ACM SIGACT News Distributed Computing Column 17

Sergio Rajsbaum*

Abstract

The Distributed Computing Column covers the theory of systems that are composed of a number of interacting computing elements. These include problems of communication and networking, databases, distributed shared memory, multiprocessor architectures, operating systems, verification, Internet, and the Web.

This issue consists of:

- "A Short Introduction to Failure Detectors for Asynchronous Distributed Systems," an introductory survey by Michel Raynal for readers who want to quickly understand the aim, the basic principles, the power and limitations of the failure detector concept.

Many thanks to Michel for his contribution to this issue.

Request for Collaborations: Please send me any suggestions for material I should be including in this column, including news and communications, open problems, and authors willing to write a guest column or to review an event related to theory of distributed computing.

A Short Introduction to Failure Detectors for Asynchronous Distributed Systems

Michel Raynal [1]

Abstract

Since the first version of Chandra and Toueg's seminal paper titled "*Unreliable failure detectors for reliable distributed systems*" in 1991, the failure detector concept has been extensively studied and investigated. This is not at all surprising as failure detection is pervasive in the design, the analysis and the implementation of a lot of fault-tolerant distributed algorithms that constitute the core of distributed system middleware.

The literature on this topic is mostly technical and appears mainly in theoretically inclined journals and conferences. The aim of this paper is to offer an introductory survey to the *failure detector* concept for readers who are not familiar with it and want to quickly understand its aim, its basic principles, its power and limitations. To attain this goal, the paper first describes the motivations that underlie the concept, and then surveys several distributed computing problems showing how they can be solved with the help of an appropriate failure detector. So, this short paper presents motivations, concepts, problems, definitions, and algorithms. It does not contain proofs. It is aimed at people who want to understand basics of failure detectors.

*Instituto de Matemáticas, UNAM. Ciudad Universitaria, Mexico City, D.F. 04510 `rajsbaum@math.unam.mx`.

[1] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, `raynal@irisa.fr`.

1 Why Failure Detectors?

To stop waiting or not to stop waiting? that is the question! Asynchronous distributed systems are characterized by the fact there is no bound on the time it takes for a process to execute a computation step, or for a message to go from its sender to its receiver. This is why these systems are usually called "*time-free*" systems. The major part of the software that addresses non-realtime problems implicitly considers a time-free underlying system. This has several advantages. The main one is "generality": as it is does not require that the underlying system satisfies specific timing assumptions, the software can be safely executed on any system. Moreover, the understanding and the correctness proof are usually easier as they don't rest on particular timing assumptions.

Unfortunately, the previous advantage can become useless as soon as there are failures. Assuming there is a process per node (processor), let us consider the case where a node can crash. The problem is then for a process p to know if another process q has or has not crashed. The bad news is that the combination of crashes and asynchrony creates a context where p has no safe means to know whether q has or has not crashed. If, thinking q has crashed, p stops waiting from q after some time, it can be wrong as maybe q has not crashed and the message from q to p is only very slow. If, after it stops waiting from q and before it gets q's message, p takes an irrevocable decision (motivated by the fact that it thinks that q has crashed), this decision is wrong (and the safety property of the upper layer application can consequently be violated[2]). On the other side, let us assume that q has crashed. To prevent the bad previous scenario from occurring, p must wait until it gets q's message. It is easy to see that p will wait forever, and the liveness property of the application will never be satisfied. This is one of the main problems we are faced with when designing fault-tolerant distributed algorithms in asynchronous systems prone to failures [56].

Do the same as ancient Greeks did: Ask an oracle! To solve the previous dilemma, Chandra and Toueg have introduced and investigated the notion of failure detectors [13]. A failure detector can be seen as a distributed oracle related to the detection of failures[3]. Such oracles do not change the pattern of failures that affect the execution in which they are used. Their essential characteristic is related to the guess they provide about failures. As defined by Chandra and Toueg [13], a failure detector class is basically defined by two properties, namely, a completeness property and an accuracy property. *Completeness* is on the actual detection of failures, while *accuracy* restricts the mistakes a failure detector can make.

Why use failure detectors? There are several good reasons for using a failure detector. One lies in the design approach it favors. More precisely, a failure detector is not defined in terms of a particular implementation (involving network topology, message delays, local clocks, etc.) but in terms of abstract properties (related to the detection of failures) that allow problems to be solved despite process crashes. Thus, the failure detector approach allows a modular decomposition that not only simplifies protocol design but also provides general solutions. More specifically, during a first step, a protocol is designed and proved correct assuming only the properties provided by a failure detector class. So, this protocol is not expressed in terms of low-level parameters, but depends only on a well defined set of abstract properties. The implementation of a failure detector

[2]A problem can be defined with a *safety* and a *liveness* property. The safety property stipulates that "nothing bad ever happens", while the liveness property stipulates that "something good eventually happens" [37].

[3]Let us notice that the *oracle* notion has first been introduced in language theory. An oracle is a *language* whose words can be recognized in one step from a particular state of a Turing machine [33]. The main characteristic of such oracles is to hide in a single "observed" step a sequence of computation steps or the use of an uncomputable function. They have been used to provide hierarchies of problems with respect to complexity or computability.

FD of the assumed class can then be addressed independently: additional assumptions can be investigated and the ones that are sufficient to implement FD can be added to the underlying distributed system in order to get an augmented system on top of which FD can be implemented. In that way, FD can be implemented in one way in some context and in another way in another context, according to the particular features of the underlying system. It follows that this layered approach favors the design, the proof and the portability of protocols.

Another important advantage of failure detectors lies in the approach they promote to address problems that are impossible to solve in time-free asynchronous distributed systems prone to failures. One of the most famous of them is the consensus problem which cannot be solved in asynchronous systems as soon as a process can crash [23][4]. When faced with a problem *Pb* that cannot be solved in an asynchronous system prone to failures, a natural and fundamental question that comes to mind is:

> *Which is the weakest failure detector $FD_{min}(Pb)$ the underlying asynchronous system has to be equipped with in order for the problem Pb to be solved?"*

Answering this question is important from both a practical and a theoretical point of view. From a theoretical point of view, the properties defining such a weakest failure detector state the necessary and sufficient conditions under which the problem *Pb* can be solved. This means that $FD_{min}(Pb)$ defines the borderline beyond which *Pb* cannot be solved. More precisely, *Pb* can be solved only in asynchronous systems enriched with additional mechanisms able to implement $FD_{min}(Pb)$. This has an immediate practical consequence: to solve *Pb* we need a system where the $FD_{min}(Pb)$ properties can be implemented[5]. Of course, if the system we are provided with satisfies stronger properties, *Pb* can be solved. This means that when a (provably correct) protocol does not work in a system, it is only because the properties assumed by this protocol are not satisfied by the underlying system. (Let us notice that the same argument applies to hierarchies of failure modes [53].)

Failure detectors further a deeper understanding of distributed computing problems in presence of failures, in the sense that they allow us to know whether a problem Pb_1 is "more difficult" to solve than a problem Pb_2 or whether Pb_1 and Pb_2 require different kinds of assumptions. Pb_1 is said to be more difficult to solve than Pb_2 if it requires that the underlying system satisfies additional assumptions not necessary for solving Pb_2, that is, formally, $FD_{min}(Pb_1) \Rightarrow FD_{min}(Pb_2)$. If we have neither $FD_{min}(Pb_1) \Rightarrow FD_{min}(Pb_2)$, nor $FD_{min}(Pb_2) \Rightarrow FD_{min}(Pb_1)$, Pb_1 and Pb_2 are incomparable, which means that they require underlying systems with different properties in order for them to be solved.

Can failure detectors be implemented? Guided by practical motivations, we only consider failure detectors that cannot guess the future. Those failure detectors have been called *realistic* [17]. They actually do correspond to the failure detectors that can be implemented in a synchronous system (i.e., a system with known upper bounds on both message delays and the time it requires for a process to execute a step).

So, a simple way to implement a failure detector is to use an underlying synchronous system as additional subsystem. This subsystem is only used to implement the required failure detector and is not directly accessible by the processes. They do not ever know the existence of it: they evolve in a computation model defined by an asynchronous system enriched with the appropriate failure detector.

[4]Failure detectors have initially been introduced to cope with the impossibility to solve consensus in time-free systems prone to process crashes [13].

[5]Or (in some cases) approximated, as we will see later.

As we will see later, it is possible to solve some problems with *eventually accurate* failure detectors. Those failure detectors are assumed to satisfy their accuracy property (restriction on the mistakes they can make) only after some finite but unknown time. It appears that the use of such failure detectors generally requires a majority of correct processes (this constraint can be seen as the price that has to be paid to cope with *eventual* accuracy).

Interestingly, the use of eventually accurate failure detectors allows the design of *indulgent* algorithms [26], i.e., algorithms that never violate their safety property, whatever the behavior of the failure detector they use. This means that, if the failure detector never meets its accuracy property, these algorithms cannot terminate, but if they terminate they terminate correctly. Such eventually accurate failure detectors are very interesting for a simple reason. They have best effort implementations in asynchronous systems, namely, these implementations provide failure detector outputs that a priori can only be considered as *approximate* outputs. But, very interestingly, when the underlying system behaves synchronously during a long enough period, the outputs are no longer approximate but become correct. The periods during which the underlying asynchronous system behaves synchronously are usually called "*stable*" periods. An interesting consequence is that, as we can see, the protocols implementing such failure detectors can run concurrently with the application processes on the same asynchronous system. This is practically relevant.

2 Asynchronous System Models

Process model We consider a system consisting of a finite set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$. A process can fail by crashing, i.e., prematurely halting. It behaves correctly (i.e., according to its specification) until it (possibly) crashes. By definition a process is *correct* (during a run) if it does not crash (during that run); otherwise, it is *faulty*. There is no assumption on the time it takes for a (non-crashed) process to execute a step. In the following, $t \leq n-1$ denotes the maximum number of processes that can crash, and $f \leq t$ the actual number of process crashes during a given run.

Communication model Processes communicate and synchronize by exchanging message through links. Every pair of processes is connected by a link. We consider two types of links.

- The link connecting p_i to p_j is *reliable* if it does not create or duplicate messages, and every message sent by p_i to p_j is eventually received by p_j (if p_j is correct).
- The link connecting p_i to p_j is *fair lossy* if, while it does not create or duplicate messages, it can lose messages but, if p_i sends an infinite number of messages to p_j and p_j executes receive actions infinitely often, then it receives an infinite number of messages from p_i.

A process p_i sends a message m to a process p_j by invoking "*send*(m) *to* p_j"; p_j receives it when it terminates the invocation of "*receive*()". The *send*() and *receive*() primitives are provided by the underlying communication network. The notation "*broadcast*(m)" is used as a shortcut for "**forall** $j \in \{1, \ldots, n\}$ **do** *send*(m) *to* p_j **enddo**". If p_i crashes while executing "*send*(m) *to* p_j", either m is sent, or m is not sent at all (i.e., *send*() is atomic, while *broadcast*() is not.)

Computation models In the following we consider two types of asynchronous computation models.

- The FLP computation model that considers crash-prone processes and reliable links[6].
- The FLL computation model that considers crash-prone processes and fair lossy links.

3 Solving Consensus

3.1 The Consensus Problem

The *consensus* problem is a paradigm of agreement problems. It appears, in one form or another, as soon as processes have to agree, e.g., on a common action to execute, on the same decision to take, etc. A well-known example where consensus appears is *atomic broadcast*. That problem, which appears as a basic software layer in a lot of replication-based fault-tolerant distributed systems, requires that the correct processes deliver the same set of messages in the same order. So, it is at the same time a communication problem (all the correct processes have to deliver the same set of broadcast messages), and a consensus problem (as they have to deliver them in the same order) [13]. So, in the consensus problem, every correct process p_i *proposes* a value v_i and all correct processes have to *decide* on some value v, in relation to the set of proposed values. More precisely, the *consensus problem* is defined by the following three properties [13, 23]:

- C-Termination: Every correct process eventually decides on some value.
- C-Validity: If a process decides v, then v was proposed by some process.
- C-Agreement: No two correct processes decide differently.

The agreement property applies only to correct processes. So, it is possible that a process decides on a distinct value just before crashing. *Uniform consensus* prevents such a possibility. It has the same Termination and Validity properties plus the following agreement property:

- C-Uniform Agreement: No two processes (correct or not) decide differently.

In the following we consider the uniform consensus problem.

3.2 An Eventually Accurate Failure Detector

As indicated in the Introduction, the consensus problem cannot be solved in asynchronous systems prone to even a single process failure [23]. It is to circumvent this impossibility that Chandra and Toueg proposed the failure detector concept [13]. Among the several classes of failure detectors they have proposed, the one denoted $\Diamond\mathcal{S}$ has been shown to be the weakest to solve consensus [12]. Each process p_i is equipped with a local failure detector module that provides it with a set $suspected_i$; p_i can only read this set that contains the identities of the processes that are currently suspected to have crashed. Any failure detector module is inherently unreliable: it can make mistakes by not suspecting a crashed process or by erroneously suspecting a correct one. Moreover, suspicions are not necessarily stable: a process p_j can be added to or removed from a set $suspected_i$ according to whether p_i's failure detector module currently suspects p_j or not. We say "process p_i suspects process p_j" at some time, if at that time we have $p_j \in suspected_i$.

To be useful a failure detector class has to satisfy some properties, and those have to be as weak as possible while allowing the problem of interest to be solved. The class $\Diamond\mathcal{S}$ includes all the failure detectors satisfying the following properties:

[6]The name "FLP" is coined from the first letters of Fischer, Lynch and Paterson who proved the impossibility of solving consensus in this system model [23]. This acronym is of general use in the literature.

- Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.
- Eventual Weak Accuracy: There is a time after which some correct process is never suspected by the correct processes.

The implementation of failure detectors of the class $\diamond\mathcal{S}$ has been addressed in [22, 30, 38, 39, 41]. As noted in the Introduction, all these implementations assume that the underlying system is eventually *stable*. If the stability assumption is satisfied during a long enough period, the sets $suspected_i$ satisfy the properties defining $\diamond\mathcal{S}$. "Long enough" means here "a duration allowing the protocol using the failure detector to terminate".

3.3 A $\diamond\mathcal{S}$-based Consensus Protocol

The protocol that follows considers the FLP model. It is indulgent, so it enjoys the nice property of never violating consensus safety (validity and uniform agreement), whatever the sequence of (correct or bad) values read from the $suspected_i$ sets; moreover, it terminates (at least) when these sets contain correct values during a long enough period (namely, the period during which the consensus protocol needs the failure detector).

The protocol presented in Figure 1 is a particular instance of the generic protocol introduced in [46]. It requires a majority of correct processes ($t < n/2$), which has been shown to be a necessary requirement for indulgent protocols [26]. Its principles are surprisingly simple. The processes (p_i) proceed by asynchronous consecutive rounds (r_i). Each round r is coordinated by a process p_c such that $c = (r \bmod n) + 1$ (hence, if the round number never stops increasing, each process is ensured to be the coordinator of a future round).

Let v_i be the value initially proposed by p_i. The local variable est_i represents p_i's estimate of the decision value. During a round r, its coordinator p_c tries to impose its current estimate as the decision value. To attain this goal, a round is made up of two phases. During the first phase (1) p_c sends est_c to all the processes (line 4), and (2) any process p_i waits until it receives p_c's estimate or suspects it (line 5). According to the result of its waiting, a process p_i sets a local variable aux_i to the received value $v = est_c$, or sets it to a default value $\perp$ (line 6). It is important to notice that due to the completeness property of the underlying failure detector no process can block forever at line 5.

Then, the processes start the second phase of round r, during which they exchange the values of their aux_i variables (line 7). Let us observe that, due to the "majority of correct processes" assumption, no process can block forever at line 8. Moreover, it is important to notice that only two values can be exchanged: $v = est_c$ or $\perp$. Consequently, the set rec_i of values received by a process p_i can only have the values $\{v\}$, $\{v, \perp\}$, or $\{\perp\}$. Moreover, due to the "majority of correct processes" assumption it is impossible for two sets rec_i and rec_j to be such that $rec_i = \{v\}$ and $rec_j = \{\perp\}$, so we also have the following invariant (for each pair of processes p_i and p_j that have not crashed):

$$\begin{array}{rcl} rec_i = \{v\} & \Rightarrow & (\forall\, p_j : \ (rec_j = \{v\}) \ \vee \ (rec_j = \{v, \perp\})) \\ rec_i = \{\perp\} & \Rightarrow & (\forall\, p_j : \ (rec_j = \{\perp\}) \ \vee \ (rec_j = \{v, \perp\})). \end{array}$$

This invariant dictates the behavior of p_i:

- $rec_i = \{v\}$ (line 10). In this case, p_i decides the value v. It can safely do so, since in this case, a process that does not decide adopts v as its new estimate value. Moreover, to prevent possible deadlock situations, p_i broadcasts its decision value.

```
Function Consensus($v_i$)

Task $T1$:
 (1)  $r_i \leftarrow 0$; $est_i \leftarrow v_i$;
 (2)  while $true$ do
 (3)      $c \leftarrow (r_i \bmod n) + 1$; $r_i \leftarrow r_i + 1$; % $1 \leq r_i < +\infty$ %

          ————— Phase 1 of round $r$: from $p_c$ to all —————
 (4)      if $(i = c)$ then broadcast PHASE1$(r_i, est_i)$ endif;
 (5)      wait until (PHASE1$(r_i, v)$ has been received from $p_c \vee c \in suspected_i$);
 (6)      if (PHASE1$(r_i, v)$ received from $p_c$) then $aux_i \leftarrow v$ else $aux_i \leftarrow \perp$ endif;

          ————— Phase 2 of round $r$: from all to all —————
 (7)      broadcast PHASE2$(r_i, aux_i)$;
 (8)      wait until (PHASE2 $(r_i, aux)$ msgs have been received from a majority of proc.);
 (9)      let $rec_i$ be the set of values received by $p_i$ at line 8;
          % We have $rec_i = \{v\}$, or $rec_i = \{v, \perp\}$, or $rec_i = \{\perp\}$ where $v = est_c$ %
 (10)     case $rec_i = \{v\}$      then $est_i \leftarrow v$; broadcast DECISION$(est_i)$; stop $T1$
 (11)          $rec_i = \{v, \perp\}$ then $est_i \leftarrow v$
 (12)          $rec_i = \{\perp\}$    then skip
 (13)     endcase
 (14) endwhile

Task $T2$: when DECISION$(est)$ is received: broadcast DECISION$(est_i)$; return$(est)$
```

Figure 1: A Simple $\diamond\mathcal{S}$-Based Consensus Protocol ($t < n/2$) [46]

- $rec_i = \{v, \perp\}$: (line 11). In this case, consistently with the previous item, p_i adopts v as its new estimate value, and proceeds to the next round.
- $rec_i = \{\perp\}$ (line 12). In this case, p_i proceeds to the next round without modifying est_i.

The proof that this $\diamond\mathcal{S}$-based consensus protocol is correct is relatively easy. It is left to the reader (who can also find it in [46]). The strong completeness property is used to show that the protocol never blocks. The eventual weak accuracy property is used to ensure termination (there will be a round coordinated by a correct non-suspected process). The majority of correct processes is used to prove consensus agreement.

Other $\diamond\mathcal{S}$-based consensus protocols can be found in [13, 35, 34, 58].

3.4 Interactive Consistency

This problem has first been introduced in the context of synchronous systems where some processes can behave in a Byzantine way [51]. Here we consider the interactive consistency problem in the FLP model.

This problem is harder than consensus in the following sense: the processes have to agree not on a proposed value but on the *vector* of proposed values. So, each process p_i proposes a value v_i and has to decide a vector D_i such that the following properties are satisfied (we consider here the *uniform* version of the problem):

- IC-Termination: Every correct process eventually decides on a vector.
- IC-Validity: Any decided vector D is such that $D[i] \in \{v_i, \perp\}$, and is v_i if p_i does not crash.

- IC-Agreement: No two processes decide differently.

It is shown in [32] that the weakest failure detector class that allows the interactive consistency problem to be solved in the FLP model is the class of perfect failure detectors. This class, denoted $\mathcal{P}$, contains all the failure detectors that satisfy the following properties [13]:

- Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.

- Strong Accuracy: No process is suspected before it crashes.

As we can see, a perfect failure detector never makes mistakes. A $\mathcal{P}$-based interactive consistency protocol is described in [21]. Interestingly, this protocol which proceeds by consecutive asynchronous rounds, is as efficient as the "best" synchronous interactive consistency protocol ("best" from a time complexity point of view, i.e., when we count the maximum number of rounds that are required, namely, $\min(f+2, t+1, n)$).

While consensus can be solved in the FLP model (with a majority of correct processes) equipped with $\diamond\mathcal{S}$, it is not possible, assuming a solution to the consensus problem, to design a protocol building a failure detector of $\diamond\mathcal{S}$. On the contrary, we show here that, in the FLP model, the construction of a perfect failure detector and interactive consistency are equivalent problems in the sense that one can solve either of them as soon as we are provided with a solution to the other.

Interactive consistency protocols based on a perfect failure detector are described in [21, 32]. A protocol providing the inverse construction is described here (Figure 2): assuming a solution to the interactive consistency problem (subroutine protocol called IC_Protocol(x, v)), this protocol implements a perfect failure detector. The protocol consists of two tasks and is very simple[7]. Task $T1$ repeatedly invokes the interactive consistency protocol and suspects a process p_j as soon as the output D_i returned by an invocation is such that $D_i[j] = \perp$. Task $T2$ processes the queries issued by the upper layer: it returns the current value of $suspected_i$. The reader can easily check that the sets $suspected_i$ satisfy strong completeness and strong accuracy.

```
init: $suspected_i \leftarrow \emptyset$; $seq_i \leftarrow 0$

task $T1$: while $true$ do
                $seq_i \leftarrow seq_i + 1$; % IC instance number %
                $D_i \leftarrow$ IC_Protocol$(seq_i, v_i)$; % $v_i \neq \perp$ %
                $suspected_i \leftarrow \{j \mid D_i[j] = \perp\}$
           enddo

task $T2$: when $p_i$ issues $QUERY$: $return(suspected_i)$
```

Figure 2: From Interactive Consistency to a Perfect Failure Detector [32]

4 Solving Non-Blocking Atomic Commit

4.1 The Non-Blocking Atomic Commit Problem

Originated from databases, the *non-blocking atomic commit* problem (NBAC) is certainly one of the oldest agreement problems encountered in distributed computing. According to its local state,

[7]The difficult construction is the other one: solving interactive consistency from a perfect failure detector [21, 32].

each process first issues a vote (*yes* or *no*). Then, according to the set of votes and the fact that some processes possibly crashed, the non-crashed processes have to decide on a single value, namely, *commit* or *abort*. More precisely, the problem is defined by the following properties:

- NBAC-Termination: Every correct process eventually decides.
- NBAC-Validity: A decided value is *commit* or *abort*. Moreover:
  - NBAC-Justification: If a process decides *commit*, all processes have voted *yes*.
  - NBAC-Obligation: If all processes vote *yes* and there is no crash, then the decision value is *commit*.
- NBAC-Agreement: No two processes decide differently.

It is easy to see that the justification property relates the *commit* decision to the *yes* votes, while the obligation property eliminates the trivial and useless solution where all processes would always decide *abort*. Actually, this property defines what is a "good" run: it is a run in which all the processes want to commit (they voted *yes*) and the environment behaves correctly (no process crashes). The decision can only be *commit* in good runs.

As the reader can see, a major difference between the specification of consensus and the specification of NBAC lies in the fact that the latter mentions explicitly process crashes occurring during a protocol execution.

4.2 An Appropriate Failure Detector

Solving NBAC in the FLP model requires the model to be enriched with appropriate failure detectors. Such failure detectors are studied and investigated in [27, 28]. We consider here *timeless* failure detectors, i.e., failure detectors that do not provide information on when exactly (in the sense of global time) failures occurred. (Let us notice that $\mathcal{P}$ and $\diamond\mathcal{S}$ define classes of timeless failure detectors.)

To address this question, a failure detector class, denoted $?\mathcal{P}$ and called the class of *anonymously perfect* failure detectors, has been proposed in [27]. This class is defined as follows:

- Anonymous completeness: If a crash occurs, eventually every correct process is permanently informed that some crash occurred.
- Anonymous accuracy: No crash is detected unless some process crashed.

The class of failure detectors denoted $?\mathcal{P} + \diamond\mathcal{S}$ includes all the failure detectors that satisfy both $?\mathcal{P}$ and $\diamond\mathcal{S}$. It has then been been shown [28] that this is the weakest class of timeless failure detectors to solve NBAC, assuming a majority of correct processes ($t < n/2$). In the following, a failure detector module of that class is represented at p_i as a boolean variable ap_flag_i ("approximate flag") that is *true* iff a crash is detected.

Figure 3 describes a NBAC protocol based on a failure detector of $?\mathcal{P} + \diamond\mathcal{S}$. This protocol actually reduces NBAC to consensus (which, as we have seen, can be solved in the FLP model enriched with $\diamond\mathcal{S}$ when $t < n/2$). The protocol is pretty simple. From a methodological point of view, it is interesting to see how each of $?\mathcal{P}$ and $\diamond\mathcal{S}$ are used: the first is for ensuring the validity of NBAC, the second to be able to use the subroutine consensus protocol.

```
Function Nbac($vote_i$)

    broadcast MY_VOTE($vote_i$);
    wait until (MY_VOTE($vote$) has been received from each process $\vee$ $ap\_flag_i$);
    if (a vote yes has been received from each of the $n$ processes)
            then $output_i \leftarrow$ Consensus($commit$)
            else  $output_i \leftarrow$ Consensus($abort$)
    endif;
    return($output_i$)
```

Figure 3: A Simple $?\mathcal{P} + \diamond\mathcal{S}$-Based NBAC Protocol ($t < n/2$) [27]

5 Implementing Quiescent Communication

This section continues our visit of the failure detector concept by considering the problem that consists of implementing *quiescent communication* despite process crashes and fair lossy links, i.e., in the FLL model [2].

5.1 The Quiescence Problem

Considering two processes p_i and p_j that do not crash connected by a fair lossy link, a basic communication problem consists in building a reliable link on top of that fair lossy link. This problem is well-known and basic mechanisms such as retransmission and acknowledgments allow it to be solved. Retransmission allows message losses to be tolerated, while acknowledgments allow their retransmission to be eventually stopped: when a receiver receives a message m, it sends back $ack(m)$, and for each message m it wants to send, the sender repeatedly resends it until it gets an $ack(m)$. This simple protocol is *quiescent* in the sense that, after some time, no process sends or receives messages related to the transmission of m.

Let us now consider the case where the receiver p_j can crash. In this case, it is possible that p_j crashes before receiving m and the sender will consequently send copies of m forever. The protocol is no longer quiescent. So, the problem is to provide quiescent implementations of communication primitives in the FLL model. This is the problem addressed and solved in [2]. This paper first shows that the *quiescent communication* problem cannot be solved in a pure FLL model. To solve it, that model has to be appropriately enriched with a failure detector. This is not at all counter-intuitive since, to stop retransmitting a message, the sender has to know -in a way or another- whether the receiver has crashed.

[2] shows that the weakest class of failure detectors solving the quiescent communication problem is the class of eventually perfect failure detectors, i.e., the failure detectors that, after some unknown but finite time, suspect all the crashed processes and only them. Unfortunately, such a failure detector cannot be implemented in FLL. So, the authors investigated another class of implementable failure detectors capable of providing quiescent communication protocols. They called it the class of *heartbeat* failure detectors.

5.2 A Heartbeat Failure Detector

A heartbeat failure detector outputs at each process p_i an array $HB_i[1..n]$ of non-decreasing counters satisfying the following properties:

- HB-completeness: If p_j crashes, then $HB_i[j]$ stops increasing.

- HB-accuracy: If p_j is correct, then $HB_i[j]$ never stops increasing.

As we can see, heartbeat failure detectors can be implemented, but their implementation is not quiescent. In that sense these failure detectors allow the non-quiescent part of a communication protocol to be isolated. Moreover, the use of a heartbeat failure detector favors design modularity and eases correctness proofs. Additionally, a single heartbeat failure detector "service" can be used by several upper layer applications.

5.3 A Quiescent Implementation

Figure 4 presents a quiescent protocol providing a reliable link in an FLL system model equipped with a heartbeat failure detector. The protocol on the sender side provides an implementation of the SEND() primitive invoked by the upper layer application. To that end, it uses the send() primitive provided by the underlying communication layer. Similarly, on the receiver side, RECEIVE() notifies the upper layer that a new message has arrived, while receive() is used to receive a message from the underlying communication layer. The protocol is particularly simple and self-explanatory. The seq_i variable (initialized to 0) is used by the sender as a sequence number generator. It is easy to see that, after some unknown but finite time, p_i either receives $ack(m)$ (due to the fairness of the underlying channel) or stops retransmitting as $HB_i[j]$ no longer increases (when p_j has crashed).

This protocol shows an important difference between a *quiescent* protocol and a *terminating* protocol. The protocol described in Figure 4 is quiescent as for each message m sent by p_i to p_j, there is a time after which no more protocol messages are exchanged. However this protocol is not terminating. This is because, until it receives $ack(m)$, p_i has no means to know if it has to retransmit m: if p_j crashes and all $ack(m)$ it sent before are lost, p_i will never terminate the task *repeat_send* (m, seq_i). The reader can observe that this is the best that can be done. This important difference is discussed in detail in [36].

```
Sender $p_i$:
    when SEND($m$) TO $p_j$ is invoked:
        $seq_i \leftarrow seq_i + 1$;
        fork task $repeat\_send\ (m, seq_i)$

    task $repeat\_send\ (m, seq_i)$:
        $prev\_hb \leftarrow -1$;
        repeat periodically $hb \leftarrow HB_i[j]$;
                            if $(prev\_hb < hb)$ then $send\ msg(m, seq)\ to\ p_j$;
                                                      $prev\_hb \leftarrow hb$
                            endif
        until ($ack(m, seq)$ is $received$)


Receiver $p_j$:
    when $msg(m, seq)$ is $received\ from\ p_i$:
        if (first reception of $msg(m, seq)$) then $m$ is RECEIVED endif;
        $send\ ack(m, seq)\ to\ p_i$
```

Figure 4: A Quiescent Implementation of a Reliable Link [2]

6 Failure Detectors in Synchronous Systems

While atomic broadcast, consensus, NBAC, etc. cannot be solved in the FLP computation model, they can be solved in synchronous systems, i.e., in systems where there are bounds on processing time and message transfer delay. So, failure detectors are useless in these systems from a decidability point of view: they add no computational power. Nevertheless, failure detectors suited to synchronous systems have recently been introduced. Their aim is to help design more efficient protocols, i.e., protocols with a "best case" time complexity that can not be attained in pure synchronous systems. To illustrate this idea, we show here, how *fast failure detectors* can be used to expedite consensus in synchronous systems [6].

6.1 Synchronous System Model

As previously mentioned, a synchronous systems is characterized by the existence of a bound on the time it takes to receive and process a message, and the fact that this bound is known by the processes. In order to simplify the presentation and without loss of generality, we assume in the following that local computations take no time and transfer delays are upper bounded by D. Thus, a message sent at time t is not received after $t + D$ (D-timeliness). The links are reliable (no creation, duplication or loss). Moreover, processes have access to a common clock.

The combination of D-timeliness and no-loss properties with the possibility of process crashes, makes possible the following behaviors when, at time t, a process p_i sends a message m to processes p_j and p_k. If p_i does not crash at time t, both p_j and p_k receive m by time $t + D$. However, if p crashes at time t, different scenarios are possible. Namely, it is possible that neither p_j nor p_k receives m, or that only one of them receives m (by time $t + D$) while the other does not receive it, or that both of them receive m (by time $t + D$).

6.2 Fast Failure Detectors

Such failure detectors have been introduced in [6]. A *fast perfect* failure detector provides the processes p_i with sets $suspected_i$ that satisfy the following properties (where $d < D$):

- d-Timely completeness. If a process p_j crashes at time t, then, by time $t + d$, every alive process suspects it permanently.
- Strong accuracy. No process is suspected before it crashes.

Let us observe that, if a process crashes between times t and $t + d$ then some, but not necessarily all, processes may suspect it at $t + d$.

As indicated in [6], fast failure detectors can be implemented with specialized hardware (with provides $d << D$). From a user point of view (the one in which we are interested here) they can be used to attain time complexity lower bounds that are better than what can be attained in a pure synchronous system.

To illustrate this, let us consider the fast failure detector-based synchronous consensus protocol described in Figure 5 [6]. This protocol enjoys the following early deciding property: started at time $T = 0$, it allows the processes to decide by time $D + fd$ (let us remind that f is the actual number of process crashes). Thus, its time complexity is $D + fd$ which is much better than the best that can be done without using failure detectors, namely, $\min(f + 2, t + 1)D$.

Let us now describe in detail the behavior of a process p_i. Let us first observe that, during the time period $[0, (i - 1)d)$, p_i can only receive messages. Then, at time $(i - 1)d$, if p_i suspects all the processes with a smaller id, it sends its current estimate of the decision value (est_i) to all the

```
Function Consensus($v_i$)

    init $est_i \leftarrow v_i$; $max_i \leftarrow 0$

    when $(est, j)$ is received:
        if $(j > max_i)$ then $est_i \leftarrow est$; $max_i \leftarrow j$ endif

    at time $(i-1)d$ do
        if $(\{p_1, p_2, \ldots, p_{i-1}\} \subseteq suspected_i)$ then broadcast $(est_i, i)$ endif

    at time $(j-1)d + D$ for every $1 \leq j \leq n$ do
        if $\big((p_j \notin suspected_i) \wedge (p_i$ has not yet decided$)\big)$ then return $(est_i)$ endif
```

Figure 5: Synchronous Consensus with a Fast Failure Detector [6]

processes. When a process p_i receives an estimate value (est), it updates its own estimate (est_i) only if it is coming from a process whose id is larger than max_i (a local variable initialized to a value smaller than any id). In that way, the successive values of est_i are coming from processes with increasing ids. Finally, at times $(j-1)d + D$, for $j = 1, \ldots, n$, p_i decides if it trusts the corresponding process p_j. A proof of the protocol can be found in [6]. It is easy to see that the processes decide by D time units when the process p_1 does not crash (in that case they decide the value v_1 proposed by p_1). If p_1 crashes while p_2 does not, they decide by time $d + D$; according to the failure pattern, the decided value is the value v_1 proposed by p_1 or the value v_2 proposed by p_2 (it is the value v_1 proposed by p_1, if p_1 succeeded in sending v_1 to p_2). Etc.

7 Additional Remarks

Other problems In addition to the previous distributed computing problems that we have shortly visited, the failure detector approach has been used to circumvent other impossibility results. We list here two of them.

The first is the construction of a *reliable atomic register* in the FLP model. It has been shown that such a construction is possible if and only if $t < n/2$. Intuitively, the majority of correct processes assumption can be used to ensure that the last value[8] of the register can always be accessed [9]. A natural question is then: "Which is the weakest failure detector to implement an atomic register in the FLP model when $n/2 \leq t < n$ (i.e., when any number of processes can crash)?" This question is answered in [18] with the failure detector, denoted $\mathcal{P}^x$, defined by the following properties:

- Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.
- x-Accuracy: At any time, a process falsely suspects at most $n - x - 1$ alive processes.

A $\mathcal{P}^t$-based algorithm building an atomic register is described in [18]. (Interestingly, $\mathcal{P}^t$ can be built in the FLP model when $t < n/2$. This is not at all surprising as the atomic register problem can be solved in the same context without the help of a failure detector.) For the interested reader, let us mention that a protocol solving the consensus problem despite up to $t < n$ crashes, in the

[8] "Last" refers here to physical time, as the consistency criterion considered for the register is atomicity.

FLP model enriched with both $\mathcal{P}^t$ and $\Diamond\mathcal{S}$ is described in [24].

The second problem we mention here is related to communication, namely the design of a *uniform reliable broadcast* primitive. This primitive allows the processes to broadcast messages, and ensures that (1) at least the messages broadcast by the correct processes are delivered to all the correct processes, and (2) if a process (correct or faulty) delivers a message m then all correct processes deliver m. So, this primitive ensures that no message from a correct process is "lost", and that no message delivered by a process is missed by a correct processes: the correct processes deliver the same set of messages, and a faulty process delivers a subset of it.

As previously, this problem can be easily solved in the FLP model when $t < n/2$ and requires additional assumptions when $n/2 \leq t < n$. The main difficulty lies in ensuring item (2). The interested reader will find in [8] an appropriate (optimal) failure detector and a uniform reliable broadcast protocol based on such a failure detector. For the interested reader, a uniform reliable broadcast protocol that additionally satisfies the quiescence property is described in [41, 55].

Other failure detectors Other failure detectors have been proposed. One of the most well-known provides the processes with a function leader satisfying the following properties:

- Validity: Each invocation of leader returns a process name.
- Eventual Leadership: There is a time t and a correct process p such that, after t, every invocation of leader by a correct process returns p.

The oracles that satisfy this property define the class of failure detector oracles denoted Ω [12]. A failure detector of this class actually provides the processes with an eventual leader election capability. But, let us notice that there is no knowledge of when the leader is elected. This means that several leaders can coexist during an arbitrarily long period of time, and there is no way for the processes to learn when this "confusing" period is over.

Ω-based consensus protocols are described in [29, 48]. The requirement $t < n/2$ is a necessary for such protocols [13]. Moreover, it has been shown that Ω and $\Diamond\mathcal{S}$ have the same computational power [12, 16]: no one allows the solution of a problem that could be solved without the other.

On the methodology Since the implementation of some failure detectors (e.g., eventually accurate failure detectors such as $\Diamond\mathcal{S}$) can be only approximate during some periods (when the underlying system is unstable), it is interesting to use a failure detector only in "extreme" cases, which means that the use of a failure detector has to be avoided whenever possible.

Considering the atomic broadcast problem, several papers [3, 47, 52] have provided atomic broadcast implementations that use a failure detector-based consensus black box only in extreme cases. Such protocols are said to be *thrifty* (or non-trivial) with respect to the underlying oracle.

A few references The reader interested in the implementation of failure detectors should consult the following references [4, 11, 14, 22, 30, 38, 41, 42, 57]. Protocols implementing failure detectors of the class Ω can be found in [4, 5, 39, 49]. The reader interested in the the classification of distributed computing problems in presence of failures can consult [19, 25, 31]. The reader interested in the weakest failure detector classes to solve some fundamental distributed computing problems can consult [12, 20].

Random oracles have also been investigated to solve distributed computing problems in presence of crash failures [10, 54]. Combination of random oracles with failure detectors is addressed in [7, 43, 50].

The condition-based approach to solve agreement problems consists in characterizing the largest set of input vectors for which it is possible to solve the problem [44]. As a "trivial" example we can consider the case where we know that more than a majority of processes do propose the same value. It is easy to solve consensus for such input vectors despite one process crash. Roughly speaking, a condition-based protocol solves the corresponding agreement problem each time the input vector belongs to the condition (or when there are no failures), and does its best to terminate when the input vector does not belong to the condition and there are failures. Very recently, has been proposed a new class of failure detectors that allows combining the power of conditions with the information on failures required to solve agreement problems [45].

8 Conclusion

The failure detector approach was introduced by Sam Toueg. Initially designed for asynchronous systems, it allows a statement, of the weakest assumptions that have to be added to these systems in order to solve problems that otherwise could not be solved. So, in this type of system, they allow the barrier separating impossibility and decidability to be crossed. From a more practical software engineering point of view, they strongly favor a modular approach. Failure detectors have then been extended to synchronous systems. In these systems, they allow the attainment of time complexity lower bounds that could not be attained in purely synchronous systems.

Acknowledgments

I would like to thank David Powell (from LAAS) for interesting discussions, constructive remarks and useful comments that helped improve the paper.

References

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

[2] Aguilera M.K., Chen W. and Toueg S., On Quiescent Reliable Communication. *SIAM Journal of Computing*, 29(6):2040-2073, 2000.

[3] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Thrifty Generic Broadcast. *Proc. 14th Symposium on Distributed Computing (DISC'00)*, Springer-Verlag LNCS #1914, pp. 268-282, 2000.

[4] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., On Implementing Ω with Weak Reliability and Synchrony Assumptions. *Proc. 22th ACM Symposium on Principles of Distributed Computing (PODC'03)*, ACM Press, pp. 306-314, Boston (MA), 2003.

[5] Aguilera M.K., Delporte-Gallet C., Fauconnier H. and Toueg S., Communication-Efficient Leader Election and Consensus with Limited Link Synchrony. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 328-337, St-John's (Newfoundland, Canada), 2004.

[6] Aguilera M.K., Le Lann G. and Toueg S., On the Impact of Fast failure Detectors on Real-Time Fault-Tolerant Systems. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 354-369, 2002.

[7] Aguilera M.K. and Toueg S., Aguilera M.K. and Toueg S., Failure Detection and Randomization: a Hybrid Approach to Solve Consensus. *SIAM Journal of Computing*, 28(3):890-903, 1998.

[8] Aguilera M.K., Toueg S. and Deianov B., Revisiting the Weakest Failure Detector for Uniform Reliable Broadcast. *Proc. 13th Int. Symposium on DIStributed Computing (DISC'99)*, Springer-Verlag LNCS #1693, pp. 21-34, 1999.

[9] Attiya H. and Welch J. Distributed Computing: Fundamentals, Simulations and Advanced Topics. *McGraw-Hill*, 451 pages, 1988.

[10] Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *2nd ACM Symposium on Principles of Distributed Computing, (PODC'83)*, Montréal (CA), pp. 27-30, 1983.

[11] Bertier M., Marin O. and Sens P., Implementation and Performance Evaluation of an Adaptable Failure Detector. *Proc. Int. IEEE Conference on Dependable Systems and Networks (DSN'02)*, IEEE Computer Society Press, pp. 354-363, Washington D.C., 2002.

[12] Chandra T.D., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996.

[13] Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996. (First version published in the proceedings of the *10th ACM Symposium on Principles of Distributed Computing*, 1991.)

[14] Chen W., Toueg S. and Aguilera M.K., On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, 51(5):561-580, 2002.

[15] Chor M., and Dwork C., Randomization in Byzantine Agreement. *Adv. in Comp. Research*, 5:443-497, 1989.

[16] Chu F., Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 76(6):293-298, 1998.

[17] Delporte-Gallet C., Fauconnier H. and Guerraoui R., A Realistic Look at Failure Detectors. *Proc. IEEE Inter. Conference on Dependable Systems and Networks (DSN'02)*, IEEE Computer Society Press, pp. 345-352, Washington D.C., 2002.

[18] Delporte-Gallet C., Fauconnier H. and Guerraoui R., Failure Detection Lower Bounds on Registers and Consensus. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 237-251, 2002.

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

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

[21] Delporte-Gallet C., Fauconnier H., Helary J.-M. and Raynal M. Early Stopping in Global Data Computation. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):909-921, 2003.

[22] Fetzer C., Raynal M. and Tronel F., An Adaptive Failure Detection Protocol. *Proc. 8th IEEE Pacific Rim Int. Symposium on Dependable Computing (PRDC'01)*, IEEE Computer Society Press, pp. 146-153, Seoul (Korea), 2001.

[23] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[24] Friedman R., Mostefaoui A. and Raynal M., A Weakest Failure Detector-Based Asynchronous Consensus Protocol for $f < n$. *Information Procesing Letters*, 90(1):39-46, 2004.

[25] Fromentin E., Raynal M. and Tronel F. On Classes of Problems in Asynchronous Distributed Systems with Process Crashes. *19th IEEE Int. Conf. on Distributed Computing Systems (ICDCS'99)*, Austin, TX, pp. 470-477, 1999.

[26] Guerraoui R., Indulgent Algorithms. *Proc. 19th ACM Symposium on Principles of Distributed Computing, (PODC'00)*, ACM Press, pp. 289-298,Portland (OR), 2000.

[27] Guerraoui R., Non-Blocking Atomic Commit in Asynchronous Distributed Systems with Failure Detectors. *Distributed Computing*, 15:17-25, 2002.

[28] Guerraoui R. and Kouznetsov P., On the Weakest Failure Detector for Non-Blocking Atomic Commit. *Proc. 2nd Int. IFIP Conference on Theoretical Computer Science (TCS'02)*, pp. 461-473, Montréal (Canada), August 2002.

[29] Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers.* 53(4), 53(4):453-466, April 2004.

[30] Gupta I., Chandra T.D. and Goldszmidt G.S., On Scalable and Efficient Distributed Failure Detectors. *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, ACM Press, pp. 170-179, Newport (RI), 2001.

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

[32] Hélary J.-M., Hurfin M., Mostefaoui A., Raynal M. and Tronel F., Computing Global Functions in Asynchronous Distributed Systems with Process Crashes. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):897-909, 2000.

[33] Hopcroft J.E. and Ullman J.D. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, Reading, Mass., 418 pages, 1979.

[34] Hurfin M., Mostefaoui A. and Raynal M., A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors. *IEEE Transactions on Computers*, 51(4):395-408, 2002.

[35] Hurfin M. and Raynal M., A simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector. *Distributed Computing*, 12(4):209-223, 1999.

[36] Koo R. and Toueg S., Effects of Message Loss on the Termination of Distributed Protocols. *Information Processing Letters*, 27:181-188, 1987.

[37] Lamport L., Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125-143, 1977.

[38] Larrea M., Arèvalo S. and Fernández A., Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. *Proc. 13th Symposium on Distributed Computing (DISC'99)*, Bratislava (Slovakia), Springer Verlag LNCS #1693, pp. 34-48, 1999.

[39] Larrea M., Fernández A. and Arèvalo S., Optimal Implementation of the Weakest Failure Detector for Solving Consensus. *Proc. 19th Symposium on Reliable Distributed Systems (SRDS'00)*, IEEE Computer Society Press, pp. 52-60, Nuremberg (Germany), 2000.

[40] Mostefaoui A., Mourgaya E. and Raynal M., An Introduction to Oracles for Asynchronous Distributed Systems. *Future Generation Computer Systems*, 18(6):757-767, 2002.

[41] Mostefaoui A., Mourgaya E., and Raynal M., Asynchronous Implementation of Failure Detectors. *Proc. Int. IEEE Conference on Dependable Systems and Networks (DSN'03)*, IEEE Computer Society Press, pp. 351-360, San Francisco (CA), 2003.

[42] Mostefaoui A., Powell D., and Raynal M., A Hybrid Approach for Building Eventually Accurate Failure Detectors. *10th IEEE Pacific Rim Int. Symposium on Dependable Computing (PRDC'2004)*, IEEE Computer Society Press, pp. 57-65, Papeete (Tahiti, France), 2004.

[43] Mostefaoui A., S. Rajsbaum S. and Raynal M., Versatile and Modular Consensus Protocol. *Int. IEEE/IFIP Conf. on Dependable Systems and Networks (DSN'02)*, IEEE Computer Society Press, pp. 364-373, Washington DC, 2002.

[44] Mostefaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Journal of the ACM,* 50(6):922-954, 2003.

[45] Mostefaoui A., S. Rajsbaum S. and Raynal M., The Combined Power of Conditions and Information on Failures to Solve Asynchronous Set Agreement. *Tech Report #1688*, IRISA, Université de Rennes (France), 2005. http://www.irisa.fr/bibli/publi/pi/2005/1688/1688.html

[46] Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Symp. on DIStributed Computing (DISC'99)*, Springer Verlag LNCS #1693, pp. 49-63, Bratislava (Slovakia), 1999.

[47] Mostefaoui A. and Raynal M., Low-Cost Consensus-Based Atomic Broadcast. *7th IEEE Pacific Rim Int'l Symposium on Dependable Computing (PRDC'2000)*, IEEE Computer Society Press, UCLA, Los Angeles (CA), pp. 45-52, 2000.

[48] Mostefaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.

[49] Mostefaoui A., Raynal M. and Travers C., Crash-Resilient Time-free Eventual Leadership. *Proc. 23th IEEE Symposium on Reliable Distributed Systems (SRDS'04)*, IEEE Computer Society Press, pp. 208-217, Florianõpolis (Brasil), October 2004.

[50] Mostefaoui A., Raynal M. and Tronel F., The Best of Both Worlds: a Hybrid Approach to Solve Consensus. *Proc. Int. Conference on Dependable Systems and Networks (DSN'00)*, IEEE Computer Society Press, pp. 513-522, New-York City, 2000.

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

[52] Pedone F. and Schiper A., Handling Message Semantics with Generic Broadcast Protocols. *Distributed Computing*, 15(2):97-107, 2002.

[53] Powell D., Failure Mode Assumptions and Assumption Coverage. *Proc. of the 22nd Int'l Symposium on Fault-Tolerant Computing (FTCS-22)*, Boston, MA, pp.386-395, 1992.

[54] Rabin M., Randomized Byzantine Generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, pp. 116-124, Los Alamitos (CA), 1983.

[55] Raynal M., Quiescent Uniform Reliable Broadcast as an Introduction to Failure Detector Oracles. *Proc. 6th Int. Conference on Parallel Computing Technologies (PaCT'01)*, Novosibirsk, Springer Verlag LNCS #2127, pp. 98-111, 2001.

[56] Raynal M., Detecting Crash Failures in Asynchronous Systems: What? Why? How? Tutorial given at *Proc. Int. Conference on Dependable Systems and Networks (DSN'04)*, Florence (Italy), 2004.

[57] Raynal M. and Tronel F., Group Membership Failure Detection: a Simple Protocol and its Probabilistic Analysis. *Distributed Systems Engineering Journal*, 6(3):95-102, 1999.

[58] Schiper A. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10:149-157, 1997.