Consensus in Asynchronous Distributed Systems: a Concise Guided Tour

R. Guerraoui*, M. Hurfin†, A. Mostefaoui†, R. Oliveira*, M. Raynal† and A. Schiper*

* EPFL, Département d'Informatique, 1015 Lausanne, Suisse
† 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 can not be trusted by the process when it has to take a system-wide consistent decision. Similarly, *failure occurrences* can not 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 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. And (2), apply the updates in the same order to each copy. The first problem can be solved by using a *reliable multicast* primitive [18]. 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:

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

which enforces the same decision on any process that decides.

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) can be found in [3, 15, 17, 9, 21]. So Consensus can be viewed 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 can not 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 [10]. 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) $\forall\ p_j$ **do** send($vote$) to p_j **end do**;
(2.1) **wait** ((delivery of a vote NO)
(2.2) **or** ($\exists\ p_j$: 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 $\rightarrow v_i :=$ ABORT
(3.3) a process is perceived as bad $\rightarrow v_i :=$ ABORT
(3.4) all votes are YES $\rightarrow v_i :=$ COMMIT
(3.5) **end case**;
(4) propose(v_i); $decision$:=decide(); % Consensus execution %

Figure 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, 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 [11, 14, 24, 25].

3 The Crash/no Recovery Model

3.1 Good and Bad Processes

We consider here the Crash/no Recovery model. When a process crashes, it definitely stops working. So, a *good* process is a process that never crashes. 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.

3.2 A Fundamental Impossibility Result

A fundamental result on the Consensus problem has been proved by Fischer, Lynch and Paterson [8]. 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 and a very slow process or a process with which communications are very slow.

This impossibility result has been misunderstood by a large community of system implementors [16], 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 [5], partial synchrony [7] 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 provide 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 detectors 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, is natural to expect the failure detector to make mistakes: it 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 is satisfied by some system executions, there is no way to guarantee that it will be satisfied by all system executions. This observation shows the limit of asynchronous systems subject to process crashes, as far as crash detection is concerned: there is no mean to ensure safe process crash detection. Ultimately, such a detection can be at best approximate.

3.4 Consensus Algorithms Based on Unreliable Failure Detectors

The first Consensus algorithm designed to work with a failure detector belonging 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 [26], another one by Hurfin and Raynal [19]. 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 (eg., using $c = (r \bmod n) + 1$) process.
- 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 algorithms 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 detectors. 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 detectors' 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 $\diamond S$.

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 can not 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 [11] proved that any algorithm that solves Consensus using failure detectors of the class $\diamond\mathcal{S}$, also solves Uniform Consensus.

4 About Channel Semantics

The algorithms mentioned in Section 3 assume reliable channels [3, 26, 19]. 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, 26, 19] 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 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 [12]. 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 [12] 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

[1] Or *correct-restricted* reliable channel.

illustrate the problem consider a message m sent by p_i to p_j and that p_j crashes and may 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 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 the Crash/Recovery model, according to their crash patterns, processes can be classified into four categories:

- *NC* processes that never crash.

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

- ENC processes that eventually recover and no longer crash
- EO processes that crash and recover infinitely often.
- EC processes that eventually crash and no longer recover.

Particularly different from the process classification in the Crash/no Recovery model is the EO set of processes. These processes indefinitely oscillate between up and down periods and, due to the asynchrony of the model (which makes no assumptions regarding process speeds), one may be tempted to consider EO processes entirely capable of contributing to the computation of a decision value. However, because such a process is infinitely often down, and due to the unpredictability of the crash and communication patterns occurring during an execution, it is possible that the process is down whenever a message is delivered to it. This scenario renders the process unable to receive any message addressed to it and therefore incapable to contribute to the progression of the algorithm.

From the above categories, *good* processes are those in the NC and ENC sets, and *bad* processes those in the EO and EC sets. From a practical point of view, the good processes are the processes that are eventually up during a long enough period of time to allow Consensus to be solved. Bad processes are either eventually crashed forever, or are never up long enough to allow Consensus to be solved. As in the Crash/no Recovery model, the relevant period during which process crashes are observed spans only the execution of the Consensus algorithm.

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 EO set)[3]. Recall that *strong completeness* requires good processes to eventually suspect bad processes *permanently* which would imply to safely[4] eventually distinguish between ENC and EO processes. Since there is no bound for the number of times a ENC process may crash and recover, this distinction would mean predicting the crash pattern of the process.

For the Crash/Recovery model we consider the $\diamond\mathcal{S}_r$ class of failure detectors defined in [23]. $\diamond\mathcal{S}_r$ differs from $\diamond\mathcal{S}$ in the completeness property. Any failure detector of class $\diamond\mathcal{S}_r$ satisfies *Eventual weak accuracy* and

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

As with $\diamond\mathcal{S}$ failure detectors, completeness can be realized by using "I am alive" messages and timeouts for detecting EC processes. Detecting EO 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 these monitoring messages are handled by each process failure detector module which is part of the process and thus subject to its crash pattern.

[3]In [6], the defined Crash/Recovery model does not consider EO processes which allows the adoption of $\diamond S$ failure detectors.

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

Finally, it should be noted that due to the absence of an eventually stable sequence of values from $\diamond\mathcal{S}_r$ failure detectors regarding the suspicion of EO processes, the output of the failure detector module has to be adequately defined so that the sequence of values perceived by the algorithm also satisfies *recurrent strong completeness*.

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 possible. Therefore, a pertinent question is whether can Consensus be solved in the Crash/Recovery model without using stable storage at all? This question has been answered by Aguilera, Chen and Toueg [1] who have proved that, Consensus can be solved without using stable storage provided that the number of processes that never crash ($|NC|$) is greater than the number of bad processes ($|EO \cup EC|$).

This result shows that, even without resorting to stable storage, it is possible to solve Consensus in the presence of transient process crashes (with complete loss of state) which otherwise, with algorithms designed for the Crash/no Recovery model, would not be tolerated. 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 that contributing to the a decision, that is, data which reflects a championed or detracted proposed estimate of the decision.

6.4 Algorithms

Algorithms for solving Consensus in the Crash/Recovery model without requiring stable storage have been proposed in [1]. These algorithms are bound to the results on the requirements of stable storage (Section 6.3) and thus, to terminate, require the number of processes that never crash to be greater than the number of bad processes ($|NC| > |EO \cup EC|$).

Several Consensus algorithms releasing the assumption of processes that never crash (NC) have been proposed in [22, 1, 20]. In practice, albeit the cost of using stable storage, these algorithms are better suited for the Crash/Recovery model as they tolerate the crash and recovery of any process, and allow any recovering process to actively take part of the computation.

These algorithms borrow their design principles from the Consensus algorithms for the Crash/no Recovery model [3, 26, 19]. 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 [22] and Hurfin, Mostefaoui and Raynal [20] were designed using failure detectors satisfying *strong completeness* and can be proved correct with failure detectors satisfying *Recurrent strong completeness* [23]. The algorithm of Aguilera, Chen and Toueg [1] uses a hybrid failure detector which satisfies *strong completeness* regarding EC processes and handles the detection of EO processes by providing an estimate count of the number of recoveries of all processes.

With regards to stable storage, these algorithms all require each process to log critical data in every round. The algorithm in [20] is particularly efficient since stable storage is accessed 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 Realtime 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] 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.

[6] 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.

[7] Dwork C., Lynch N. and Stockmeyer L. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[8] 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.

[9] 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.

[10] Gray J.N. and Reuter A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1070 pages, 1993.

[11] 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.

[12] Guerraoui R., Oliveira R. and Schiper A., Stubborn Communication Channels. *Research Report*, Département d'informatique, EPFL, Lausanne, Switzerland, July 1997.

[13] 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.

[14] 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.

[15] Guerraoui R. and Schiper A., Total Order Multicast to Multiple Groups. *Proc. 17th IEEE Int. Conf. on Distributed Computing Systems (ICDSC-17)*, Baltimore, MD, 1997, pp. 578-585.

[16] 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.

[17] 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.

[18] 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.

[19] Hurfin M. and Raynal M., A Simple and Fast Asynchronous Consensus Based on a Weak Failure Detector. *Research Report 1118*, IRISA, Rennes, (July 1997), 19 pages.

[20] 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.

[21] Hurfin M., Macedo R., Raynal M. and Tronel F., A General Framework to Solve Agreement Problems. *Research Report*, IRISA, Rennes, January 1999.

[22] Oliveira R., Guerraoui R. and Schiper A., Consensus in the Crash/Recovery Model. *Research Report 97-239*, EPFL, Lausanne, Suisse, 1997.

[23] Oliveira R., Solving Asynchronous Consensus with the Crash and Recovery of Processes. *PhD Thesis, EPFL Département d'Informatique, 1999* (to appear).

[24] Raynal M., Consensus-Based Management of Distributed and Replicated Data. *IEEE Bulletin of the TC on Data Engineering*, 21(4):31-37, December 1998.

[25] Raynal M., Non-Blocking Atomic Commitment in Distributed Systems: A Tutorial Based on a Generic Protocol. To appear in *Journal of Computer Systems Science and Engineering*, Vol.14, 1999.

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