Quorum-Based Replication in Asynchronous Crash-Recovery Distributed Systems

Luís RODRIGUES
Universidade de Lisboa
ler@di.fc.ul.pt

Michel RAYNAL
IRISA
raynal@irisa.fr

Abstract

This paper describes a solution to the replica management problem in asynchronous distributed systems in which processes can crash and recover. Our solution is based on a Atomic Broadcast primitive which, in turn, is based on an underlying Consensus algorithm. The proposed technique makes a bridge between established results on Weighted Voting and recent results on the Consensus problem.

Keywords: Asynchronous Distributed Systems, Atomic Broadcast, Consensus, Crash/Recovery, Quorum, Replica Management, Weighted Voting.

1 Introduction

Replication is a well known technique to increase the reliability and availability of data [10]. Replication involves coordination among replicas. For instance, replicas may need to agree on a common state after a given action or on the order by which requests will be processed. Several of these coordination activities are instances of the Consensus problem, that can be 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. Unfortunalety, this problem has no deterministic solution in assynchronous systems where processes may fail, a result known as the Fischer-Lynch-Paterson's (FLP) impossibility result [6]. Intuitively, this negative result is due to the impossibility to safely distinguish (in an asynchronous setting) a faulty process from a slow process (or from a process with which communications are very slow).

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 [4]. 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 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, 5, 12, 17].

Weighted voting [8] is a well known technique to manage replication in the crash-recovery model. The technique consists in assigning votes to each replica and define quorums for *read* and *write* operations. Quorums for conflicting operations, namely read/write and write/write must overlap such that conflicts can be detected. Typically, voting algorithms are applied in the same context of transactions [9]: quorums ensure one-copy equivalence for each replica, concurrency control techniques ensure mutual consistency of data and atomic commitment protocols ensure updates persistence (write operations are performed in the write quorum). It should be noted that, in asynchronous systems, these solutions must also rely on variants of Consensus to decide the outcome of transactions [18].

This paper explores an alternative path to the implementation of quorum-based replication that relies on the use of an *Atomic Broadcast* primitive. An *Atomic Broadcast* primitive allows processes 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. By employing this primitive to disseminate updates, all correct copies of a service are delivered the same set of updates in the same order, and consequently the state of the service is kept consistent. The proposed technique makes: *i)* a bridge between established results on Weighted Voting and recent results on the Consensus problem; *ii)* 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 Section 3 describes the building blocks required by our ptotocol. The algorithm to perform quorum-based management of replicated data is described in section 4. Section 5 discusses the properties of this solution and Section 6 concludes the paper.

2 System Model

The Crash-Recovery Model We consider a system consisting of a finite set of processes $\Pi = \{p, \ldots, q\}$. 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* (resp. *recover*) generated by the local environment of a process, makes this process transit from up to down (resp. from down to up).

Stable storage 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.

Channels 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.

Good and Bad Processes A *good* process is a process that eventually remains permanently up. A *bad* process is a process that is not 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 permanently 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. 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.

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 [12, 17, 1].

3 Building Blocks

The quorum-based replica management algorithm requires the use of an unreliable transport protocol and of a atomic broadcast protocol. It has been shown that the atomic broadcast problem is equivalent to Consensus in asynchronous crash-recovery systems [19]. By resourcing to the atomic broadcast protocol our algorithm does not use a Consensus protocol explicitly (the Consensus is encapsulated by the atomic broadcast primitive).

Transport Protocol The transport protocol allows processes to exchange messages. A process sends a message by invoking a send or multisend primitive (this primitive is actually a macro that allows a process p to send a message to a

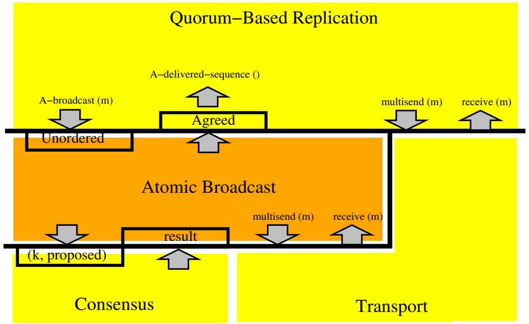

Figure 1: Protocol Interfaces

set of processes). 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 is received infinitely often by its receiver [15]. 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.

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).

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 [4], 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" if $m \in \mu_p$ or "false" otherwise. We also say that some process p "A-delivers" m if A-delivered(m, A-deliver-sequence()) is "true" at p. The complete protocol interface is givenin Figure 1.

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.

- 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.*

[1] We actually consider the definition of the *Uniform* Atomic Broadcast problem [11].

- Termination: *For any message m, (1) if the process that issues* A-broadcast(m) *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 can not 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 non-trivial. An atomic broadcast algorithm for asynchronous crash-recovery systems can be found in [19].

4 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.

4.1 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 change to the algorithm described in [19].

4.2 Atomic Broadcast for Quorum-Based Voting

Weighted voting [8] is a popular technique to increase the availability of replicated data in networks subject to node crashes or network partitions. The technique consists in assigning votes to each replica and define quorums for read *read* and *write* operations. Quorums for conflicting operations, namely read/write an write/write must overlap such that conflicts can be detected. Typically, voting algorithms are applied in the context of transactions [9]: quorums ensure one-copy equivalence for each replica, concurrency control techniques ensure mutual consistency of data and atomic commitment protocols ensure updates persistence (write operations are performed in the write quorum).

Here we propose a weighted voting variant based on our atomic broadcast primitive. 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, 12, 17]).

The client of the replicated service does not need to participate in the atomic broadcast protocol. The client code is depicted on Figure 2. 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 has discussed in [14].

The server code is depicted in the bottom part of Figure 2. 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

```
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, version_nb) from q
        if r.id = request.id and r.version_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(version_nb);
        return val such that val.version_nb= timestamp;
    end read

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

Server q:
    Initialization(myvote, initval):
        vote ← myvote;
        (value,version_nb, processed) ← (initval, 0, ∅);
        log(vote, value, version_nb, processed)

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

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

    upon A-deliver UPDATE(p, id, val):
        if id ∉ processed then
            (value,version_nb,processed) ← (val,version_nb + 1, processed ∪ {id});
            log(value,version_nb, processed);
            send REPLY(id,vote, value,version_nb) to p
        fi

    upon recovery:
        retrieve(vote, value, version_nb, processed)
```

Figure 2: Managing Replicated Objects using Weighted Voting

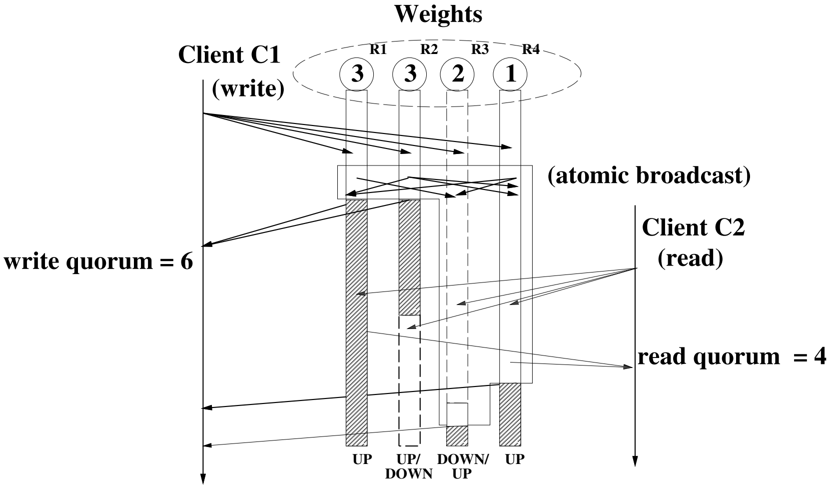

Figure 3: Example of Quorum-Based Replication

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 *processed* 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.

Figure 3 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 9 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 [8]).

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 re-established 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 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.

5 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 [9] 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 [3].

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 [14]. Our work can be seen as a modular extension of the work described in [13]. 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 have been designed in the no-failure model (*e.g.*, [2, 16]). Let us finally remark that votes can be dynamically modified by adopting techniques described in [7].

6 Conclusion

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.

References

[1] M. Aguilera, W. Chen and S. Toueg, Failure Detection and Consensus in the Crash-Recovery Model. *Proc. 12th Int. Symposium on DIStributed Computing*

(formerly WDAG)), Andros, Greece, Sringer-Verlar LNCS 1499, pp. 231-245, September 1998.

[2] H. Attiya and J. Welch, Sequential Consistency versus Linearizability. *ACM TOCS*, 12(2):91-122, 1994.

[3] Ö Babaoğlu and S. Toueg, Understanding Non-Blocking Atomic Commitement. *Chapter 6, Distributed Systems (2nd edition)*, ACM Press (S. Mullender Ed.), New-York, pp. 147-168, 1993.

[4] T. Chandra and S. Toueg, Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2): 225-267, 1996.

[5] D. Dolev, R. Friedman, I. Keidar and D. Malkhi, Failure Detectors in Omission Failure Environments. *Proc. 16th Annual ACM Symposium on Principles of Distributed Computing*, Santa Barbara, California, page 286, 1997.

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

[7] H. Garcia-Molina, D. Barbara and A.M. Spauster, Increasing Availability Under Mutual Exclusion Constraints with Dynamic Vote Assignment. *ACM TOCS*, 7(4):394-426, 1989.

[8] D. Gifford, Weighted Voting For Replicated Data. *Proc. 7th ACM Symposium on Operating Systems Principles*, pp. 150-162, 1979.

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

[10] R. Guerraoui and A. Schiper, Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68-74, 1997.

[11] V. Hadzilacos and S. Toueg, Reliable Broadcast and Related Problems. *Chapter 4, Distributed Systems (2nd Edition)*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

[12] M. Hurfin, A. Mostéfaoui and M. Raynal, Consensus in Asynchronous Systems Where Processes Can Crash and Recover. *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette (IN), October 1998.

[13] L. Lamport, The Part-Time Parliament *ACM Transactions on Computer Systems*, 16(2):133-169, 1998.

[14] R. Ladin, B. Liskov, B. Shrira and S. Ghemawat, Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360-391, 1992.

[15] N. Lynch, Data Link Protocols. *Chapter 16, Distributed Algorithms*, *Morgan-Kaufmann Pub.*, pp. 691-732, 1996.

[16] M. Mizuno, M. Raynal and J. Zhou, Sequential Consistency in Distributed Systems. *Proc. Int. Workshop on Theory and Practice in Distributed Systems*, Springer-Verlag LNCS 938, Dagsthul (Germany), pp. 224-241, 1994.

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

[18] F. Pedone, R. Guerraoui and A. Schiper, Exploiting Atomic Broadcast in Replicated Databases, *Proc. Europar Conference*, Springer-Verlag LNCS 1470, pp.513-520, 1998.

[19] L. Rodrigues and M. Raynal. Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems. *In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, Taipe, Taiwan, April, 2000. (to appear).