Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems and Its Use in Quorum-Based Replication

Luís Rodrigues, *Member*, *IEEE*, and Michel Raynal

Abstract—*Atomic Broadcast* is a fundamental problem of distributed systems: It states that messages must be delivered in the same order to their destination processes. This paper describes a solution to this problem in asynchronous distributed systems in which processes can crash and recover. A Consensus-based solution to *Atomic Broadcast* problem has been designed by Chandra and Toueg for asynchronous distributed systems where crashed processes do not recover. We extend this approach: It transforms any Consensus protocol suited to the crash-recovery model into an Atomic Broadcast protocol suited to the same model. We show that Atomic Broadcast can be implemented requiring few additional log operations in excess of those required by the Consensus. The paper also discusses how additional log operations can improve the protocol in terms of faster recovery and better throughput. To illustrate the use of the protocol, the paper also describes a solution to the replica management problem in asynchronous distributed systems in which processes can crash and recover. The proposed technique makes a bridge between established results on Weighted Voting and recent results on the Consensus problem.

Index Terms—Distributed fault-tolerance, asynchronous systems, atomic broadcast, consensus, crash/recovery, quorum, replica management, weighted voting.

1 Introduction

Atomic Broadcast is one of the most important agreement problems encountered in the design and in the implementation of fault-tolerant distributed systems. This problem consists of providing processes with a communication primitive that allows them to broadcast and deliver messages in such a way that processes agree not only on the set of messages they deliver but also on the order of message deliveries. Atomic Broadcast (sometimes called Total Order Broadcast) has been identified as a basic communication primitive in many systems [27]. It is particularly useful to implement fault-tolerant services by using software-based replication [16]. By employing this primitive to disseminate updates, all correct copies of a service deliver the same set of updates in the same order, and, consequently, the state of the service is kept consistent.

Solutions to the Atomic Broadcast problem in asynchronous systems prone to process crash (no-recovery) failures are now well-known [8], [11], [28]. In this model, process crashes are definitive (i.e., once crashed, a process never recovers), so a failed process is a crashed process. Unfortunately, the crash no-recovery model is unrealistic for the major part of applications. That is why, in this paper, we consider the more realistic crash-recovery model. In this model, processes can crash and later recover. We assume that when a process crashes 1) it loses the content of its volatile memory and 2) the set of messages that has been delivered while it was crashed is also lost. This model is well-suited to feature real distributed systems that support user applications. Real systems provide processes with stable storage that make them able to cope with crash failures. A stable storage allows a process to log critical data. But, in order to be efficient, a protocol must not consider all its data as critical and must not log a critical data every time it is updated (the protocol proposed in this paper addresses these efficiency issues).

It has been shown in [8] that *Atomic Broadcast* and *Consensus* are equivalent problems in asynchronous systems prone to process crash (no-recovery) failures. The Consensus problem is defined in the following way: Each process proposes an initial value to the others, and, despite failures, all correct processes have to agree on a common value (called decision value), which has to be one of the proposed values. Unfortunately, this apparently simple problem has no deterministic solution in asynchronous distributed systems that are subject to even a single process crash failure: this is the so-called Fischer-Lynch-Paterson's (FLP) impossibility result [10]. The FLP impossibility result has motivated researchers to find a set of minimal assumptions that, when satisfied by a distributed system, makes Consensus solvable in this system. The concept of unreliable failure detector introduced by Chandra and Toueg constitutes an answer to this challenge [8]. 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 information regarding the status of other processes. An oracle can make mistakes, for instance, by not suspecting a failed process or by suspecting a not

- *L. Rodrigues is with the Faculdade de Ciencias, Universidade de Lisboa, Bloco C5, Campo Grande, 1749-016 Lisboa, Portugal. E-mail: ler@di.fc.ul.pt.*
- *M. Raynal is with IRISA, Campus de Beaulieu, Université de Rennes 1, Avenue du Gnral Leclerc, 35042 Rennes Cedex, France. E-mail: raynal@irisa.fr.*

Manuscript received 18 Sept. 2000; revised 2 May 2001; accepted 7 May 2001. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 112877.

failed one. Although failure detectors were originally defined for asynchronous systems where processes can crash but never recover, the concept has been extended to the crash-recovery model [1], [19], [25].

Chandra and Toueg have also shown how to transform any Consensus protocol into an Atomic Broadcast protocol in the asynchronous crash (no-recovery) model [8]. In the present paper, we follow a similar line of work and we show how to transform a protocol that solves Consensus into the crash-recovery model in a protocol that solves Atomic Broadcast in the same model. Thus, our protocol assumes a solution to the Consensus problem in the crash-recovery model (such protocols are described in [1], [19]). A less modular approach to solve Atomic Broadcast in the crash-recovery model has been described in [21]. An attempt to decompose this protocol is described in [7].

Our transformation owns several interesting properties. First, it does not require the explicit use of failure detectors (although those are required to solve the Consensus problem). Thus, it is not bound to any particular failure detection mechanism. It relies on a gossip mechanism for message dissemination, avoiding the problem of reliable multicast in the crash-recovery model. Also, it allows recovering processes to avoid explicitly invoking Consensus for those instances that already have a decided outcome. Additionally, our solution is nonblocking [4], i.e., as long as the system allows Consensus to terminate, the Atomic Broadcast is live. Finally, but not the least, we show that Atomic Broadcast can be implemented with few additional log operations in excess of those required by the Consensus.

Chandra and Toueg's approach and ours are similar in the sense that both of them transform a Consensus protocol into an Atomic Broadcast protocol. But, as they consider different models (crash no-recovery and crash-recovery, respectively), they use different techniques. This comes from the fact that we have to cope with process crashes and message losses (that is why our protocol requires a gossiping mechanism, which is not necessary in a crash-no recovery + no message loss model). Additionally, while being crashed, a process may miss a large number of message exchanges; therefore, efficient methods to recover its state must be defined. Finally, subtle problems arise in the definition of the properties of the atomic multicast primitive since message deliveries must be logged. Actually, when solving a distributed system problem, modifying the underlying system model very often requires the design of new protocols more appropriate to those models [17].

This paper also shows how the primitive can be applied to implement quorum-based replication. Weighted voting [13] is a well-known technique to manage replication in the crash-recovery model. The technique consists of assigning votes to each replica and defining quorums for *read* and *write* operations. Quorums for conflicting operations, namely read/write and write/write, must overlap such that conflicts can be detected. To ensure that an operation only succeeds if a quorum is achieved, each operation must be encapsulated within a transaction [5]. It should be noted that, in asynchronous systems, these solutions must also rely on variants of Consensus to decide the outcome of transactions [15]. This paper explores an alternative path to the implementation of quorum-based replication that relies on the use of our *Atomic Broadcast* primitive. By employing this primitive to disseminate updates, all correct copies of a service process the same set of updates in the same order and the service state is kept consistent. The proposed technique makes: 1) a bridge between established results on Weighted Voting and recent results on the Consensus problem and 2) a bridge between the active replication model in the synchronous crash (no-recovery) model and the asynchronous crash-recovery model.

The paper is organized as follows: Section 2 defines the crash-recovery model and the Atomic Broadcast problem in such a model. Then, Section 3 presents the underlying building blocks on top of which the proposed protocol is built, namely, a transport protocol and a Consensus protocol suited to the crash-recovery model. A minimal version of our Atomic Broadcast protocol for the crash-recovery model is then presented in Section 4. The impact of additional log operations on the protocol is discussed in Section 5. Section 6 shows how the protocol can be used to implement quorum-based replication. Section 7 relates the Atomic Broadcast problem in the crash-recovery with other relevant problems. Finally, Section 8 concludes the paper.

2 Atomic Broadcast in the CR-Model

2.1 The Crash-Recovery Model

We consider a system consisting of a finite set of processes $\Pi = \{p, \dots, q\}$. At a given time, a process is either *up* or *down*. When it is *up*, a process progresses at its own speed behaving according to its specification (i.e., it correctly executes its program text). While being up, a process can fail by crashing: It then stops working and becomes *down*. A down process can later recover: It then becomes up again and restarts by invoking a recovery procedure. So, the occurrence of the local event *crash*(respectively, *recover*) generated by the local environment of a process, makes this process transit from up to down (respectively, from down to up).

A process is equipped with two local memories: a volatile memory and a stable storage. The primitives log and retrieve allow an up process to access its stable storage. When it crashes, a process definitely loses the content of its volatile memory; the content of a stable storage is not affected by crashes.

Processes communicate and synchronize by sending and receiving messages through channels. We assume there is a bidirectional channel between each pair of processes. Channels are not necessarily FIFO; moreover, they can duplicate messages. Message transfer delays are finite but arbitrary. Even if channels are reliable, the combination of crashes, recoveries, and arbitrary message transfer delays can entail message losses: The set of messages that arrive at a process while it is down are lost. Thus, the protocol must be prepared to recover from messages losses.

2.2 Atomic Broadcast

Atomic Broadcast allows processes to reliably broadcast messages and to receive them in the same delivery order. Basically, it is a reliable broadcast plus an agreement on a

single delivery order. We assume that all messages are distinct. This can be easily ensured by adding an identity to each message, an identity being composed of a pair (local sequence number, sender identity). To ensure that sequence numbers are unique despite crash and recoveries, the local sequence number must include an incarnation number that is logged in stable memory and incremented whenever a process recovers.

At the syntactical level, Atomic Broadcast is composed of two primitives: A-broadcast(m) (used to send messages) and $\mu_p =$ A-deliver-sequence() (used by the upper layer to obtain the sequence of ordered messages). As in [8], when a process executes A-broadcast(m) we say that it "A-broadcasts" m. We also define a Boolean predicate A-delivered (m, μ_p) which evaluates to "true" is $m \in \mu_p$ or "false" otherwise. We also say that some process p "A-delivers" m if A-delivered$(m,$ A-deliver-sequence$())$ is "true" at p.

In the context of asynchronous distributed systems where processes can crash and recover, the semantics[1] of *Atomic Broadcast* is defined by the four following properties: Validity, Integrity, Termination, and Total Order. This means that any protocol implementing these communication primitives in such a crash/recovery context has to satisfy these properties:

- Validity. If a process A-delivers a message m, then some process has A-broadcast m.
- Integrity. Let μ_p be the delivery sequence at a given process p. A message m appears at most once in μ_p.
- Termination. For any message m, 1) if the process that issues A-broadcast (m) returns from A-broadcast (m) and eventually remains permanently up, or 2) if a process A-delivers a message m, then all processes that eventually remain up A-deliver m.
- Total Order. Let μ_p be the sequence of messages A-delivered to process p. For any pair (p, q), either μ_p is a prefix of μ_q or μ_q is a prefix of μ_p.

The validity property specifies which messages can be A-delivered by processes: It states that the set of A-delivered messages cannot contain spurious messages. The integrity property states there are no duplicates. The termination property specifies the situations where a message m has to be A-delivered. The total order property specifies that there is a single total order in which messages are A-delivered. This is an Agreement property that, joined to the termination property, makes the problem nontrivial.

Note the subtle difference in the definition of the Integrity property with regard to its formulation in the crash-no recovery model. Typically, Integrity is formulated in terms of preventing the same message from being delivered twice (to avoid duplication). However, consider the following two scenarios:

- **Case 1**. p delivers a given message m and crashes; p recovers.
- **Case 2**. p is about to deliver m (but has not done so yet); p crashes; p recovers.

1. We actually consider the definition of the *Uniform* Atomic broadcast problem [18].

These two scenarios are indistinguishable. We address this problem by avoiding an explicit A-deliver primitive and by using instead the $\mu_p =$ A-deliver-sequence() call. Note also that an application that crashes and recovers may have to maintain its own log to memorize which messages have been processed before the crash.

3 Underlying Building Blocks

The protocol proposed in Section 4 is based on two underlying building blocks: a *Transport Protocol* and a protocol solving the *Uniform Consensus* problem. This section describes their properties and interfaces.

3.1 Transport Protocol

The transport protocol allows processes to exchange messages. A process sends a message by invoking a send or multisend primitive.[2] Both primitives are unreliable: The channel can lose messages but it is assumed to be fair, i.e., if a message is sent infinitely often by a process p, then it can be received infinitely often by its receiver [23]. When a message arrives at a process, it is deposited in its input buffer that is a part of its volatile memory. The process will consume it by invoking a receive primitive. If the input buffer is empty, this primitive blocks its caller until a message arrives.

3.2 Consensus Interface

In the *Consensus* problem, each process proposes a value and all correct processes have to decide on some value v that is related to the set of proposed values [10]. The interface with the *Consensus* module is defined in terms of two primitives: propose and decided. As in previous works (e.g., [8]), when a process p invokes propose(w), where w is its proposal to the Consensus, we say that p "proposes" w (when multiple instances of consensus are required, propose accepts an additional parameter, the instance identifier, propose(k, w)). A process proposes by logging its initial value on stable storage; this is the only logging required by our basic version of the protocol. In the same way, when p invokes decided and gets v as a result, we say that p "decides" v (denoted decided(v)).

The definition of the Consensus problem requires a definition of a "correct process." As the words "correct" and "faulty" are used with a precise meaning in the crash (no-recovery) model [8], for clarity purpose, we define their equivalents in the crash-recovery model, namely, "good" and "bad" processes (we use the terminology of [1]).

Note that a process may crash and later recover and reissue propose for a given instance of consensus more than once. In any case, if the consensus has terminated, the consensus module will always return the result of that instance to the invoking process. In our solution, we will enforce that if a recovered process reissues propose for a given instance of consensus, it proposes exactly the same values as in the previous invocations.

3.3 Good and Bad Processes

A *good* process is a process that eventually remains permanently up. A *bad* process is a process that is not

2. The primitive multisend is actually a macro that allows a process p to send (by using the basic send primitive) a message to all processed (including itself).

good. So, after some time, a good process never crashes. On the other hand, after some time, a bad process either permanently remains crashed or oscillates between crashes (down periods) and recoveries (up periods). From a practical point of view, a good process is a process that, after some time, remains up long enough to complete the upper layer protocol. In the Atomic Broadcast problem, for example, this means that a good process that invokes A-broadcast(m) will eventually terminate this invocation (it is possible that this termination occurs only after some (finite) number of crashes). It is important to note that, when considering a process, the words "up" and "down" refer to its current state (as seen by an external observer), while the words "good" and "bad" refer to its whole execution.

A bad process can oscillate between crashes and recoveries. Such a behavior can, in some circumstances, prevent the progress of good processes. Hence, in the following, we assume that a bad process eventually remains crashed forever. It follows that a good process eventually remains permanently up, while a bad process eventually remains permanently down. The difficulty in designing protocols in such a context lies in the fact that, at any time, given any process p (that is currently up or down), it is not known if p is actually good or bad.

3.4 Consensus Definition

The definition of the Consensus problem in the crash-recovery model is obtained from the one given in the crash (no-recovery) model by replacing "correct process" by "good process." Each process p_i has an initial value v_i that it *proposes* to the others, and all good processes have to *decide* on a single value that has to be one of the proposed values. More precisely, the Consensus problem is defined by the following three properties (we actually consider the *Uniform* version [8] of the Consensus problem):

- Termination. If all good processes propose, every good process eventually decides some value.
- Uniform Validity. If a process decides v, then v was proposed by some process.
- Uniform Agreement. No two processes (good or bad) decide differently.

3.5 Enriching the Model to Solve Consensus

As noted previously, the Consensus problem has no deterministic solution in the simple crash (no-recovery) model. This model has to be enriched with a failure detector that, albeit unreliable, satisfies some minimal conditions in order that the Consensus be solvable. The crash-recovery model has also to be augmented with a failure detector so that the Consensus can be solved. With this aim, different types of failure detectors have been proposed. The protocol proposed in [19] uses failure detectors that outputs list of "suspects"; so, their outputs are bounded. Aquilera et al. [1] uses failure detectors whose outputs are unbounded (in addition to lists of suspects, the outputs include counters). The advantage of the later is that they do not require the failure detector to predict the future behavior of bad processes. A positive feature of our protocol is that it does not require the explicit use of failure detectors (although these are required to solve the Consensus problem). Thus, it is not bound to any particular failure detector mechanism.

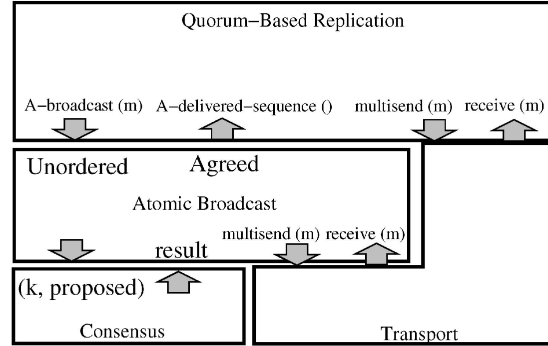

Fig. 1. Protocol interfaces.

4 A Crash-Recovery Atomic Broadcast Protocol

4.1 Basic Principles

The proposed protocol borrows some of its principles from the total order protocol designed for the crash (no-recovery) model that is described in [8].

As illustrated in Fig. 1, the protocol interfaces the upper layer through two variables: the *Unordered* set and the *Agreed* queue. Messages requested to be atomically broadcast are added to the *Unordered* set. Ordered messages are inserted in the *Agreed* queue, according to their relative order. The *Agreed* is a representation of the delivery sequence. Operations on the *Unordered* and *Agreed* variables must be idempotent, i.e., if the same message is added twice, the result is the same as if it is added just once (since message have unique identifiers, duplicates can be detected and eliminated).

The protocol requires the use of a Consensus protocol and of an unreliable (but fair) transport protocol offering the send, multisend, and receive primitives described in Section 3. The transport protocol is used to gossip information among processes. The interface with the Consensus protocol is provided by the propose and decided primitives. The propose primitive accepts two parameters: an integer, identifying a given instance of the Consensus, and a proposed value (a set of messages) for that instance. When a Consensus execution terminates, the decided primitive returns the messages decided by that instance of the Consensus in the variable *result*. The Consensus primitives must also be idempotent: Upon recovery, a process may (re-)invoke these primitives for a Consensus instance that has already started or even terminated.

The Atomic Broadcast protocol works in consecutive rounds. In each round, messages from the *Unordered* set are proposed to Consensus and the resulting decided messages moved to the *Agreed* queue. Processes periodically gossip their round number and their *Unordered* set of messages to other processes. This mechanism provides the basis for the dissemination of unordered messages among good processes.

4.2 Protocol Description

We now provide a more detailed description of the protocol. The protocol is illustrated in Figs. 2 and 3. The state of each process p is composed of:

- k_p: the round counter (initialized to 0);
- $Proposed_p$: an array of sets of messages proposed to Consensus. $Proposed_p[k_p]$ is the set of messages

```
Initial values:
    k_p = 0; ∀k : Proposed_p[k] = ⊥;
    Unordered_p=∅; Agreed_p=⊥; gossip-k_p = 0;

procedure replay ():
    // may be shortened by logging k_p and Agreed_p
    // (see discussion in Section 5)
    k_p ← 0;
    while Proposed_p[k_p] ≠ ⊥ do
        propose(k_p,Proposed_p[k_p]); wait until decided (k_p, result);
        k_p ← k_p + 1; Agreed_p ← Agreed_p⊕ result
        Unordered_p ← Unordered_p ∖ Agreed_p
    end while

upon initialization or recovery:
    retrieve (Proposed_p); replay ();
(a) fork task { sequencer and gossip }

Task gossip:
    repeat forever
        multisend GOSSIP-R(k_p)
        ∀m ∈ Unordered_p multisend GOSSIP-M(m)
```

Fig. 2. Atomic broadcast protocol.

proposed to the k_pth Consensus. All entries of the array are initialized to $\perp$ ($\perp$ means "this entry of the array has not yet been used");

- $Unordered_p$: a set of unordered messages, requested for broadcast (initialized to $\emptyset$);
- $Agreed_p$: a queue of already ordered messages (initialized to $\perp$);
- $gossip\text{-}k_p$: a variable that keeps the value of the highest Consensus round known as already decided (this value is obtained via the gossiping mechanism).

The first four variables can be structured as two pairs of variables. The $(k_p, Proposed_p)$ pair is related to the current (and previous) Consensus in which p is (was) involved. The ($Agreed_p$, $Unordered_p$) pair is related to the upper layer interface. Statements associated with message receptions and transmission (A-broadcast) are executed with mutual exclusion with regard to each other. The *sequencer* task and the *gossip* task constitute the core of the protocol. Both tasks access atomically the variables k_p and $Unordered_p$.

A-broadcast(m) issued by process p consists of adding m to its set $Unordered_p$. Then, the protocol constructs the common delivery order. A-deliver issued by p takes the next message from the $Agreed_p$ queue and A-delivers it to the upper layer application. The activation of the protocol is similar in the initial case and in the recovery case: the *gossip* and *sequencer* tasks are started (line a).

4.3 The *Gossip* Task

This task is responsible for disseminating periodically a relevant part of the current state of processes. The gossip messages sent by a process p, namely GOSSIP-R(k_p) and GOSSIP-M(m), contain its round number and the messages in its set of unordered messages. The goal of the *gossip* task is twofold. First, it ensures the dissemination of data messages, such that they are eventually proposed to Consensus by all good processes. Second, it allows a process that has been down to know which is the most up-to-date round.

Upon reception of a GOSSIP message, an active process p updates its $Unordered_p$ set and checks if the sender q has a higher round number ($k_p > k_q$). In this case, p records that it has lagged behind by updating the $gossip\text{-}k_p$ variable. This variable is used by the *sequencer* task to get the result of the Consensus p has missed.

4.4 The *Sequencer* Task

This task is the heart of the ordering protocol [8]. The protocol proceeds in rounds. In the round k, a process p proposes its $Unordered_p$ set to the kth instance of Consensus. Before starting the Consensus, the proposed value is saved in stable storage. Note that the propose primitive must be idempotent: In case of crash and recovery, it may be called for the same

```
upon A-broadcast(m): // (issued by the upper layer)
    Unordered_p ← (Unordered_p ∪ {m}) ∖ Agreed_p;
    wait until (m ∈ Agreed_p) // see discussion

upon receive GOSSIP-M(m) from q:
    Unordered_p ← (Unordered_p ∪ {m}) ∖ Agreed_p;
upon receive GOSSIP-R(k_q) from q:
    if (k_q > k_p) then
        gossip-k_p ← max (gossip-k_p,k_q) fi // q was ahead

Task sequencer:
    repeat forever
        if Proposed_p[k_p]=⊥ then
            // Process p has to define its initial value for
            //the next Consensus
            wait until ((Unordered_p ≠ ∅) or (gossip-k_p > k_p));
            Proposed_p[k_p] ← Unordered_p;
            log(Proposed_p[k_p]); propose(k_p,Proposed_p[k_p]);
        fi;
        wait until decided (k_p, result);
        // Initializes the new round and
        // commits results from previous round
        [ k_p ← k_p + 1; Agreed_p ← Agreed_p⊕ result ];
        Unordered_p ← Unordered_p ∖ Agreed_p
    end repeat

upon A-delivered-sequence: // (issued by the upper layer)
        return Agreed_p
```

Fig. 3. Atomic broadcast protocol (cont.).

round more than once. The result of the Consensus is the set of messages to be assigned sequence number k. These messages are moved (according to a deterministic rule) from the $Unordered_p$ set to the $Agreed_p$ queue. Then, the round number k_p is incremented and the messages that remain in the $Unordered_p$ set are proposed by process p during the next Consensus. The *sequencer* task has to execute some statements atomically with respect to the processing of GOSSIP messages. This is indicated by bracketing with "[" and "]" the corresponding statements in the *sequencer* task.

To avoid running unnecessary instances of Consensus, a process does not start a new round unless it has some messages to propose or it knows it has lagged behind other processes. In the later case, it can propose an empty set as the initial value for those Consensus it has missed (this is because for those Consensus a decision has already been taken without taking p's proposal into account).

4.5 Logging into Stable Storage

Logging is used to create checkpoints from which a recovering process can continue its execution and consequently make the protocol live. So, at a critical point, the values of relevant variables are logged into stable storage. In this paper, we are interested in discussing a protocol that makes a minimal number of checkpoints (independently of those required by the underlying Consensus protocols). Thus, we only log the initial value proposed for each Consensus round. This guarantees that if process p crashes before the Consensus decides, p will propose the same value again after recovering. We will later argue that this logging step cannot be avoided.

Note that we do not log the $Unordered_p$ set or the $Agreed_p$ queue. The $Agreed_p$ queue is reconstructed upon recovery from the results of past Consensus rounds by the replay procedure. To ensure that messages proposed to Atomic Broadcast are not lost, the A-broadcast(m) primitive does not return until the message m is in the agree queue. If the process fails before that, there is no guaranty that the message has been logged, so the message may have or may have not been A-broadcasted. In the latter case, it is the same as if the process has failed immediately before calling A-broadcast(m). Note that these design options that aim at minimizing the number of logging operations, do not necessarily provide the more efficient implementation. Alternative designs are discussed below.

4.6 Recovery

Since the protocol only logs the initial values proposed for each instance of Consensus, the current round k_p and the $Agreed_p$ queue have to be reconstructed upon recovery. The current round is simply the round for which no initial value has been proposed yet. The agreed queue can be reconstructed by reading the results of the Consensus instances that have terminated. Thus, before forking the *sequencer* and *gossip* tasks, the process parses the log of proposed and agreed values (kept internally by Consensus).

4.7 Discussion

The following observations can be made on the protocol behavior:

- (P1). The sequence of consecutive round numbers logged by a process p is not decreasing.
- (P2). If process p has logged k_p whose value is k, then its variable k_p will always be $\geq k$.
- (P3). If a good process joins round k, then all good processes eventually join a round $\geq k$.
- (P4). For any k, whatever the number of times p participates in consensus numbered k, the value it proposes to this consensus is always the same (despite crashes and despite total/partial consensus reexecutions).
- (P5). For any k, whatever the number of times p participates in consensus numbered k, the $result$ value is the same each time the invocation of decided$(k, .)$ terminates at p ([3]). (This property follows from the consensus specification.)
- (P6). Any message m A-broadcast by a good process is eventually deposited in $Unordered_p$ or $Agreed_p$ by any good process p.
- (P7). Any message m A-delivered by a process is eventually deposited in $Agreed_p$ by any good process p.

The Integrity property follows from the $\oplus$ operation on the $Agreed_p$ queue that adds any message m at most once into this queue. The Validity property directly follows from the fact the protocol does not create messages. The Total Order property follows from the use of the underlying consensus and from the appropriate management of the $Agreed_p$ queue. The Termination property follows from the previous observations, the fact consensus executions terminate, and the assumption that a bad process eventually remains crashed.

4.8 On the Minimal Logging

Our solution only requires the logging of the initial proposed value for each round of Consensus. We argue that this logging operation is required for every atomic protocol that uses Consensus as a black box. In fact, all Consensus protocols for the crash-recovery model we are aware of assume that a process p proposes a value by writing it on stable storage. For instance, upon recovery, the protocol of [1] checks the stable storage to see if a initial value has been proposed.

4.9 Performance

Since we assume that processes can crash and recover and that channels are lossy, and no bounds are established for maximum number of faults a correct process/channel may exhibit, we can only evaluate the performance of the algorithm in terms of the number of messages m, protocol steps s, and the log operations l, required to perform a given operation. Also, since we use the consensus module as a black box, the performance of our protocol is a function of the performance figures of the consensus box

$$(m_c(n), s_c(n), l_c(n)).$$

3. Using this terminology used in [8], this means that, after the first consensus execution numbered k, the *result* value associated with round k is "*locked*."

```
Initial values:
      k_p = 0; ∀k : Proposed_p[k] = ⊥;
      Unordered_p=∅; Agreed_p=(A-checkpoint(⊥), VC(⊥));
      gossip-k_p = 0;

procedure replay ():
**** same as before without its first line *****
      while Proposed_p[k_p] ≠ ⊥ do
            propose(k_p,Proposed_p[k_p]);
            wait until decided (k_p, result);
            k_p ← k_p + 1; Agreed_p ← Agreed_p⊕ result

upon initialization or recovery:
**** same as before with the addition of the first line *****
      retrieve (k_p,Agreed_p);
      retrieve (Unordered_p);
      retrieve (Proposed_p);
      replay ();
(a)   fork task { sequencer and gossip and checkpoint }

upon A-broadcast(m): // (issued by the upper layer)
**** first line: same as before *****
      Unordered_p ← (Unordered_p ∪ {m}) \ Agreed_p;
      log(Unordered_p)

**** first two lines: same as before *****
upon receive GOSSIP-M(m) from q:
      Unordered_p ← (Unordered_p ∪ {m}) \ Agreed_p;
upon receive GOSSIP-R(k_q) from q:
      if (k_q > k_p) then
            gossip-k_p ← max(gossip-k_p,k_q) // q is ahead
(d)   else if (k_p > k_q + δ) then// δ is a configuration parameter
            send STATE(k_p − 1, Agreed_p) TO q
      fi    fi

upon receive STATE(k_q,A_q) from q:
**** new message *****
      if k_p < k_q − δ then // p is late
(e)         terminate task {sequencer};
            // Skip Consensus whose number k is such
            // that k_p ≤ k < k_q so, during the processing of the
            // STATE message, the sequencer task is aborted
            k_p ← k_q + 1; Agreed_p ← A_q;
(f)         fork task { sequencer }
      else
            // small de-synchronization
            gossip-k_p ← max(gossip-k_p,k_q)
      fi
```

Fig. 4. Alternative protocol.

```
Task gossip:
**** same as before *****
      repeat forever
            multisend GOSSIP-R(k_p)
            ∀m ∈ Unordered_p multisend GOSSIP-M(m)
Task checkpoint: ***** new task *****
      repeat forever // implementation dependent frequency
(b)         [ Agreed_p ← (A-checkpoint(Agreed_p), VC(Agreed_p)) ]
            log (k_p,Agreed_p)
(c)         // Proposed_p[i], i < k_p can be discarded from the log

Task sequencer: ***** same as before *****
      repeat forever
            if Proposed_p[k_p]=⊥ then
                  // Process p has to define its initial value
                  // for the next Consensus
                  wait until ((Unordered_p ≠ ∅) or (gossip-k_p > k_p));
                  Proposed_p[k_p] ← Unordered_p;
                  // Ensure that despite crashes p always proposes
                  // the same input to the k_p^th Consensus
                  log(Proposed_p[k_p]);
                  propose(k_p,Proposed_p[k_p]);
            fi;
            wait until decided (k_p, result);
            // Initializes the new round and
            // commits results from previous round
            [ k_p ← k_p + 1; Agreed_p ← Agreed_p⊕ result ];
            Unordered_p ← Unordered_p \ Agreed_p
      end repeat

upon A-deliver-sequence: // (issued by the upper layer)
            ***** same as before *****
            return Agreed_p
```

Fig. 5. Alternative protocol (cont.).

The protocol requires the dissemination of each message to all processes before the consensus, for a total of $n + n_c(n)$ messages and $1 + s_c(n)$ steps. Since only a log operation is required in each round, the total number of log operations is $1 + l_s(n)$. Note that several messages may be ordered in a single consensus round, therefore the number of messages, steps, and log operations does not necessarily increase linearly with the number of atomic broadcast operations. When recovering, a process must first replay all previous consensus and then obtain all broadcast that it has missed while crashed. The second phase of recovery requires at least two messages (and two steps).

5 An Alternative Atomic Broadcast Protocol

We now present a number of modifications to our basic protocol that, although increasing slightly, the complexity and the number of log operations may provide some benefits in practical systems. The protocol proposes a state transfer mechanism and additional log operations to reduce the recovery overhead and increase throughput. Additionally, the protocol shows how to prevent the number of entries in the logs from growing indefinitely, by taking application-level checkpoints. The version of the protocol that takes into account the previous considerations is illustrated in Figs. 4 and 5.

5.1 Avoiding the Replay Phase

In the previous protocol, we have avoided any logging operation that is not strictly required to ensure protocol correctness. In particular, we have avoided logging the current round (k_p) and agreed queue ($Agreed_p$) since they can be recomputed from the entries of the array $proposed_p$ that have been logged. This forces the recovering process to replay the actions taken for each Consensus result (i.e., insert the messages in the agreed queue according to the predetermined deterministic rule).

Faster recovery can be obtained at the expense of periodically checkpointing both variables. The frequency of this checkpointing has no impact on correctness and is an implementation choice (that must weight the cost of checkpointing against the cost of replaying). Note that old proposed values that are not going to be replayed can be discarded from the log (line c).

5.2 Size of Logs and Application-Level Checkpoint

A problem with the current algorithm is that the size of the logs grows indefinitely. A way to circumvent this behavior is to rely on an application-level checkpointing. In some applications, the state of the application will be determined by the (totally ordered) messages delivered. Thus, instead of logging all the messages, it might be more efficient to log the application state which logically "contains" the *Agreed* queue. For instance, when the Atomic Broadcast is used to update replicated data, the most recent version of the data can be logged instead of all the past updates. Thus, a checkpoint of the application state can substitute the associated prefix of the delivered message log.

In order to exploit this property, one needs to augment the interface with the application layer with an upcall to

obtain the application state (see Fig. 1). The upcall, $state =$ A-checkpoint(μ_p), accepts as an input parameter a sequence of delivered messages and returns the application state that "contains" those updates. A-checkpoint$(\perp)$ returns the initial state of the application. In order to know which messages are associated with a given checkpoint, a *checkpoint vector clock* $VC(\mu_p)$ is associated to each checkpoint. The vector clock stores the sequence number of the last message delivered from each process "contained" in the checkpoint.[4] An application-level checkpoint is defined by the pair (A-checkpoint $(\mu_p), VC(\mu_p)$). The sequence of messages delivered to a process is redefined to include an application checkpoint plus the sequence of messages delivered after the checkpoint. A message m belongs to the delivery sequence if it appears explicitly in the sequence or if it is logically included in the application checkpoint that initiates the sequence (this information is preserved by the checkpoint vector clock).

In our protocol, the application state is periodically checkpointed and the delivered messages in the *Agreed* queue are replaced by the associated application-level checkpoint. This not only offers a shorter replay phase but also prevents the number of entries in the logs from growing indefinitely.

5.3 State Transfer

In the basic protocol, a process that has been down becomes aware that it has missed some Consensus rounds when it detects that some other process is already in an higher round of Consensus (through the gossip messages). When this happens, it activates the Consensus instances that it has missed in order to obtain the correspondent agreed messages. A process that has been down for a long period may have missed many Consensus and may require a long time to "catch-up."

An alternative design consists of having the most up-to-date process to send a STATE message containing its current round number k_p and its $Agreed_p$ queue. When a process p that is late receives a STATE message from a process q with a higher round number ($k_p < k_q$), it stops its *sequencer* task (line e), updates its state such that it catches up with that process, and restarts its *sequencer* task from the updated state (line f), effectively skipping the Consensus instances it has missed.

Both approaches coexist in the final protocol. A late process can recover by activating the Consensus instances that it has missed or by receiving a STATE message. The amount of desynchronization that triggers a state transfer can be tuned through the variable δ (line d).

Note that, for clarity, we have made the STATE message to carry the complete *Agreed* queue. Simple optimizations can minimize the amount of state to be transfered. For instance, since the associated GOSSIP messages carries the current round number of the late process, the STATE message can be made to carry only those messages that are not known by the recipient.

4. Although each checkpoint can be uniquely identified by the sequence number of the last message included in the checkpoint, a vector is needed to verify if a given message has already been ordered.

5.4 Sending Message Batches

For better throughput, it may be interesting to let the application propose batches of messages to the Atomic Broadcast protocol, which are then proposed in batch to a single instance of Consensus. However, the definition of Atomic Broadcast implies that every message that has been proposed by a good process be eventually delivered. When there are crashes, a way to ensure this property is not to return from A-broadcast(m) before m is logged. In the basic protocol, we wait until the message is ordered (and internally logged by the Consensus). In order to return earlier (and allow more messages to be proposed to be include in the batch), the A-broadcast interface needs to log the $Unordered_p$ set.

5.5 Incremental Logging

As described, the protocol emphasizes the control locations where values have to be logged. The actual size of these values can be easily reduced. When logging a queue or a set (such as the *Unordered* set), only its new part (with respect to the previous logging) has to be logged. This means that a log operation can be saved each time the current value of a variable that has to be logged does not differ from its previously logged value.

6 Quorum-Based Replica Management

In this section, we discuss how our atomic broadcast primitive can be used to manage replicated objects in crash-recovery systems. We use a client-server model, where clients interact with servers using the send and multisend primitives and server coordinate their actions using atomic broadcast. We start by extending our definition of atomic broadcast to allow different processes to initiate the *same* broadcast (this will allow server replicas to initiate broadcasts on behalf of client processes). Then, we show how the extended primitive can be used to implement a quorum-based voting technique to manage replicated servers.

6.1 Extending Atomic Broadcast

Usually, the Atomic Broadcast problem is defined assuming that a single process in the system initiates a given broadcast, i.e., that a single process issues A-broadcast(m) for a given message m. In this section, we extend this definition, by allowing the *same* broadcast to be initiated by more than one process. We recall that each message has a unique identifier that distinguishes it from any other message in the system.

The purpose of this extension is to allow the set of processes participating in the broadcast to play the role of a group of servers, providing services to external client processes. Broadcast messages are generated and uniquely identified by these client processes which forward them to the server replicas using an unreliable channel. One or several servers will initiate the atomic broadcast to the set of replicas, by invoking locally the A-broadcast primitive on behalf of the (remote) clients. It should be noted that this extension does not require any significant change to the algorithm described previously.

```
Client p: // sequential process

Initialization:
    request ← ⊥;
    timestamp ← 0;
    fork task gossip

Task gossip:
    repeat forever
        if request ≠ ⊥ then multisend request to set of replicas fi

upon receive r = REPLY(id, vote, val, ver_nb) from q
    if r.id = request.id and r.ver_nb ≥ timestamp then
        replies ← replies ∪ {r} fi

function read(id) begin
    replies ← ⊥; request ← READ(id);
    wait until replies contains a read quorum;
    request ← ⊥;
    timestamp ← max_replies(ver_nb);
    return val such that val.ver_nb= timestamp;
end read

function write(id, val) begin
    replies ← ⊥; request ← WRITE(id, val);
    wait until replies contains a write quorum;
    request ← ⊥
end write
```

Fig. 6. Managing replicated objects using weighted voting (client).

```
Server q:

Initialization(myvote, initval):
    vote ← myvote;
    (value,ver_nb, done) ← (initval, 0, ∅);
upon recovery:
    retrieve(vote, value, ver_nb, done)

    log(vote, value, vers_nb, done)

upon receive READ(id) from p
    send REPLY(id, vote, value, ver_nb) to p

upon receive WRITE(id, val) from p
    if id ∉ done then
        A-broadcast(UPDATE(p, id, val)) to the set of replicas
    else
        send REPLY(id,vote, value,ver_nb) to p // retransmission
    fi

Task update:
    repeat forever
        μ =A-deliver-sequence();
        *** process new updates according to their total order ***
        ∀ UPDATE(p,id,val)∈ μ : id ∉ done do
            (value,ver_nb,done) ← (val,ver_nb + 1, done ∪ {id});
            log(value,ver_nb, done);
            send REPLY(id,vote, value,ver_nb) to p
    end repeat
```

Fig. 7. Managing replicated objects using weighted voting (server).

6.2 Atomic Broadcast for Quorum-Based Voting

Weighted voting [13] is a popular technique to increase the availability of replicated data in networks subject to node crashes or network partitions. The technique consists of assigning votes to each replica and define quorums for read *read* and *write* operations. Quorums for conflicting operations, namely read/write and write/write, must overlap such that conflicts can be detected. To ensure that an operation only succeeds if a quorum is achieved, each operation must be encapsulated within a transaction [5].

Here, we propose a weighted voting variant based on our atomic broadcast primitive. The technique can be used to replicate objects where each transaction is either a read or a write operation. Votes and quorums are assigned exactly as in the transaction-based weighted-voting algorithms. The atomic broadcast (and the underlying consensus) is defined for the set of data replicas. To maximize availability, the majority condition used in the consensus protocol must be defined using the weights assigned to each replica (this can be achieved with a trivial extension to the protocols of [1], [19], [25]).

6.2.1 Client Protocol

The client of the replicated service does not need to participate in the atomic broadcast protocol. The client code is depicted in Fig. 6. Since the channels are lossy and processes can crash, the client periodically retransmits its request until a quorum of replies is received. We assume that each client assigns a unique identifier to each request. This identifier is used by the servers to discard duplicate requests and by the client to match replies with the associated request. The read and write procedures simply wait for a read quorum (or write quorum) of replies to be collected. The reply carries the identifier of the request, the data value, and version number. The highest version number corresponds to the most recent value of the data, which is returned by the read operation.

We avoid locking and keep data available during updates. Thus, reads that are executed concurrently with writes can either read the new or the old data value. To ensure consistency of reads from the same process, each client records the last version read in a variable *timestamp* and discards replies containing older versions. It should be noted that if clients communicate, either directly or by writing/reading other servers, the timestamp must be propagated as discussed in [22].

6.2.2 Server Protocol

The server code is depicted in the bottom part of Fig. 7. Each replica keeps the data value and an associate version number. All updates are serialized by the atomic broadcast algorithm. Read operations do not need to be serialized and are executed locally: The quorum mechanism ensures that the client will get the most updated value. Upon reception of a READ request, each replica simply sends a REPLY to the client with its vote, data value, and version number. Upon reception of a WRITE request, the replica first checks if the associated update has already been processed (since the system is asynchronous, the write request can be received *after* the associated update): in such a case, it simply acknowledges the operation. Otherwise, an UPDATE message is created from the write request and atomically broadcast in the group of replicas. Whenever an update is delivered, the value of the data is updated accordingly and the version number incremented. The fact that this update has been applied is logged in the *done* variable. The processing of the UPDATE reveals a subtle point regarding the black-box interface between the atomic broadcast protocol and the replication algorithm: When a server recovers, it has to parse the sequence of delivered messages, discarding already processed messages.

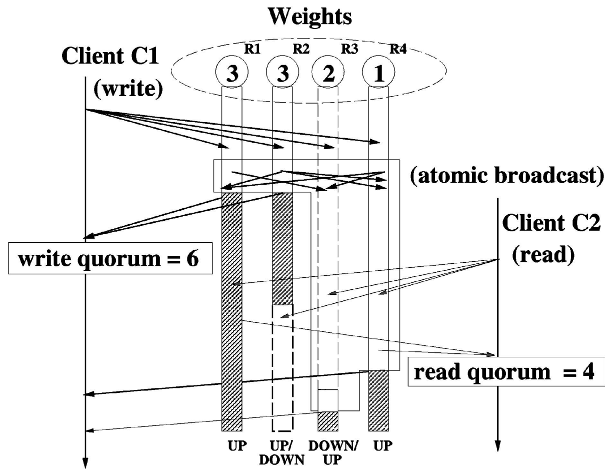

Fig. 8. Example of quorum-based replication.

6.2.3 Example

Fig. 8 illustrates the algorithm. The replicated service is provided by four replicas R1, R2, R3, and R4 with assigned votes of 3, 3, 2 and 1, respectively, for a total of nine votes. In the example, we use a read quorum of 4 and a write quorum of 6 (the sum of quorums for conflicting operations must exceed the total number of votes in the system [13]).

In the example, replicas R1 and R4 are always up, replica R2 crashes after the write but before the read operation, and replica R3 is initially down and later recovers (dotted lines denote crash periods). During the period in which replicas R2 and R3 are both down, there is no write quorum but the data is still available for read operations. Note that while there is no write quorum, the consensus module (encapsulated by the atomic broadcast primitive) blocks. The write quorum is reestablished with the recovery of R3.

The example uses two distinct client processes: client C1 issues a write operation and client C2 a read operation. The write operation is initiated by a multisend to all replicas which, in turn, triggers an A-broadcast to disseminate the update (and serialize it with regard to other updates). Since the system is asynchronous, different replicas may receive the update in different points in time. (In our example, replica R4 receives the update later. The shaded zones indicate that the update has been delivered to the corresponding replica.) Nevertheless, the write operation terminates as soon as a write quorum of replies is received; delayed replies can be discarded by the client. The read operation is initiated by a multisend to all replicas and terminates when a read quorum is received. In our example, replica R4 replies to the read request before receiving the update from client C1, but since a read quorum is awaited, at least one of the replies has the most recent version (R1 in the example). The example also illustrates the recovery of replica R3: When it recovers, both write and read operations have terminated but the atomic broadcast algorithm ensures that the missed update is eventually delivered and that the replica state becomes consistent.

6.3 Discussion

Quorum-based techniques to manage replicated data require the write operation to be applied to a number of replicas satisfying a write quorum or applied to none. When operations are performed in the context of a transaction, a distributed atomic commit protocol [14] is used to decide the outcome of the transaction. Naturally, the atomic commit protocol must be carefully selected to preserve the desired availability, otherwise, the execution of this protocol introduces a window of vulnerability in the system. For instance, if a simple two-phase commit protocol is used, the protocol may block even if a replica with a majority of votes remain up [4].

The protocol proposed in this paper shows that weighted voting can also be applied to a strategy that relies on atomic broadcast to manage replicated data in asynchronous crash-recovery systems. An advantage of this approach is that locking is not required during updates. On the other hand, logical clocks are required to ensure consistent reads [22]. Our work can be seen as a consensus-based extension of the work described in [21], [7]. The proposed protocol can easily be tailored to implement the ROWA strategy (Read One replica, Write All). In that case, it encompasses distributed data management protocols based on an atomic broadcast primitive that has been designed in the no-failure model (e.g., [3], [24]). Let us finally remark that votes can be dynamically modified by adopting techniques described in [12].

7 Related Problems

7.1 Consensus versus Atomic Broadcast

In this paper, we have shown how to transform a Consensus protocol for the asynchronous crash-recovery model into an atomic broadcast protocol. It is easy to show that the reduction in the other direction also holds [8]. To propose a value, a process atomically broadcasts it; the first value to be delivered can be chosen as the decided value. Thus, both problems are equivalent in asynchronous crash-recovery systems.

7.2 Atomic Broadcast and Transactional Systems

It has been shown that a deferred update replication model for fully replication databases can exhibit a better throughput if implemented with an Atomic Broadcast-based termination protocol than if implemented with Atomic Commitment [26]. The same report also proposes designs for Atomic Broadcast protocols in the crash-recovery model, but these solutions are not Consensus-based.

7.3 Total Order Multicast to Distinct Groups

In this paper, we have focused on the Atomic Broadcast problem for a single group of processes. The problem of efficiently implementing atomic multicast across different groups in crash (no-recovery) asynchronous systems has been solved in several papers [11], [28]. Since these solutions are based on a Consensus primitive, it is possible to extend them to crash-recovery systems using an approach similar to the one that has been followed here.

7.4 Recovery in Group Communication Systems

In our approach, a process is never expelled from the group of processes participating in Atomic Broadcast. An alternative approach consists of excluding crashed or

slow processes from the group and makes progress only within the group of mutually-reachable processes that remain active. In such case, the application must explicitly execute some reintegration algorithm, to bring recovering replicas up-to-date [6], [2], [20].

8 Conclusion

This paper has proposed an Atomic Broadcast primitive for asynchronous crash-recovery distributed systems. Its concept has been based on a building block implementing Consensus. This building block is used as a black box, so our solution is not bound to any particular implementation of Consensus. The protocol is nonblocking in the following sense: As long as the underlying Consensus is live, the Atomic Broadcast protocol does not block good processes despite the behavior of bad processes. Moreover, our solution does not require the explicit use of failure detectors (even though those are required to solve the underlying Consensus). Thus, it is not bound to a particular failure detection mechanism. Also, we have shown that Atomic Broadcast can be solved with few additional log operations in excess of those required by the Consensus. Finally, we have discussed how additional log operations can improve the protocol.

Weighted voting techniques are used to increase the availability of replicated objects. This paper has shown that an atomic broadcast primitive can be used to implement weighted replicated objects in asynchronous crash-recovery systems. In the proposed approach, updates are serialized but reads can be performed in parallel.

Acknowledgments

The authors are grateful to Rachid Guerraoui for his constructive comments on an earlier version of the paper. The authors would also like to thank the referees for their comments which helped to improve both the presentation and content of the paper. This work was partially supported by the 234/J4 Franco/Portuguese Grant and by Praxis/C/EEI/12202/1998, TOPCOM. Earlier versions of selected sections of this paper were published separately in [29] and [30].

References

[1] M. Aguilera, W. Chen, and S. Toueg, "Failure Detection and Consensus in the Crash-Recovery Model," *Distributed Computing,* vol. 13, no. 2, pp. 99-125, 2000.

[2] Y. Amir, "Replication Using Group Communication over a Partitioned Network," PhD thesis, Hebrew Univ. of Jerusalem, 1995.

[3] H. Attiya and J. Welch, "Sequential Consistency versus Linearizability," *ACM Trans. Computer Systems,* vol. 12, no. 2, pp. 91-122, 1994.

[4] O. Babaoglu and S. Toueg, "Understanding Non-Blocking Atomic Commitement," *Distributed Systems,* (second ed.), chap. 6, S. Mullender, ed., pp. 147-168, 1993.

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

[6] *Reliable Distributed Computing With the ISIS Toolkit,* K. Birman and R. van Renesse, eds. IEEE CS Press, 1994.

[7] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui, "Deconstructing Paxos," technical report, Distributed Programming Laboratory, EPFL, ID:200106, Jan. 2001.

[8] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM,* vol. 43, no. 2, 225-267, 1996.

[9] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi, "Failure Detectors in Omission Failure Environments," *Proc. 16th Ann. ACM Symp. Principles of Distributed Computing,* p. 286, 1997.

[10] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374-382, 1985.

[11] U. Fritzke Jr., Ph. Ingels, A. Mostefaoui, and M. Raynal, "Consensus-Based Fault-Tolerant Total Order Multicast," *IEEE Trans. Parallel and Distributed Systems,* vol. 12, no. 2, 147-156, 2001.

[12] H. Garcia-Molina, D. Barbara, and A.M. Spauster, "Increasing Availability Under Mutual Exclusion Constraints with Dynamic Vote Assignment," *ACM Trans. Computer Systems,* vol. 7, no. 4, pp. 394-426, 1989.

[13] D. Gifford, "Weighted Voting For Replicated Data," *Proc. Seventh ACM Symp. Operating Systems Principles,* pp. 150-162, 1979.

[14] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* p. 1070, Morgan Kaufmann Publishers, 1993.

[15] R. Guerraoui and A. Schiper, "The Generic Consensus Service," *IEEE Trans. Software Eng.,* vol. 27, no. 1, 29-41, Jan. 2001.

[16] R. Guerraoui and A. Schiper, "Software-Based Replication for Fault Tolerance," *Computer,* vol. 30, no. 4, 68-74, 1997.

[17] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper, "Consensus in Asynchronous Distributed Systems: A Concise Guided Tour," *Chapter 1: Recent Advances in Large Scale Distributed Systems,* Springer-Verlag, LNCS 1752, pp. 33-47, 1999

[18] V. Hadzilacos and S. Toueg, "Reliable Broadcast and Related Problems," chapter 4, Distributed Systems (second ed.), S. Mullender, ed., pp. 97-145, ACM Press, 1993.

[19] M. Hurfin and A. Mostéfaoui, and M. Raynal, "Consensus in Asynchronous Systems Where Processes Can Crash and Recover," *Proc. 17th IEEE Symp. Reliable Distributed Systems,* Oct. 1998.

[20] B. Kemme, A. Bartoli, Ö. Babaoglu, "Online Reconfiguration in Replicated Databases Based on Group Communication," *Proc. Int'l Conf. Dependable Systems and Networks, Digest of Papers,* July 2001.

[21] L. Lamport, "The Part-Time Parliament," *ACM Trans. Computer Systems,* vol. 16, no. 2, pp. 133-169, 1998.

[22] R. Ladin, B. Liskov, B. Shrira, and S. Ghemawat, "Providing Availability Using Lazy Replication," *ACM Trans. Computer Systems,* vol. 10, no. 4, pp. 360-391, 1992.

[23] N. Lynch, *Data Link Protocols,* chapter 16, Distributed Algorithms, Morgan-Kaufmann Publishers. pp. 691-732, 1996.

[24] M. Mizuno, M. Raynal, and J. Zhou, "Sequential Consistency in Distributed Systems," *Proc. Int'l Workshop Theory and Practice in Distributed Systems,* pp. 224-241, 1994.

[25] R. Oliveira, R. Guerraoui, and A. Schiper, "Consensus in the Crash-Recovery Model," Research report 97-239, EPFL, Lausanne, Switzerland 1997.

[26] F. Pedone, R. Guerraoui, and A. Schiper, "Exploiting Atomic Broadcast in Replicated Databases," *Proc. Europar Conf.* pp. 513-520, 1998.

[27] *Comm. ACM,* special issue on Group Communication, D. Powell, guest ed., vol. 39, no. 4, pp. 50-97, 1996.

[28] L. Rodrigues, R. Guerraoui, and A. Schiper, "Scalable Atomic Multicast," *Proc. Seventh Int'l Conf. Computer Communications and Networks (IC3N '98),* pp. 840-847, Oct. 1998.

[29] L. Rodrigues and M. Raynal, "Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems," *Proc. 20th IEEE Int'l Conf. Distributed Computing Systems,* pp. 288-295, Apr. 2000.

[30] L. Rodrigues and M. Raynal, "Quorum-Based Replication in Asynchronous Crash-Recovery Distributed Systems," *Proc. Sixth European Conf. Parallel Computing (Euro-Par 2000),* pp. 605-609, Aug. 2000.

Luís Rodrigues graduated in (1986) and received the master's (1991) and PhD (1996) degrees in electrotechnic and computers engineering, respectively, from the Instituto Superior Técnico de Lisboa (IST). He is an associate professor in the Department of Informatics, Faculty of Sciences, University of Lisbon. Previously, he was at the Electrotechnic and Computers Engineering Department, Instituto Superior Técnico de Lisboa (he joined IST in 1989). From 1986 to 1996, he was a member of the Distributed Systems and Industrial Automation Group at INESC and, since 1997, he is a (founding) member of the LASIGE laboratory at the University of Lisbon. His current interests include fault-tolerant and real-time distributed systems, group membership and communication, and replicated data management. He has more than 50 publications in these areas. He is coauthor of a book on distributed systems. He is a member of the Ordem dos Engenheiros, IEEE, and ACM.

Michel Raynal has been a professor of computer science at the University of Rennes, France, since 1984. At IRISA (CNRS-INRIA-University joint computing laboratory located in Rennes), he is the leader of the ADP (Distributed Algorithms and Protocols) research group that he created in 1986. He has served as program cochair of WDAG (now DISC, the Symposium on Distributed Computing) in 1989 and 1995. He has served several times as vice-chair for the "Distributed Algorithms" track of the IEEE International Conference on Distributed Computing Systems. He has served as a conference chair for the IEEE ISORC '2000 conference (Saint-Malo, France) and will serve as a program co-chair for ICDCS'2002 (Vienna, Austria). He has served as a PC member in many international conferences (among them, many DISC and PODC'2001). Dr. Raynal has written seven books (two published by Wiley & Sons, and two by the MIT Press). He has published more than 50 papers in journals and 100 in conferences. Together with other european leaders, he is currently a member of the ESPRIT Basic Research Network of excellence in Distributed Computing Architectures (CABERNET) currently headed by B. Randell. On the theoretical side, Dr. Raynal's research interests include distributed algorithms, distributed systems, distributed computing, and fault-tolerance. His main interest lies in the fundamental concepts, principles, and mechanisms that underly the design and the construction of distributed systems. Among them, he is currently interested in the study of the *Causality* concept and in the *Consensus* problem. On the practical side, Dr. Raynal is interested in the implementation of reliable communication primitives, the consistency of distributed data, the design and the use of checkpointing protocols, and the set of problems that can be solved on top of a consensus "building block."

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.