a main contribution of the paper concerns the implementation of an atomic register in this system model. More precisely, a new failure detector (denoted  $M\Sigma$ ) is introduced and it is shown that, when considering the information on failures needed to implement a register, this failure detector is the weakest. To that end, the paper presents an  $M\Sigma$ -based algorithm that builds a register in the considered hybrid system model and shows that it is possible to extract  $M\Sigma$  from any failure detector-based algorithm that implements a register in this model. The paper also (a) shows that  $M\Sigma$  is strictly weaker than  $\Sigma$  (which is the weakest failure detector to implement a register in a classical message-passing system) and (b) presents a necessary and sufficient condition to implement  $M\Sigma$  in a hybrid communication system.

**Key-words:** Asynchronous message-passing system, Atomic register, Distributed algorithm, Failure detector, Fault-tolerance, Hybrid communication, Necessity proof, Process crash, Shared memory system, Weakest failure detector.

---

*Le plus faible détecteur de fautes pour construire un registre dans un système à communication hybride*

**Résumé :** Ce rapport introduit un détecteur de fautes pour les systèmes répartis à communication hybride (mémoire partagée et passage de messages) et montre qu'il est le plus faible détecteur de fautes possible pour construire un registre dans un tel système.

**Mots clés :** Système réparti asynchrone, mémoire partagée, passage de messages, détecteur de fautes, optimalité.

---

<sup>\*</sup> Membre du Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes, [damien.imbs@irisa.fr](mailto:damien.imbs@irisa.fr).

<sup>\*\*</sup> Membre senior de l'IUF et Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes, [raynal@irisa.fr](mailto:raynal@irisa.fr).

# 1 Introduction

## 1.1 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  $REG.read()$  and  $REG.write(v)$ . 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 (possibly concurrent) 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). Let us observe that concurrent operations can be ordered in any way as long as the legality property stated in item (1) is satisfied.

## 1.2 Building an atomic register in a message-passing system

**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 proved 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 reader interested by a pedagogical introduction to these issues will consult [3, 12, 13].

**The failure detector approach to circumvent the “ $t \geq n/2$ ” impossibility** The failure detector approach [5, 6] 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 be defined.

The class of *quorum* failure detectors, denoted  $\Sigma$ , has been introduced by Delporte-Gallet, Fauconnier and Guerraoui in [7]. (A quorum is a set of processes. Quorums have first been introduced by Gifford [8].) It is shown in [4, 7] that  $\Sigma$  is the weakest class of failure detectors that allow 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  $\Sigma$  captures the minimal information on failures that has to be known by the processes in order to implement a register. The definition of  $\Sigma$  is given below. It is important to notice that, due to the results of [2] and [7], it follows that  $\Sigma$  cannot be implemented in asynchronous message-passing systems despite any number of crashes.

## 1.3 Content of the paper

**Towards new system models** The advent of multicore architectures where processors share a common memory and the design of clusters (where, for example, each cluster is a multicore system) communicating by message-passing opens the door for the design of new computing models where processes communicate both by shared memory (intra-cluster communication) and message passing (point-to-point communication).

**Context and content of the paper** This paper is on the construction of atomic registers in hybrid models (such as the one previously described). It has several contributions.

- It first introduces a simple asynchronous crash-prone model, denoted  $SM\_MP_{n,m}[\emptyset]$ , that captures the previous intra-cluster and point-to-point communication types (the meaning of  $m$  will be defined later).
- The paper then introduces a new failure detector, denoted  $M\Sigma$ , and
  - Presents and proves correct an algorithm that builds an atomic register in  $SM\_MP_{n,m}[M\Sigma]$  ( $SM\_MP_{n,m}[\emptyset]$  enriched with  $M\Sigma$ ),
  - Shows that  $M\Sigma$  is the weakest information on failures  $SM\_MP_{n,m}[\emptyset]$  has to be enriched with in order an atomic register can be implemented.
- The paper finally shows that  $M\Sigma$  is strictly weaker than  $\Sigma$ . It also presents a necessary and sufficient condition to implement  $M\Sigma$  in a hybrid communication system.

**Roadmap** The paper is made up of 7 sections. Section 2 presents the computation model  $SM\_MP_{n,m}[\emptyset]$ . The new failure detector class  $M\Sigma$  is introduced in Section 3. Then, Section 4 presents an algorithm that builds an atomic register in  $SM\_MP_{n,m}[M\Sigma]$  and Section 5 shows that  $M\Sigma$  is optimal. Section 6 presents a necessary and sufficient condition to implement  $M\Sigma$  in a hybrid communication system. Finally, Section 7 concludes the paper.

## 2 A hybrid communication system model

### 2.1 System model with hybrid communication

**Process model** The system comprises  $n$  processes denoted  $p_1, \dots, p_n$ . Each process  $p_i$  is asynchronous (i.e., it proceeds to an arbitrary speed) and sequential (it executes one step-base action- at a time).  $\Pi = \{1, \dots, n\}$  is the set of process identities.

A process can crash. A crash is a premature halt (after it has crashed, if it ever does, a process issues no more step). Let  $t$  be the upper bound on the number of processes that are allowed to crash. We assume here  $t = n - 1$  (this is sometimes the *wait-free* process model).

**Progress condition** In the following we are interested in a system model whose algorithms satisfy the *wait-freedom* progress condition [9]. When considering an algorithm implementing an atomic register  $REG$ , this means that a process that does not crash must return from all its invocations of the operations  $REG.read()$  and  $REG.write()$ .

**Message-passing communication** Processes can send and receive messages through reliable channels. It is assumed that any pair of processes is connected by a bidirectional channel. Channels are reliable but asynchronous. Reliable means that messages are neither corrupted, nor duplicated nor lost. Asynchronous means that, albeit finite, message transfer delays are arbitrary.

The sending and the reception of a message are atomic steps. The processes can also use a broadcast operation, but this operation is not atomic (if a process crashes during a broadcast, an arbitrary subset of the processes receive the corresponding message).

**Partially shared memory communication** The  $n$  processes are partitioned into  $m$ ,  $1 \leq m \leq n$ , non-empty subsets  $P[1], \dots, P[m]$  called clusters (i.e.,  $\cup_{1 \leq x \leq m} P[x] = \Pi$  and  $\forall x, y : (x \neq y) \Rightarrow (P[x] \cap P[y] = \emptyset)$ ).

Inside each cluster  $x$ ,  $1 \leq x \leq m$ , the processes in  $P[x]$  share a common read/write memory denoted  $MEM_x$ .  $MEM_x$  is composed of a set of 1WMR (single-writer/multi-reader) atomic registers (this assumption is without loss of generality as multi-writer/multi-reader atomic registers can be built on top single-writer/multi-reader atomic registers [3, 11, 12]). For notational convenience, we use an index/array notation for every register of  $MEM_x$ : if  $i \in P[x]$ ,  $MEM_x[i]$  can be written only by  $p_i$  and read by all processes in  $P[x]$  (if  $i \notin P[x]$ ,  $MEM_x[i]$  is meaningless and  $p_i$  cannot access  $MEM_x$ ).

Two examples of partially shared memory are depicted in Figure 1 where the communication channels are not depicted. In both cases, we have  $n = 7$  and  $m = 3$  but the partitions are different.

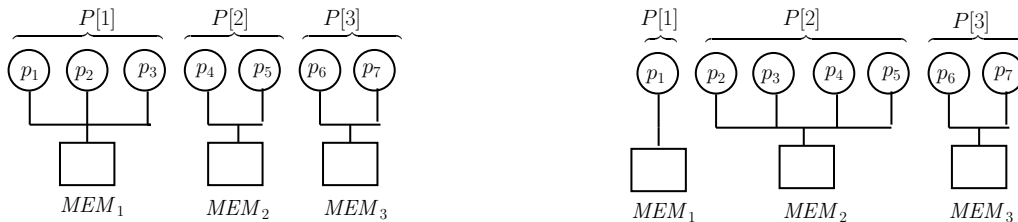


Figure 1: Two examples of partially shared memories

**Notation** As already indicated in the introduction,  $SM\_MP_{n,m}[\emptyset]$  is used to denote the previous base wait-free hybrid distributed computing model. In the following  $\emptyset$  will be replaced by a failure detector to denote the corresponding enriched model. In Figure 1 we have two instances of  $SM\_MP_{7,3}[\emptyset]$ .

**Two particular cases** The two extreme cases  $m = 1$  and  $m = n$  are particularly interesting. The case  $m = 1$  corresponds to the case where all processes share a common read/write memory. In that case, as the read/write communication model is stronger than the message-passing model, message-passing communication becomes useless and, consequently,  $SM\_MP_{n,1}[\emptyset]$  is the classical shared memory model.

When  $m = n$ , there is a single process in each partition and for each  $x$ ,  $1 \leq x \leq n$ ,  $MEM_x$  boils down to the local memory of a single process. Hence,  $SM\_MP_{n,n}[\emptyset]$  is the classical send/receive message-passing model.

## 2.2 An atomic register cannot be built in $SM\_MP_{n,m}[\emptyset]$ when $m > 1$

**Theorem 1** *Let  $1 < m \leq n$ . It is impossible to build an atomic register in  $SM\_MP_{n,m}[\emptyset]$ .*

**Proof** The proof is a simple reduction to the impossibility theorem stating that there is no wait-free implementation of a register in an asynchronous send/receive message-passing system [2, 3, 12, 13].

To that end, let us assume that there is an algorithm  $A$  that builds a register in  $SM\_MP_{n,m}[\emptyset]$  and consider its executions in  $SM\_MP_{n,m}[\emptyset]$  where, in each partition, all processes but one crash before taking any step. As  $A$  is wait-free, it follows that the  $m > 1$  remaining processes implement an atomic register in the system model  $SM\_MP_{m,m}[\emptyset]$ , i.e., in a pure message-passing system model. This contradicts the existence of algorithm  $A$  and concludes the proof.  $\square_{Theorem 1}$

## 3 A new failure detector class

### 3.1 Failure pattern and failure detector

The underlying *time model* is the set  $\mathbb{N}$  of natural 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  $\tau, \tau'$ , etc.

**Formal definitions** The notions introduced here are from [6].

A *failure pattern* is a function  $F()$  such that  $F(\tau)$  denotes the set of processes that have crashed by time  $\tau$ . As crashes are stable, we have  $\forall \tau: F(\tau) \subseteq F(\tau + 1)$ . Given a run, let  $\mathcal{F}$  be the set of processes that crash in that run (these are the faulty processes) and  $\mathcal{C}$  the set of processes that do not crash (these are the correct processes). We have  $\mathcal{F} = \cup_{\tau} F(\tau)$  and  $\mathcal{C} = \Pi \setminus \mathcal{F}$ .

A *failure detector history*  $H$  with range  $\mathcal{R}$  is a function from  $\Pi \times \mathbb{N}$  to  $\mathcal{R}$  whose meaning can be interpreted as follows:  $H(i, \tau)$  is the output of the considered failure detector at process  $p_i$  at time  $\tau$ .

A *failure detector*  $\mathcal{D}$  with range  $\mathcal{R}$  is a function that maps each failure pattern  $F()$  to a non-empty set of failure detector histories with range  $\mathcal{R}$ .  $\mathcal{D}(F)$  is the set of behaviors (possible failure detector histories) that  $\mathcal{D}$  can exhibit when the failure pattern is  $F$ .

**On the operational side** From an algorithm point of view, a failure detector can be seen as a distributed device that provides each process  $p_i$  with a read-only local variable whose value at time  $\tau$  is  $H(i, \tau)$ .

### 3.2 The failure detector class $\Sigma$

As already indicated, this failure detector class [7] is the class of the weakest failure detectors that allow an atomic register to be implemented in the base send/receive message-passing system model. Using the formalism introduced in Section 2, this means that  $SM\_MP_{n,n}[\Sigma]$  is the weakest failure detector-based system model in which an atomic register can be built.

The range of  $\Sigma$  is the set of all non-empty subsets of processes ( $2^{\Pi} \setminus \emptyset$ ). Let  $\Sigma_i$  be the read-only local variable provided to  $p_i$  by  $\Sigma$ . Such a local output is called a *quorum*. This failure detector class is defined by the two following properties where  $\Sigma_i^{\tau}$  denotes the value of  $\Sigma_i$  at time  $\tau$ , i.e.,  $\Sigma_i^{\tau} = H(i, \tau)$ .

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

The intersection property states that any two quorums taken at any times intersect. This property prevents partitioning and is used to maintain the consistency of the atomic register. The liveness property states that eventually a quorum contains only correct processes. This property is used to allow a process to stop waiting for messages from crashed processes. Because any two majorities always intersect, it is easy to see that  $\Sigma$  can be easily implemented in a message-passing system in which a majority of process never crashes.

### 3.3 The failure detector class $M\Sigma$

**Definition** This failure detector class is for the system model  $SM\_MP_{n,m}[\Sigma]$ . It consists of all the failure detectors that satisfy the following properties where the quorum  $M\Sigma_i$  is the local output at process  $p_i$  and  $M\Sigma_i^{\tau}$  its value at time  $\tau$ .

- Liveness.  $\exists \tau \forall \tau' \geq \tau \forall \mathcal{C} \in \mathcal{C} \exists x \in \mathcal{C} \exists k \in M\Sigma_i^{\tau} \subseteq \mathcal{C} \wedge (k \in M\Sigma_i^{\tau}) \wedge (\ell \in M\Sigma_j^{\tau'}) \wedge (k, \ell \in P[x])$ .

The liveness property is the same as the one of  $\Sigma$ . The intersection property is more general. It states that any pair of quorums (whose values are taken at any times) is such that each one contains a process and these two, processes share the same common memory. This can be seen as an “indirect” intersection:  $M\Sigma_i$  and  $M\Sigma_j$  are not required to intersect “directly” but must include processes that share the same memory.

**Particular cases** Let us first consider the case  $m = 1$  (the model is then the classical base read/write shared memory model). In that case, there is a single shared memory ( $MEM_1$ ) and taking always  $M\Sigma_i = \{i\}$  for each  $p_i$ , both properties are always satisfied. Hence, there is a trivial implementation of  $M\Sigma$  in  $SM\_MP_{n,1}[\emptyset]$  which means that  $M\Sigma$  adds no computational power when  $m = 1$ . This is in perfect agreement with the fact that  $SM\_MP_{n,1}[\emptyset]$  is the base read/write shared memory model in which atomic register are given for free.

Let us now consider the case  $m = n$  (the model is then the classical send/receive message-passing model). In that case, there is a single process per cluster  $x$  (e.g.,  $P[x]$  contains only  $p_x$ ). It follows that, for the intersection property to be true, we need to have  $\forall i, j \in \Pi, \forall \tau, \tau' : \exists k : (k \in M\Sigma_i^\tau) \wedge (k \in M\Sigma_j^{\tau'})$ , i.e.,  $\forall i, j, \forall \tau, \tau' : M\Sigma_i^\tau \cap M\Sigma_j^{\tau'} \neq \emptyset$ . Hence, when considering  $m = n$ ,  $M\Sigma$  boils down to  $\Sigma$ , which means that  $SM\_MP_{n,n}[M\Sigma]$  and  $SM\_MP_{n,n}[\Sigma]$  define the same computational model.

## 4 $M\Sigma$ is sufficient: building an atomic register in $SM\_MP_{n,m}[M\Sigma]$

This section presents and proves correct an algorithm that builds an 1WMR atomic register in  $SM\_MP_{n,m}[M\Sigma]$ . The writer is denoted  $p_w$ . The atomic register that is constructed is denoted  $REG$ .

### 4.1 Description of the algorithm

The algorithm, described in Figure 2, is a simple adaptation to the hybrid model of the algorithm described in [2] that builds an atomic register in a message-passing system where a majority of processes are correct. As already indicated, while the operation send is atomic, the operation broadcast is not.

This algorithm is not designed with efficiency in mind. Its aim is only to show that an atomic register can be built in  $SM\_MP_{n,m}[M\Sigma]$ , and consequently show that  $M\Sigma$  is sufficient. (Let us remember that, when  $m = 1$ , the underlying message-passing system can be easily simulated on top of the shared memory.)

**The variables implementing the atomic register  $REG$**  Let  $p_i$  be a process and  $x$  its cluster (i.e.,  $i \in P[x]$ ). Process  $p_i$  stores its “local copy” of  $REG$  in  $MEM_x[i]$ . More precisely, this base register has two fields  $MEM_x[i].val$  (which stores the last value of  $REG$  known by  $p_i$ ) and  $MEM_x[i].sn$  (which stores the corresponding sequence number).

The variables in italics with subscript  $s$  are variables that are local to process  $p_s$ . These local variables are used to generate local sequence numbers.

**The operation  $REG.write(v)$**  This operation (which can be issued only by  $p_w$ ) first associates a new sequence number ( $sn_w$ ) with its current invocation (line 01). Then, it sends the message  $WRITE(v, sn_w)$  to all the processes to inform them on the new write (line 02). When,  $M\Sigma_w$  is such that  $p_w$  has received a matching acknowledgment from each of its processes, the operation returns  $ok$  and terminates (lines 02-04).

Let  $p_i$  be a process such that  $i \in P[x]$ . When  $p_i$  ( $p_i$  can be  $p_w$ ) receives a message  $WRITE(v, seqnb)$  from a process  $p_j$  it updates  $MEM_x[i]$  if this message carries a more recent write (line 12). Moreover,  $p_i$  always sends by return an acknowledgment carrying  $seqnb$  (line 13) to inform  $p_j$  that its “local copy” of  $REG$  has now a sequence number which is  $\geq seqnb$ .

**The operation  $REG.read()$**  This operation proceeds in two phases. In the first phase (lines 05-08),  $p_i$  broadcasts a message  $READ(r\_sn_i)$  where  $r\_sn_i$  is used to identify all its read invocations (lines 05-06) and waits until  $M\Sigma_i$  contains only processes from which  $p_i$  has received a matching acknowledgment (line 07).

When a process  $p_k$  receives such a message  $READ(r\_sn)$  from a process  $p_j$ , it computes the most recent value of  $REG$  stored in the cluster shared memory  $MEM_x$ , i.e., such that  $k \in P[x]$  (lines 14-15) and sends back to  $p_j$  this most recent value (line 16).

Finally,  $p_i$  determines the most recent value of  $REG$  it has received from the processes in  $M\Sigma_i$  (line 08). That value will be returned by the read operation (line 11), but before,  $p_i$  has to execute the second phase (lines 08-10) whose aim is ensure that no overwritten value is ever returned by a read operation. To that end,  $p_i$  simulates a write of the value it is about to return.

### 4.2 Proof of the algorithm

**Theorem 2** Let  $1 \leq m \leq n$ . The algorithm described in Figure 2 is wait-free construction of a 1WMR atomic register in  $SM\_MP_{n,m}[M\Sigma]$ .

**Proof** We have to show that (a) any invocation of an operation by a correct process terminates whatever the number of process crashes (wait-freedom) and (b) all operation invocations (except possibly, for each process, its last operation if it is faulty) can be totally ordered in such a way that (when considering this total order) no operation returns an overwritten value and if the operation invocation  $inv1$  terminates before the operation invocation  $inv2$  starts then  $inv1$  appears before  $inv2$  in the total order.

```

operation REG.write(v):  % This code is only for the single writer  $p_w$  %
(01)   $sn_w \leftarrow sn_w + 1$ ;
(02)  broadcast WRITE( $v, sn_w$ );
(03)  wait until ( $M\Sigma_w$  is such that  $\forall j \in M\Sigma_w$ :  $p_w$  has received ACK( $sn_w$ ) from  $p_j$ );
(04)  return(ok).

% The code snippets that follow are for every process  $p_i$ ,  $1 \leq i \leq n$  %
% Moreover, the value  $x$  denotes  $p_i$ 's partition number i.e.,  $x$  is such that  $i \in P[x]$  %

operation REG.read() :
(05)   $r\_sn_i \leftarrow r\_sn_i + 1$ ;
(06)  broadcast READ( $r\_sn_i$ );
(07)  wait until ( $M\Sigma_i$  is such that  $\forall j \in M\Sigma_i$ :  $p_i$  has received VAL( $v, sn, r\_sn_i$ ) from  $p_j$ );
(08)   $\langle v, sn \rangle \leftarrow (\langle v, sn \rangle \mid \text{VAL}(v, sn, r\_sn_i) \text{ received} \wedge \nexists sn' > sn : \text{VAL}(-, sn', -) \text{ received})$ ;
(09)  broadcast WRITE( $v, sn$ );
(10)  wait until ( $M\Sigma_i$  is such that  $\forall j \in M\Sigma_i$ :  $p_i$  has received ACK( $sn$ ) from  $p_j$ );
(11)  return(v).

Task T1: when WRITE( $v, seqnb$ ) is received from  $p_j$ :
(12)  if ( $MEM_x[i].sn < seqnb$ ) then  $MEM_x[i] \leftarrow \langle v, seqnb \rangle$  end if;
(13)  send ACK( $seqnb$ ) to  $p_j$ .

Task T2: when READ( $r\_sn$ ) is received from  $p_j$ :
(14)   $mem \leftarrow \{MEM_x[k] \text{ such that } k \in P[x]\}$ ;
(15)   $\langle v, sn \rangle \leftarrow (mem[k] \mid \nexists \ell : mem[\ell].sn > mem[k].sn)$ ;
(16)  send VAL( $v, sn, r\_sn$ ) to  $p_j$ .

```

Figure 2: Building an atomic 1WMR register  $SM\_MP_{n,m}[M\Sigma]$

Proof of Item (a). Let  $p_i$  be a correct process that invokes an operation on *REG*. Let us observe that the only location where  $p_i$  could block forever is when it executes a **wait** statement where it is waiting for messages from the processes in  $M\Sigma_i$ . The proof follows from the following observations: (1) Each **wait** statement follows a broadcast identified by a sequence number, (2) each broadcast message is answered by all correct processes, (3) channels are reliable, and (4) eventually  $M\Sigma_i$  contains only correct processes.

Proof of Item (b). Let us observe that, as there is a single writer, and it is sequential, the invocations of *REG.write*() are totally ordered and this order is in agreement with their sequence numbers. Hence, let us initialize *S* to the sequence of all write invocations ordered according to their sequence numbers (that sequence respects their real-time occurrence order).

Considering an invocation of *REG.read*() issued by a process  $p_i$ , let  $sn$  be the sequence number of the value returned by that invocation. Each invocation of *REG.read*() is added to *S* after the write whose sequence number is  $sn$  and before (if it exists) the one with sequence number  $sn + 1$ . Moreover, if two read invocations obtains the same sequence number, the one that started first is placed in *S* before the other one. It follows from its definition that *S* is a correct register history (i.e., no read obtains an overwritten value in *S*).

It remains to show that *S* respects the real-time occurrence order on the operation invocations. As we have seen, this is already the case for the invocations of the write operation. For the invocations of *REG.read*() there are two cases to consider.

- The read invocation is issued (in real-time) after the  $\alpha$ th write has terminated. To prove that it appears in *S* after this write we show that the sequence number obtained by this read invocation is  $\geq \alpha$ .

Let  $p_i$  be the process that issued this read operation and  $M\Sigma_i^\tau$  be the quorum value that allowed it to stop waiting at line 07 of its read invocation. Let  $M\Sigma_w^{\tau'}$  be the quorum value that allowed  $p_w$  to stop waiting at line 03 of its  $\alpha$ th write invocation. As  $p_i$  started reading after  $p_w$  has terminated the  $\alpha$ th write, we have  $\tau > \tau'$ . Moreover, due to the intersection property of  $M\Sigma$ ,  $\exists x, k, \ell : (k \in M\Sigma_i^\tau) \wedge (\ell \in M\Sigma_w^{\tau'}) \wedge (k, \ell \in P[x])$ . It then follows from lines 12-13 executed by  $p_\ell$  just before it sent back ACK( $\alpha$ ) to  $p_w$  that we have  $MEM_x[\ell].sn \geq \alpha$ .

Moreover, as  $\tau > \tau'$  and  $k \in M\Sigma_i^\tau$ , it follows that when  $p_k$  (triggered by the message READ( $r\_sn$ ) from  $p_i$ ) executes lines 14-15, we have  $MEM_x[\ell].sn \geq \alpha$ . Hence,  $p_k$  sends to  $p_i$  the message VAL( $-, \beta, r\_sn$ ) where  $\beta \geq \alpha$  (line 16). Consequently, the sequence number computed by  $p_i$  at line 07 of the read operation is  $\geq \alpha$  which proves the case.

- The second case is when there are two non-concurrent read invocations such that the first one obtains the sequence number  $sn1$  and the second one obtains the sequence number  $sn2$  (these sequential read invocations can be from different processes and concurrent with write invocations). We need to show that  $sn1 \geq sn2$ .

Let  $p_j$  be the process that issued the first read invocation. The reasoning is the same as in the previous item after having replaced in that item the write invocation issued by  $p_w$  by the lines 09-10 executed by  $p_j$  during its read invocation.  $\square_{Theorem 2}$

## 5 $M\Sigma$ is necessary

### 5.1 $M\Sigma$ is the weakest failure detector for a register in a hybrid communication model

The previous section has shown that an atomic register can be built in  $SM\_MP_{n,m}[M\Sigma]$ , thereby showing that enriching  $SM\_MP_{n,m}[\emptyset]$  with  $M\Sigma$  is sufficient (from an “information on failures” point of view) when one wants to build an atomic register. This section addresses the necessity side. It shows that any failure detector  $D$  such that an atomic register can be built in  $SM\_MP_{n,m}[D]$  provides enough information on failures in order  $M\Sigma$  can be built in  $SM\_MP_{n,m}[D]$ .

Let  $D$  be any failure detector such that there is an algorithm  $A$  that allows building an atomic register in  $SM\_MP_{n,m}[D]$ . The proof of the “necessity” part consists in showing that it is possible to build a failure detector of the class  $M\Sigma$  from  $A$  executed in  $SM\_MP_{n,m}[D]$ . In the failure detector parlance, we say that it is possible to “extract”  $\Sigma$  from  $A$ . In a very interesting way, the proposed extraction algorithm is the one we have presented in [4, 13] (for  $\Sigma$ ) but its proof is different. Hence, the current paper shows that the extraction algorithm introduced in [4] has a generic dimension with respect to failure detectors.

### 5.2 Bonnet-Raynal’s extraction algorithm

This section presents Bonnet-Raynal’s extraction algorithm introduced in [4] where it is assumed that the underlying  $D$ -based algorithm  $A$  builds an atomic register in  $SM\_MP_{n,m}[D]$ . Albeit not new, this presentation is needed for completeness of the minimality proof.

**Arrays of atomic registers** Let  $Q$  be a non-empty set of processes, and  $REG_Q[1..n]$  an array of  $n$  atomic registers (initialized to  $[\perp, \dots, \perp]$ ), such that each atomic register  $REG_Q[x]$  is implemented by the  $n$ -process algorithm  $A$  executed only by  $|Q|$  threads, each one associated with a process of  $Q$ .

**A simple register-based algorithm** Let  $WR_Q$  be the following register-based algorithm (also called a task) 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**  $WR_Q$ :

$REG_Q[i].write(\top)$ ; **for each**  $x \in \{1, \dots, n\}$  **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  $REG_Q[x]$  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  $WR_Q$ .

Let us remember that  $\mathcal{C}$  is the set of identities of the processes that are correct in the considered run. 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  $A$ . As an example, let us consider the task  $WR_Q$ , and two correct processes  $p_i$  and  $p_j$  such that  $i \in Q$  and  $j \notin Q$ . Let  $th_{i,Q}$  be the thread of  $p_i$  involved in  $Q$ . As  $j \notin Q$ , the thread  $th_{j,Q}$  does not exist. The thread  $th_{i,Q}$  can block forever when it executes  $A$  to read or write a register of  $REG_Q[1..n]$  if, due to the output of  $D$  and the code of  $A$ , it is directed to wait for a message from  $th_{j,Q}$  -that does not exist-<sup>1</sup>).

**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  $2^{\Pi}$ . To that end, 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.

**The extraction algorithm** The algorithm that extracts  $\Sigma$  is described in Figure 3. Let us recall that the aim is to provide each process  $p_i$  with a local variable  $\Sigma_i$  such that the  $(\Sigma_x)_{1 \leq x \leq n}$  variables satisfy the intersection and liveness properties of  $\Sigma$ .

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 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 an element of  $quorum\_sets_i$  as the current output of  $\Sigma_i$ . 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.

<sup>1</sup>A similar blocking can happen when the processes use an underlying  $\Omega$ -based algorithm [5] and, in the considered run, the correct process that is eventually elected as a leader does not participate in the algorithm.

The main issue is to ensure the liveness property of  $\Sigma_i$  (eventually  $\Sigma_i$  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  $\Sigma_i$  is the set (quorum) of  $quorum\_sets_i$  that appears as being the “first” in  $queue_i$ . The formal definition of “first element 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  $quorum\_sets_i = \{\{3, 4, 9\}, \{2, 3, 8\}, \{1, 2, 4, 7\}\}$ , and  $queue_i = \langle 4, 8, 3, 2, 7, 5, 9, 1, \dots \rangle$ . 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  $\{1, 2, 4, 7\}$  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). The notion of *first quorum* is used to ensure the liveness of  $\Sigma$ , i.e., the set  $\Sigma_i$  of any correct process  $p_i$  eventually contains only correct processes.

```

Init:  $quorum\_sets_i \leftarrow \{\{1 \dots, n\}\}$ ;  $queue_i \leftarrow \langle 1, \dots, n \rangle$ ;
for each  $Q \in (2^{\mathbb{I}} \setminus \{\emptyset, \{1, \dots, n\}\})$  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  $\Sigma_i$ :
  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 3: Extracting  $\Sigma$  from a failure detector-based algorithm  $A$  that implements a register (code for  $p_i$ )

**Remark 1** Initially  $quorum\_sets_i$  contains the set  $\{1, \dots, n\}$ . 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  $WR_{\{1, \dots, n\}}$  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 terminate in the task  $WR_{\{1, \dots, n\}}$ . This case is directly taken into account in the initialization of  $quorum\_sets_i$  (thereby saving the execution of the task  $WR_{\{1, \dots, n\}}$ ).

**Remark 2** 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.

### 5.3 Minimality of $M\Sigma$

**Theorem 3** Let  $1 \leq m \leq n$ .  $M\Sigma$  is the weakest failure detector  $SM\_MP_{n,m}[\emptyset]$  has to be enriched with in order an atomic register can be built.

**Proof** Let  $D$  be any failure detector such that there is an algorithm  $A$  that builds an atomic register in  $SM\_MP_{n,m}[D]$ . The proof consists in showing that, given such an algorithm  $A$ , the algorithm described in Figure 3 builds a failure detector  $M\Sigma$ .

Proof of the intersection property.

The proof is by contradiction. Let us first observe that the set  $\Sigma_i$  returned to a process  $p_i$  is a set of  $quorum\_set_i$  (that contains the set  $\{1, \dots, n\}$  -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)  $Q_1, Q_2 \in \bigcup_{1 \leq j \leq n} (quorum\_set_j)$ , and (2)  $\forall x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$ . Let us notice that the first item means that  $Q_1$  and  $Q_2$  can be returned to some processes as their local value for  $\Sigma$ . The second item means that at least one of  $k$  and  $\ell$  is not in  $P[x]$ , from which we conclude that the processes in  $Q_1$  and  $Q_2$  cannot communicate via the shared memory cluster  $P[x]$ .

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 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, 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  $D$ . Since (a)  $p_i \in Q_1$  and  $p_j \in Q_2$ , and (b)  $\forall x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$ ,  $p_i$  does not write to  $REG_{Q_2}[i]$  and  $p_j$  does not write to  $REG_{Q_1}[j]$ . Thus,  $p_i$  reads  $\perp$  from  $REG_{Q_1}[j]$ , and  $p_j$  reads  $\perp$  from  $REG_{Q_2}[i]$ .

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  $REG_{Q_{12}}[1..n]$  contain 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  $\forall x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$ ) 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  $\forall x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$  is false, which completes the proof of the intersection property of  $M\Sigma$ .

Proof of the liveness property.

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

Moreover, after some finite time, each correct process  $p_i$  receives  $ALIVE(j)$  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  $C$  has been added to  $quorum\_set_i$ , the quorum  $Q$  selected by the task  $T4$  is always such that  $Q \subseteq C$ , which proves the liveness property of  $M\Sigma$ .  $\square_{Theorem 3}$

## 5.4 $M\Sigma$ is strictly weaker than $\Sigma$ when $m < n$

**Theorem 4** *Let  $m < n$ . The model  $SM\_MP_{n,m}[M\Sigma]$  is strictly weaker than the model  $SM\_MP_{n,m}[\Sigma]$ .*

### Proof

To prove the theorem we have to show that (a) it is possible to build  $M\Sigma$  in  $SM\_MP_{n,m}[\Sigma]$  and (b) it is impossible to build  $\Sigma$  in  $SM\_MP_{n,m}[M\Sigma]$  when  $m < n$ .

- Proof of Item (a). For any  $i$  and  $\tau$ , let us define  $M\Sigma_i^\tau = \Sigma_i^\tau$ . As  $\Sigma_i^\tau \cap \Sigma_j^{\tau'} \neq \emptyset$ , it follows that  $\exists k \in M\Sigma_i^\tau \cap M\Sigma_j^{\tau'}$  and there is trivially a partition  $x$  such that  $k \in P[x]$  which proves the intersection property of  $M\Sigma$ . The liveness property of  $M\Sigma$  follows directly from its  $\Sigma$  counterpart.
- Proof of Item (b). The proof is by contradiction. let us assume that there is a wait-free algorithm  $A$  that builds  $\Sigma$  in  $SM\_MP_{n,m}[M\Sigma]$  when  $m < n$ .

As  $m < n$ , there is a partition  $P[x]$  and a pair of processes  $p_i$  and  $p_j$  such that  $i, j \in P[x]$  (i.e.,  $p_i$  and  $p_j$  belong to the same memory partition  $x$ ). Let us consider a run in which  $p_i$  and  $p_j$  are correct and all the other processes crash before taking any step. As  $i, j \in P[x]$ ,  $\forall \tau, \tau', M\Sigma_i^\tau = \{i\}$  and  $M\Sigma_j^{\tau'} = \{j\}$  are correct local outputs of  $M\Sigma$  (they satisfy its intersection and liveness properties).

Let us suppose that, while it is executing  $A$ ,  $p_j$  pauses during an arbitrary long but finite period during which  $p_i$  runs solo and (due to asynchrony) receives no message from  $p_j$ . As  $\forall \tau$  we have  $M\Sigma_i^\tau = \{i\}$ ,  $p_i$  cannot distinguish this execution of  $A$  from the one in which it is the only correct process. Hence, after some finite time, because it is wait-free,  $A$  has to output  $\{i\}$  at  $p_i$  in order the liveness property of  $\Sigma$  be satisfied. Hence, there is a time  $\tau_i$  such that  $\Sigma_i^{\tau_i} = \{i\}$ .

Let us now suppose that, after time  $\tau_i$ ,  $p_i$  pauses for an arbitrary long but finite period during which  $p_j$  runs solo and (due to asynchrony) receives no message from  $p_i$ . It follows from the same reasoning as before that there is a time  $\tau_j$  at which we have  $\Sigma_j^{\tau_j} = \{j\}$ .

It follows that  $\Sigma_i^{\tau_i} \cap \Sigma_j^{\tau_j} = \emptyset$ , and the intersection property of  $\Sigma$  is violated which concludes the proof of the theorem.  $\square_{Theorem 4}$

**Remark** Let us observe from the second part previous proof (Item b) that, when the processes of all but one memory clusters crash,  $M\Sigma$  is too weak to give information on failures. Moreover, The next corollary follows from the previous theorem when we consider the case  $m = 1$  (read/write shared memory model in which  $M\Sigma$  can be trivially implemented).

**Corollary 1**  $\Sigma$  cannot be built from atomic registers only.

## 6 On the implementability of $M\Sigma$ despite asynchrony and failures

When  $m = n$  (pure asynchronous message-passing system),  $M\Sigma$  boils down to  $\Sigma$  and it is known that  $\Sigma$  can be implemented in a pure message-passing asynchronous system where a majority of processes are correct. Hence, the question: Is there a necessary and sufficient condition  $C$  on  $n$ ,  $m$  and a system parameter associated with failures such that  $M\Sigma$  can be implemented in  $SM\_MP_{n,m}[C]$  (where  $SM\_MP_{n,m}[C]$  denotes the system model  $SM\_MP_{n,m}[\emptyset]$  restricted to the runs where  $C$  is satisfied)? This section presents such a necessary and sufficient condition  $C$ .

**Notion of a faulty cluster** Let us say that a cluster  $x$  is faulty in a run if all processes of  $P[x]$  are faulty in that run. Let  $t$ ,  $1 \leq t < m$  be the upper bound on the number of faulty clusters.

The next theorem shows that  $C = (t < m/2)$  is a necessary and sufficient condition to implement  $M\Sigma$  in  $SM\_MP_{n,m}[\emptyset]$ .

**Theorem 5** Let  $COND$  be the set of all the predicates on  $n$ ,  $m$  and  $t$ ,  $C' \in COND$  and  $C = (t < m/2)$ .  $M\Sigma$  can be built in  $SM\_MP_{n,m}[C']$  if and only if  $C' \Rightarrow C$ .

**Proof** The proof of the theorem is made up of two parts: (a)  $M\Sigma$  can be built in all the runs in which  $C$  is satisfied and (b)  $M\Sigma$  cannot be built in all the runs in which  $C$  is not satisfied.

Proof of Item (a). The algorithm described in Figure 4 builds  $M\Sigma$  in  $SM\_MP_{n,m}[t < m/2]$ . Initially, each process  $p_i$  initializes  $M\Sigma_i$  to  $\Pi$  (the set of all process identities). Then, repeatedly,  $p_i$  broadcasts a message `ALIVE( $i$ )`, waits until it has received a message from  $(m - t)$  processes belonging to different clusters and sets  $M\Sigma_i$  to set of processes. It is easy to show that the intersection and liveness properties of  $M\Sigma$  are satisfied.

- Let us first observe that, due to the assumption  $t < m/2$ , no correct process remains blocked forever in the **wait** statement. Moreover, after some finite time, a correct process receives message only from correct processes. It follows directly from these two observations that, after some finite time,  $M\Sigma_i$  contains only correct processes which is the liveness property of  $M\Sigma$ .
- $\forall i, j \in \Pi$ ,  $\forall \tau, \tau'$ , let us consider the values of  $M\Sigma_i^\tau$  and  $M\Sigma_j^{\tau'}$ . It follows from  $t < m/2$  that  $M\Sigma_i^\tau$  contains processes belonging to a majority of clusters, and similarly for  $M\Sigma_j^{\tau'}$ . As any two majorities intersect, we conclude that there is cluster  $x$  such that  $k \in M\Sigma_i^\tau \wedge \ell \in M\Sigma_j^{\tau'} \wedge \{k, \ell\} \subseteq P[x]$  which proves the intersection property of  $M\Sigma$ .

```

MΣi ← Π;
repeat forever
  broadcast ALIVE(i);
  wait until (messages received from processes in (m - t) different clusters);
  MΣi ← the set of processes from which messages have been received at previous line
end repeat.

```

Figure 4: Building  $M\Sigma$  in  $SM\_MP_{n,m}[t < m/2]$  (code of  $p_i$ )

Proof of Item (b). Considering that  $t \geq m/2$ , let us partition the set of clusters in two sets  $QC_1$  and  $QC_2$  (i.e.,  $QC_1 \cap QC_2 = \emptyset$  and  $QC_1 \cup QC_2 = \cup_{1 \leq x \leq m} P[x]$ ). Due to asynchrony it is possible to delay for an arbitrary long period all the messages from the processes in  $QC_1$  to the processes in  $QC_2$  and all the messages from the processes in  $QC_2$  to the processes in  $QC_1$ . Then, the processes in  $QC_1$  cannot distinguish the case where the processes in  $QC_2$  have crashed or are only very slow and similarly for the processes of  $QC_2$  with respect to the processes of  $QC_1$ . The impossibility follows from this classical partitioning argument.  $\square_{Theorem 5}$

## 7 Conclusion

This paper has first introduced a new distributed computing model with hybrid communication. On the one side, any pair of processes can communicate by asynchronous message-passing. On the other side, processes are partitioned into clusters and, insider each cluster, processes can communicate through a read/write shared memory.

Then, the paper has investigated the minimal information on failures that allows an atomic register to be implemented in such a hybrid communication model. This minimal information on failures is captured by a new failure detector denoted  $M\Sigma$  (which generalizes the failure detector  $\Sigma$ ). The paper has also presented a necessary and sufficient condition on the number of faulty shared memory clusters that, when satisfied, allows  $M\Sigma$  to be implemented despite the net effect of asynchrony and failures.

## 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] Bonnet F. and Raynal M., A Simple Proof of the Necessity of the Failure Detector  $\Sigma$  to Implement an Atomic Register in Asynchronous Message-passing Systems. *Information Processing Letters*, 110(4): 153-157, 2010.
- [5] Chandra T., Hadzilacos V. and Toueg S.: The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.
- [6] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
- [7] Delporte-Gallet C., Fauconnier H. and Guerraoui R., Tight Failure Detection Bounds on Atomic Object Implementations. *Journal of the ACM*, 57(4), Article 22, 32 pages, 2010.
- [8] Gifford D.K., Weighted Voting for Replicated Data. *Proc. 7th ACM Symposium on Operating System Principles (SOSP'79)*, ACM Press, pp. 150-172, 1979.
- [9] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.
- [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.
- [13] Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. *Morgan & Claypool Publishers*, 251 pages, 2010 (ISBN 978-1-60845-293-4).