Optimistic Active Replication

Pascal Felber
Bell Laboratories – Lucent Technologies
600 Mountain Avenue, Room 2B-303
Murray Hill, NJ 07974, USA

André Schiper
Swiss Federal Inst. of Technology (EPFL)
Operating Systems Laboratory
1015 Lausanne, Switzerland

Abstract

Replication is a powerful technique for increasing availability of a distributed service. Algorithms for replicating distributed services do however face a dilemma: they should be (1) efficient (low latency), while (2) ensuring consistency of the replicas, which are two contradictory goals. The paper concentrates on active replication, where all the replicas handle the clients' requests. Active replication is usually implemented using the Atomic Broadcast primitive. To be efficient, some Atomic Broadcast algorithms deliberately sacrifice consistency, if inconsistency is likely to occur with a low probability. We present in the paper an algorithm that handles replication efficiently in most scenarios, while preventing inconsistencies. The originality of the algorithm is to take the client-server interaction into account, while traditional solutions consider Atomic Broadcast as a black box.

1. Introduction

Replication is a widely used technique for providing high-availability and fault-tolerance of critical services. Nevertheless, developing replicated services is a challenging task: a replicated service must appear as a single highly-available logical entity to its client, which specifically means that the different copies must remain synchronized and consistent with each other. In the so-called *active* replication technique the client request is sent to all the server replicas using an Atomic Broadcast primitive (also called Total Order Broadcast) [6], which ensures that the requests are delivered in the same order by all replicas. Every replica handles the request and sends back the reply to the client. While Atomic Broadcast preserve the consistency of a replicated service, it is considered costly to implement (leading to high latency). To reduce the latency, some Atomic Broadcast algorithms (e.g., [2, 13]) have sacrificed consistency: in some scenarios the total order delivery property of messages can be violated, leading to inconsistencies in the state of the servers. However, the fact that the servers are not always consistent with each other is not a problem in itself, as long as (1) inconsistencies can be repaired, and (2) they do not propagate to the clients. This requires however integration of the Atomic Broadcast algorithm with the client-server interaction schema, i.e., Atomic Broadcast can no more be considered as a black box.

The paper present an active replication algorithm, inspired by the Atomic Broadcast algorithm of [2, 13], which has a low latency in "good" runs with no failures and no failure suspicions, while ensuring that, in runs with failures or failure suspicions, inconsistencies can be repaired and do not propagate to clients. Repairing the inconsistencies requires the ability for the replicas to rollback, which can be done for instance using transactions. The algorithm is optimistic in the sense that it assumes that failures are rare and is optimized for this case.

The rest of the paper is organized as follows. Section 2 introduces background concepts and related work about replication and optimistic algorithms. Section 3 describes the system model. Section 4 presents an overview of the optimistic active replication algorithm, and Section 5 gives a formal description of that algorithm. Due to space constraints, the proof of the algorithm has been omitted. It can be found in [8]. Finally, Section 6 concludes the paper.

2. Background and related work

2.1. Replication techniques

Fault-tolerance in distributed systems is typically achieved through replication. The literature distinguishes between two main classes of replication techniques: passive replication and active replication [15]. In passive replication the client only interacts with one replica, called the *primary*: the primary handles the client request and sends back the response. The primary also issues messages to the secondaries (the other replicas) in order to update their state. In active replication the client sends its request to all the replicas, which all handle the request and send back the response to the client. The client waits only for the first reply. Note that active replication requires the servers to be deterministic.

Ensuring consistency of the replicas is the main difficulty of replication techniques. One well known technique are *quorum systems* [10]. However, quorum systems typically require a transactional infrastructure. This is not the case for group communication, another infrastructure for managing replication, which we consider here.

With active replication, consistency is ensured by having the clients invoke a group communication primitive called *Atomic Broadcast* (also called *Total Order Broadcast*). Atomic Broadcast guarantees that the requests sent by the clients are received by all replicas in the same order. With passive replication, consistency requires a group communication infrastructure that provides a *group membership service* (to select the primary), and a *view synchronous broadcast* (to be used by the primary to update the state of the secondaries) [11].

2.2. Design issues for Atomic Broadcast algorithms

We consider in the paper only active replication and Atomic Broadcast. Numerous Atomic Broadcast algorithms have been published in the last 15 years. A good survey can be found in [5]. The different algorithms differ mainly by the assumptions they make with respect to the system model: they typically assume either a synchronous system, or an asynchronous system augmented with failure detectors. From a practical point of view, modeling the system as synchronous when the network and processor load are variable requires to be pessimistic for the bounds on message transmission delay and relative processor speeds. This leads to a large crash detection time, i.e., a large failover time, which is inadequate for time critical applications.

A better approach consists in assuming an asynchronous model augmented with a failure detector, which makes Atomic Broadcast solvable [3]. In this context, Atomic Broadcast algorithms can further be classified in two categories: (1) those that rely on a group membership oracle,[1] and (2) those that rely on a failure detector oracle. The Isis Atomic Broadcast algorithm [2] belongs to the first category, while the Chandra-Toueg Atomic Broadcast algorithm [3] belongs to the second category. MULTI-PAXOS [18], which can be seen as an Atomic Broadcast algorithm, also belongs to the second category and has a cost similar to the Chandra-Toueg Atomic Broadcast algorithm. As argued in [4], the algorithms in the first category force the crash of processes that have been incorrectly suspected, which is not the case of the algorithms in the second category. This has an important consequence: the overhead due to an incorrect failure suspicion is higher with algorithms in the first category, where the incorrectly suspected processes join again after being excluded (to keep the same degree of replication), which includes the costly state transfer operation.[2] For this reason we consider in the paper an optimistic active replication technique that does not rely on a group membership oracle.

[1] Also called group membership *service*.

2.3. Optimistic algorithms

Achieving low cost in the absence of failures is an important — but not new — idea. Achieving fault-tolerance is often considered to be expensive and to lead to a significant overhead. The same holds for Atomic Broadcast, which was often criticized as being too expensive. In spite of that, designing *optimistic* Atomic Broadcast algorithms or *optimistic* active replication techniques was largely ignored until recently [17], even though optimistic algorithms were known since several years in the context of concurrency control [1] and file system replication [12]. In the context of active replication, Pedone distinguishes two dimensions of optimism [16]:

i) Optimism at the level of the Atomic Broadcast algorithm.
ii) Optimism at the level of the treatment of the client request by the replicated service.

The Optimistic Atomic Broadcast algorithm [17] is an example of (i). The algorithm makes the optimistic assumption that in a LAN messages are spontaneously received in total order with high probability, which is experimentally confirmed. If this assumption is met, the algorithm delivers messages faster than known Atomic Broadcast algorithms. However, if the assumption does not hold, the algorithm is less efficient than other algorithms (but still delivers messages in total order).

The optimistic processing of transactions over Atomic Broadcast [14] is an example of (ii). [14] distinguishes two delivery events following the Atomic Broadcast of message m: the optimistic delivery denoted by *Opt-*$deliver(m)$ and the traditional total order delivery denoted by $Adeliver(m)$. *Opt-*$deliver(m)$ occurs upon reception of m. Even though the order of m is not yet decided, the processing of the request contained in m is optimistically started. As in [17], the optimism is related to the spontaneous total order property of LANs. If $Adeliver(m)$ invalidates on some server s_i the temporary order defined by *Opt-*$deliver(m)$, then the replica s_i must rollback and undo the processing of request m. Notice that in this case the inconsistency is *internal* to the server s_i: no response is sent to a client before the Adelivery of m.

[2] As explained in [4], the only advantage of the algorithms in the first category over those in the second category is the "finite storage" issue related to the implementation of reliable channels over fair-lossy channels. It is trivial to extend algorithms that belong to the second category with a group membership oracle in order to address the "finite" storage problem (without requiring a perfect failure detector). A full discussion of this issue is out of the scope of the paper.

This last example leads us to suggest another classification of optimism in the context of active replication and/or Atomic Broadcast:

a) Optimism that never leads to inconsistency.
b) Optimism that leads to internal inconsistencies only (server inconsistencies only).
c) Optimism that leads to external inconsistencies (server and client inconsistencies).

The optimistic Atomic Broadcast algorithm of [17] is an example of (a). The optimistic delivery and processing of [14] is an example of (b). The optimistic concurrency control in the context of transactions is another example of (b) (optimistic concurrency control never leads a client to see an inconsistent state of the data). The Isis Atomic Broadcast algorithm based on a sequencer [2] is an example of (c): in some runs the total order delivery of messages can be violated leading clients, in the context of active replication, to receive inconsistent responses (see Section 2.4). The violation of total order can occur with a variable probability, depending on the network/processor load, and on the timeout value chosen for suspecting crashed processes. Despite the potential inconsistencies the algorithm was chosen in Isis for its low cost in absence of failures.

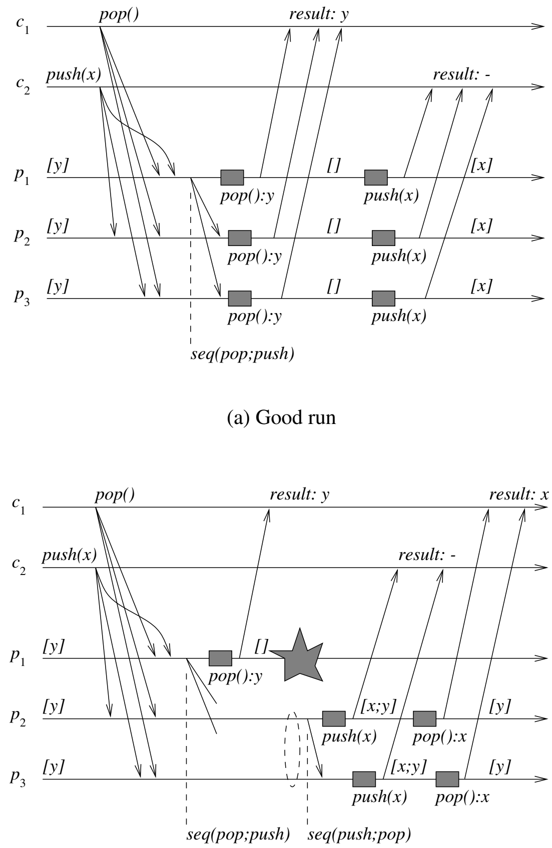
The optimistic active replication technique given in the paper builds on this last algorithm. It prevents however external inconsistencies, even though internal inconsistencies are possible. For this reason the technique is to be classified under category (b). In the next section we briefly recall the Isis Atomic Broadcast algorithm [2]. The same idea appears in [13].

2.4. The Isis sequencer-based atomic broadcast algorithm

Consider a group G of replicated servers, and a client process issuing an Atomic Broadcast of m to G. The sequencer algorithm works as follows:

1. The message m is sent to the replicas in G.
2. One of the replicas in G, called the sequencer, assigns sequence numbers to messages and sends these numbers to G.
3. Each replica in G delivers the messages according to their sequence numbers.

A run of this algorithm, with no failure and no failure suspicion, is shown in Figure 1(a): two clients c_1 and c_2 perform $push$ and pop operations on a stack replicated on three processes p_1, p_2, and p_3. Process p_1 is the sequencer, and it orders the pop operation of c_1 before the $push$ operation of c_2. A run with a failure leading to external inconsistency is depicted in Figure 1(b). The sequencer p_1 crashes (or is incorrectly suspected) after sending the reply to the client; the sequencing message is not received by the other replicas. The new sequencer p_2 decides upon a different ordering of messages: the reply previously received by the client is inconsistent with the state of p_2 and p_3.



(a) Good run

(b) Inconsistent run

Figure 1. Sequencer-based Atomic Broadcast algorithm

3. System model

For our optimistic active replication algorithm we consider an asynchronous system with processes that communicate by message passing. Processes are either *client* processes or *server* processes. For simplification, we consider one single replicated service, with server processes denoted by $\Pi = \{p_1, \dots, p_n\}$. The client processes, which are not part of Π, are denoted by $c_1, \dots, c_n$. Each server process maintains a replica which supports a limited form of *rollback*: at any time a suffix of the sequence of messages may be undone, which returns the replica to a former state. Note that we never need to undo messages other than a suffix.

Processes only fail by crashing (i.e., we do not consider Byzantine failures). Processes are connected by reliable FIFO channels, defined in terms of the two primitives $send$ and $receive$. Moreover, we assume the existence of a Reliable Multicast primitive, denoted by *R-multicast(m,*Π*)*, defined by the following properties: *(Validity)* if a correct process executes R-multicast(m,Π), then every correct process in Π eventually R-delivers m, *(Agreement)* if a correct process R-delivers m, then all correct processes in Π eventually R-deliver m, and *(Integrity)* for any message m, every process R-delivers m only once, and only if m was previously R-multicast.

To handle the cases where the current sequencer process crashes or is incorrectly suspected (and only in these cases), our optimistic active replication algorithm relies on a consensus *oracle*. It is well known that such an oracle is not implementable in an asynchronous system [9]. However, as shown in [3], the consensus oracle is implementable in an asynchronous system augmented with the failure detector $\diamond\mathcal{S}$ and a majority of correct processes. The paper assumes the failure detector $\diamond\mathcal{S}$ among Π and a majority of correct processes in Π.

4. Overview of the algorithm

Like most optimistic algorithms, the optimistic active replication (OAR) algorithm is based on the assumption that failures are infrequent, i.e., the algorithm is optimized for failure-free runs. It uses a lightweight sequencer protocol similar to the protocol described in Section 2.4, which requires a minimal number of communication phases in absence of failures. The originality of the algorithm is that, despite the fact that the replies sent to a client may be different, it guarantees that the client will never "adopt" an inconsistent reply. The algorithm also includes mechanisms for resolving the temporary inconsistencies that may affect some servers.

To send its request to the servers, a client uses a *Reliable Multicast* primitive, which ensures that if one correct server receives the request, all correct servers eventually receive the request. The client then waits for replies from the servers. Contrary to the usual active replication techniques, the replies might here not be identical. To allow the client to select a "correct" reply, each server reply r contains an additional *weight* field. This field identifies the set of servers that endorse reply r. The client waits for a quorum of replies, and selects the reply according to a majority rule. The rule ensures the selection of the "correct" reply. The details are given later.

The algorithm responsible for ordering the messages among the servers proceeds in a sequence of *epochs*. Each epoch has two *phases*. In phase 1 — the optimistic phase — the algorithm uses a sequencer to optimistically order the messages fast, assuming no failure. The optimistic message delivery is called *Opt-deliver*. As soon as a request message is Opt-delivered by some server, the server processes the request and generates a reply, which is sent back to the client.

If the sequencer crashes or is incorrectly suspected, the algorithm proceeds to phase 2 — the conservative phase — where it uses a different paradigm (based on consensus) to conservatively order messages. The conservative message delivery is called *A-deliver*. As with optimistic delivery, upon A-delivery of a request message, the request is immediately processed and the reply sent back to the client. If the sequences of messages Opt-delivered by each server during phase 1 are not identical, the conservative ordering of phase 2 might invalidate the optimistic ordering of phase 1. However, the following safety property holds:

Majority guarantee. If a majority of processes Opt-deliver m_1 before m_2, then no process A-delivers m_2 before m_1.

For the (rare) cases where — because of the crash or the (false) suspicion of the sequencer — the conservative order is different from the optimistic order, we introduce the *Opt-undeliver(m)* primitive. This primitive notifies the server that message m has been Opt-delivered in a wrong order, and that the effects induced by the processing of m must be undone. A message Opt-undelivered by a correct process will eventually be delivered again (Opt-delivered or A-delivered). The A-delivery of a message can never be undone.

Phase 2 is handled by the problem that we call *Conservative-order* (or simply *Cnsv-order*), which is solved by reduction to a consensus problem. *Cnsv-order* has two input parameters ($O\_delivered$, $O\_notdelivered$), and outputs two sequences of messages (Bad, New):

$$\{Bad; New\} \leftarrow \textit{Cnsv-order}(O\_delivered, O\_notdelivered)$$

For each server p, $O\_delivered$ is the sequence of messages Opt-delivered by p during the current epoch; $O\_notdelivered$ is the sequence of messages received but not yet delivered by p; Bad is the sequence that p has to Opt-undeliver, and New is the sequence that p has to A-deliver (after the Opt-undelivery operation).

Note that a minority of processes can deliver messages out of order only if the minority is suspected by the majority (e.g., a minority partition cannot communicate with the majority partition). However, this minority partition does not need to be declared faulty and commit suicide, like in the primary partition paradigm [2, 19].

Figures 2, 3, and 4 illustrate different runs of the OAR algorithm. Servers p_i receive incoming messages from clients

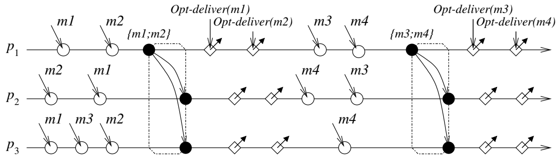


Figure 2. The OAR algorithm with no failure nor suspicion.

unordered (white circles in the figures). Process p_1 is the sequencer. Black circles represent the reception of the ordering message from the sequencer. Message delivery (Opt-delivery, A-delivery) is represented by white diamonds, and message undelivery is represented by grey diamonds. Each time a message is delivered to a server, it is immediately processed, and a reply is sent back to the client (this is represented in the figures by the outgoing arrow on the delivery events). In Figure 2, no failure occurs, processes only execute phase 1 of the OAR algorithm, and the client receives the reply after three communication steps *(request, sequencing message, response)*.[3]

In Figure 3, the sequencer p_1 fails just after Opt-delivery of m_3 and m_4, or is incorrectly suspected. Only process p_2 receives ordering information from p_1 and Opt-delivers m_3 and m_4. As a result of the suspicion, processes proceed to phase 2. In this example, since a majority of processes (p_1 and p_2) have Opt-delivered m_3 before m_4, no process can deliver these messages in a different order. Thus, *Cnsv-order* returns $Bad = \epsilon$ (the empty sequence), $New = \epsilon$ for p_2 and $Bad = \epsilon$, $New = \{m_3; m_4\}$ for p_3.

The scenario of Figure 4 is similar to the one of Figure 3, except that the sequencing message $\{m_3; m_4\}$ is not received by p_2. Therefore *Cnsv-order* may decide on a different ordering and returns $Bad = \epsilon$, $New = \{m_4; m_3\}$ at p_2 and p_3. If p_1 does not crash but is incorrectly suspected (e.g., because of a network partition), *Cnsv-order* returns $Bad = \{m_3; m_4\}$ and $New = \{m_4; m_3\}$ at p_1, which leads p_1 to undeliver m_3 and m_4.

5. The optimistic active replication algorithm

5.1. Notation

The OAR algorithm manages sequences of messages. This leads us to introduce the following notation. Sequences (of messages) are denoted as follows: $\{m_1; m_2; m_3\}$. Sets are denoted as usual using commas: $\{m_1, m_2, m_3\}$. An empty sequence is represented by ϵ, and an empty set by the usual $\emptyset$ symbol.

[3]Both the Chandra-Toueg Atomic Broadcast algorithm [3] and the MULTIPAXOS algorithm [18] require in the best case one additional communication step.

As in [17], we use the operators $\oplus$ and $\ominus$ for representing concatenation and decomposition of sequences, and the function $\odot$ for representing the common prefix of a set of sequences. More precisely, $seq_1 \oplus seq_2$ is the sequence of all the messages from seq_1 followed by all the messages of seq_2; $seq_1 \ominus seq_2$ is the sequence of all messages from seq_1 that are not in seq_2; and $\odot(seq_1, \ldots, seq_n)$ is the longest sequence that is a common prefix to $seq1, \ldots, seq_n$.

Additionally, we use the symbol $\uplus$ to designate a function that takes a list of sequences $seq_1, \ldots, seq_n$ as argument and produces a new sequence by appending all sequences together and removing duplicates. We also assume an implicit conversion from a sequence to a set, whenever we use the following set operators: $\cap$, $\cup$, $\in$, and $\notin$. For instance, $seq_1 \cap seq_2 = \emptyset$ means that there is no common element in seq_1 and seq_2.

5.2. The client-side algorithm

Figure 5 describes the client-side code of the OAR algorithm. The client R-multicasts its request to all processes of Π (line 2) and waits for a quorum of replies (line 3). The *weight* W_i^m (m is the request message) of $reply_i^m$ is a set that identifies the servers that endorse $reply_i^m$. Said differently, let p_i be the server that has sent $reply_i^m$: the weight W_i^m contains the identifiers of all servers that p_i knows to deliver the request in the same order as itself, i.e., to generate the same reply. The value k in the reply identifies the epoch number during which the servers generate the reply.

The client waits until it receives a set of replies that contribute to a total weight greater than or equal to $\left\lceil \frac{|\Pi|+1}{2} \right\rceil$ (majority weight). Our algorithm guarantees that individual replies of an epoch k that have a minority weight are all identical. Similarly, individual replies that have a majority weight are all identical. If replies with a majority weight are different from the replies that have a minority weight for an epoch k, the latter cannot have together a total weight greater than or equal to $\left\lceil \frac{|\Pi|+1}{2} \right\rceil$ and the former are the correct replies. Therefore, when the set of replies received by the client reaches a majority weight, the client adopts any reply with the largest weight (lines 5–6).

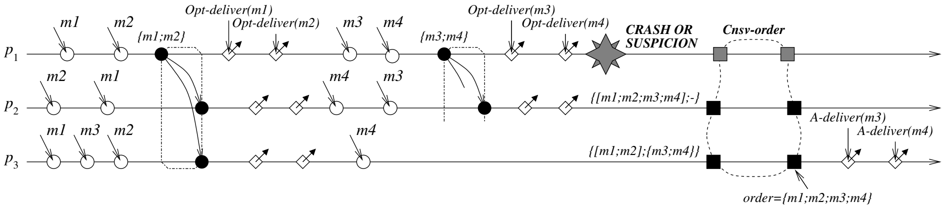


Figure 3. The OAR algorithm with the crash or suspicion of the sequencer (no Opt-undelivery).

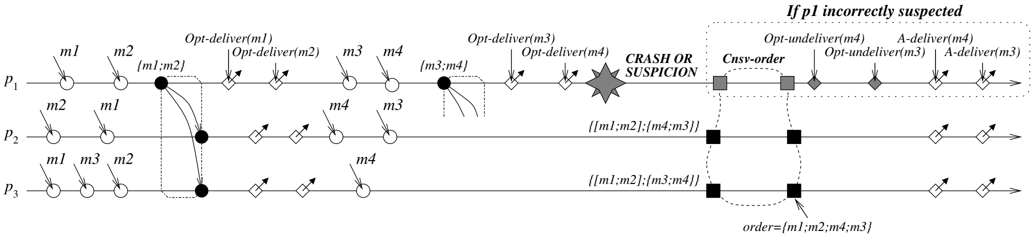


Figure 4. The OAR algorithm with the crash or suspicion of the sequencer (if the sequencer is incorrectly suspected, then Opt-undelivery of m_3 and m_4 at the sequencer).

---

procedure *OAR-multicast* (m, Π)
  R-multicast (m, Π)
  wait until for some k, for j processes $p_1, \ldots, p_j$:
    received $(reply_i^m, W_i^m, k)$ from p_i **and** $\cup_{i=1}^{j} W_i^m \geq \left\lceil \frac{|\Pi|+1}{2} \right\rceil$
  $W_{max} \leftarrow$ largest W_i^m such that received $(reply_i^m, W_i^m, k)$ from p_i
  $r \leftarrow$ select one $reply_i^m$ such that received $(reply_i^m, W_{max}, k)$ from p_i
  return r

Figure 5. The OAR algorithm: client code

5.3. The server-side algorithm

Figure 6 gives the server-side code of the OAR algorithm. Each process of Π maintains a list (i) of incoming messages ($R\_delivered$, line 2), (ii) of messages conservatively delivered ($A\_delivered$, line 2), and (iii) of messages optimistically delivered during the current epoch ($O\_delivered$, line 3). The algorithm progresses through a sequence of epochs, represented by a monotonously increasing integer k (line 4). An epoch is composed of two parts: the optimistic phase (phase 1) and the conservative phase (phase 2).

The OAR algorithm consists of several tasks. These tasks can execute in any order, but in mutual exclusion. Task 0 receives incoming messages and adds them to $R\_delivered$ (line 7). This task is active in phase 1 and in phase 2.

Task 1a, 1b, and 1c are active during phase 1. In Task 1a, the sequencer s periodically checks whether there are some messages which have been received but not yet ordered (line 8). If so, s orders these messages in a sequence and sends the sequence to all processes of Π (line 10). After that, and in order to simplify the algorithm, we assume that the sequencer immediately delivers this message, i.e., immediately executes Task 1b.

In Task 1b, when some process p — including the sequencer itself — delivers the sequence $msgSet^k$ from the sequencer (line 11), it iterates through this sequence (line 16) and performs the following operations: for each message m, process p Opt-delivers m, processes the request and generates the reply (line 17), adds m to $O\_delivered$ (line 18), and sends a reply to the client (line 19). The weight (lines 12–15) is equal to $\{s\}$ if p is the sequencer, and to $\{s, p\}$ otherwise (p knows that s delivers m in the same order).

In Task 1c, when a process suspects the sequencer to have failed (i.e., when the sequencer belongs to the list $\mathcal{D}_p$ of processes suspected by p's failure detector), it R-multicasts a PHASEII message to notify the other processes to proceed to the conservative phase 2 (line 21).

Task 2 is the task of the conservative phase 2. Process p proceeds to that phase upon delivery of a PHASEII message (line 22).[4] Process p then invokes the *Cnsv-order* function, which conservatively orders messages (line 24). The function *Cnsv-order* takes as argument two sequences: the sequence of messages Opt-delivered by p during epoch k

[4]Note that the reception of multiple PHASEII messages for epoch k is not a problem: at line 32, the algorithm proceeds to epoch $k + 1$, thus making redundant messages obsolete.

1: Initialization:
2: $R\_delivered \leftarrow \epsilon$; $A\_delivered \leftarrow \epsilon$
3: $O\_delivered \leftarrow \epsilon$
4: $k \leftarrow 0$ {*current epoch*}
5: $s \leftarrow p_1$ {*p_1 is the first sequencer*}

6: **when** *R-deliver* (m) {*Task 0: buffer incoming client message*}
7: $R\_delivered \leftarrow R\_delivered \oplus \{m\}$

8: **when** $p = s$ **and** $(R\_delivered \ominus A\_delivered) \ominus O\_delivered \neq \epsilon$ {*Task 1a: sequencer orders messages*}
9: $O\_notdelivered \leftarrow (R\_delivered \ominus A\_delivered) \ominus O\_delivered$
10: send $(k, O\_notdelivered)$ to all

11: **when** *deliver* $(k, msgSet^k)$ {*Task 1b: processes opt-deliver messages*}
12: **if** $p = s$ **then**
13: $W \leftarrow \{s\}$
14: **else**
15: $W \leftarrow \{p, s\}$ {*p knows that s delivers m in the same order*}
16: **for all** $m \in msgSet^k$ **do**
17: $reply \leftarrow$ *Opt-deliver* (m)
18: $O\_delivered \leftarrow O\_delivered \oplus \{m\}$
19: send $(reply, W, k)$ to $sender(m)$

20: **when** $s \in \mathcal{D}_p$ {*Task 1c: suspicion*}
21: *R-multicast* $(k,$PHASEII$, \Pi)$ {*to all servers*}

22: **when** *R-deliver* $(k,$PHASEII$)$ {*Task 2: conservative ordering*}
23: $O\_notdelivered \leftarrow (R\_delivered \ominus A\_delivered) \ominus O\_delivered$
24: $\{Bad; New\} \leftarrow$ *Cnsv-order* $(k, O\_delivered, O\_notdelivered)$
25: **for all** $m \in Bad$ **do**
26: *Opt-undeliver* (m)
27: **for all** $m \in New$ **do**
28: $reply \leftarrow$ *A-deliver* (m)
29: send $(reply, \Pi, k)$ to $sender(m)$
30: $A\_delivered \leftarrow A\_delivered \oplus (O\_delivered \ominus Bad) \oplus New$
31: $O\_delivered \leftarrow \epsilon$
32: $k \leftarrow k + 1$
33: $s \leftarrow p_{(k \bmod n)+1}$ {*next sequencer*}

Figure 6. The OAR algorithm: code of server process p

($O\_delivered$), and the sequence of messages that have been R-delivered by p but not yet ordered. The function returns two values: a sequence of messages that p Opt-delivered in the wrong order (Bad) and a sequence of messages that have just been conservatively ordered but not yet delivered (New). The *Cnsv-order* function is defined in Section 5.4.

In the (unlikely) event of the sequence Bad being not empty, process p first Opt-undelivers these messages[5] (line 26). Then p A-delivers sequentially all the messages in New (line 28) and sends a reply with a weight equals to Π to the client (line 29). The weight indicates agreement of the order that has been decided. Finally, p adds the messages delivered during epoch k to $A\_delivered$, clears $O\_delivered$, and proceeds to epoch $k + 1$ (lines 30–33).

Epoch $k + 1$ starts in the "optimistic" mode, with another process acting as the sequencer: the new sequencer is the next process modulo n (line 33). Another policy can be easily implemented, e.g., the next sequencer can be decided as part of the agreement problem solved in phase 2. In the new epoch the algorithm behaves exactly as in the previous epoch, i.e., client requests are R-delivered (line 6), the sequencer sends sequencing messages (line 10), etc.

Remark. In the OAR algorithm of Figure 6, execution of phase 2 allows to "forget" about messages Opt-delivered (line 31). So, if phase 2 is executed only rarely, the sequence $O\_delivered$ can become extremely long, which might slow down the execution of the next instance of *Cnsv-order* in phase 2. The problem can easily be solved by having the sequencer R-multicast a PHASEII message on a regular basis (e.g., every n requests or every t seconds) to explicitly execute phase 2. More lightweight solutions to garbage collect the $O\_delivered$ sequence also exist, but are not detailed here.

5.4. Specification of *Cnsv-order*

We specify the *Cnsv-order* problem by the following properties, which are commented below:

Termination. If a correct process p calls *Cnsv-order* then eventually p gets the result $\{Bad_p; New_p\}$.

Agreement. For any two correct processes p and q, we have $(O\_delivered_p \ominus Bad_p) \oplus New_p = (O\_delivered_q \ominus Bad_q) \oplus New_q$.

Unicity. For all processes p, we have $New_p \cap (O\_delivered_p \ominus Bad_p) = \emptyset$.

Non-triviality. If for a majority of processes q, $m \in O\_delivered_q \cup O\_notdelivered_q$, then for all correct processes p, we have $m \in (O\_delivered_p \ominus Bad_p) \oplus New_p$.

Validity. If for any process p, $m \in New_p$, then for at least one process q, we have $m \in O\_delivered_q \cup O\_notdelivered_q$.

Undo legality. For all processes p, we have $(O\_delivered_p \ominus Bad_p) \oplus Bad_p = O\_delivered_p$.

Undo consistency. For all processes p, if $m \in Bad_p$ then for a majority of processes q, we have $m \notin O\_delivered_q$.

The termination property ensures progress. The agreement property ensures agreement on the sequence of messages delivered in each epoch k. The unicity property forbids a message to be delivered twice during epoch k (a message can be Opt-delivered and A-delivered only if it is meanwhile Opt-undelivered, i.e., in the sequence Bad_p). The non-triviality property states that any message that has been received by a majority of processes during epoch k will also

[5] Although not shown in the algorithm, undelivery of messages should generally be performed in the reverse order of delivery, i.e., starting by the last message of Bad.

be delivered during epoch k, and thus prohibits the trivial solution where New_p is always empty. The validity property prevents arbitrary messages from being in New_p. The undo legality guarantees that Bad_p is well formed, i.e., a suffix of $O\_delivered_p$. The undo consistency property prevents messages from being in Bad_p if they have been Opt-delivered by a majority of processes.

The above properties are sufficient to ensure the correctness of the OAR algorithm. However, we add the following property, which guarantees that no optimistic delivery will be unnecessarily undone.

Undo thriftiness. For all processes p, we have $\odot(Bad_p, New_p) = \epsilon$.

5.5. Implementation of *Cnsv-order*

The *Cnsv-order* problem can be solved by reduction to a consensus problem. Recall that consensus is defined in terms of two primitives $propose(v)$ and $decide(v)$, and specified as follows:

Termination. Each correct process eventually decides.

Validity. If a process executes $decide(v)$, then some process has executed $propose(v)$.

Agreement. No two correct processes decide differently.

For solving the *Cnsv-order* problem, we use a slightly different specification. The *Validity* property is replaced by the following *Maj-validity* property, in which the decision V is a sequence of initial values:

Maj-validity. If a process executes $decide(V)$, then V is a sequence of values such that, for a majority of processes p_i, if p_i has executed $propose(v_i)$, then $v_i \in V$.

Said differently, the decision sequence contains the initial value of a majority of processes. The consensus with the Maj-validity property can be solved by minor modifications to the consensus algorithm based on $\Diamond\mathcal{S}$ [3]. For a description of these modifications, see [7].

The implementation of the *Cnsv-order* function is given in Figure 7. Each process computes three sequences: *Good* (messages Opt-delivered in the right order during epoch k), Bad (messages Opt-delivered in the wrong order during epoch k), and New (messages to A-deliver during epoch k). At line 3, the processes propose their initial value for consensus, which consists of a pair of sequences of messages (O_dlv, O_notdlv). The decision at line 4 is a sequence of pairs denoted by $D^k \equiv \{(dlv_1, notdlv_1); (dlv_2, notdlv_2); \ldots\}$. Upon decision, p selects the longest sequence dlv_i denoted by dlv_{max} (line 5).[6]

[6]The sequences dlv_i can differ only by their length, i.e., given any two sequences dlv_i, dlv_j, if they are not equal, one is a prefix of the other.

1: **procedure** *Cnsv-order* (k, O_dlv, O_notdlv)
2: $\quad Bad \leftarrow New \leftarrow \epsilon$
3: $\quad$ *propose* $(k, (O\_dlv, O\_notdlv))$
4: $\quad$ **wait until** *decide* (k, D^k)
$\qquad\qquad \{D^k$ *is a sequence* $\{(dlv_1, notdlv_1); (dlv_2, notdlv_2); \ldots\}\}$
5: $\quad dlv_{max} \leftarrow$ longest dlv_i such that $(dlv_i, notdlv_i) \in D^k$
6: $\quad$ **if** $O\_dlv = \odot(O\_dlv, dlv_{max})$ **then**
7: $\qquad New \leftarrow dlv_{max} \ominus O\_dlv$
8: $\qquad Good \leftarrow O\_dlv$
9: $\quad$ **else**
10: $\qquad Good \leftarrow \odot(O\_dlv, dlv_{max})$
11: $\qquad Bad \leftarrow O\_dlv \ominus Good$
12: $\quad notdlv \leftarrow \uplus_i(notdlv_i$ such that $(dlv_i, notdlv_i) \in D^k)$
13: $\quad notdlv \leftarrow notdlv \ominus dlv_{max}$
14: $\quad New \leftarrow New \oplus notdlv$
15: $\quad$ **if** $\odot(Bad, New) \neq \epsilon$ **then**
16: $\qquad prefix = \odot(Bad, New)$
17: $\qquad Good \leftarrow Good \oplus prefix$
18: $\qquad Bad \leftarrow Bad \ominus prefix$
19: $\qquad New \leftarrow New \ominus prefix$
20: $\quad$ **return** $\{Bad; New\}$

Figure 7. The conservative ordering procedure

If for p, O_dlv is a subsequence of dlv_{max} (line 6), then p sets New equal to the subsequence of dlv_{max} that it did not yet deliver (line 7). *Good* is set to the sequence already delivered (line 8). However, if for p, dlv_{max} is shorter that O_dlv (line 6) (i.e., p's initial value is not contained in the decision), there may be a risk of inconsistent ordering. In this case, *Good* is set to the sequence of messages delivered in the correct order (line 10), and Bad is set to the sequence of wrongly-ordered messages (line 11).

Process p then generates deterministically a sequence $notdlv$ with all $notdlv_i$ sequences in the decision D^k of the consensus $\#k$ (line 12), makes sure that this sequence does not contain any message correctly Opt-delivered or already scheduled for delivery (line 13), and adds this sequence to New (line 14).

Finally, at line 15, p checks if Bad and New have a common prefix, i.e., if it will undeliver some messages and re-deliver them in the same order. This may happen if some messages are added to Bad because they are not part of any dlv_i in D^k, but are incidentally rescheduled for delivery in the same order. In that case, p adds these messages to *Good* and removes them from both New and Bad (lines 17–19). This ensures that the implementation of *Cnsv-order* satisfies the undo thriftiness property of Section 5.4.

6. Conclusion

Our optimistic active replication algorithm solves Atomic Broadcast as a subproblem.[7] However, the origi-

[7]Where the atomic delivery of a

nality of the OAR algorithm is to handle Atomic Broadcast as a white box, rather than as black box as usually done. This allows us to have an algorithm that is both efficient in terms of latency in "good" runs, while always preserving consistency at the client level. Similarly to sequencer-based Atomic Broadcast algorithms (e.g., [2, 13]), our algorithm requires only one communication step for ordering messages in absence of failures, but unlike sequencer-based protocols it prevents inconsistencies that may occur with these algorithms.

Reconciliation among the servers is handled thanks to the *Opt-undeliver* primitive. The probability of having to Opt-undeliver a message is very low. It requires a combination of three events: (1) the sequencer s fails or is suspected in such a way that only a minority of processes (call them P_{min}) have received ordering information from s, (2) no process of P_{min} has its initial value in the decision of the consensus, and (3) the messages Opt-delivered only by the processes of P_{min} are conservatively ordered differently by *Cnsv-order*. Events (1) and (2) can happen for example if $s \in P_{min}$ and P_{min} is partitioned from other processes of Π.

The OAR algorithm is well-adapted to a transactional environment with a save-point facility. For each replica p, all changes are made in the context of a single transaction, and a new savepoint is declared before each Opt-delivery of a message in phase 1. During phase 2, the system returns to the savepoint before the first message which is in Bad_p, and then it applies the messages in $Good_p$. At this time the transaction can be committed, and a new transaction begin.

We believe that the OAR algorithm presented in this paper offers a good compromise between efficiency (low latency) and consistency, by not trying to preserve server consistency by all means, but always ensuring consistency at the client level.

Acknowledgement. We would like to thank the anonymous reviewers for their useful comments.

References

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

[2] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic multicast. *Transactions on Computer Systems*, 9(3):272–314, Aug. 1991.

[3] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[4] B. Charron-Bost, X. Défago, and A. Schiper. Time vs. Space in Fault-Tolerant Distributed Systems. In *Proceedings of the 6th International Workshop on Object-oriented Real-Time Dependable Systems (WORDS)*, Rome, Italy, Jan. 2001. to appear.

[5] X. Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Aug. 2000. Number 2229.

[6] X. Défago, A. Schiper, and P. Urban. Totally Ordered Broadcast and Multicast Algorithms: a Comprehensive Survey. TR DSC/2000/036, Dept. of Communication Systems, EPFL, Switzerland, October 2000.

[7] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.

[8] P. Felber and A. Schiper. Optimistic active replication. Technical Report DSC/2000/035, École Polytechnique Fédérale de Lausanne, Switzerland, Sept. 2000.

[9] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):217–246, 1985.

[10] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.

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

[12] R. Guy, G. Popek, and T. Page Jr. Consistency algorithms for optimistic replication. In *1st IEEE Int. Conference on Network Protocols*, October 1993.

[13] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, pages 222–230, Arlington, TX, USA, May 1991.

[14] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over Optimistic Atomic Broadcast Protocols. In *IEEE 19th Intl. Conf. Distributed Computing Systems (ICDCS-19)*, pages 424–431, June 1999.

[15] S. Mullender, editor. *Distributed Systems*, chapter 7 and 8. Addison-Wesley, 2nd edition, 1993.

[16] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Dec. 1999. Number 2090.

[17] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In *12th. Intl. Symposium on Distributed Computing (DISC'98)*, pages 318–332. Springer Verlag, LNCS 1499, September 1998.

[18] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos Algorithm. In *11th. Intl. Symposium on Distributed Computing (DISC'97)*, pages 111–125. Springer Verlag, LNCS 1320, September 1997.

[19] R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Tr, Dept. of Computer Science, Technion, Israel, September 99.

message corresponds to the message being either (1) A-delivered or (2) Opt-delivered but not Opt-undelivered.