Consensus-Based Management of Distributed and Replicated Data

Michel RAYNAL
IRISA, Campus de Beaulieu
35 042 Rennes-cedex, France
raynal@irisa.fr

Abstract

Atomic Broadcast and *Atomic Commitment* are fundamental problems that have to be solved when managing distributed/replicated data. This short note aims at showing that solutions to these problems can benefit from results associated with the *Consensus* problem. Such an approach helps gain a better insight into distributed/replicated data management problems.

More precisely, this note addresses one of the most important issues one is faced to when designing distributed/replicated data management protocols, namely, their *Non-Blocking* property. This property stipulates that the crash of nodes participating in a protocol must not prevent the non-crashed nodes from terminating the protocol execution. Results from the Consensus study allow to know the *minimal assumptions* a system must satisfy in order its distributed/replicated data management protocols be non-blocking despite process crash and asynchrony.
Keywords: Asynchronous Distributed Systems, Atomic Broadcast, Atomic Commitment, Consensus, Crash/no Recovery, Crash/Recovery, Data Management, Non-Blocking, Replicated Data, Transaction.

Asynchronous Distributed Systems

Systems that span large geographical areas (*e.g.*, through Internet), or systems that are subject to unpredictable loads are fundamentally *asynchronous*. This means that it is not possible to assume and to rely on upper bounds for message transfer delays or for process scheduling delays when one has to implement an application (*e.g.*, banking, booking-reservations or point-of-sale commerce) on such a system. This does not mean that timeout mechanisms do not have to be used. It only means that any value used to set a timer can not be trusted when this value is given a system-wide meaning. More precisely, as values used by timers do not define correct upper bounds, the taking of a critical decision by the system or by the upper layer application can not be based on them. That is why asynchronous distributed systems are sometimes called *time-free* systems.

When the system, although asynchronous, suffers no failure (an unrealistic case in practice!) it is possible to compute a consistent snapshot of its state from which consistent decisions can be taken. *Consistent* means here that the snapshot is a past global state the system has or could have passed through. This uncertainty (has/could have) is intrinsically due to the asynchrony of the system.

When failures can occur, the situation becomes much more complicated. The combination of failures (whose occurrences are unpredictable!) and asynchrony create a stronger uncertainty on the system state. Consequently,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

this can make very difficult or even impossible for an observer (*e.g.*, a process of the application layer) to determine a consistent global state. To address this problem theoreticians have introduced an *abstract* problem, namely the *Consensus* problem, which can be instantiated to solve practical problems dealing with globally consistent decision takings. Among those problems, the *Atomic Broadcast* (AB) problem and the *Non-Blocking Atomic Commitment* (NBAC) problem are particularly important in the distributed/replicated data management context.

This note briefly discusses the connections of the AB and NBAC problems with the *Consensus* problem. More precisely, the *Consensus* problem is used to throw light on these data management problems. It appears that the knowledge of both practical and theoretical results associated with *Consensus* provides basic principles that can help better understand not only solutions to distributed/replicated data management problems but also their intrinsic difficulty. One of the main issues encountered when designing a protocol (that solves such a data management problem in a failure-prone asynchronous system) concerns its *Non-Blocking* property. this property states that, despite the actual but unpredictable pattern of failure occurrences, the protocol must terminate at all non-crashed nodes. The focus of this note is on this *Non-Blocking* property. When a protocol works (correctly terminates), we must understand *why* it does work (correctly terminate). When a protocol does not work, we must understand *why* it does not work. *Knowing the assumptions that are required for data management protocols to correctly terminate in failure-prone asynchronous distributed systems is a fundamental issue.* These assumptions actually define the "limits" beyond which a data management problem may not be solved. Such a knowledge should help researchers and engineers (1) to get a deeper insight into the intrinsic difficulty one has to master when solving distributed/replicated data management problems, and (2) to state the precise assumptions under which their particular distributed/replicated data management problem can be solved.

What is the Consensus Problem?

The Problem

In the *Consensus* problem, each process p_i proposes a value v_i and all non-crashed processes have to decide on some value v that is related to the set of proposed values [10]. Actually, v_i is the *view* the process p_i has of (a relevant[1] part of) the system state. The consensus imposes one of these views to processes. Its aim is, despite failures and asynchrony, to provide processes with the *same view* of (the relevant part of) the system state. After a consensus execution, as processes have the same view of the system state, they can proceed in a globally consistent way (*e.g.*, by taking identical decisions).

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, we say that p "proposes" v_i. In the same way, when p invokes decide and gets v as a result, we say that p "decides" v. It is assumed that links are reliable and that processes can crash. Moreover, a crashed process does not recover. This is the *Crash/no Recovery* model. In this model, a correct process is a process that does not crash[2]. (Lossy links and process recovery are addressed below.) The semantics of propose and decide is defined by the following properties:

- C-Termination. *Every correct process eventually decides.*
- C-Agreement. *No two processes (correct or not) 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 property.

[1]The word "relevant" has to be appropriately instantiated for each particular problem.

[2]Practically, this means that the process does not crash during the execution of the consensus protocol, or during the execution of the upper layer application that uses the consensus protocol.

About Failures

What is surprising with this apparently simple problem is that it has no solution when processes can crash. This impossibility result is from Fischer, Lynch and Paterson [10] who have formally shown that the *Consensus* problem has no deterministic solution in asynchronous distributed systems that are subject to even a single process crash failure. Intuitively, this negative result is due to the impossibility to safely distinguish (in an asynchronous setting) a crashed process from a slow process, or from a process with which communications are very slow.

This impossibility result has motivated researchers to find a set of minimal assumptions that, when satisfied by a distributed system, makes the Consensus problem solvable in this system. Chandra-Toueg's *Unreliable Failure Detector* concept constitutes an answer to this challenge [7]. From a practical point of view, an unreliable failure detector can be seen as a set of oracles: each oracle is attached to a process and provides it with a list of processes it suspects to have crashed. An oracle can make mistakes by not suspecting a crashed process or by suspecting a non-crashed one. By restricting the domain of mistakes they can make, several classes of failure detectors can be defined. From a formal point of view, a failure detector class is defined by two properties: a *Completeness* property which addresses detection of actual failures, and an *Accuracy* property which restricts the mistakes a failure detector can make.

Among the classes of failure detectors defined by Chandra and Toueg, the class $\diamond\mathcal{S}$ is characterized by the two following properties. The *Strong Completeness* property states: *Eventually, every crashed process is permanently suspected by every correct process.* The *Eventual Weak Accuracy* property states: *There is a time after which some correct process is never suspected.* It has been shown in [8] that, provided a majority of processes are correct, these conditions are the *weakest* ones that allow to solve the Consensus problem. This means that the Consensus problem can not be solved in runs that do not satisfy these two properties. Furthermore, while the completeness property can always be realized by using timeouts, it is important to note that the Eventual Weak Accuracy property can only be approximated in asynchronous distributed systems[3]. So, when the Eventual Weak Accuracy property can not be guaranteed, it is possible that the Consensus problem can not be solved. This means that the Eventual Weak Accuracy property defines very precisely the borderline beyond which the consensus problem can not be solved. This is a very important result that can help understand why some protocols (which can be shown to be instantiations of consensus protocols) work (do terminate) in some circumstances and do not work (can not terminate) in other circumstances.

It is important to note that several consensus protocols based on unreliable failure detectors of the class $\diamond\mathcal{S}$ have been proposed [7, 19, 25]. They all are based on the rotating coordinator paradigm and proceed in consecutive asynchronous rounds. Each round, coordinated by a predetermined process, implements a majority vote strategy. During a round, the associated coordinator tries to impose a value as the decision value. If the Eventual Weak Accuracy property is satisfied, there will be a round during which the associated coordinator will not be erroneously suspected, and this coordinator will succeed in establishing a decision value.

Current Research Efforts

(1) Tradeoff Safety *vs* Liveness. The crucial problem when implementing a consensus protocol lies in the fact its liveness can not be guaranteed in time-free asynchronous unreliable systems. All the previous protocols always guarantee safety, but their liveness can not be taken for granted. When the Eventual Weak Accuracy property of the underlying failure detector is not satisfied, they may not terminate. In that case the protocol does not satisfy the C-Termination property. More precisely, any protocol that requires a failure detector of the class $\diamond\mathcal{S}$ may lose its liveness (*i.e.*, it may not terminate), but will never lose its safety (*i.e.*, if it gives a result, this result is correct) when the underlying failure detector malfunctions by failing to meet its properties. A current research effort concerns the design of consensus protocols that trade liveness against safety. In that case, the resulting protocol always terminates, but there is a price that has to be paid: the safety property is ensured only with some probability; this probability depends on the density function associated with the random variable representing

[3]Otherwise, this would contradict the Fisher-Lynch-Paterson's impossibility result!

message transfer delays [6].
(2) Towards More Realistic models. The previous results have been established in a relatively simplistic model, namely, the *Crash/no Recovery* model. This model assumes that communication is reliable and that a process that crashes during the protocol execution does no recover.

Research is now focusing on more realistic models in which messages can be lost and processes can recover during the protocol execution. This is the *Crash/Recovery* model. Current results concerning this realistic model are : (1) Methods and principles to cope with message loss [2, 4, 9, 15, 20]; (2) Definitions of appropriate failure detectors [2, 9, 20]; (3) Consensus protocols [2, 20]; (4) The statement of a necessary and sufficient condition on the use by each process of a local stable storage in which it can save critical state data during its "crash" periods [2].

The Non-Blocking Atomic Commitment Problem

The Problem

The *Non-Blocking Atomic Commitment* problem (NBAC) has originated from databases, more precisely from transactions. In a distributed system, a transaction usually involves several participant sites (*i.e.*, several processes). At the end of a transaction, its participants are required to enter a commitment protocol in order to commit it (when enough things went well) or to abort it (when too many things went wrong). Each participant votes YES or NO. If for any reason (deadlock, storage problem, concurrency control conflict, etc.) a participant can not locally commit the transaction, it votes NO. Otherwise a vote YES means that the participant commits locally to make updates permanent if it is required to do so. Then, the decision to commit or to abort is determined. The decision must be COMMIT if enough participants (usually all) voted YES. It must be ABORT if too many participants (usually one) voted NO [3, 5, 12].

More formally, NBAC in an asynchronous distributed system, can be defined by the following properties, where the parameter x is used to capture several instances of the problem within a single definition ($x = n$ characterizes the classical *All-or-none* NBAC problem, while $x = \lceil (n+1)/2 \rceil$ characterizes the *Majority* NBAC problem, where n is the total number of participants):

- NBAC-Termination. *Every correct participant eventually decides.*
- NBAC-Agreement. *No two participants 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 participant decides* COMMIT*, then at least x participants have voted* YES.
  - Obligation. *If x participants vote* YES *and are not suspected (to have crashed), 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.*, enough processes voted YES). Finally, the obligation property eliminates the trivial solution where the decision value would be ABORT even when the situation is good enough to commit.

NBAC **is a Consensus Problem**

Actually the NBAC is a particular instance of the Consensus problem. We provide here a simple protocol that reduces NBAC to Consensus. Figure 1 describes this reduction when we consider an All-or-none NBAC problem (*i.e.*, x is equal to the number of participants).

(1) $\forall q \in$ participants **do** send$(vote)$ **end do**;
(2.1) **wait** ((delivery of a vote NO from a participant)
(2.2) **or** ($\exists q \in$ participants: p suspects q to have crashed)
(2.3) **or** (**from each** $q \in$ participants: delivery of a vote YES)
(2.4));
(3.1) **case**
(3.2) a vote NO has been delivered $\rightarrow view$:= ABORT
(3.3) a participant has been suspected $\rightarrow view$:= ABORT
(3.4) all votes are YES $\rightarrow view$:= COMMIT
(3.5) **end case**;
(4) propose$(view)$; $decision$:=decide; % Consensus execution %

Figure 1: From Consensus to NBAC in Asynchronous Systems

The behavior of every participant p is made of 4 steps. First (line 1), p disseminates its vote to all participants. Then (lines 2.*), p waits until either it has received a NO vote (line 2.1), or it has received a YES vote from each participant (line 2.3), or it has suspected a participant to have crashed (line 2.2). Then (lines 3.*), the participant p builds its own $view$ of the global state: this view 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 participates in a consensus. After having proposed its local view of the global state (invocation of propose$(view)$), p waits for the result of the consensus execution (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.

This reduction shows that, in asynchronous distributed systems, the NBAC problem[4] can be solved each time the Consensus problem can be solved. Actually, they are equivalent [13]. It is important to note that the NBAC protocols described in [27] assume a synchronous distributed system: they rely on timeouts to detect participant crashes. So, due to these timeouts, their termination is always guaranteed. But, if the values used to set timers reveal to be erroneous (this occurs when they are not correct upper bounds), those protocols can not satisfy their safety property. In time-free asynchronous systems, the satisfaction of the termination property of NBAC protocols relies on the fact the underlying failure detectors satisfy the Strong Completeness property and the Eventual Weak Accuracy property. Basing these protocols on a solution to the Consensus problem help understand the minimal assumptions that have to be satisfied in order protocols solving NBAC work. More information on the relations between the NBAC problem and the Consensus problem can be found in [13, 17, 23].

The Case of Replicated Data

The replication of a data does not change the problem. In that case, the NBAC problem usually requires that a *majority* of the replicas agree to commit updates. The appropriate version of the NBAC problem is the *Majority* version [5]. The reduction of the *Majority* NBAC problem to the Consensus problem requires to add an additional step to the protocol described in Figure 1. Such reduction protocols are described in [14, 16]. This shows that, although not often claimed in a clear way, the NBAC problem (be it used for committing updates on distinct data

[4]Note that the definition we have considered for the NBAC problem includes the "suspicion of a participant" notion (see the Obligation property) which is less precise than the "crash of a participant" notion: due to asynchrony, a participant q can be suspected by another participant p to have crashed while actually q has not crashed. This shows that, in an asynchronous system, the notion of "suspicion of a participant" is the best approximation we can have for "crash of a participant".

or on copies of a replicated data) is an instantiation of the Consensus problem. This is particularly important as it allows us to know the exact conditions under which a NBAC protocol is non-blocking.

The Atomic Broadcast/Multicast Problem

An increasing number of works consider an *Atomic Broadcast* (AB) primitive to disseminate updates to copies of replicated data [1, 21, 22, 29]. A suite of broadcast primitives has been formally defined in [18].

The Atomic Broadcast Problem

The AB problem is defined by two primitives, namely A_Broadcast and A_Deliver. Informally, the effect of these primitives is to guarantee that processes are delivered messages in the same order. Processes have to agree not only on the set of messages they are delivered but also on their delivery order. More precisely, the AB problem is defined by the three following properties:

- AB-Termination. This property is composed of two parts.
  - Correct Sender. *If a correct process executes* A_Broadcast(m)*, then eventually all correct processes execute* A_Deliver(m).
  - Delivery Occurrence. *If a process executes* A_Deliver(m)*, then all correct processes execute* A_Deliver(m).
- AB-Agreement. *If two processes p and q execute* A_Deliver(m_1) *and* A_Deliver(m_2)*, then p executes* A_Deliver(m_1) *before* A_Deliver(m_2) *if and only if q executes* A_Deliver(m_1) *before* A_Deliver(m_2).
- AB-Validity. *If a process executes* A_Deliver(m)*, then m has been sent by $sender(m)$* (where $sender(m)$ denotes the sender of m).

As the previous definitions, the definition of what is a "*correct* process" depends on the model we consider. It has been shown that the Consensus problem and the AB problem are equivalent problems: any solution to one of them solves the other [7]. So, any AB protocol designed for asynchronous distributed systems has the same limitations as a Consensus protocol. An AB protocol for Crash/no Recovery systems is described in [7] (here, a correct process is a process that never crash). An AB protocol for Crash/Recovery systems is described in [24] (here, a correct process is a process that can crash and recover an arbitrary number of times, but that eventually recovers and remains "up" long enough for the upper layer application to be able to terminate).

Atomic Multicast

Atomic Multicast to multiple groups (AM) is a primitive particularly useful to manage a set of replicated data. Actually, each data is replicated on a set of sites; these sites define a replication group. The AM primitive allows to reliably send a message to several groups in such a way that (1) all the sites of the destination groups are delivered the message, and (2) there is a single delivery order for all the messages. This means that if 2 copies (of the same data or of different data) receive m_1 and m_2 then they deliver these messages in the same order. When there is a single group, AM confuses with AB. The AM primitive has been investigated in [26] from where the following example is taken.

Let us consider a classical transaction that transfers $1,000 from bank account #1 to bank account #2. To achieve fault-tolerance, assume that each bank account is replicated on several nodes, and assume that every replica is managed by a process. Let g_1 be the fault-tolerant group of processes that manage bank account #1, and let g_2 be the fault-tolerant group of processes that manage bank account #2. The two operations (withdrawal and deposit) can be aggregated into a single message by defining m as: (*remove $1,000 from account #1; add $1,000 to account #2*). When a process in g_1 delivers m, it removes $1,000 from the bank account it manages;

when a process in g_2 delivers m, it adds $1,000 to the bank account it manages. In this distributed setting, the money transfer transaction can be expressed as the atomic multicast of m to the groups g_1 and g_2. Either both of the groups or none of them deliver m. Moreover, if both groups are destination of the same messages m_1 and m_2, they deliver them in the same order. It is easy to see that the total order property of multicast ensures the serializability of transactions.

An efficient protocol implementing the AM primitive is described in [11]. This protocol is a combination of consensus protocols with a total order multicast protocol designed by Skeen for failure-free systems [28]. What is important is that consensus allows to solve the problem in failure-prone systems. The price that has to be paid is always the same: it is related to the termination property of the AM problem. The knowledge of the Consensus problem indicates to us which are the assumptions related to the detection of failures that have to be satisfied by the underlying system for the AM protocol to terminate.

By way of Conclusion

The aim of this short discussion was to show that the Consensus problem has to be considered as a fundamental problem as soon as one is interested in managing distributed/replicated data despite asynchrony and failure occurrences. In such a context, one of the main problems consists in ensuring the termination (*Non-Blocking*) property of distributed/replicated data management protocols: failed nodes must not prevent non-failed nodes from terminating.

There are two bad news. The first one is that the Consensus problem has no solution in an asynchronous distributed system model as simplistic as the Crash/no Recovery model. This means that it is not possible to design a Consensus protocol that will *always* satisfy the *Non-Blocking* property in presence of process crashes. The second one is that two very important practical problems encountered in the management of distributed/replicated data (NBAC and AB) can be solved in a given system only when the Consensus problem can be solved in this system.

The good news is that a deep knowledge of the Consensus problem (*i.e.*, a knowledge of the assumptions it requires to be solved, and a knowledge of the principles that underlie the protocols that solve it) can provide engineers with concepts and mechanisms that allow them to master problems encountered in the management of distributed/replicated data. As indicated at the beginning of this note, when a protocol works, we must understand *why* it does work. And, when a protocol does not work, we must understand *why* it does not work. Data engineering has to be based on both Technology and Science.

References

[1] Agrawal D., Alonso G., El Abbadi A. and Stanoi I., Exploiting Atomic Broadcast in Replicated databases. *Proc. 1997 EURO-PAR Conference on Parallel Processing*, August 1997, pp. 496-503.

[2] Aguilera M.K., Chen W. and Toueg S., Failure Detection and Consensus in the Crash-Recovery Model. *Proc. of the 12th Int. Symposium on Distributed Computing (DISC'98, ex-WDAG)*, Andros (Greece), Springer Verlag LNCS 1499 (S. Kutten Ed.), pp. 231-245.

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

[4] Basu A., Charron-Bost B., Toueg S., Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes. *Proc. of the 10th Int. Workshop on Distributed Algorithms (WDAG)*, Springer-Verlag, LNCS 1151 (. Babaoğlu and K. Marzullo Eds), pp. 105-122, Bologna, Italy, October 1996.

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

[6] Bollo R., Le Narzul J.-P., Raynal M. and Tronel F., Probabilistic Analysis of a Group Failure Detection Protocol. *Research Report 1209*, IRISA, Rennes, October 1998.

[7] 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).

[8] 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).

[9] Dolev D., Friedman R., Keidar I and Malkhi D., Failure Detectors in Omission Failure Environments. *Short paper in Proc. of the 16th ACM Symposium on Principles of Distributed Computing*, pp. 286, August 1997.

[10] 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.

[11] 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.

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

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

[14] Guerraoui R., Oliveira R. and Schiper A., Atomic Updates of Replicated Data. *Proc. of the 2th European Dependable Computing Conference (EDCC2)*, Springer-Verlag, LNCS 1150, pp. 1365-382, Taormina, Italy, October 1996.

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

[16] Guerraoui R., Raynal M. and Schiper A., Atomic Commit And Consensus: a Unified View. (In French) *Technique et Science Informatiques*, 17(3):279-298, 1998.

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

[18] 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] Kemme B. and Alonso G., A Suite of Database Replication Protocols Based on Group Communication Primitives. *Proc. 18th Int. IEEE Conference on Distributed Computing Systems*, Amsterdam, May 1988, pp. 156-163.

[22] Pedone F., Guerraoui R. and Schiper A., Exploiting Atomic Broadcast in Replicated databases. *Proc. 4th EURO-PAR Conference on Parallel Processing*, Southampton, September 1998, Springer Verlag LNCS 1470, pp. 513-520.

[23] 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.

[24] Rodrigues L. and Raynal M., Non-Blocking Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems. *Research Report*, IRISA, Rennes, October 1998, 16 pages.

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

[26] Schiper A. and Raynal M., From Group Communication to Transactions in Distributed Systems. *Communications of the ACM*, 39(4):84–87, April 1996.

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

[28] Skeen D., Unpublished communication. Cited in: Birman K. and Joseph T., Reliable Communication in presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47-76, 1987.

[29] Stanoi I., Agrawal D. and El Abbadi A., Using Broadcast Primitives in Replicated databases. *Proc. 18th IEEE Int. Conference on Distributed Computing Systems (IC DCS'98)*, Amsterdam, May 1988, pp. 148-155.