Chapter 7

Circumventing the $t < n/2$ Read/Write Register Impossibility: the Failure Detector Approach

This chapter presents the failure detector class (denoted Σ) that allows us to circumvent the impossibility of building an atomic read/write register in an asynchronous message-passing system in which half or more processes may commit crash failures (system model $CAMP_{n,t}[t \geq n/2]$). (The reader is referred to Section 3.3 for formal definitions related to failure detectors.) This chapter first introduces the class Σ, and shows how it allows us to implement an atomic register for any value of t. Then, it shows that Σ is the failure detector class that provides us with the weakest information on failures that allows an atomic read/write register to be built despite asynchrony and any number of process crashes. Finally, the chapter compares the failure detectors classes Σ and Θ on the one side, and Σ and the URB-broadcast communication abstraction on another side (Θ, introduced in Section 3.4, is the weakest failure detector class that allows URB-broadcast to be built on top of fair channels in the presence of any number of process crashes).

Keywords Asynchronous system, Atomic register, Extraction algorithm, Impossibility, Process crash failure, Quorum failure detector Σ, Uniform reliable broadcast, Weakest failure detector.

7.1 The Class Σ of Quorum Failure Detectors

7.1.1 Definition of the Class of Quorum Failure Detectors

A quorum is a non-empty set of processes. (The majority sets of processes used in the algorithms of the previous chapter are sometimes called *majority* quorums.)

The class of *quorum failure detectors*, denoted Σ, was introduced by C. Delporte, H. Fauconnier, and R. Guerraoui (2004 and 2010). It contains all the failure detectors that provide each process p_i with a quorum local variable, denoted $sigma_i$, which p_i can only read, and such that the set of local variables $\{sigma_i\}_{1 \leq n}$ collectively satisfy the intersection and liveness properties stated below. Let us remember that F denotes the failure pattern associated with a given execution, and $Correct(F)$ is the set of processes that do not crash in this failure pattern.

Let us denote $sigma_i^\tau$ the output of Σ at process p_i at time τ (using the formalism introduced in the previous section we have $sigma_i^\tau = H(p_i, \tau)$).

- Intersection. $\forall i, j \in \{1, \ldots, n\}$: $\forall \tau, \tau' \in \mathbb{N}$: $sigma_i^\tau \cap sigma_j^{\tau'} \neq \emptyset$.
- Liveness. $\exists \tau \in \mathbb{N}$: $\forall \tau' \geq \tau$: $\forall i \in Correct(F)$: $sigma_i^{\tau'} \subseteq Correct(F)$.

M. Raynal, *Fault-Tolerant Message-Passing Distributed Systems*,
https://doi.org/10.1007/978-3-319-94141-7_7

The intersection property states that any two quorum values intersect, whatever the times at which they are output. As it has to always be satisfied, this property in called a *perpetual* property: it is an invariant provided by Σ. A Σ-based algorithm that aims to build an atomic register will rely on this invariant to prevent partitioning (and consequently prevent the bad scenario described in the proof of Theorem 18 from occurring), thereby guaranteeing the required atomicity (safety) property of a register.

The second property states that, after some finite time, the quorum values output at any non-faulty process contain only non-faulty processes. These processes are not required to be the same forever. They can change as long as the intersection property remains satisfied. This property is called an *eventual* property: it states that, after some finite time, "something" has to be forever satisfied. Its aim is to allow a Σ-based algorithm to guarantee that the read and write operations issued by the non-faulty processes always terminate.

7.1.2 Implementing a Failure Detector Σ When $t < n/2$

There is a very simple algorithm that builds a failure detector of the class Σ in $CAMP_{n,t}[t < n/2]$ (Fig. 7.1). Each process p_i manages a queue (denoted $queue_i$) that contains the n process identities. The initial value is any permutation of these identities. Each process broadcasts forever (i.e., until it crashes, if it ever crashes) ALIVE () messages to indicate it has not crashed. When a process p_i receives such a message from a process p_j, it moves j in $queue_i$ from its current position to the head of $queue_i$. Finally, it defines the current value of $sigma_i$ as the majority of the processes that are at the head of $queue_i$.

```
background task: repeat forever broadcast ALIVE () end repeat.

when ALIVE () is received from $p_j$ ($j \in \{1, \ldots, n\}$:
    suppress $j$ from $queue_i$; add $j$ at the head of $queue_i$;
    $sigma_i \leftarrow$ the $\lceil \frac{n+1}{2} \rceil$ processes at the head of $queue_i$.
```

Figure 7.1: Building a failure detector of the class Σ in $CAMP_{n,t}[t < n/2]$

The intersection property trivially follows from the fact that any two majorities intersect. As far as the liveness property is concerned, let c be the number of correct processes. We have $c > n/2$, i.e., $c \geq \lceil \frac{n+1}{2} \rceil$. Let us observe that, after some time, only the c non-faulty processes send messages, and consequently, only these processes will appear in the first c positions of the queue of any non-faulty process. The liveness follows immediately from $c \geq \lceil \frac{n+1}{2} \rceil$.

Remark As we have seen, it is possible to build an atomic register in $CAMP_{n,t}[t < n/2]$, and as we are about to see, it is also possible to build an atomic register in $CAMP_{n,t}[\Sigma]$. Hence, it is not counter-intuitive that a failure detector of the class Σ can be built in $CAMP_{n,t}[t < n/2]$. Let us also observe that this algorithm is the same as the one presented in Fig. 3.2, which builds a failure detector of the class Θ in $CAMP_{n,t}$[- FC; $t < n/2$] (a weaker system model than $CAMP_{n,t}[t < n/2]$).

However, thanks to Theorem 18, and the fact that Σ allows the construction of an atomic register for any value of t, we can conclude that it is not possible to build a failure detector of the class Σ in $CAMP_{n,t}[\emptyset]$. Such a construction requires additional assumptions that the underlying system has to satisfy. Hence, Σ is more powerful than the assumption "$t < n/2$".

The fundamental added value supplied by a failure detector, is that it provides us with the weakest information on failures the processes have to be provided with in order to build an atomic register. The model assumption "$t < n/2$" does not characterize the weakest information on failures that allows the construction of an atomic register.

7.1.3 A Σ-based Construction of an SWSR Atomic Register

This section presents a Σ-based algorithm that builds an SWSR atomic register REG (i.e., it builds a register in the system model $CAMP_{n,t}[\Sigma]$). The algorithm appears in Fig. 7.2. Extending this algorithm to build an MWMR atomic register is straightforward. It can be easily done using an incremental construction similar to the one described in the previous chapter.

One writer, one reader, but all the processes must participate The writer is denoted p_w, while the reader is denoted p_r. It is important to notice that all the processes have to participate in the algorithm. This is because the output domain of Σ is the set of the identities of all the processes, p_1, ..., p_n, and both $sigma_w$ and $sigma_r$ can a priori contain the identities of any subset of p_1, ..., p_n. The progress of p_w depends on the values returned by $sigma_w$, and, similarly, the progress of p_r depends on the values returned by $sigma_r$, which are not known in advance. Hence, to cope with any subset of faulty processes, each process must participate in the construction of the atomic register REG. Each process p_i has consequently to manage a local copy reg_i of REG, and a local variable wsn_i, as in the register algorithms of the previous chapter.

```
operation $REG$.write ($v$) is      % This code is for the single writer $p_w$ %
(1)  $wsn_w \leftarrow wsn_w + 1$;
(2)  broadcast WRITE ($v$, $wsn_w$);
(3)  wait ($sigma_i$ is such that $\forall p_j \in sigma_i$ : ACK_WRITE ($wsn_w$) received from $p_j$);
(4)  return().

operation $REG$.read () is      % This code is for the single reader $p_r$ %
(5)  $reqsn_i \leftarrow reqsn_i + 1$;
(6)  broadcast READ_REQ ($reqsn_i$);
(7)  wait ($sigma_i$ is such that $\forall p_j \in sigma_i$ : ACK_READ_REQ ($wsn_w, -, -$) received from $p_j$);
(8)  let $msn$ be greatest sequence number received in an ACK_READ_REQ ($reqsn_i, -, -$) message;
(9)  if ($msn > wsn_i$) then $reg_i \leftarrow v$; $wsn_i \leftarrow msn$ end if;
(10) return ($reg_i$).

% The code snippets that follow are for every process $p_i$, $i \in \{1, \dots, n\}$.

when WRITE ($val, wsn$) is received from $p_w$ do
(11) if ($wsn \geq wsn_i$) then $reg_i \leftarrow val$; $wsn_i \leftarrow wsn$ end if;
(12) send ACK_WRITE ($wsn$) to $p_w$.

when READ_REQ ($rsn$) is received from $p_r$ do
(13) send ACK_READ_REQ ($rsn$, $wsn_i$, $reg_i$) to $p_r$.
```

Figure 7.2: An algorithm for an atomic SWSR register in $CAMP_{n,t}[\Sigma]$

The algorithm The code of the algorithm is very close to that of the algorithms in the previous chapter. The local variables have the same meaning, and the basic structure is also the same. There are only two differences:

- The first is the use of a quorum failure detector of the class Σ instead of the majority of non-faulty processes assumption. Let us observe that the value of the quorum failure detector module $sigma_i$ can change forever (lines 3 and 7). A process p_i waits until there is a set output by the local failure detector module such that it has received an appropriate message (ACK_WRITE or ACK_READ_REQ) from each process of this set.
- The second difference is not related to the use of Σ, but to the fact that there is a single reader. As p_r is the only reader, when it invokes REG.read(), it is not necessary for it to execute the second phase of the REG.read() operation (the write phase), whose aim was to ensure that the

value kept in the local memories of the other processes is at least as recent as the value it is about to return. As no other process is allowed to read, it is sufficient that p_r keeps a local copy of the value it is about to return, in order to prevent new/old inversions. So, the second phase of a read operation required to guarantee atomicity is now simply a local write (that actually depends on the sequence number of the returned value).

The proof is a simplified version of the proof of the algorithm described in Fig. 6.5 of the previous chapter, where the majority of correct processes assumption is replaced by the properties of Σ. It is left to the reader as an exercise.

7.2 Σ Is the Weakest Failure Detector to Build an Atomic Register

7.2.1 What Does "Weakest Failure Detector Class" Mean

Notion of extraction algorithm The previous section has shown that it is possible to build an atomic register in $CAMP_{n,t}[\Sigma]$, i.e. Σ is sufficient to implement an atomic register in an asynchronous system prone to any number of process crashes. This section shows that, as soon as we rely on information on failures when we want to build a register, Σ is also necessary.

Let D be a failure detector class such that it is possible to build a register in $CAMP_{n,t}[D]$. Intuitively, "necessary" means that the information on failures provided by D "includes" information on failures provided by Σ. More precisely, let D be any failure detector class such that it is possible to build an atomic register in $CAMP_{n,t}[D]$, and A be any algorithm that builds a register in $CAMP_{n,t}[D]$. Proving the necessity of Σ to build an atomic register consists in designing an algorithm that, given the previous D-based algorithm A as an input, builds a failure detector of the class Σ. We say that this algorithm *extracts* Σ from the D-based algorithm A (see Fig. 7.3).

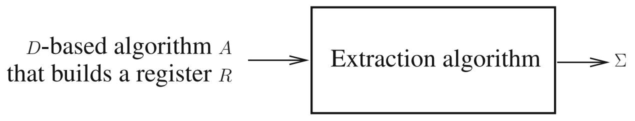

Figure 7.3: Extracting Σ from a register D-based algorithm A

Remark It is important to understand that the notion of *weakest* used here is related to information on failures only. Nothing prevents us from designing an oracle that does not provide processes with hints on failures but with another type of information (e.g., about the synchrony of the system) that would allow the construction of an atomic register despite any number of process crashes. "Weakest" means that any oracle that (1) provides processes only with information on failures (i.e., any failure detector class), and (2) allows processes to build an atomic register, allows the construction of a failure detector of class Σ.

7.2.2 The Extraction Algorithm

Aim As previously indicated, 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 an n-process D-based algorithm that implements an atomic register in an n-process asynchronous message-passing system prone to any number of crashes.

The following extraction algorithm is due to F. Bonnet and M. Raynal (2010). It has the property to be a bounded construction (every local variable or message content is bounded).

An array 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 $[\bot, \ldots, \bot]$) such that each atomic register $REG_Q[x]$ is implemented by the n-process algorithm A executed only by $|Q|$ threads, each associated with a process of Q.

A simple register-based algorithm (task) Let WR_Q be the register-based algorithm (called a task) where each process p_i, such that $i \in Q$, executes the following statements (where $reg_i[1..n]$ is an array local to p_i):

$REG_Q[i]$.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 note 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 (history) of WR_Q is denoted E_Q. In that run, no process outside Q sends or receives messages related to the task WR_Q. 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, we could say that, in WR_Q, the processes of Q "omit" sending messages to the processes that are not in Q.)

Let $\mathcal{C}$ denote the set of non-faulty processes in the run we consider. 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 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 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 such that $Q \in 2^\Pi$. To this end, each process p_i manages 2^{n-1} threads, one for each subset Q such that $i \in Q$. Let us note that the crash of a process p_i entails the crash of all its threads.

An extraction algorithm The algorithm that extracts Σ is described in Figure 7.4. Let us recall that its aim is to provide each process p_i with a local variable $sigma_i$ such that the $(sigma_x)_{1 \le x \le n}$ variables satisfy the intersection and liveness properties defined in Section 7.1.

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 has terminated WR_Q (task $T1$), while $queue_i$ is managed in such a way that eventually any correct process appears in it before any faulty process (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.

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 "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 $queue_i =< 4,8,3,2,7,5,9,1,\cdots >$. 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 $\{1, 2, 4, 7\}$ includes an element (e.g., 9 and 7, respectively) that appears in $queue_i$ after the elements of S. (If several sets are "first", any of them can be selected). The notion of "first quorum in $queue_i$" is used to ensure that Σ_i eventually includes only correct processes.

```
Init: $quorum\_sets_i \leftarrow \{\{1,\ldots,n\}\}$; $queue_i \leftarrow \langle 1,\ldots,n\rangle$;
      for each $Q \in \big(2^\Pi \setminus \{\emptyset, \{1,\ldots,n\}\}\big)$ 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 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 7.4: Extracting Σ from a failure detector-based register algorithm A (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 processes participate. This is because, as the underlying failure detector-based algorithm A (which implements a register) is correct, it follows that each correct process decides in 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 $WR_{\{1,\ldots,n\}}$).

7.2.3 Correctness of the Extraction Algorithm

Let us recall that a *bounded* construction is an algorithm in which all variables and all messages have a bounded size.

Theorem 28. *Let A be any failure detector-based algorithm that implements an atomic register in the system model $CAMP_{n,t}[\emptyset]$. Given A, the algorithm described in Fig. 7.4 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 $sigma_i$ returned to a process p_i is a set of $quorum\_set_i$ (which contains the set $\{1, \ldots, n\}$ – its initial value – plus all the sets Q such that p_i has terminated 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, messages sent by processes in Q_1 to processes in 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 $quorum\_set_j$. Let us similarly delay messages sent by processes in Q_2 to processes in Q_1 when they execute A for each register of the array REG_{Q_2}.

Let us observe that, in concurrent runs E_{Q_1} and E_{Q_2}, algorithm A, which is executed only by (1) processes of Q_1 in E_{Q_1} to build registers $REG_{Q_1}[1..n]$, and (2) processes of Q_2 in E_{Q_2} to build 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) messages from Q_1 to Q_2 and from Q_2 to Q_1 are delayed, 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 $Q_{12} = Q_1 \cup Q_2$, which is a simple merge of E_{Q_1} and E_{Q_2} defined as follows. In this run, algorithm A (which 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, 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 hand, we have the following. As process p_i receives the same messages and the same failure detector outputs in $E_{Q_{12}}$ as in E_{Q_1}, 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 hand, we have the following. In $E_{Q_{12}}$, 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 (namely, the 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 (the corollary 2 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 by assumption, each correct process p_i terminates its $REG_{\mathcal{C}}[i]$.write($\top$) and $REG_{\mathcal{C}}[x]$.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 ALIVE(j) messages only from correct processes. This means that, at each correct process p_i, every correct process eventually precedes every faulty process in $queue_i$. Due to the definition of "first set of $quorum\_set_i$ with respect to $queue_i$" stated in task $T4$, it follows that, from the time at which $\mathcal{C}$ 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 $sigma_i$.

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. $\Box_{Theorem\ 28}$

An additional property 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}; hence, the following corollary.

Corollary 2. *Let two sets Q_1 and Q_2 be 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).*

7.3 Comparing the Failure Detectors Classes Θ and Σ

The failure detector class Θ provides us with the weakest information on failures needed to implement the URB-broadcast abstraction in $CAMP_{n,t}$[- FC, $t \geq n/2$] (see Section 3.4.1). Let us remember that the output of such a failure detector at a process p_i is a set of processes, denoted $trusted_i$, that always contains a non-faulty process, though not necessarily always the same non-faulty process (accuracy), and eventually contains only correct processes (liveness).

We have also seen in Section 7.1.2 that both Θ and Σ can be implemented in $CAMP_{n,t}[t < n/2]$. Which raises the question: Do Θ and Σ have the same computational power, is one stronger than the other, or are they incomparable? The theorem that follows answers this question.

Theorem 29. *In any system where $t \geq n/2$, Σ is strictly stronger than Θ (i.e., Θ can be built in $CAMP_{n,t}[\Sigma]$, while Σ cannot be built in $CAMP_{n,t}[\Theta]$).*

Proof Let us first observe that it follows from their definitions that Σ is at least as strong as Θ. This comes from the following two observations. First, their liveness properties are the same. Second, the combination of the intersection and liveness properties of Σ implies that any set $sigma_i$ contains a correct process, which is the accuracy property of Θ (let us observe that this is independent of the value of t).

The rest of the proof shows that, when $t \geq n/2$, the converse is not true, from which it follows that Σ is strictly stronger than Θ in systems where $t \geq n/2$.

The proof is by contradiction. Let us assume that there is an algorithm A that, accessing any failure detector of the class Θ, builds a failure detector of the class Σ. Let us partition the processes into two subsets $P1$ and $P2$ (i.e., $P1 \cap P2 = \emptyset$ and $P1 \cup P2 = \{p_1, \ldots, p_n\}$) such that $|P1| = \lceil n/2 \rceil$ and $|P2| = \lfloor n/2 \rfloor$.

Let FD be a failure detector such that, in any failure pattern in which at least one process $p_x \in P1$ (resp., $p_y \in P2$) is non-faulty, outputs p_x (resp. p_y) at all the processes of $P1$ (resp., $P2$). Moreover, in the failure patterns in which all the processes of $P1$ (resp., $P2$) are faulty, FD outputs the same non-faulty process $\in P2$ (resp., $P1$) at all the processes.

It is easy to see that FD belongs to the class Θ: no faulty process is ever output (hence we have the liveness property), and at least one non-faulty process is always output at any non-faulty process (hence we have the accuracy property).

Let us consider a failure pattern F where some process $p_x \in P1$ is non-faulty, and FD outputs $trusted_x = \{x\}$, and some process $p_y \in P2$ is non-faulty, and FD outputs $trusted_y = \{y\}$. The process p_x cannot distinguish the failure pattern F from the failure pattern in which all the processes of $P2$ are faulty. Similarly, p_y cannot distinguish the failure pattern F from the failure pattern in which all the processes of $P1$ are faulty. It follows from these observations and the fact that $trusted_x \cap trusted_y = \emptyset$, that the intersection of Σ cannot be ensured, which concludes the proof of the theorem.
$\square_{Theorem\ 29}$

The previous theorem actually shows that Σ is Θ enriched with the property that any two sets output by Θ have a non-empty intersection.

7.4 Atomic Register Abstraction vs URB-broadcast Abstraction

7.4.1 From Atomic Registers to URB-broadcast

The URB-broadcast communication abstraction has been defined in Section 2.1.2. This section presents a direct construction of this communication abstraction in any system where the atomic register abstraction can be built. (This construction corresponds to the bottom left-to-right arrow in Fig. 7.6.)

The construction uses an array of SWMR atomic registers $REG[1..n]$ such that $REG[i]$ can be read by any process but written only by p_i. Moreover, each process p_i manages a local variable denoted $sent_i$ and a local array $reg_i[1..n]$. Each atomic register $REG[x]$, and each local variable

```
operation URB_broadcast ($m$) is
(1) $sent_i \leftarrow sent_i \oplus m$; $REG[i]$.write($sent_i$).

background task $T$ is
(2) repeat forever
(3)     for each $j \in \{1, \dots, n\}$ do
(4)         $reg_i[j] \leftarrow REG[j]$.read();
(5)         for each $m \in reg_i[j]$ not yet urb-delivered do URB_deliver ($m$) end for
(6) end repeat.
```

Figure 7.5: From atomic registers to URB-broadcast (code for p_i)

$sent_x$ or $reg_i[x]$ contains a sequence of messages. Each is initialized to the empty sequence; $\oplus$ denotes message concatenation.

To urb-broadcast a message m a process p_i appends m to the local sequence $sent_i$ and writes its new value into $REG[i]$ (line 1). The urb-deliveries occur in a background task T. This task is an infinite loop that reads all the atomic registers $REG[j]$ (line 4), and urb-delivers all the messages they contain exactly once (line 5).

Theorem 30. *The algorithm described in Fig. 7.5 constructs an* URB-broadcast *communication abstraction in any system in which atomic registers can be built.*

Proof As the algorithm does not forge new messages, the validity property of URB-broadcast is trivial. Similarly, it follows directly from the text of the algorithm that a message is urb-delivered at most once; hence, the integrity property of URB-broadcast.

For the termination property of URB-broadcast, let us observe that a non-faulty process p_i that urb-broadcasts a message m adds this message to the sequence of messages contained in $REG[i]$. Then, when p_i executes the background task T, it reads $REG[i]$, and consequently $reg_i[i]$ contains m. According to the text of the algorithm, p_i eventually urb-delivers m.

The previous observation has shown that, if a non-faulty process urb-broadcasts a message m, it eventually urb-delivers it. It remains to show that, if any process urb-delivers a message m, then every non-faulty process urb-delivers m. So, let us assume that a (faulty or non-faulty) process p_x urb-delivers a message m. It follows that p_x has read m from an atomic register $REG[j]$. Due to the atomicity property of $REG[j]$, (1) the process p_j has executed a $REG[j]$.write($sent_j$) operation such that $sent_j$ contains m, and (2) each $REG[j]$.read() operation issued after this write operation obtains a sequence that contains m. As any non-faulty process p_y reads the atomic registers infinitely often, it will obtain infinitely often m from $REG[j]$.read(), and will urb-deliver it, which concludes the proof of the theorem. $\square_{Theorem\ 30}$

7.4.2 Atomic Registers Are Strictly Stronger than URB-broadcast

An immediate consequence of Theorem 29 is that, whatever the value of $t \geq n/2$, Θ can be built in $CAMP_{n,t}[\Sigma]$ and $CAMP_{n,t}[$- FC; $\Sigma]$, while a failure detector Σ can be built neither in $CAMP_{n,t}[\Theta]$ nor in $CAMP_{n,t}[$- FC; $\Theta]$.

On the one hand, as we have seen, Σ is the weakest failure detector class that needs to be added to $CAMP_{n,t}[\emptyset]$ in order to build an atomic register whatever the value of $t \in \{1, ...n-1\}$. On another hand, Θ is the weakest failure detector class that allows the construction of the URB-broadcast communication abstraction in this type of system.

This means that, when looking from a *failure detector class point of view*, as the atomic register abstraction requires a stronger failure detector class than the one required by URB-broadcast, it is a problem strictly stronger than the URB-broadcast abstraction. This is depicted in Fig. 7.6 where an arrow from X to Y means that Y can be built on top of X.

$$\begin{tikzcd}
\Sigma \arrow[r, "\text{Theorem 29}"'] \arrow[d] & \Theta \arrow[d] \\
\text{Atomic register} \arrow[r, "\text{Construction of Fig. 7.5}"] & \text{URB abstraction}
\end{tikzcd}$$

Figure 7.6: From the failure detector class Σ to the URB abstraction ($1 \leq t < n$)

7.5 Summary

This chapter introduced the failure detector class Σ, and showed that Σ allows an atomic register to be implemented in an asynchronous message-passing system prone to any number of process crashes. It also proved that, when one wants to build a register this context enriched with the computability power provided by an oracle giving information on failures, Σ is the weakest such oracle required. The chapter has also shown that, from an information on failures point of view, the construction of an atomic read/write register is a stronger problem than the implementation of the URB-broadcast communication abstraction.

7.6 Bibliographic Notes

- Quorums were introduced by D. Gifford in [187] in the context of duplicated data management. General methods to define quorums can be found in [290, 345]. Quorums suited to Byzantine failures (which are more severe than crash failures) have been introduced in [277].
- Relations between quorums and voting systems are investigated in [23, 52, 187].
- The notion of a failure detector was introduced by T. Chandra and S. Toueg in [102].
- Pedagogic presentations of the failure detector concept can be found in [195, 306, 365, 369].
- Weakest failure detectors to solve several fundamental distributed computing problems (such as consensus, non-blocking atomic commitment, and quittable consensus) are presented in [123].
- It is shown in [242] that any non-trivial distributed computing problem has a weakest failure detector.
- The class Σ of quorum failure detectors was introduced by C. Delporte, H. Fauconnier, and R. Guerraoui [122, 123].
- The first proof that shows that Σ is the weakest class of failure detectors to build a register despite asynchrony and any number of process crashes was given by C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui [122]. The proof presented in this chapter is due to F. Bonnet and M. Raynal [73].
- A general method to extract quorum failure detectors is presented in [63].
- An extension of the class Σ, where the intersection property is no longer required to be perpetual, is presented in [169].
- The weakest failure detector to build an atomic register in a hybrid system was introduced in [234].

7.7 Exercise and Problem

1. Prove that the algorithm described in Fig. 7.2 is correct.
2. Construction of an atomic register in a hybrid communication model.

Hybrid communication model Let us consider the following hybrid distributed computing model $CAMP_{n,t}[\emptyset]$, where the n processes are partitioned into m, $1 \leq m \leq n$, non-empty subsets $P[1], \ldots, P[m]$ 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 at least one atomic SWMR (single-writer/multi-reader) register per process p_i belonging to $P[x]$. For notational convenience, we use an array notation for every register of MEM_x: if $i \in P[x]$, $MEM_x[i]$ can only be written 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).

Initially, each process knows the indexes of the processes that are in its partition. They do not know the composition of the other clusters.
Two examples of partially shared memory are depicted in Fig. 7.7 where the communication channels are not depicted. In both cases we have $n = 7$ and $m = 3$ but the partitions are different.

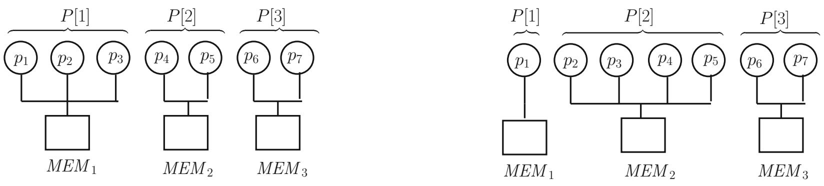

Figure 7.7: Two examples of the hybrid communication model

The Failure Detector Class $M\Sigma$ This class of failure detectors consists of all the failure detectors that satisfy the following properties where the quorum $msigma_i$ is the local output at process p_i and $msigma_i^\tau$ its value at time τ:

- Intersection. $\forall\, i, j \in \Pi,\ \forall\, \tau, \tau'$:
  $\exists x, k, \ell :\ (x \in [1..m]) \wedge (k \in msigma_i^\tau) \wedge (\ell \in msigma_j^{\tau'}) \wedge (k, \ell \in P[x])$.
- Liveness. $\exists\, \tau :\ \forall\, \tau' \geq \tau :\ \forall\, i \in Correct(F) :\ msigma_i^\tau \subseteq Correct(F)$.

The liveness property is the same as the one of Σ. 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: $msigma_i$ and $msigma_j$ are not required to intersect "directly" but must include processes that share the same memory.

What has to be done

- Implement an atomic SWMR read/write register in the previous hybrid communication model, enriched with a failure detector of the class $M\Sigma$.
- Show that $M\Sigma$ is the weakest failure detector class to build an atomic SWMR read/write register in the previous hybrid communication model.

Solution in [234].