Consensus in Asynchronous Distributed Systems: A Concise Guided Tour

Rachid Guerraoui[1], Michel Hurfin[2], Achour Mostefaoui[2], Riucarlos Oliveira[1], Michel Raynal[2], and Andre Schiper[1]

[1] EPFL, Département d'Informatique, 1015 Lausanne, Suisse
[2] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract. It is now recognized that the Consensus problem is a fundamental problem when one has to design and implement reliable asynchronous distributed systems. This chapter is on the Consensus problem. It studies Consensus in two failure models, namely, the Crash/no Recovery model and the Crash/Recovery model. The assumptions related to the detection of failures that are required to solve Consensus in a given model are particularly emphasized.

Keywords: Asynchronous Distributed Systems, Atomic Broadcast, Atomic Commitment, Consensus, Crash/no Recovery, Crash/Recovery.

1 Introduction

Distributed applications are pervading many aspects of everyday life. Booking-reservations, banking, electronic and point-of-sale commerce are noticeable examples of such applications. Those applications are built on top of distributed systems. When building such systems, system designers have to cope with two main issues: asynchrony and failure occurrence. *Asynchrony* means that it is impossible to define an upper bound on process scheduling delays and on message transfer delays. This is due to the fact that neither the input load from users nor the precise load of the underlying network can be accurately predicted. This means that whatever is the value used by a process to set a timer, this value cannot be trusted by the process when it has to take a system-wide consistent decision. Similarly, *failure occurrences* cannot be predicted. The net effect of asynchrony and failure occurrences actually create an *uncertainty* on the state of the application (as perceived by a process) that can make very difficult or even impossible to determine a system view that can be validly shared by all non-faulty processes. The mastering of such an uncertainty is one of the main problems that designers of asynchronous systems have to solve.

As a particular example, let us consider the case of a service whose state has been distributed on several nodes. To maintain a consistent copy of the service state, each node must apply to its copy the same sequence of the updates that have been issued to modify the service state. So, there are two problems to solve. (1) Disseminate the updates to the nodes that have a copy of the service state.

S. Krakowiak, S.K. Shrivastava (Eds.): Distributed Systems, LNCS 1752, pp. 33–47, 2000.

And (2), apply the updates in the same order to each copy. The first problem can be solved by using a *reliable multicast* primitive [21]. The second problem is more difficult to solve. The nodes have to agree on a common value, namely, the order in which they will apply the updates. This well known problem (namely, the *Atomic Broadcast* problem) is actually a classical *Agreement problem.*

It appears that any agreement problem can be seen as a particular instance of a more general problem, namely, the *Consensus* problem. In the Consensus problem, each process proposes a value, and all non-faulty processes have to agree on a single decision which has to be one of the proposed values. This chapter presents a few results associated with the Consensus problem. It is composed of seven sections. Section 2 defines the Consensus problem. Section 3 studies Consensus in the Crash/no Recovery model. Section 4 discusses issues related to the communication channel semantics. Section 5 introduces the differences between two main distributed computing models: (1) the Crash/no Recovery model, and (2) the Crash/Recovery model. Section 6 studies Consensus in the Crash/Recovery model, and Section 7 concludes the chapter.

2 The Consensus Problem

2.1 General Model

A *distributed system* is composed of a finite set of n sites interconnected through a communication network. Each site has a local memory (and possibly a stable storage according to the needs of applications) and executes one or more processes. To simplify and without loss of generality, we assume that there is only one process per site. Processes synchronize and communicate by exchanging messages through channels of the underlying network.

We consider *asynchronous* distributed systems: there are bounds neither on communication delays, nor on process speeds. The interest of the asynchronous model comes from its practicability. Open distributed systems such as systems covering large geographic areas, or systems subject to unpredictable loads that may be imposed by their users, are basically asynchronous due to the unpredictability of message transfer delays and process scheduling delays in those systems [2]. This makes the asynchronous model a very general model.

A process is either a *good* process or a *bad* process. What determines a process as being good or bad depends on the failure model. Section 3 and Section 6 provide instantiations of what is a good/bad process, in the Crash/no Recovery model and in the Crash/Recovery model, respectively. Roughly speaking, a *good* process is a process that behaves as expected. A *bad* process is a process that is not good. In both cases, a process is fail-silent: (1) until it crashes, a process behaves according to its specification, and (2) when crashed, it does nothing.

2.2 What Is the Consensus Problem?

In the *Consensus* problem, defined over a set $\{p_1, p_2, \ldots, p_n\}$ of processes, each process p_i proposes initially a value v_i, and all good processes have to decide on some common value v that is equal to one of the proposed values v_i [3].

Formally, the *Consensus* problem is defined in terms of two primitives: propose and decide. When a process p_i invokes propose(v_i), where v_i is its proposal to the Consensus problem, we say that p_i "proposes" v_i. When p_i invokes decide() and gets v as a result, we say that p_i "decides" v. The semantics of propose() and decide() is defined by the following properties:

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

While C-Termination defines the liveness property associated with the *Consensus* problem, C-Agreement and C-Validity define its safety properties.

The C-Agreement property allows bad processes to decide differently from good processes. This fact can be sometimes undesirable as it does not prevent a bad process to propagate a different decision throughout the system before crashing. In the *Uniform Consensus* problem, agreement is defined by the following property, which enforces the same decision on any process that decides:

- C-Uniform-Agreement. *No two processes (good or bad) decide differently.*

Actually, all Consensus algorithms discussed in this chapter solve the Uniform Consensus algorithm.

2.3 From an Agreement Problem to Consensus

When practical agreement problems have to be solved in real systems, a transformation is needed to bring them to the Consensus problem specified in the previous section. We illustrate below such a transformation on the Atomic Commitment problem. Transformation of other agreement problems to Consensus (*e.g.*, Group Membership to Consensus, View Synchronous Communication to Consensus, Atomic Broadcast to Consensus, Atomic Multicast to Consensus, Clock Management to Consensus) can be found in [3,5,10,18,20,24,25,29]. So Consensus can be viewed as the common denominator of the different agreement problems. This explains the importance of Consensus, and justifies the large interest in the literature for this problem.

The Atomic Commitment Problem As an example of agreement problem let us consider the *Non-Blocking Atomic Commitment* Problem. At the end of a computation, processes are required to enter a commitment protocol in order to commit their local computations (when things went well) or to abort them (when things went wrong). So, when it terminates its local computation each process has to vote YES or NO. If for any reason (deadlock, storage problem, concurrency control conflict, local failure, etc.) a process cannot locally commit its local computation, it votes NO. Otherwise a vote YES means that the process commits locally to make its updates permanent if it is required to do so. Based on these votes, the decision to commit or to abort is taken. The decision must be COMMIT if things went well (all process are good and voted YES). It must

be ABORT if things went wrong [12]. We consider here that a good process is a process that does not crash.

More formally, NBAC in an asynchronous distributed system can be defined by the following properties:

- NBAC-Termination. *Every good process eventually decides.*
- NBAC-Agreement. *No two processes decide differently.*
- NBAC-Validity. This property gives its meaning to the decided value. It is composed of three parts.
  - Decision Domain. *The decision value is* COMMIT *or* ABORT.
  - Justification. *If a process decides* COMMIT*, then all processes have voted* YES.
  - Obligation. *If all participants vote* YES *and none of them is perceived as bad, then the decision value must be* COMMIT.

The justification property states that the "positive" outcome, namely COMMIT, has to be justified: if the result is COMMIT, it is because, for sure, things went well (*i.e.*, all processes voted YES). Finally, the obligation property eliminates the trivial solution where the decision value would be ABORT even when the situation is satisfactory to commit.

Reducing Atomic Commit to Consensus Actually the NBAC is a particular instance of the Consensus problem. Figure 1 describes a simple protocol that reduces NBAC to Consensus.

```
(1)   ∀ p_j  do send(vote) to p_j end do;
(2.1) wait (   (delivery of a vote NO)
(2.2)       or (∃ p_j:  p_i perceives p_j as a bad process)
(2.3)       or (from each p_j:  delivery of a vote YES from p_j)
(2.4)       );
(3.1) case
(3.2)       a vote NO has been delivered     → v_i := ABORT
(3.3)       a process is perceived as bad    → v_i := ABORT
(3.4)       all votes are YES                → v_i := COMMIT
(3.5) end case;
(4)   propose(v_i); decision:=decide(); % Consensus execution %
```

Fig. 1. From Consensus to NBAC in Asynchronous Systems (code of process p_i)

The behavior of every process p_i is made of 4 steps. First (line 1), p_i disseminates its vote to all processes. Then (lines 2.*), p_i waits until either it has received a NO vote (line 2.1), or it has received a YES vote from each process (line 2.3), or it perceives a process as being (crashed) bad (line 2.2). Then (lines

3.*), p_i builds its own view v_i of the global state: v_i is COMMIT if from its point of view everything went well (line 3.4), and ABORT if from its point of view something went wrong (lines 3.2 and 3.3). Finally (line 4), p_i participates in a Consensus. After having proposed v_i, process p_i waits for the result of the Consensus (invocation of *decide*) and saves it in the local variable *decision*. It can be easily shown that this reduction protocol satisfies the NBAC-Termination, the NBAC-Agreement and the NBAC-Validity properties. More information on the relations between the NBAC problem and the Consensus problem can be found in [13,14,16,17,30,31].

3 The Crash/no Recovery Model

3.1 Good and Bad Processes

We consider here the Crash/no Recovery model. This model is characterized by the following process behavior: when a process crashes, it permanently stops working, and a process that does not crash always follows its specification. So, in this model, a *good* process is a process that never crashes, and (as in any model) a *bad* process is a process that is not good. From a practical point of view, this means that a good process does not crash during the execution of the Consensus algorithm. A process that crashes is a *bad* process. Moreover, this section assumes that each pair of processes is connected by a reliable channel. Roughly speaking, a reliable channel ensures that no message is created, corrupted or duplicated by the channel, and that any good process eventually receives every message sent to it. Readers interested by a theoretical classification of problems in the Crash/Recovery model can consult [11,21].

3.2 A Fundamental Impossibility Result

A fundamental result on the Consensus problem has been proved by Fischer, Lynch and Paterson [9]. This result states that it is impossible to design a deterministic Consensus algorithm in an asynchronous distributed system subject to even a single process crash failure.

The intuition that underlies this impossibility result lies in the impossibility, in an asynchronous distributed system, to safely distinguish between a crashed process, a very slow process, and a process with which communications are very slow.

This impossibility result has been misunderstood by a large community of system implementors [19], but has challenged other researchers to find a set of minimal assumptions that, when satisfied by an asynchronous distributed system, makes the Consensus problem solvable in this system. Minimal synchronism [6], partial synchrony [8] and unreliable failure detectors [3] constitute answers to this challenge. In this chapter, we consider the unreliable failure detectors formalism.

3.3 Unreliable Failure Detectors

The unreliable failure detectors formalism, introduced by Chandra and Toueg in [3], is a powerful abstraction for designing and building reliable distributed applications. Conceptually, a failure detector is a distributed oracle which provides processes with an approximate view of the process crashes occurring during the execution of the system. With respect to its structure, a failure detector is usually seen and used as a set of n, one per process, failure detector modules. These modules are responsible for providing their associated processes with the set of processes they currently *suspect* to have crashed. When the failure detector module of process p_i suspects p_j to have crashed, we say that p_i *suspects* p_j.

Due to asynchrony, and consistently with the impossibility result of Section 3.2, it is natural to expect the failure detector to make mistakes: a failure detector may not suspect a bad (crashed) process or, erroneously suspect a good one. However, to be useful, failure detectors have to eventually provide some correct information about process crashes during the execution and thus, their mistakes are typically bounded by a *completeness* and an *accuracy* properties. The completeness property requires bad processes to be eventually suspected, and accuracy restricts the erroneous suspicions of good processes. Combining different definitions for the completeness and accuracy properties, several classes of failure detectors can be defined [3]. In the following we consider the class of *Eventual Strong* failure detectors, which is denoted by $\diamond S$ and defined by:

- Strong completeness: Eventually every bad process is permanently suspected by every good process.
- Eventual weak accuracy: Eventually some good process is never suspected by any good process.

Note that, in practice, strong completeness can be easily satisfied using "I am alive" messages and timeouts. On the other hand, even if eventual weak accuracy might be satisfied by some executions, it cannot be ensured that it will be satisfied by all executions. This observation shows the limit of asynchronous systems, as far as crash detection is concerned: there is no mean to ensure accurate process crash detection.

3.4 Consensus Algorithms Based on Unreliable Failure Detectors

The first Consensus algorithm designed to work with a failure detector belonging to the class $\diamond\mathcal{S}$ was proposed by Chandra and Toueg [3]. Since then, other algorithms based on $\diamond\mathcal{S}$ have been proposed: one of them has been proposed by Schiper [32], another one by Hurfin and Raynal [22]. All these algorithms share the following design principles:

- The algorithm is based on the *rotating coordinator* paradigm and proceeds in consecutive asynchronous rounds. Each round is coordinated by a process. The coordinator of round r, process p_c, is a predetermined process (*e.g.*, $c = (r \bmod n) + 1$).

- Each process p_i manages a local variable est_i that represents p_i's current estimate of the decision value (initially, est_i is the value v_i proposed by p_i). This value is updated as the algorithm progresses and converges to the decision value.
- During a round r, the coordinator proposes its estimate est_c as the decision value. To this end processes have to cooperate:
  - Processes that do not suspect p_c to have crashed, eventually receive its proposal and *champion* it, adopting est_c as their own estimate of the decision. The proposal of the coordinator becomes the decision value as soon as a majority of processes champion it. The termination of the algorithm directly depends on the *accuracy* property of $\diamond S$ which ensures that, eventually, there is a round during which the coordinator is not suspected by any good process.
  - The crash of the coordinator is dealt by moving to the next round (and coordinator). By the *completeness* property of $\diamond S$, if the coordinator crashes, every good process eventually suspects the coordinator. When this happens, processes *detract* the coordinator's proposal and proceed to the next round.

It is possible that not all processes decide in the same round, depending on the pattern of process crashes and on the pattern of failure suspicions that occur during the execution. One important point which differentiates the algorithms is the way they solve this issue, while ensuring that there is a single decision value (*i.e.*, without violating the agreement property of Consensus).

Other differences between these Consensus algorithms lie in the message exchange pattern they generate and in the way they use the information provided by the failure detector. Chandra-Toueg's algorithm is based on a centralized scheme: during a round all messages are from (to) the current round coordinator to (from) the other processes. In Schiper's and Hurfin-Raynal's algorithms, the message exchange pattern is decentralized: the current coordinator broadcasts its current estimate to all processes, and then those cooperate in a decentralized way to establish a decision value. An important difference between Schiper's algorithm and Hurfin-Raynal's algorithm is the way each algorithm behaves with respect to failure suspicions. Basically, a design principle of Schiper's algorithm is not to trust the failure detector: a majority of processes must suspect the current coordinator to allow a process to proceed to the next round, and to consider another coordinator. Differently, a basic design principle of Hurfin-Raynal's algorithm is to trust the failure detector. Consequently, Hurfin-Raynal's algorithm is particularly efficient when the failure detector is reliable. Schiper's algorithm resists in a better way to failure detector mistakes.

What makes these algorithms far from being trivial is the fact that they can tolerate an unbounded number of incorrect failure suspicions, while ensuring the agreement property of the Consensus problem. This is particularly important from a practical point of view, as it allows to define aggressive time-out values, that might be met only whenever the system is stable, without having the risk of violating the agreement property during unstable periods of the system.

Finally, the algorithms satisfy the validity and agreement properties of Consensus despite the number of bad processes in the system, and satisfy termination whenever a majority of processes are good and the failure detector is of class $\diamond S$.

The S class includes all failure detectors that satisfy strong completeness and *perpetual* weak accuracy (this means that, from the beginning of the system excution, there is a correct process that is never suspected). A generic Consensus algorithm that works with S (whatever the number of crashes is) and with $\diamond S$ (when a majority of processes is correct) has been proposed by Mostefaoui and Raynal [26]. This surprisingly simple generic algorithm is based on the use of quorums. Its respective instantiations in systems equipped with S and with $\diamond S$ require only to modify the quorum definition.

3.5 Other Fundamental Results

Three important results are associated with the class $\diamond\mathcal{S}$ of failure detectors:

- Chandra, Hadzilacos and Toueg [4] showed that the $\diamond S$ class is the weakest class of failure detectors allowing to solve Consensus. This indicates that, as far as the detection of process crashes is concerned, the properties defined by $\diamond\mathcal{S}$ constitute the borderline beyond which the Consensus problem cannot be solved.
- Chandra and Toueg [3] proved that a majority of processes must be good (*i.e.*, must not crash) to solve Consensus using failure detectors of the $\diamond\mathcal{S}$ class.
- Guerraoui [13] proved that any algorithm that solves Consensus using failure detectors of the class $\diamond\mathcal{S}$, also solves Uniform Consensus.

4 On Channel Semantics

The algorithms mentioned in Section 3 assume reliable channels [3,22,32]. However, a reliable channel is an abstraction whose implementation is problematic. Consider for example a reliable channel between processes p_i and p_j. If p_i sends message m to p_j, and crashes immediately after having executed the send primitive, then p_j eventually receives m if p_j is good (*i.e.*, does not crash). This means that the channel is not allowed to lose m because retransmission of m is not possible since p_i has crashed. Indeed, the reliable channel abstraction assumes that the underlying communication medium does not lose a single message, which is an unreasonable assumption given the *lossy* communication channels offered by existing network layers.

It turns out that the algorithms in [3,22,32] are correct with a weaker channel semantics, which is sometimes called *eventual* reliable channel[1]. An eventual reliable channel is *reliable* only if both the sender and the receiver of a message

[1] Also *quasi-reliable* channel, or *correct-restricted* reliable channel.

are good processes. Implementation of eventual reliable channels is straightforward. Messages are buffered by the sender, and retransmitted until they are acknowledged by the receiver. However, what happens if the destination process crashes? If the system is equipped with a perfect failure detector (a failure detector that does not make mistakes), then the sender stops retransmitting messages once it learns that the receiver has crashed. If the failure detector is unreliable, the sender has to retransmit messages forever, which might require unbounded buffer space!

Fortunately, a weaker channel semantics, called *stubborn channels*, is sufficient for solving Consensus [15]. Roughly speaking, a k-stubborn channel retransmits only the k most recent messages sent through it. Contrary to reliable channels or eventual reliable channels, a stubborn channel may lose messages if the sender is a good process. It is shown in [15] that Consensus can be solved with 1-stubborn channels and $\diamond\mathcal{S}$ failure detectors, and that the required buffer space is logarithmically bounded by the number of rounds of the algorithm.

Being able to solve Consensus in the Crash/no Recovery model with lossy channels is a first step towards solving Consensus in the Crash/Recovery model (Section 6). Indeed, solving Consensus in the Crash/Recovery model, among other difficulties requires to cope with the loss of messages. To illustrate the problem consider a message m sent by p_i to p_j and assume that p_j crashes and afterwards recover from the crash. If m arrives at p_j while p_j is crashed, then p_j cannot receive m, *i.e.*, m is lost. If p_j never recovers then the loss of m is not a problem. This is no more the case if p_j eventually recovers. Notice that in this case the loss of m is not the fault of the channel. However, the reason for the loss of the message does not make any difference for the Consensus algorithm.

5 Crash/no Recovery Model *vs* Crash/Recovery Model

While in Section 2 we have defined *one* instance of the Consensus problem, in a real system Consensus is a problem that has to be solved multiple times. Solving multiple instances of the Consensus problem is called *Repeated Consensus*. Repeated Consensus allows us to clarify the difference between the Crash/no Recovery model and the Crash/Recovery model.

In the context of Repeated Consensus, let us consider instance $\#k$ of the Consensus problem. In the Crash/no Recovery model a process p_i that crashes while solving Consensus $\#k$ is excluded forever from Consensus $\#k$, even if p_i recovers before Consensus $\#k$ is solved[2]. Notice that this does not prevent process p_i from learning the decision of Consensus $\#k$, neither does this prevent p_i from taking part in Consensus $\#(k+1)$. In contrast, in the Crash/Recovery model a process p_i that crashes while solving Consensus $\#k$ remains allowed to take part in Consensus $\#k$ after its recovery. Of course, this helps only if Consensus $\#k$ is not yet solved when p_i recovers. This is typically the case

[2] This can easily been achieved making p_i to exclude itself from actively participating in the algorithm upon recovery.

whenever the crash of p_i prevents the other processes from solving Consensus $\#k$.

As an example, consider a Consensus algorithm that requires a majority of processes to take part in the algorithm (let us call such an algorithm *Maj-C-Algorithm*), and the case in which three processes ($n = 3$) have to solve Consensus $\#k$. If we assume that no more than one single process crashes during the execution of Consensus $\#k$, a Maj-C-Algorithm based on the Crash/no Recovery model is perfectly adequate. However, if we admit now that *more* than one process crashes, Consensus $\#k$ is not solvable with a Maj-C-Algorithm based on the Crash/no Recovery model. Such an algorithm leads the whole system to block whenever a majority of processes crash: (1) the surviving process cannot solve Consensus alone, (2) waiting for the recovery of the crashed processes would not help, and (3) if Consensus $\#k$ cannot be solved, none of the subsequent instances of Consensus $\#(k + 1)$, $\#(k + 2)$, etc., will ever be launched.

To overcome the above situation, an algorithm based on the Crash/Recovery model is required. With such an algorithm, the assumption of failure free processes can be released and processes that recover are allowed to actively participate in the instance of Consensus being currently solved. These advantages have certainly a price: apart from the issue of message loss (Section 4), appropriate failure detectors have to be defined, and stable storage becomes necessary.

6 The Crash/Recovery Model

6.1 Good and Bad Processes

In this model, a process can recover after a crash. So, this new particularity of a process behavior has to be taken into account to define what is a *good/bad* process. Actually, according to its crash pattern, a process belongs to one of the following four categories:

- AU: the set of processes that never crash (AU stands for "Always Up").
- EAU: the set of processes that eventually recover and no longer crash (EAU stands for "Eventually Always Up").
- AO: the set of processes that crash and recover infinitely often (AO stands for "Always Oscillating").
- EAD: the set of processes that (maybe after a finite number of oscillations) eventually crash and no longer recover (EAD stands for "Eventually Always Down").

Let us observe that processes that (after several possible recoveries) remain permanently crashed, can be detected. So, the set EAD can eventually be detected. More difficult to manage is the set of processes that forever oscillate between up and down. Actually, it is possible that those processes be never up "long enough" to contribute to the computation of a decision value. Moreover, on one side, it is possible that their "oscillations" perturbe the computation of correct processes. On the other side, it is also possible that, due to the unpredictability

of the crash and communication patterns occurring during an execution, an AO process is always down whenever a message is delivered to it. In this scenario the process is is unable to contribute to the progress of the algorithm.

The previous discussion provides us with an insight sufficient to define which processes have to be considered as *good* (resp. *bad*) in the Crash/Recovery model. *Good* processes are those that are eventually up during a period of time "long enough" to allow Consensus to be solved. So, *good* processes are those of the set $AU \cup EAU$. Consequently, the set of *bad* processes is the set $AO \cup EAD$. Let us finally note that, as in the Crash/no Recovery model, the relevant period during which process crashes are observed spans only the execution of the Consensus algorithm (this gives its practical meaning to the words "long enough period" used previously).

6.2 Failure Detection

Solving Consensus in the Crash/Recovery model requires the definition of appropriate failure detectors. From a practical point of view, it is unreasonable to assume failure detectors satisfying *strong completeness* (such as those in the $\diamond\mathcal{S}$ class) in the presence of processes that crash and recover infinitely often (processes in the AO set)[3]. Recall that *strong completeness* requires good processes to eventually suspect bad processes *permanently* which would imply to safely[4] eventually distinguish between EAU and AO processes. Since there is no bound for the number of times a process may crash and recover, this distinction would mean predicting the crash pattern of the process.

We present here the $\diamond\mathcal{S}_r$ class of failure detectors [28]. $\diamond\mathcal{S}_r$ differs from $\diamond\mathcal{S}$ in the completeness property. Any failure detector of the class $\diamond\mathcal{S}_r$ satisfies *Eventual weak accuracy* and the following completeness property:

- Recurrent strong completeness: Every bad process is infinitely often suspected by every good process[5].

As with $\diamond\mathcal{S}$ failure detectors, completeness can be realized by using "I am alive" messages and timeouts for detecting EAD processes. Detecting AO processes however requires a different scheme. It can be accomplished by having each process to broadcast a "I recovered" message each time the process recovers from a crash. It is worth to notice that those control messages are handled by each process failure detector module which is part of the process and thus subject to its crash pattern.

[3] In [7], the defined Crash/Recovery model does not consider AO processes which allows the adoption of $\diamond S$ failure detectors.

[4] Without compromising the accuracy property of the failure detector.

[5] Let us note that this property is weaker than reclaiming eventual "permanent suspicion".

6.3 Stable Storage

In practice, processes have their state on local volatile memory whose contents is lost in the event of a crash. To overcome this loss and to be able to restore their state when recovering from crashes, processes need to be provided with some sort of stable storage.

Access to stable storage is usually a source of inefficiency and should be avoided as much as possible. Therefore, a pertinent question is whether Consensus can be solved in the Crash/Recovery model without using stable storage at all? This question has been answered by Aguilera, Chen and Toueg [1]. Let $x = |AU|$ (number of processes that never crash) and $y = |AO \cup EAD|$ (number of bad processes). They have shown that Consensus can be solved without using stable storage if and only if $x > y$. Intuitively, this means that the number of processes that (with the help of their volatile memory) can simulate a "stable storage for the whole system" (this number is x) has to exceed some threshold (defined by y) for a physical stable storage be useless.

This result shows that, even without resorting to stable storage, it is possible (when $x > y$) to solve Consensus in the presence of transient process crashes (with complete loss of state) which otherwise would not be possible with algorithms designed for the Crash/no Recovery model. On the other hand, allowing any good process to crash and recover at least once, requires processes to periodically log critical data. When and what data needs to be logged obviously depends on each particular algorithm. Critical process data that has invariably to be persistent to crashes is data contributing to the decision, that is, data which reflects a championed or detracted proposed estimate of the decision.

6.4 Algorithms

An algorithm for solving Consensus in the Crash/Recovery model without requiring stable storage has been proposed in [1]. This algorithm is bound to the requirement of stable storage (Section 6.3) and thus, to terminate, requires that the number of processes that never crash be greater than the number of bad processes ($|AU| > |AO \cup EAD|$).

Several Consensus algorithms suited to Crash/Recovery distributed systems equipped with a stable storage have been proposed [1,23,27]. They tolerate the crash and recovery of any process, and allow recovering process to take part in the computation.

These algorithms borrow their design principles from the Consensus algorithms for the Crash/no Recovery model [3,22,32]. All algorithms require a majority of good processes and rely on the semantics of stubborn communication channels. Apart from their structure, their major differences lie in the failure detectors they assume and on the use processes make of stable storage. The algorithms of Oliveira, Guerraoui and Schiper [27] and Hurfin, Mostefaoui and Raynal [23] were designed using failure detectors satisfying *strong completeness* and can be proved correct with failure detectors satisfying *Recurrent strong completeness* [28]. The algorithm of Aguilera, Chen and Toueg [1] uses a hybrid

failure detector which satisfies *strong completeness* regarding EAD processes and handles the detection of AO processes by providing an estimate count of the number of recoveries of all processes.

With regards to stable storage, the algorithms described in [1,27] require each process to log critical data in every round. The algorithm in [23] is particularly efficient since each process accesses its stable storage at most once during a round.

7 Conclusion

The Consensus problem is a fundamental problem one has to solve when building reliable asynchronous distributed systems. This chapter has focused on the definition of Consensus and its solution in two models: the Crash/no Recovery model and the more realistic Crash/Recovery model. Theoretical results associated with Consensus have also been presented. A fundamental point in the study of the Consensus problem lies in the *Non-Blocking* property. An algorithm is non-blocking if the good (non-faulty) processes are able to terminate the algorithm execution despite bad (faulty) processes. The termination property of the Consensus problem is a non-blocking property. From a theoretical point of view, there are two main results associated with the Consensus problem. The first is due to Fischer, Lynch and Paterson who proved that there is no deterministic non-blocking Consensus algorithm in a fully asynchronous distributed system. The second one is due to to Chandra, Hadzilacos and Toueg who have exhibited the minimal failure detector properties (namely, $\diamond\mathcal{S}$) for solving the non-blocking Consensus problem with a deterministic algorithm. From a practical point of view, it is important to understand the central role played by the Consensus problem when building reliable distributed systems.

References

1. Aguilera M.K., Chen W. and Toueg S., Failure Detection and Consensus in the Crash-Recovery Model. In *Proc. 11th Int. Symposium on Distributed Computing (DISC'98, formerly WDAG)*, Springer-Verlag, LNCS 1499, pp. 231-245, Andros, Greece, September 1998.
2. Bollo R., Le Narzul J.-P., Raynal M. and Tronel F., Probabilistic Analysis of a Group Failure Detection Protocol. *Proc. 4th Workshop on Object-oriented Real-time Distributed Systems (WORDS'99)*, Santa-Barbara, January 1999.
3. Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(1):225–267, March 1996 (A preliminary version appeared in *Proc. of the 10th ACM Symposium on Principles of Distributed Computing*, pp. 325–340, 1991).
4. Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, July 1996 (A preliminary version appeared in *Proc. of the 11th ACM Symposium on Principles of Distributed Computing*, pp. 147–158, 1992).

5. Défago X, Schiper A., Sergent N., Semi-Passive Replication. *Proc. 17th IEEE Symp. on Reliable Distributed Systems*, West Lafayette, Indiana, USA, October 1997, pp. 43-50.
6. Dolev D., Dwork C. and Stockmeyer L., On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
7. Dolev D., Friedman R., Keidar I. and Malkhi D., Failure Detectors in Omission Failure Environments. *Technical Report 96-1608*, Department of Computer Science, Cornell University, Ithaca, NY, September 1996.
8. Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
9. Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.
10. Fritzke U., Ingels Ph., Mostefaoui A. and Raynal M., Fault-Tolerant Total Order Multicast to Asynchronous Groups. *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, Purdue University (IN), pp.228-234, October 1998.
11. Fromentin E., Raynal M. and Tronel F., On Classes of Problems in Asynchronous Distributed Systems with Process Crashes. *Proc. 19th IEEE Int. Conf. on Distibuted Computing Systems (ICDCS-19)*, Austin, TX, pp. 470-477, June 1999.
12. Gray J.N. and Reuter A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1070 pages, 1993.
13. Guerraoui R., Revisiting the Relationship between Non-Blocking Atomic Commitment and Consensus. *Proc. 9th Int. Workshop on Distributed Algorithms* (WDAG95), Springer-Verlag LNCS 972 (J.M. Hélary and M. Raynal Eds), Sept. 1995, pp. 87-100.
14. Guerraoui R., Larrea M. and Schiper A., Reducing the Cost for Non-Blocking in Atomic Commitment. *Proc. IEEE 16th Intl. Conf. Distributed Computing Systems*, Hong-Kong, May 1996, pp. 692-697.
15. Guerraoui R., Oliveira R. and Schiper A., Stubborn Communication Channels. *Research Report*, Département d'informatique, EPFL, Lausanne, Switzerland, July 1997.
16. Guerraoui R., Raynal M. and Schiper A., Atomic Commit And Consensus: a Unified View. (In French) *Technique et Science Informatiques*, 17(3):279-298, 1998.
17. Guerraoui R. and Schiper A., The Decentralized Non-Blocking Atomic Commitment Protocol. *Proc. of the 7th IEEE Symposium on Parallel and Distributed Systems*, San Antonio, TX, 1995, pp. 2-9.
18. Guerraoui R. and Schiper A., Total Order Multicast to Multiple Groups. *Proc. 17th IEEE Int. Conf. on Distributed Computing Systems (ICDCS-17)*, Baltimore, MD, 1997, pp. 578-585.
19. Guerraoui R. and Schiper A., Consensus: the Big Misunderstanding. *Proc of the Sixth IEEE Workshop on Future Trends of Distributed Computing Systems*, Tunis, 1997, pp. 183-186.
20. Guerraoui R. and Schiper A., The Generic Consensus Service. *Research Report 98-282*, EPFL, Lausanne, Suisse, 1998. A previous version appeared in *Proc. IEEE 26th Int Symp on Fault-Tolerant Computing (FTCS-26)*, June 1996, pp. 168-177.
21. Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems (Second Edition)*, ACM Press (S. Mullender Ed.), New-York, 1993, pp. 97-145.
22. 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.

23. Hurfin M., Mostefaoui A. and Raynal M., Consensus in Asynchronous Systems Where Processes Can Crash and Recover. *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, Purdue University (IN), pp. 280-286, October 1998.
24. Hurfin M., Macedo R., Raynal M. and Tronel F., A General Framework to Solve Agreement Problems. *Proc. 18th IEEE Symposium on Reliable Distributed Systems*, Lausanne, October 1999.
25. Mostefaoui A., Raynal M. and Takizawa M., Consistent Lamport's Clocks for Asynchronous Groups with Process Crashes. *Proc. 5th Int. Conference on Parallel Computing Technologies (PACT'99)*, St-Petersburg, Springer Verlag LNCS 1662, pp. 98-107, 1999.
26. Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Int. Symposium on Distributed Computing (DICS'99, formerly WDAG)*, Springer-Verlag LNCS 1693, pp. 49-63, Bratislava (Slovakia), September 1999.
27. Oliveira R., Guerraoui R. and Schiper A., Consensus in the Crash/Recovery Model. *Research Report 97-239*, EPFL, Lausanne, Suisse, 1997.
28. Oliveira R., Solving Asynchronous Consensus with the Crash and Recovery of Processes. *PhD Thesis, EPFL Département d'Informatique, 1999* (to appear).
29. Pedone F. and Schiper A., Generic Broadcast. *Proc. 13th Int. Symposium on Distributed Computing (DICS'99, formerly WDAG)*, Springer-Verlag LNCS 1693, pp. 94-108, Bratislava (Slovakia), September 1999.
30. Raynal M., Consensus-Based Management of Distributed and Replicated Data. *IEEE Bulletin of the TC on Data Engineering*, 21(4):31-37, December 1998.
31. Raynal M., Non-Blocking Atomic Commitment in Distributed Systems: A Tutorial Based on a Generic Protocol. *Journal of Computer Systems Science and Engineering*, Vol.14, 1999.
32. Schiper A., Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10:149-157, 1997.