Fault-Tolerant Distributed Systems: a Modular Approach to the Non-Blocking Atomic Commitment Problem

Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fault-Tolerant Distributed Systems: a Modular Approach to the Non-Blocking Atomic Commitment Problem

Michel Raynal

N ° 2973

Septembre 1996

THÈME 1

ISSN 0249-6399

INRIA
RENNES

Fault-Tolerant Distributed Systems: a Modular Approach to the Non-Blocking Atomic Commitment Problem

Michel Raynal*

Thème 1 — Réseaux et systèmes
Projet Adp

Rapport de recherche n ° 2973 — Septembre 1996 — 18 pages

Abstract: Agreement problems allow a set of processes to agree on a common output value. These problems are of primary importance in distributed systems and difficult to solve in presence of failures. This paper considers one of these problems whose practical interest is well known, namely the *Non-Blocking Atomic Commitment Problem*. First, a generic protocol solving this problem is given and then instantiations of its generic statements are provided for both synchronous and asynchronous distributed systems. These instantiations use timeout mechanism, reliable multicast primitives and unreliable failure detectors as basic components. Incidentally, this paper can also be considered as an introduction to state-of-the-art concepts and mechanisms of distributed fault tolerance.

Key-words: Atomic Commitment, Consensus Problem, Distributed Systems, Failure Detection, Fault-Tolerant Protocols, Non-blocking Protocols, Reliable Multicast, Transaction.

(Résumé : tsvp)

* IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France. raynal@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

La validation atomique non-bloquante dans les systèmes répartis.

Résumé : Les protocoles non-bloquants de validation atomique constituent un composant essentiel de tout système de gestion de données. Ce rapport en offre une présentation synthétique.
Un modèle général et modulaire est tout d'abord proposé pour de tels protocoles. Ce modèle utilise quatre primitives génériques qui sont ensuite particularisées pour les sytèmes distribués synchrones et pour les sytèmes distribués asynchrones. Ces diverses instances mettent l'accent sur les primtives de communication et sur les mécanismes de détection de défaillances. Il est intéressant de noter que cette étude peut également être vue comme une introduction à un certain nombre de concepts fondamentaux relatifs à la tolérance aux défaillances dans les systèmes répartis.

Mots-clé : Consensus, Détection de défaillances, Diffusion fiable, Protocole non bloquant, Systèmes répartis, Tolérance aux défaillances, Transaction, Validation atomique.

1 Introduction

Systems as diverse as banking, booking-reservations or point-of-sale commerce actually constitute data management systems from a computer engineering or research point of view. Because data are geographically distributed or because they are located at different places to improve availability, data management systems are usually distributed. A fundamental issue in these systems is to ensure that data always remain consistent. This is the aim of the *transaction* concept introduced in the 1970s ([13, 20]). A transaction is a data manipulation unit (1) that, when executed alone, keeps data consistent and (2) to which two atomicity properties are attached. The first one, *concurrency atomicity* ensures that a parallel execution of a set of transactions cannot make data inconsistent: a set of transactions executed in parallel must appear as if these transactions had been executed serially; the underlying theory, called *serializability theory*, has been extensively studied. The second one, called *failure atomicity*, ensures that failures cannot make data inconsistent: all the update operations of a transaction are performed or none of them is performed; this atomicity property is sometimes called *all-or-nothing* property.

In a distributed system a transaction usually involves several sites as participants. At the end of a transaction its participants are required to enter a *commitment protocol* in order to commit it (when everything went well) or to abort it (when something went wrong). This protocol obeys usually a two phase pattern (the so-called Two Phase Commit, in short 2PC). In the first phase each participant votes either YES or NO. If for any reason (deadlock, storage problem, concurrency control conflict, etc) a participant cannot locally commit the transaction, it votes NO; otherwise, a YES vote means the participant commits to locally make updates permanent if it is required to do so. Then the second phase pronounces the order to commit the transaction if all participants voted YES or to abort it if some participant voted NO. Of course this protocol sketch has to be enriched to take into account failures in order to correctly implement the failure atomicity property. The underlying idea is that a failed participant is considered as having voted NO.

Several atomic commitment protocols have been proposed and implemented in various systems ([3, 13]). Unfortunately some of them (such as the 2PC with a main coordinator) are blocking in some failure scenarios ([1, 23]). "Blocking" means that non-failed participants have to wait for the recovery of failed participants in order to terminate their commit procedure, *i.e.*, a commitment protocol is *blocking* if it admits executions in which non-failed participants cannot decide. When such a situation occurs, non-failed participants cannot release resources they acquired for exclusive use on behalf of the transaction. This not only prevents the concerned transaction from terminating but also prevents other transactions from accessing locked data. Thus it is highly desirable to devise non-blocking atomic commitment (in short NBAC) protocols, *i.e.*, protocols that ensure transactions will terminate (by committing or aborting) despite any failure scenario. Several NBAC protocols (called 3PC protocols) have been designed and implemented. Basically they add handling of failure scenarios to a 2PC-like protocol and use complex sub-protocols that make them difficult to understand, program, prove and test. Moreover these protocols assume more or less implicitly that the underlying distributed system is synchronous (*i.e.*, process scheduling delays and message transfer delays are upper-bounded, and these bounds are known and used by the protocols).

This paper focuses on the NBAC problem and on protocols that solve it in synchronous and asynchronous distributed systems[1]. Actually, non-blocking protocols necessitate that the underlying system offers some *liveness guarantee* about failure detection. Synchronous systems provide such a guarantee thanks to the upper bounds on scheduling and message transfer delays (protocols use timeouts to safely detect failures in bounded time). Asynchronous systems do not have such bounded delays, but can be equipped with unreliable failure detectors. A failure detector associated with a site can be seen as an "oracle" that gives it hints on sites or participants it suspects to have crashed. Failure

[1] We do not address in this study the NBAC problem in the context of real-time systems. In those systems deadlines are attached to commitments. The interested reader can consult [9, 25].

detectors can be implemented by using timeouts but their suspicions can be erroneous (as the values they use in timeouts cannot fit scheduling or message transfer delays as those are arbitrary!). Undetected failures and false failure suspicions may cause degradation of performance. However, Chandra and Toueg have shown in [4] that if these unreliable failure detectors satisfy some properties then they provide the liveness guarantee we need to design protocols in unreliable asynchronous distributed systems. So, timeouts in synchronous systems (in this context they implement reliable failure detectors) and unreliable failure detectors in asynchronous systems constitute *building blocks* offering the liveness guarantee from which solutions can be designed. The use of such unreliable failure detectors in the design of asynchronous NBAC protocols has been advocated and used for the first time by Guerraoui and Schiper in [15, 16].

The paper is composed of seven sections. Section 2 exposes the differences between synchronous and asynchronous systems and presents a model for distributed transactions. Section 3 addresses crash failures and their detection in both types of system. Section 4 presents multicast communication primitives used to disseminate information in distributed systems; these primitives will be used to get solutions to the NBAC problem. Section 5 gives a specification of the NBAC problem for both types of system and introduces a generic NBAC protocol. Instantiations of this protocol for synchronous and asynchronous distributed systems are given in Sections 6 and 7, respectively. The design of the generic protocol and of its instances follows the modular approach introduced by Babaoğlu and Toueg in [1] in the context of unreliable synchronous systems. This modular approach has been generalized by Guerraoui and Schiper in [17] where they introduce a methodolgy to define very general Consensus services in the context of unreliable asynchronous distributed systems.

Actually, this paper can be seen as a state-of-the-art introduction (based on previous works described in [1, 4, 14, 16, 17, 19, 23]) to a class of agreement problems and fault-tolerance techniques for distributed systems, with a special emphasis on the NBAC problem.

2 Distributed Systems and Transactions

2.1 Distributed Systems

A *distributed* system is composed of a finite set of sites interconnected through a communication network. Each site has local memory and stable storage and executes one or more processes. To simplify we assume there is only one process per site. Processes communicate and synchronize by exchanging messages through channels of the underlying network. It is the multiplicity of sites connected through a network that makes the system distributed.

A *synchronous* distributed system is characterized by upper bounds on message transfer delays, process scheduling delays and message processing time. An embedded distributed system or a process control system built on top of a local area network constitute examples of synchronous distributed systems. In the following we will call δ an upper time bound for message transfer delay + receiving process scheduling delay + message processing time. Such a value is a constant known by all sites of the system.

Most real distributed systems are asynchronous in the sense that an upper bound δ, such as the previous one, cannot be established ([7]). So, *asynchronous* distributed systems are characterized by the absence of such an a priori known bound; message transfer and process scheduling delays cannot be predicted and are considered arbitrary. Systems covering large geographic areas or subject to unpredictable loads that may be imposed by their users are basically asynchronous as no realistic upper bound δ can be a priori defined. Typically, open distributed systems are asynchronous. Some authors call them *time-free* asynchronous systems ([8]) to insist on the absence of physical time bounds that could be safely used by protocols executed on these systems. Such an absence makes some problems much more difficult to solve in asynchronous systems than in synchronous systems. More importantly, in presence of failures, some problems solvable in synchronous contexts can not be solved in purely

asynchronous ones (see Section 3.3 and [11]), unless these systems (1) be run in a context in which they satisfy some additional properties and (2) be provided with a minimal a priori "common knowledge" that acts as a seed from which a distributed agreement can be obtained.

2.2 Transactions

In a distributed system a transaction can access shared data located at multiple sites. It is assumed that each transaction is a unit of consistency in the following sense: executed alone and in the absence of failures, a transaction transforms data from a consistent global state to another consistent global state. A transaction t is initiated at some site (the initiator) and involves other sites. All these sites (including the initiator) constitute the set of *participants* associated with t (we assume a participant is implemented by a process). As indicated in the introduction, parallel execution of transactions is managed by a concurrency control protocol that ensures that a parallel execution is equivalent to a sequential one. As we focus only on the NBAC problem, we will consider in the following a single transaction; thus transaction attributes such as its unique global identifier will be omitted.

Figure 1 describes the protocol associated with the execution of a transaction t; *trans* is the type of a message whose contents include the transaction body *trans_body* and the set of its *participants* ([1]). This message is sent to each participant by the transaction initiator when the transaction execution is started; this is expressed by a call to the multicast primitive multicast_1 *trans(trans_body, participants)* (line 1) that will be instantiated in an appropriate way in synchronous and in asynchronous systems. When it is delivered the *trans* message, a participant performs local computations as defined in *trans_body* and then prepares its vote. If it can locally commit to make the updates permanent it votes YES; in the other cases (*e.g.*, to solve a deadlock) it votes NO. Then the participant enters the NBAC protocol by calling the procedure nbac. The aim of this protocol is to provide the same outcome decision to all participants so they uniformly commit or abort the transaction (see Section 5).

```
% The initiator executes %
(1)     multicast_1 trans (trans_body, participants)

% All participants (including the initiator) execute %
upon delivery of trans (trans_body, participants) do
(2)     perform operations requested by trans_body;
(3)     if (able to make updates permanent)
            then vote := YES
            else vote := NO
        fi;
(4)     nbac (vote,participants)
```

Figure 1: Distributed Transaction Execution Model

3 Failures in Distributed Systems

3.1 Crash Failures

The underlying communication network is assumed to be reliable, *i.e.*, it does not lose, generate or garble messages. (It is shown in [13] that distributed systems with unreliable communication do not admit solutions to the NBAC problem. This is the famous "Generals Paradox" described in [13].)

A site, or a transaction participant, can fail by crashing. Its state is correct until it crashes. A participant that does not crash during the transaction execution is said to be *correct*; otherwise it is

faulty. As we are only interested in commitment and not in recovery we assume that a faulty participant does not recover. Moreover, whatever the number of faulty participants and the configuration of the network, it is assumed that each pair of correct participants can always communicate.

3.2 Failure Detection in Synchronous Systems

In synchronous systems crashes can be easily detected by using timeout mechanisms. Each site has a real-time local clock (If a site has a local clock whose drift rate with respect to real-time is ρ, then δ has to be replaced by $(1+\rho)\delta$ in timeout delays to compensate the worst drift rate).

It is well-known that in such a context if a participant p does not receive from q a response to a message 2δ time units after its sending, then it can *safely* conclude that q crashed. Moreover, with such a failure detection mechanism, as the network is assumed reliable, the following property holds: before being notified of the failure of q by the timeout mechanism, p will receive all the messages q sent to it before crashing.

3.3 The Consensus Problem

This section defines the *Consensus Problem* ([21]) which is a fundamental problem is distributed systems (see [24] for a survey). This problem is far from being trivial in presence of failures. Solutions to and use of this problem will be addressed in Section 3.4 and in Section 7, respectively.

Let us consider a set of processes that can fail by crashing. A non failed process is said to be correct. Each process has an input value which it proposes to the other processes. The *Consensus Problem* consists in designing a protocol such that all correct processes unanimously and irrevocably decide on a common output value which is one of the input values. More precisely the four following properties specify the Consensus Problem:

- *Termination*: Every correct process eventually decides some value.
- *Integrity*: A process decides at most once.
- *Agreement*: No two correct processes decide differently.
- *Validity*: If a process decides D, then D was proposed by some process.

The *Uniform Consensus Problem* is defined by the same Termination, Integrity and Validity conditions plus the following Agreement property:

- *Uniform Agreement*: No two processes decide differently.

So, in the Uniform Consensus Problem, a correct process and a crashed process that decided just before crashing, cannot decide differently. Uniformity is important for recovery. Let us consider a process that decides and then crashes; if later this process recovers, then, thanks to the Uniform Agreement property, this process will naturally agree with the other correct processes without violating the Integrity property.

Consensus and Uniform Consensus Problems abstract several important practical agreement problems (*e.g.*, duplicated sensors processes which have to display a common output value).

Both problems have relatively simple solutions in synchronous distributed systems. Unfortunately, this is not the case in asynchronous distributed systems where the most famous result is a negative one: the so-called FLP result ([11]) states that it is impossible to design a deterministic protocol solving the Consensus Problem in an asynchronous system even with only a single process crash

failure. Intuitively this is due to the impossibility of safely distinguishing a very slow process from a crashed process in an asynchronous context.

This impossibility result has motivated researchers to discover a set of minimal properties that, when satisfied by an asynchronous distributed system, make the Consensus Problem solvable in this system. Chandra and Toueg's unreliable failure detectors ([4]) with Completeness and Accuracy properties, provide an answer to such a question (see the next section).

Remark. It is possible to design randomized protocols to solve the Consensus Problem. Such protocols use random numbers and consequently are non-deterministic. For those protocols the Termination property can only be ensured with probability 1 ([2, 22]).

3.4 Failure Detection in Asynchronous Systems

As noted in Section 2, delays for message transfer and process scheduling are arbitrary and cannot be bounded in asynchronous systems. Thus, it is not possible to differentiate a transaction participant that is very slow (due to a very long scheduling delay) or whose messages are very slow (due to long transfer delays) from a participant that crashed. This simple observation shows it is impossible to detect failures in a complete and accurate way in asynchronous distributed systems. *Completeness* means a faulty participant will eventually be detected as faulty, while *Accuracy* means a correct participant will not be considered as faulty[2].

It is however possible to equip every site with a failure detector which gives it hints on sites it suspects to be faulty. Such failure detectors can be implemented by suspecting participants that do not answer in a timely fashion. These detectors are inherently unreliable as they can make mistakes by erroneously suspecting a correct participant or by not suspecting a faulty one. In this context Chandra and Toueg [4] have refined the Completeness and Accuracy properties of failure detection and have shown that it is possible to solve the *Consensus Problem* in executions where unreliable failure detectors satisfy some of these properties. These refinements are:

- *Strong (Weak) Completeness* : Eventually every faulty participant is permanently suspected by every (some) correct participant.
- *(Eventual) Weak Accuracy* : (Eventually) There is a correct participant that is never suspected.

According to these properties, Chandra and Toueg have defined several classes of failure detectors. For example the class of "Eventual Weak failure detectors" includes all the failure detectors that satisfy Weak Completeness and Eventual Weak Accuracy. Completeness of failure detections can be met by using timeouts as a faulty participant does not answer. But, as indicated previously, Accuracy can only be approximate as, even if most messages are received within some predictable time, an answer not yet received does not mean that the sender has crashed. So, as far as Accuracy is concerned, implementations of failure detectors can only be "approximate" in purely asynchronous distributed systems. Despite such "approximate implementations", unreliable failure detectors offer some guarantees to upper layer protocols as indicated by the following set of results ([4]).

- Results related to the Consensus Problem according to the number of correct participants. These results provide a theoretical foundation to the following intuitive notion: for the Consensus Problem to have a solution, the smaller the number of correct participants is, the stronger the accuracy of failure detections has to be.

[2] For the *failure detection* problem, *Accuracy* and *Completeness* constitute its safety property and its liveness property, respectively. *Accuracy* indicates the detection must be correct, *i.e.,* only faulty processes must be detected (safety of the detection), while *Completeness* means failures must be detected (liveness of the detection).

R1. If at least one participant is correct, it is possible to solve the Consensus Problem in executions where the underlying failure detectors satisfy Weak Completeness and Weak Accuracy. A Consensus protocol for such executions is described in [4].

R2. If at least a majority of participants are correct, it is possible to solve the Consensus Problem in executions where failure detectors satisfy Weak Completeness and Eventual Weak Accuracy (*i.e.*, in executions where eventually a correct process is no longer suspected by the other correct participants). A Consensus protocol[3] for such executions is described and proved correct in [4].

- Result on the class of failure detectors defined by the Weak Completeness and Eventual Weak Accuracy properties.

R3. This class of failure detectors is the weakest that allows solutions to the Consensus Problem ([5]). Said another way, when Weak Completeness or Eventual Weak Accuracy properties can not be ensured in an asynchronous system, then the Consensus Problem cannot be solved in this system. These properties define precisely the frontier beyond which the Consensus Problem cannot be solved.

R4. Any protocol that uses a failure detector belonging to this class can lose its *liveness* (*i.e.*, it can not terminate) but will not lose its *safety* (*i.e.*, if it gives a result, this result will be correct) when the failure detector malfunctions by failing to meet its properties (*e.g.*, by not detecting a faulty participant or by repeatedly suspecting erroneously correct participants). If, after a period of malfunction, failure detectors meet again their properties for a sufficiently long period, then the upper layer protocol will regain its liveness (*e.g.*, it will decide a value in the case of the Consensus Problem). This result is very important from a practical point of view[4].

4 Dissemination of Information in Distributed Systems

The definition of appropriate communication primitives constitutes a fundamental point in the design of distributed systems. Among these primitives multicast is particularly important as it allows a process to disseminate a message m to a set $P = \{p_1, p_2, ..., p_n\}$ of processes. We consider here three multicast primitives (see [1, 19] for in-depth study of such communication primitives).

Unreliable multicast: *Multisend (m,P).* Actually this primitive is a syntactical notation that abstracts:

for each $p \in P$ **do** *send(m)* **to** p **end do**

where *send* is the usual point-to-point communication primitive. *Multisend (m,P)* is the simplest multicast primitive[5]. Its main drawback lies in the fact it is not fault-tolerant: if the sender crashes after having sent m to a subset P' of P, then all processes belonging to P but not to P' will not be delivered the message m.

[3] Appendix A2 gives a sketch of these two protocols.

[4] This means that if timers are appropriately set and the system *stabilizes* during a sufficiently long period, the properties required by "Eventual Weak failure detectors" are satisfied. In such cases protocols using those failure detectors will deliver their results in finite time ([10]).

[5] Communication primitives similar to *Multisend (m,P)* have been used in the V-kernel ([6]) and in the Parallel Virtual Machine package PVM ([12]).

Reliable multicast in asynchronous systems: *AS_Rel_Multicast(m,P).* The aim of this primitive is to reliably send a message m to all processes of P in an asynchronous system. Basically it adds an "all-or-none atomicity" property on deliveries to the *Multisend* primitive. This property is usually called Uniform Agreement. More formally this primitive is defined by the following set of properties:

- *Termination*: If a correct process multicasts a message m to P, then some correct process of P eventually delivers m (or all processes of P are faulty).
- *Validity*: if a process p delivers a message m, then m has been multicast to a set P and p belongs to P (there is no spurious message).
- *Integrity*: A process p delivers a message m at most once (no duplication).
- *Uniform Agreement*: If any (correct or not) process of P delivers message m, then all correct processes of P deliver m.

This primitive can be easily implemented in asynchronous systems where partitions are eventually repaired. When a process receives a message m for the first time, it first forwards m to the other processes of P and only then considers m's delivery. It is easy to see this protocol guarantees Uniform Agreement as, if a message m is delivered to a process that crashed after the delivery, this process has forwarded m before. The forwarding can be done by using the underlying network primitive (*e.g.*, by repeatedly using a simple point-to-point *send* primitive to send a copy of the message to each destination process). Interested readers may consult [19] for specifications of reliable multicast and broadcast primitives and [19, 17] for efficient implementations.

Reliable multicast in synchronous systems: *S_Rel_Multicast(m,P).* This primitive sends reliably a message m to a set of processes P in a synchronous system; it is defined by the four previous properties (Termination, Validity, Integrity and Uniform Agreement) plus the following one:

- *Timeliness*: There is a time constant Δ such that, if the multicast of m is initiated at real-time T, then no process delivers m after $T+\Delta$.

The implementation of this primitive is similar to the previous one. Let f be the maximum number of processes that may crash; δ is the a priori known upper bound defined in Section 2.1. It is possible to show that $\Delta = (f+1)\delta$. Several message- and time-efficient implementations of *S_Rel_Multicast(m,P)* are described in [1].

Weakening reliable multicast. In some applications the last two previous reliable multicast primitives are stronger than necessary in the following sense. These applications require the same Termination, Validity and Integrity properties (plus Timeliness if the system is synchronous) but a weaker Agreement. Only correct processes have to agree and so Agreement is no more Uniform ([19]):

- Simple Agreement: If a correct process of P delivers a message m, then all correct processes of P deliver m.

5 The Non-Blocking Atomic Commitment Problem

5.1 Problem Definition

As described in the introduction the NBAC problem consists of ensuring that all correct participants of a transaction take the same decision, namely commit or abort the transaction. If the decision is COMMIT then all participants make their updates permanent; if the decision is ABORT then no change

is operated on the data (the transaction has no effect). The COMMIT/ABORT value of the outcome of the NBAC protocol depends on the votes of participants and on failures. More precisely the NBAC problem is defined by the following set of properties.

- *Termination*: Every correct participant eventually decides.
- *Integrity*: A participant decides at most once.
- *Uniform Agreement*: No two participants decide differently.
- *Validity*: If a participant decides COMMIT then all participants have voted YES.
- plus a *Non-Triviality* property.

The aim of the Non-Triviality condition is to eliminate "non-expected" solutions in which the decision would be independent of votes (*e.g.*, always deciding on ABORT) and of failure scenarios. If all participants voted YES and "everything went well" the decision must be COMMIT. The sentence "everything went well" is related to failures. As failures can be safely detected in synchronous systems, the Non-Triviality property for these systems is ([18]):

- *S-Non-Triviality*: If all participants vote YES and there is no failure then the outcome decision is COMMIT.

As in asynchronous systems failures can only be suspected, possibly erroneously, this condition has to be weakened for the problem to be solvable ([14]). So, for these systems the Non-Triviality becomes:

- *AS-Non-Triviality*: If all participants vote YES and there is no failure suspicion then the outcome decision is COMMIT.

As we can see, the uncertainty inherent to asynchronous distributed systems (does a suspicion indicate a failure?) does not exist in synchronous systems where timeout mechanisms always allow complete and accurate failure detection.

5.2 A Generic NBAC Protocol

This section describes a generic protocol for the NBAC problem. As indicated in the Introduction, this protocol follows the modular approach advocated in [17]. It is described by the procedure nbac *(vote,participants)* introduced in Section 2.2. Each participant that has not crashed when it arrives at line 4 of Figure 1 calls this procedure with its vote and the set of participants as parameters. This procedure is described in Figure 2. It uses three generic statements (multicast_2, exception, propose) whose instantiations are specific to synchronous or to asynchronous distributed systems. Each participant has a variable *outcome* that will locally indicate the final decision (COMMIT or ABORT) at the end of the protocol execution.

A participant p first sends its vote to all participants (including itself) by using the multicast_2 statement (Figure 2, line 1.1). Then p waits until[6]: (1) either it has been delivered a vote NO from a participant (line 2.1) or (2), it has been notified of an exception concerning a participant q (lines 2.2) or (3) it has been delivered a vote YES from each participant (line 2.3). At line 2.2 the notification exception(q) concerns q's failure and will be instantiated in an appropriate manner in each type of system. Finally, according to the votes and the exception notifications it received, the participant p

[6] Such a use of the **wait** statement (on an **or** of several conditions) has been abstracted in [17] under the name *filter* and used as a building block to implement very general *Consensus Services*.

executes the statement *outcome:=* propose*(x)* (lines 3.1-3.5) with COMMIT as the value of the parameter x if a vote YES has been received from all participants, and ABORT in all other cases.

In this protocol the control is distributed ([23]): each participant sends its vote to all participants and there is no main coordinator. When compared to a central coordinator-based protocol, it requires more messages but less communication phases and so, reduces latency.

```
procedure nbac (vote,participants)
begin
(1.1) multicast_2 (vote, participants);
      % the message is sent to all participants, including its sender%
(2.1) wait ( (delivery of a vote NO from a participant)
(2.2)       or (∃ q ∈ participants: exception(q) has been notified to p)
(2.3)       or (from each q ∈ participants: delivery of a vote YES)
(2.4)       );
(3.1) case
(3.2)       a vote NO has been delivered  → outcome := propose (ABORT)
(3.3)       an exception has been notified → outcome := propose (ABORT)
(3.4)       all votes are YES             → outcome := propose (COMMIT)
(3.5) end case
end
```

Figure 2: A Generic NBAC Protocol

6 A Protocol for Synchronous Systems

6.1 The Protocol

Table 1 gives instantiations for the four generic statements, multicast_1 (Figure 1), multicast_2, exception and propose (Figure 2), that provide a NBAC protocol for synchronous systems. The resulting protocol is described by the procedure *S_nbac* (Figure 3) executed by each correct participant that received the *trans* message.

Generic Statement	Instantiation	See Comment
multicast_1	*Multisend*	S_C1
multicast_2	*S_Rel_Multicast*	S_C2
exception	*timer expiration*	S_C3
propose*(x)*	x	S_C4

Table 1: Instantiations for Synchronous Systems

S_C1 In a synchronous system it is not necessary that the primitive multicast_1 used in Figure 1 be reliable, as a correct participant can detect the crash of the sender by using a timer.

S_C2 *S_Rel_Multicast* ensures that if a correct participant is delivered a vote sent at time T, then all correct participants will deliver this vote by time $T + \Delta$.

S_C3 As in a synchronous system the crash of a participant q can be safely detected, the exception associated with q is raised if q crashed before sending its vote. This is implemented by using a

single timer for all the possible exceptions (there is one possible exception per participant): p sets a timer to $\delta + \Delta$ when it sends its vote (with the *S_Rel_Multicast* primitive); if the timer expires before a vote from each participant has been delivered, then p can safely conclude that participants from which votes have not been delivered have crashed[7].

S_C4 In this case the function propose is simply the identity function, *i.e.,* propose$(x)=x$; so *outcome* := propose(x) is instantiated by *outcome* := x.

```
procedure S_nbac (vote,participants)
begin
(1.1) S_Rel_Multicast (vote,participants);
(2.0) set timer to δ + Δ;
(2.1) wait ( (delivery of a vote NO from a participant)
(2.2)       or (timer expiration)
(2.3)       or (from each q ∈ participants: delivery of a vote YES)
(2.4)       );
(3.1) case
(3.2)       a vote NO has been delivered → outcome := ABORT
(3.3)       the timer expired            → outcome := ABORT
(3.4)       all votes are YES            → outcome := COMMIT
(3.5) end case
end
```

Figure 3: A NBAC Protocol for Synchronous Systems

6.2 Why Does it Work?

The protocol described in Figure 3 satisfies the five conditions defined in Section 5.1 and consequently solves the NBAC problem in synchronous systems.

- Termination: direct consequence of the use of timers (lines 2.0,2.2). No participant waits more than $\delta + \Delta$.
- Integrity (a participant decides at most once): follows directly from the protocol structure.
- Validity (if a participant decides COMMIT, all voted YES): follows directly from line 3.4.
- S-Non-Triviality: follows from the fact that if no participant crashes then all participants send their votes which are delivered to a participant p before p's timer expires (see SC_3). It follows that if all votes are YES, p's outcome is COMMIT (line 3.4).
- Uniform Agreement (no two participants decide differently). Suppose two participants decide differently: the outcome of p is COMMIT while the outcome of q is ABORT. It follows that p decided at line 3.4 (with all votes being YES) and q decided at line 3.2 or 3.3. It is not possible for q to decide at line 3.2 as any participant sent the same vote (namely YES) to p and q. So q decided at line 3.3, because its timer expired. This means q did not receive a vote from some participant r by $\delta+\Delta$ after setting its timer. But in the worst case r has been delivered the *trans*

[7] This is because (1) δ is an upper bound for the time elapsed between the start of the transaction (sending of the *trans (,)* message, Figure 1) and the sending of its vote by a participant and (2) Δ is an upper bound for the time elapsed between the sending of its vote by any participant p and the corresponding delivery by any participant q.

message (Figure 1) δ time units after it has been delivered at q. So, as (1) p has been delivered a vote from r (whose value is YES), (2) the Uniform Agreement property of the *S_Rel_Multicast* primitive used by r to send its vote and (3) the delivery upper bound Δ associated with this primitive, it follows that q has been delivered the vote of r before its timer (initialized to $\delta + \Delta$) expired. A contradiction.

7 A Protocol for Asynchronous Systems

7.1 The Protocol

Table 2 gives the four instantiations that provide a NBAC protocol for asynchronous systems. This protocol is described by the procedure *AS_nbac* (Figure 4) executed by each correct participant.

Generic Statement	Instantiation	See Comment
multicast_1	*AS_Rel_Multicast*	AS_C1
multicast_2	*Multisend*	AS_C2
exception	*failure suspicion*	AS_C3
propose*(x)*	*Unif_Cons(x)*	AS_C4

Table 2: Instantiations for Asynchronous Systems

AS_C1 In order all correct participants start the *AS_nbac* procedure (and so, execute the *Unif_Cons* protocol, see AS_C4) multicast_1 is instantiated by the *AS_Rel_Multicast* primitive.

AS_C2 If a participant crashes during the execution of *Multisend* it will be suspected by any failure detector that satisfies the Completeness property.

AS_C3 When the failure detector associated with participant p suspects (possibly in an erroneous way) a participant q has crashed, it raises the exception associated with q by setting a local boolean flag *suspected(q)* to the value *true*[8]. Note that if failure detectors satisfy the Completeness property, then all participants that crashed before sending their vote will be suspected (This observation will be used to show the protocol terminates).

AS_C4 The function propose is instantiated by any sub-protocol solving the Uniform Consensus problem (see Section 3.3). Let *Unif_Cons* be such a protocol; it is executed by all correct participants[9]. As indicated in Section 3.3 (R1, R2), two such protocols based on failure detectors are described in [4]. When there are neither failures nor failure suspicions, the protocol working with the assumptions described in R2 requires *3(n-1)* messages plus an *AS_Rel_Multicast* to provide the set of participants with a decision value.

Actually the protocol described in Figure 4 "reduces" the NBAC problem to the Uniform Consensus problem. This means that, in asynchronous distributed systems, if we have a solution for the Uniform Consensus problem, we can use it to solve the NBAC problem ([14]). Schiper and Guerraoui have been the first to propose such a reduction; in [16] they introduced and advocated the use of Consensus protocols to ensure proper termination of asynchronous NBAC protocols.

[8] Note that the boolean flag *suspected(q)* can alternate between *true* and *false* according to the current state of the suspicion made about q by the failure detector associated with p, *i.e.*, failure suspicions are not necessarily *stable*.

[9] It is important to note that all correct participants must propose a value for the protocol *Unif_Cons* to terminate; this is ensured when all correct participants start the *AS_nbac* procedure (hence AS_C1) and failure detectors satisfy the Completeness property.

```
procedure AS_nbac (vote,participants)
begin
(1.1) Multisend (vote,participants);
(2.1) wait ( (delivery of a vote NO from a participant)
(2.2)       or (∃ q ∈ participants: suspected(q))
(2.3)       or (from each q ∈ participants: delivery of a vote YES)
(2.4)       );
(3.1) case
(3.2)       a vote NO has been delivered → outcome := Unif_Cons (ABORT)
(3.3)       a participant has been suspected → outcome := Unif_Cons (ABORT)
(3.4)       all votes are YES            → outcome := Unif_Cons (COMMIT)
(3.5) end case
end
```

Figure 4: A NBAC Protocol for Asynchronous Systems

7.2 Why Does it Work?

The *AS_nbac* protocol satisfies the five properties defining the NBAC problem in asynchronous distributed systems.

- Integrity (a participant decides at most once): follows directly from lines 3.2-3.4.
- Validity (if a participant's outcome is COMMIT, then all votes are YES): results from line 3.4. The output of the *Unif_Cons* protocol can only be COMMIT if it was proposed by some participants (Validity property of Uniform Consensus), *i.e.*, if this participant has been delivered a vote YES from all participants.
- Uniform Agreement (no two participants decide differently). For all participants that launch *Unif_Cons*, Uniform Agreement of the protocol follows from the Uniform Agreement property of the *Unif_Cons* protocol which stipulates that the result is the same for all participants that execute it.
- AS-Non-Triviality. If all participants vote YES and no participant is suspected, then all execute the *Unif_Cons* protocol with COMMIT as input value. From Uniform Agreement and Validity properties of the Uniform Consensus, the result of the *Unif_Cons* protocol will be COMMIT and all participants will decide COMMIT as outcome.
- Termination. Because the *trans* message is sent with the *AS_Rel_Multicast* primitive, it is delivered at all correct participants and consequently they execute *AS_nbac*. As indicated in Section 3.3, Termination relies on the Completeness of the underlying failure detectors. Assume they satisfy Strong Completeness (*i.e.*, a crashed participant is eventually suspected by every correct participant; as indicated in Section 3.3 this property can be ensured by the simple use of timeouts). From this property and reliability of the communication network, it follows that a correct participant p either receives a vote from q or suspects it. So the **wait** statement (line 2.1-2.4) terminates for each correct participant. Then, due to the Termination property of the Uniform Consensus sub-protocol, all correct participants eventually decide.

7.3 Early Deciding

From the validity condition of the NBAC problem one can conclude that, if a participant q votes NO, then the final decision will inevitably be ABORT; actually, NO is *absorbent* and systematically entails

an ABORT decision[10]. Thus, a participant p receiving a vote NO can immediately decide *outcome* := ABORT. Let us consider another participant r ($r \neq p$). Three scenarios are possible for r: (1) r is delivered the vote NO from q; (2) q crashes before sending its vote to r; (3) r suspects q before being delivered its vote. In cases (2) and (3) r will launch *Unif_Cons* (ABORT) (Figure 4, line 3.3). As p does not know which scenario does occur at r, and as all correct participants must propose a value for the Uniform Consensus to terminate, it follows that p has to participate in the Uniform Consensus. So, line 3.2 can be replaced by:

(3.2.0) *a vote* NO *has been delivered* →
(3.2.1) **begin** *outcome* := ABORT
(3.2.2) *launch a thread executing Unif_Cons* (ABORT)
(3.2.3) **end**

Line 3.2.1 allows p to decide as early as possible. Line 3.2.2 ensures all correct participants will decide (namely ABORT). Instead of launching a thread, other solutions can be envisaged; *e.g.*, using the *AS_Rel_Multicast* primitive defined in Section 4 to disseminate the ABORT decision to all the participants; such a protocol is described in [16].

8 Conclusion

This paper addressed the *Non-Blocking Atomic Commitment* problem in the context of synchronous and asynchronous distributed systems where transaction participants can fail by crashing. While failures can be safely detected in synchronous systems by using timeout mechanisms (*i.e.*, a failure notification indicates a failure occurrence), failures can only be suspected, sometimes erroneously, in asynchronous systems. Such an uncertainty about the occurrence of failures is one of the main and most difficult problems encountered in asynchronous distributed systems. In this context, Chandra and Toueg's characterization of classes of unreliable failure detectors (based on Completeness and Accuracy properties) constitute an important conceptual tool to face this uncertainty and to solve fault-tolerance related problems in asynchronous systems.

A generic protocol to solve the NBAC problem has been introduced and instantiations of its generic statements have been provided for both types of distributed systems, namely synchronous and asynchronous. These instantiations have shown that timeout, multisend and reliable multicast communication primitives, unreliable failure detectors (with appropriate properties) and Uniform Consensus are basic components for fault-tolerance in distributed systems. From this point of view this paper can be seen as an introduction to these components. In the presence of process crash failures, the first important lesson is given by the generic statement propose: (1) in a synchronous system a correct participant can locally take a consistent decision when a timer expires, while (2) in an asynchronous system, it has to cooperate with the others in order to decide in a consistent way. The second important lesson is given by failure detectors; they allow to precisely characterize the set of executions in which the NBAC problem can be solved in purely asynchronous systems: Weak Completeness and Eventual Weak Accuracy (Section 3.4) delineate the precise frontier beyond which the Consensus Problem cannot be solved ([5]).

This paper referred to definitions and theoretical notions [5, 14, 19]. It seems important those notions be known as soon as one is interested in understanding and mastering the difficulty introduced by the detection of failures in distributed systems. These notions will be helpful to researchers and

[10]Note that the set of all votes equal to YES is not *absorbent*. This is because, in that case, when no participant is suspected by another one then the final decision will be COMMIT, but when a correct participant is suspected by another participant then the final decision can be either COMMIT or ABORT. Let us consider a participant p that received a vote YES from all other participants; as it does not know whether some correct participant q is suspected by another one r, p cannot unilaterally conclude that the decision value will be COMMIT.

engineers to state the *precise* assumptions under which a given problem, either theoretical or practical, can be solved. Actually, the behavior of reliable distributed applications run on asynchronous distributed systems should be predictable in spite of failures. These notions constitute a promising step towards this direction.

Acknowledgement

I would like to thank Mustaque Ahamad, André Schiper, Santosh Shrivastava, Gérard Thia-kime and Sam Toueg for interesting discussions on transactions and on the failure detection problem.

Appendix A1: The Two Phase Commit Protocol can Block

A coordinator-based Two Phase Commit Protocol obeys the following message exchange. The coordinator requests a vote from each transaction participant and waits for their answers. If all participants vote YES then the coordinator returns the decision COMMIT; if a participant voted NO or failed, the coordinator returns the decision ABORT.

Consider the following failure scenario ([1]). Participant p_1 is the coordinator and p_2, p_3, p_4 and p_5 are the other transaction participants. The coordinator p_1 sends a vote request to all participants in order to get their votes. Suppose answers from p_4 and p_5 are YES. According to the set of answers it receives, the coordinator p_1 determines the decision value D (COMMIT or ABORT), sends it to p_2 an p_3, but crashes before sending it to p_4 and p_5. Moreover p_2 and p_3 crash just after receiving the decision value D. It follows that participants p_4 and p_5 are blocked: they cannot know the decision value as (1) they voted YES and (2) p_1, p_2 and p_3 have crashed (they cannot communicate with one of them to get the decision value D). It is important to note that p_4 and p_5 cannot force a decision value as the forced value could be different from D. So, p_4 and p_5 are blocked till one of the crashed participants recovers. That is why the basic Two Phase Commit protocol is blocking: non-failed participants cannot always progress because of failure occurrences.

Appendix A2: Overview of Chandra-Toueg's Consensus Algorithms

Chandra and Toueg define two consensus protocols in [4]. The first one assumes a failure detector that satisfies strong completeness and weak accuracy. This protocol tolerates up to *n-1* faulty processes. It proceeds in *n-1* phases. At each phase each correct process p_i broadcasts to the others the new values it received during the preceeding phase (it broadcasts its initial value v_i during the first phase). The strong completeness property ensures termination of the protocol while the weak accuracy property guarantees that all correct processes will receive the value of the process that is never suspected. From these two properties the protocol builds the required agreement.

The second protocol assumes (1) a failure detector that satisfies strong completeness and *eventual* weak accuracy and (2) a majority of correct processes. It is based on the rotating coordinator paradigm and it proceeds in asynchronous rounds. All processes have a priori knowledge that during round r the coordinator is the process p_c where $c = (r \ mod \ n) + 1$. When p_i is the coordinator (so we are at round r with $i = (r \ mod \ n) + 1$) it tries to establish a consensus value in the following way:

- p_i gathers values from a majority of processes (phase 1 of round r), defines an *estimate* value from the values it received and broadcasts this *estimate* value to all the other processes (phase 2 of round r).

- During round r, after having sent a value to the current coordinator p_i, a process p_j waits to receive an *estimate* value from p_i. if it receives one, it sends back an *ack* to p_i (phase 3 of round r). If it suspects p_i to be faulty, it sends back a *nack* to p_i (phase 3 of round r) and it proceeds to round $r + 1$.
- The coordinator p_i waits to receive an *ack* or a *nack* from a majority of processes. If it received an *ack* from a majority of processes, it reliably broadcasts (with the *AS_Rel_Multicast* primitive) the *estimate* value to all processes (phase 4 of round r).

The protocol ensures consensus by guaranteing that once a majority of processes have adopted (by sending an *ack*) an *estimate* value proposed by a coordinator then this value becomes "locked" in the sense no other decision value is possible. The protocol terminates after the first round in which the coordinator is correct and is not suspected by a majority of correct processes. In the particular case where there are neither failures nor failures suspicions, the decision value is obtained at the end of the first round (in that case $3 \times (n - 1)$ messages and an execution of *AS_Rel_Multicast* are used). The interested reader will consult [4] where these protocols are detailed, proved an used to build more sophisticated protocols.

References

[1] Ö. Babaoğlu and S. Toueg. Non-Blocking Atomic Commitment. In *Distributed Systems (Second Edition)*, ACM Press (S. Mullender Ed.), New-York, 1993, pp. 147-166.

[2] M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In *Proc. 2nd* ACM *Symposium on Principles of Distributed Computing*, ACM Press, Montréal, August 1983, pp. 27-30.

[3] P.A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987, 370 pages.

[4] T.D. Chandra and S. Toueg. Unreliable Failures Detectors for Reliable Distributed Systems. *Journal of the* ACM, vol. 43(2), 1996, pp. 245-267. (A preliminary version appeared in Proc. 10th ACM *Symposium on Principles of Distributed Computing*, August 1991, ACM Press, pp. 325-340).

[5] T.D. Chandra, V. Hadzilacos and S. Toueg. The Weakest Failure Detector for Solving Consensus. To appear in *Journal of the* ACM, 1996. (A preliminary version appeared in Proc. 11th ACM *Symposium on Principles of Distributed Computing*, August 1992, ACM Press, pp. 147-158).

[6] D.R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V-kernel. *ACM Transactions on Computer Systems,* Vol.3(2), 1985, pp. 77-107.

[7] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the* ACM, Vol. 34(2), february 1991, pp. 56-78.

[8] F. Cristian. Synchronous and Asynchronous Group Communication. *Communications of the* ACM, Vol. 39(4), april 1996, pp. 88-97.

[9] S. Davidson, I. Lee and V. Wolfe. Timed Atomic Commitment. *IEEE Transactions on Computers*, vol.40(5), may 1991, pp. 573-583.

[10] C. Fetzer and F. Cristian. Fail-Awareness in Timed Asynchronous Systems. Proc. 15th ACM *Symposium on Principles of Distributed Computing*, ACM Press, may 1996, pp. 314-321.

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

[12] G.A. Geist *et al. PVM: A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[13] J. N. Gray. Notes on Database Operating Systems. In *Operating Systems : An Advanced Course*, Springer-Verlag LNCS 60 (R. Bayer, R.M. Graham and G. Seegmuller Eds), 1978.

[14] R. Guerraoui. Revisiting the Relationship between Non-Blocking Atomic Commitment and Consensus. In *Proc. 9th Int. Workshop on Distributed Algorithms* (WDAG95), Springer-Verlag LNCS 972 (J.M. Hélary and M. Raynal Eds), Sept. 1995, pp. 87-100.

[15] R. Guerraoui, M. Larea and A. Schiper. Non-Blocking Atomic Commitment with an Unreliable Failure Detector. In *Proc. of the 14th IEEE Symposium on Reliable Distributed Systems*,Bad Neuenahr, Germany, 1995, pp. 41-50.

[16] R. Guerraoui and A. Schiper. The Decentralized Non-Blocking Atomic Commitment Protocol. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Systems*, San Antonio, TX, 1995, pp. 2-9.

[17] R. Guerraoui and A. Schiper. Consensus Service: a Modular Approach for Building Agreement Protocols in Distributed Systems. In *Proc. of the 26th IEEE Symposium on Fault-Tolerant Computing Systems*, Sendai (Japan), June 1996, pp. 168-177.

[18] V. Hadzilacos. On the Relationship Between the Atomic Commitment and the Consensus Problems. In *Proc. Workshop on Fault-Tolerant Distributed Computing*, Springer-Verlag LNCS 448 (D. Simmons and A. Spector Eds), 1990, pp. 201-208.

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

[20] B. Lampson. Atomic Transactions. In *Distributed Systems Architecture and Implementation : An Advanced Course*, Springer-Verlag LNCS 105 (B. Lampson, M. Paul and H. Siegert Eds), 1981, pp. 246-265.

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

[22] M. Rabin. Randomized Byzantine Generals. In *Proc. 24th Symposium on Foundations of Computer Science*, IEEE Press, November 1983, pp. 403-409.

[23] D. Skeen. Non-Blocking Commit Protocols. In *Proc. ACM SIGMOD Int. Conference on Management of Data*, ACM Press, 1981, pp. 133-142.

[24] J. Turek and D. Shasha. The Many Faces of Consensus in Distributed Systems. *Computer*, Vol. 25(6), (June 1992), pp. 8-17.

[25] J.S.K. Wong and S. Mitra. A Non-Blocking Timed Atomic Commit Protocol for Distributed Real-Time Database Systems. *Journal of Systems Software*, vol. 34, june 1996, pp. 161-170.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399