Solving Agreement Problems with Weak Ordering Oracles[*]

Fernando Pedone[⋆] André Schiper[†] Péter Urbán[†] David Cavin[†]

[⋆]Hewlett-Packard Laboratories
Software Technology Laboratory
Palo Alto, CA 94304, USA
E-mail: fernando_pedone@hp.com

[†]Ecole Polytechnique Fédérale de Lausanne (EPFL)
Faculté Informatique & Communications
CH-1015 Lausanne, Switzerland
E-mail: {andre.schiper, peter.urban, david.cavin}@epfl.ch

Abstract

Agreement problems, such as consensus, atomic broadcast, and group membership, are central to the implementation of fault-tolerant distributed systems. Despite the diversity of algorithms that have been proposed for solving agreement problems in the past years, almost all solutions are *crash detection based* (*CDB*). We say that an algorithm is CDB if it uses some information about the status *crashed/not crashed* of processes. Randomized consensus algorithms are rare exceptions non-CDB algorithms. In this paper, we revisit the issue of non-CDB algorithms. Instead of randomization, we consider *ordering oracles.* Ordering oracles have a theoretical interest (e.g., they extend the state of the art of non-CDB algorithms) as well as a practical interest (e.g., they remove altogether the burden involved in tuning timeout mechanisms). To illustrate their use, we present solutions to consensus and atomic broadcast, and evaluate the performance of the atomic broadcast algorithm in a cluster of workstations.

[*]Also appears in EPFL Technichal Report IC/2002/010, March 2002.

1 Introduction

The paper addresses the issue of solving agreement problems, which are central to the implementation of fault-tolerant distributed systems. Consensus, atomic broadcast, and group membership are examples of agreement problems. One of the key issues when solving an agreement problem is the choice of the system model. Many system models have been proposed in the past years: synchronous models [11, 15, 16, 6], partially synchronous models [12], asynchronous models with failure detectors [9, 8, 2, 3], timed asynchronous models [10], etc. Despite the diversity of these models, almost all algorithms that have been proposed to solve agreement problems have the common point of being *Crash Detection Based* (*CDB*). We say that an algorithm is CDB if it uses some information about the status *crashed/not crashed* of processes. Typically, a CDB algorithm contains statements like "**if** p has crashed **then** ..." or "**if** p is suspected to have crashed **then** ..." There is a notable exception to the near universality of CDB algorithms: randomized consensus algorithms [7, 19], which are not CDB.

There are two motivations for this work. The first one is theoretical: it advances the state of the art of non-CDB algorithms, a class of algorithms that has been under-explored. The second motivation is practical: CDB algorithms require tuning of the failure-detection mechanism they use, which has been regarded as a nuisance for a long time [14]. To illustrate the problem, consider a system that wants to react quickly to failures. Since reaction to failures is ultimately triggered by some timer mechanism, such a system should have a very short timeout. However, due to variations in the system load, a short timeout may incur false failure suspicions. False failure suspicions are problematic because they lead to actions (e.g., determining a new coordinator) that will increase the system load and degrade performance even further. Of course, one way to reduce false failure suspicions is to increase the timeouts, but then the system no longer has a fast response to failures. By removing failure suspicions from the algorithms, we eliminate this problem of tuning: non-CDB algorithms operate in the presence of failures just as quick as they operate in their absence. Given the widespread use of computer clustering — the environment to which our algorithms are best suited, we believe that non-CDB algorithms represent an important paradigm to be exploited in the design of high-performance fault-tolerant systems in the years to come.

The non-CDB algorithms presented in the paper assume an asynchronous system model in which processes may fail by crashing. It is well known that consensus (and other agreement problems) are not solvable in an asynchronous system where processes may fail [13]. To make agreement problems solvable, we extend the asynchronous system with *ordering oracles* (Section 2), which (1) receive queries consisting of messages and (2) output messages. The specification of an oracle links the queries to the outputs. The paper defines two ordering oracles: the *k-Weak Atomic Broadcast* oracle (k-WAB oracle) where k is a positive integer, and the *Weak Atomic Broadcast* oracle (WAB oracle). Intuitively, our oracles ensure that messages are delivered in the same order from time to time. The k-WAB oracle ensures the ordering property k times. The WAB oracle ensures it an unbounded number of times.

Section 3 is devoted to consensus: we give two non-CDB algorithms, both requiring the 1-WAB oracle. The first one, called B-Consensus algorithm, is inspired by Ben-Or's randomized consensus algorithm [7] and requires $f < n/2$, where n is the total number of processes and f is the number of faulty processes. The second, called R-Consensus algorithm, is inspired by Rabin's randomized consensus algorithm [19] and requires $f < n/3$.[1] These two algorithms show an interesting resilience/complexity tradeoff: the consensus algorithm inspired by Ben-Or's algorithm has a time complexity of 3δ and $f < n/2$, while the consensus algorithm inspired by Rabin's algorithm has a time complexity of 2δ and $f < n/3$.

Our consensus algorithms can be compared to the leader-based consensus algorithms presented in [1]. Although partly similar in structure to ours[2], the consensus algorithms we propose in this paper have a better time complexity. This is because the approach in [1] relies on a leader oracle, that is, an oracle which eventually outputs the same leader process; implementing such an oracle requires a failure detection mechanism. Failure detection is not needed in our algorithms, which are based on weak ordering oracles that match the behavior of current network broadcast primitives, and so, can be efficiently implemented.

In Section 4, we consider atomic broadcast, and we extend our R-Consensus algorithm to an atomic broadcast algorithm. While the R-Consensus algorithm requires the 1-WAB oracle, the atomic broadcast algorithm requires the WAB oracle. The reduction of atomic broadcast to consensus is well known [9]. We consider here a different solution that closely integrates the ordering oracle with the atomic broadcast algorithm. Our new atomic broadcast algorithm has a time complexity of 2δ and requires $f < n/3$. Section 5 discusses some experiments we have conducted to evaluate the performance of the proposed atomic broadcast algorithms, and Section 6 concludes the paper. Proofs are given in the Appendices.

[1]Contrary to Ben-Or's and Rabin's algorithms, our algorithms solve the non-binary consensus problem.

[2]Even though this is not mentioned in [1], similarly to ours, the algorithms in [1] follow the structure of the randomized algorithms proposed by Ben-Or [7] and Rabin [19].

2 System Model and Ordering Oracles

2.1 System Model

We consider an asynchronous distributed system composed of n processes $\{p_1, \ldots, p_n\}$, which communicate by message passing. A process can only fail by crashing (i.e., we do not consider Byzantine failures). A process that never crashes is *correct*, otherwise it is *faulty*. We make no assumptions about process speeds or message transmission times.

Processes are connected through quasi-reliable channels, defined by the primitives $send(m)$ and $receive(m)$. Quasi-reliable channels have the following properties: (i) if process q receives message m from p, then p sent m to q *(no creation)*; (ii) q receives m from p at most once *(no duplication)*; and (iii) if p sends m to q, and p and q are correct, then q eventually receives m *(no loss)*.

2.2 Ordering Oracles

Every process has access to an ordering oracle, defined by properties relating queries to outputs. Queries to an oracle are requests to broadcast messages, and outputs of an oracle are messages (that were ask to broadcast by an oracle). More formally, an oracle is a set of oracle histories that satisfy properties relating queries to outputs [4].[3] We introduce the *Weak Atomic Broadcast* oracle, defined by queries of the type W-ABroadcast(r, m), and outputs of the type W-ADeliver(r, m), where r is an integer and m is a message. The parameter r groups queries and outputs, i.e., it relates different queries and outputs with the same r value. A Weak Atomic Broadcast oracle satisfies an ordering property (defined below) and the following two properties:

- **Validity:** If a correct process queries W-ABroadcast(r, m), then all correct processes eventually get the output W-ADeliver(r, m).
- **Uniform Integrity:** For every pair (r, m), W-ADeliver(r, m) is output at most once, and only if W-ABroadcast(r, m) was previously executed.

Our oracle also orders the outputs W-ADeliver(r, m). However, not all outputs need to be ordered: we call the property *weak ordering*. To define this property, we introduce the notion of *canonical sequence of queries*, and the notation $first_p(r)$. A canonical sequence of queries,

[3]In [4] an oracle is a function that takes a failure pattern F and returns a set $\mathcal{O}(F)$ of oracle histories. This is because the oracles in [4] include failure detectors. We do not consider failure detectors here as our approach does not need them.

by some process p, is a sequence of queries (1) that starts with the query W-ABroadcast$(0, -)$, and (2) where the query W-ABroadcast$(r, -)$ of p, $r \geq 0$, can only be followed by the query W-ABroadcast$(r + 1, -)$. A canonical sequence of queries can be finite or infinite. Given an integer r and a process p, we denote by $\mathit{first}_p(r)$ the message m such that (r, m) is the first pair with integer r that the oracle outputs at p. Using canonical sequences of queries, we define the following ordering properties:

- **Eventual Uniform 1-Order:** If all correct processes execute an infinite canonical sequence of queries, then there exists r such that for all processes p and q, we have $\mathit{first}_p(r) = \mathit{first}_q(r)$.

To illustrate this property, consider three processes p_1, p_2, and p_3, executing the following queries to the oracle:

- p_1 executes W-ABroadcast$(0, m_1)$; W-ABroadcast$(1, m_2)$; W-ABroadcast$(2, m_3)$.
- p_2 executes W-ABroadcast$(0, m_4)$; W-ABroadcast$(1, m_5)$; W-ABroadcast$(2, m_6)$.
- p_3 executes W-ABroadcast$(0, m_7)$; W-ABroadcast$(1, m_8)$; W-ABroadcast$(2, m_9)$.

Assume the following prefixes of sequences, obtained from the outputs of the oracles of each process (for brevity, we denote next W-ADeliver(r, m) by (r, m)):

- p_1: $(0, m_1)$; $(1, m_2)$; $(0, m_4)$; $(2, m_3)$; $(0, m_7)$; etc.
- p_2: $(0, m_4)$; $(0, m_1)$; $(1, m_5)$; $(0, m_7)$; $(2, m_3)$; etc.
- p_3: $(0, m_4)$; $(0, m_7)$; $(2, m_3)$; $(1, m_8)$; etc.

Here we have $\mathit{first}_{p_1}(0) = m_1$, $\mathit{first}_{p_2}(0) = m_4$, $\mathit{first}_{p_3}(0) = m_4$, etc. The eventual uniform 1-order property holds since we have $\mathit{first}_{p_1}(2) = \mathit{first}_{p_2}(2) = \mathit{first}_{p_3}(2) = m_3$.
We generalize the eventual uniform 1-order property as follows:

- **Eventual Uniform k-Order:** If all correct processes execute an infinite canonical sequence of queries, then there exist k values $r_1, \ldots, r_k$ such that for all processes p and q and $1 \leq i \leq k$, we have $\mathit{first}_p(r_i) = \mathit{first}_q(r_i)$.

If the oracle satisfies the eventual uniform k-order property, we will also say that the oracle *satisfies the ordering property k times*. We can now define our two oracles:

- ***k*-Weak Atomic Broadcast (*k*-WAB) Oracle:** Oracle that satisfies *eventual uniform k-order*, *validity*, and *uniform integrity* properties defined above.
- **Weak Atomic Broadcast (WAB) Oracle:** A k-WAB oracle, where $k = \infty$.

Therefore, k-WAB oracles satisfy the ordering property k times, while the WAB oracles satisfy the ordering property an infinite number of times.

2.3 Discussion

The idea of the ordering oracles stems from an experimental observation: under normal execution conditions (e.g., small or moderate load) messages broadcast in local-area networks are received in total order with high probability. We call this property *spontaneous total order*. Under high network loads, this property might be violated. More generally, one can consider that the system passes through periods when the spontaneous total order property holds, and periods when it does not hold. Our Weak Atomic Broadcast Oracles abstract this spontaneous total order property.

Figure 1 illustrates the spontaneous total order property in a system composed of a cluster of 12 PCs connected by a local-area network (see Section 5 for details about the environment). In the experiments, each workstation broadcasts messages to all the other workstations, and receives messages from all workstations over a certain period of time. Broadcasts are implemented with IP-multicast.

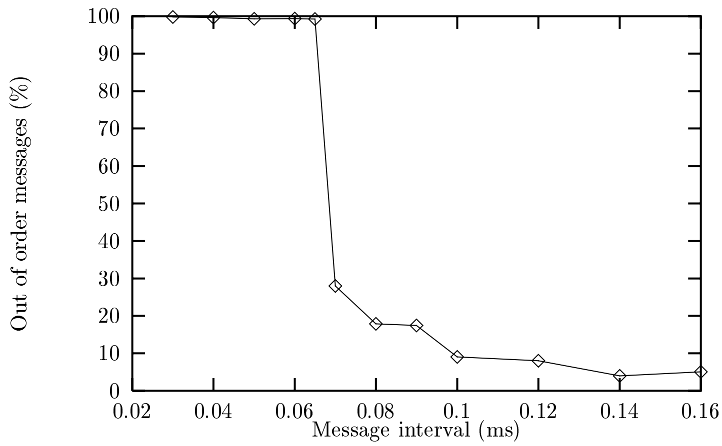

Figure 1: Spontaneous total order property

Figure 1 shows the relation between the time between successive broadcast calls and the percentage of messages that are received out of order. When messages are broadcast with a period greater than approximately 0.14 milliseconds, IP-multicast implements a WAB oracle with a very high probability (i.e., only about 5% of messages are received out of order).

3 Solving Uniform Consensus with 1-WAB Oracles

3.1 The Consensus Problem

The (uniform) consensus problem is defined over a set of n processes.[4] Each process p_i proposes an initial value v_i, and processes must eventually agree on a common value v that has been proposed by one of the processes. Formally, the problem is defined by the following three properties [9]:

- **Uniform Agreement:** No two processes decide differently.
- **Termination:** Every correct process eventually decides.
- **Uniform Validity:** If a process decides v, then v has been proposed by some process.

In this section we give two algorithms that solve consensus in an asynchronous system augmented with a 1-WAB oracle. The first algorithm, called B-Consensus algorithm, is inspired by Ben-Or's randomized consensus algorithm [7] and the second one, called R-Consensus algorithm, is inspired by Rabin's algorithm [19]. While Ben-Or's and Rabin's algorithms solve the binary consensus problem, where the initial values are 0 or 1, our algorithms solve the general (i.e., non-binary) consensus problem. We present Ben-Or's and Rabin's consensus algorithms in Appendix A, for readers not familiar with them (expressed in the same syntactic form as our algorithms).

3.2 The B-Consensus Algorithm

We initially provide an overview of the algorithm and then its description in detail (see Algorithm 1). Similarly to Ben-Or's algorithm, our algorithm requires $f < n/2$ (i.e., a majority of correct processes).

Overview of the algorithm. The algorithm executes in a sequence of rounds, where each round has three stages (see Figure 2 — for clarity, messages from a process to itself have been omitted). In the first stage of the round, processes query the 1-WAB oracle, which propagates

[4]From here on, "consensus" implicitly means "uniform consensus."

their estimates to the other processes and wait for the first message output by the oracle in the current round. The second and third stages are used to determine whether a majority of processes output the same estimate in the first stage.

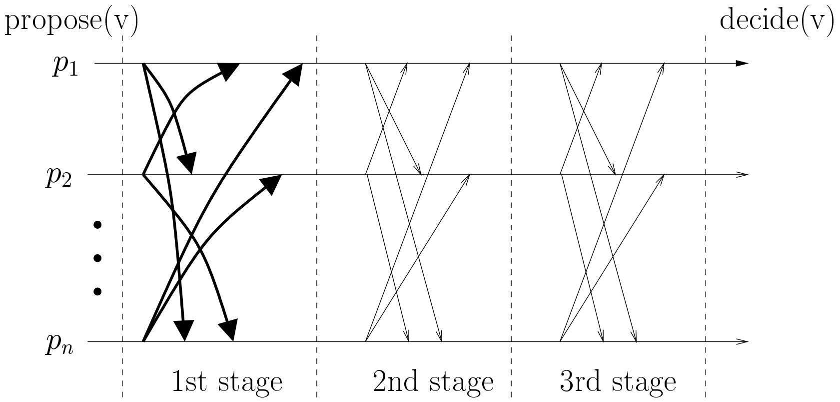

Figure 2: One round of the B-Consensus algorithm

In the second stage, a process sends its current estimate (updated in the first stage) to the other processes and waits for the first $n-f$ messages of the same kind. If the $n-f$ messages received contain the same estimate value v, the process takes v as its estimate; otherwise it takes a void value as its estimate. Notice that the majority constraint guarantees that the only possible outcomes of the second stage for all processes is either v or void.

In the third stage, each process sends its estimate to the other processes and again waits for $n-f$ responses. If the same non-void value is received from $f+1$ processes, the process decides if it has not yet decided in a previous round, and proceeds to the next round. The algorithm, as it is, requires processes to keep executing even after they have already decided on some value. We address this issue in Section 4.

B-Consensus in detail. Algorithm 1 (page 8) is the B-Consensus algorithm. In each round (lines 6–24), every process p first queries the oracle (line 6), waits for the first answer tagged with the current round number r_p (line 7) and updates its $estimate_p$ value (line 8). Then p sends $estimate_p$ to all in a message of type FIRST (line 9) and waits for $n-f$ such messages (line 10). After updating $estimate_p$, process p sends again $estimate_p$ to all in a message of type SECOND (line 15) and waits for $n-f$ such messages. If $f+1$ messages received contain a value v different from $\perp$ then p decides v (line 18). Even after deciding, p continues the algorithm.

Compared to Ben-Or's algorithm (Appendix A, Algorithm 5), lines 6–8 are new, and the coin toss (line 20 of Ben-Or's algorithm) has been replaced by an assignment of the initial value to

$estimate_p$ (line 20). Notice that while Ben-Or's algorithm solves the binary consensus problem, Algorithm 1 solves the generalized consensus problem with non-binary initial values.

It is easy to see that the validity property holds. The proof of uniform agreement is very similar to the proof of Ben-Or's algorithm, and is given, together with the proof of termination, in Appendix B.1.

3.3 The R-Consensus Algorithm

We now present the R-Consensus algorithm, inspired by Rabin's algorithm. Similarly to Rabin's algorithm, it requires $f < n/3$. As before, we first provide an overview of the algorithm and then present it in more detail.

Algorithm 1 B-Consensus algorithm ($f < n/2$)

1: To execute propose($initVal$):

2:   $estimate_p \leftarrow initVal$
3:   $decided \leftarrow false$
4:   $r_p \leftarrow 0$

5: **while** $true$ **do**

6:   W-ABroadcast$(r_p, estimate_p)$
7:   **wait until** W-ADeliver of the first message (r_p, v)
8:   $estimate_p \leftarrow v$

9:   send $(\text{FIRST}, r_p, estimate_p)$ to all
10:   **wait until** received (FIRST, r_p, v) from $n - f$ processes
11:   **if** $\exists\, v$ s.t. received (FIRST, r_p, v) from $n - f$ processes **then**
12:     $estimate_p \leftarrow v$
13:   **else**
14:     $estimate_p \leftarrow \perp$

15:   send $(\text{SECOND}, r_p, estimate_p)$ to all
16:   **wait until** received (SECOND, r_p, v) from $n - f$ processes
17:   **if not** $decided_p$ **and** ($\exists\, \overline{v} \neq \perp$ s.t. received $(second,\ r_p, \overline{v})$ from $f + 1$ processes) **then**
18:     decide $\overline{v}$ {*continue the algorithm after the decision*}
19:     $decided_p \leftarrow true$
20:   **if** $\exists\, \overline{v} \neq \perp$ s.t. received $(\text{SECOND}, r_p, \overline{v})$ **then**
21:     $estimate_p \leftarrow \overline{v}$
22:   **else**
23:     $estimate_p \leftarrow initVal$

24:   $r_p \leftarrow r_p + 1$

Overview of the algorithm. The R-Consensus algorithm also solves consensus with a 1-WAB oracle. The algorithm executes in a sequence of rounds divided in two stages (instead of three stages in the B-consensus algorithm). In the first stage, processes use the 1-WAB oracle to propagate their estimates to the other processes, and wait for the first message output by the oracle in the current round. In the second stage, processes send the estimates they received in the first stage and wait for two thirds of replies. If all values received by the process are the same, the process can decide in the round. If a majority of the values received are the same, the process adopts this value as its current estimate.

R-Consensus in detail. Algorithm 2 (page 10) is the R-Consensus algorithm. In each round (lines 6–16), just like the B-Consensus algorithm, every process first queries the oracle (line 6), waits for the first answer tagged with the current round number r_p (line 7) and updates its $estimate_p$ value (line 8). Then p sends $estimate_p$ to all in a message of type FIRST (line 9) and waits for $n - f$ such messages (line 10). If a majority of the values received are identical, p updates $estimate_p$. If $n - f$ values received are equal to $\overline{v}$, then p decides $\overline{v}$ (line 14). After deciding, p continues the algorithm. Stopping is discussed in the context of atomic broadcast (Section 4).

Compared to Rabin's algorithm (Appendix A, Algorithm 6), the lines 5–7 are new, and the coin toss (line 16 of Rabin's algorithm) has been removed. Moreover, lines 13–18 in Rabin's algorithm are no longer needed: this is because of lines 6–8 which play conceptually the role of lines 13–18 in Rabin's algorithm: ensuring that if one process decides $\overline{v}$, the other processes cannot decide differently. Notice also that, while Rabin's algorithm solves the binary consensus problem, Algorithm 2 solves the generalized consensus problem with non-binary initial values.

It is easy to see that the validity property holds. The proof of uniform agreement is very similar to the proof of Rabin's algorithm, and is given, together with the proof of termination, in Appendix B.2.

3.4 Time Complexity *vs.* Resilience

We compare now the time complexity of the B-Consensus and the R-Consensus algorithms in "good runs." In CDB algorithms, a good run is usually defined as a run in which no process fails and no process is falsely suspected by other processes. Here we define a good run as a run in which, for all processes p and q that do not crash, we have $first_p(1) = first_q(1)$. So, contrary to the definition of good runs in the context of CDB algorithms, a good run can include process

Algorithm 2 R-Consensus algorithm ($f < n/3$)

To execute propose($initVal$):

  $estimate_p \leftarrow initVal$
  $decided \leftarrow false$
  $r_p \leftarrow 0$

while $true$ **do**

  W-ABroadcast($r_p, estimate_p$)
  wait until W-ADeliver of the first message (r_p, v)
  $estimate_p \leftarrow v$

  send ($\text{FIRST}, r_p, estimate_p$) to all
  wait until received (FIRST, r_p, v) from $n - f$ processes
  if a majority of values received are equal to $\overline{v}$ **then**
    $estimate_p \leftarrow \overline{v}$

  if not $decided_p$ **and** (all values received are equal to $\overline{v}$) **then**
    decide $\overline{v}$ {*continue the algorithm after the decision*}
    $decided_p \leftarrow true$

  $r_p \leftarrow r_p + 1$

crashes.

We measure the time complexity in terms of the maximum message delay δ [4]. We assume a cost of δ for our oracle. In good runs, with Algorithm 1, every process decides after 3δ. Remember that the algorithm assumes $f < n/2$. In good runs, with Algorithm 2, every process decides after 2δ. The algorithm assumes $f < n/3$. This shows an interesting trade-off between time complexity and resilience: 3δ and $f < n/2$ *vs.* 2δ and $f < n/3$.

These time complexities are similar to the results of consensus algorithms based on failure detectors. For example, the consensus algorithms in [20, 17], based on $\diamond\mathcal{S}$, have a time complexity of 2δ and assume $f < n/2$; however, the results for B-Consensus and R-Consensus can be achieved in "less favorable" circumstances, that is, in the presence of process crashes.

4 Solving Atomic Broadcast with WAB Oracles

4.1 The Atomic Broadcast Problem

Atomic broadcast is defined by the primitives A-Broadcast and A-Deliver and the following properties:

- **Validity:** If a correct process A-broadcasts message m, then eventually it A-delivers m.
- **Uniform Agreement:** If a process A-delivers m, then all correct processes eventually A-deliver m.
- **Uniform Integrity:** Every message is A-delivered at most once at each process, and only if it was previously A-broadcast.
- **Uniform Total Order:** If two processes p and q both A-deliver messages m and m', then p A-delivers m before m' if and only if q A-delivers m before m'.

Solving atomic broadcast by reduction to a sequence of consensus is well known [9]. We consider here a different solution that closely integrates the ordering oracle with the atomic broadcast algorithm.[5] We consider hereafter an atomic broadcast algorithm based on Algorithm 2 (considering Algorithm 1 instead leads to a similar solution). Our algorithm assumes a WAB oracle, which satisfies the ordering property $\mathit{first}_p(r) = \mathit{first}_q(r)$ for an infinite number of rounds r.

Note that [5], similarly to the algorithm hereafter, describes an Atomic Broadcast algorithm based on prefix agreement. However, the structure of our algorithm is completely different (e.g., [5] is based on a variant of consensus).

4.2 Sequences of Messages

We express the atomic broadcast algorithm using message *sequences*. In addition to the traditional set operators, we use the *concatenation* operator $\oplus$ and the *prefix* operator $\otimes$ to handle sequences.

- **Concatenation** $s_1 \oplus s_2$ **:** The sequence $s \stackrel{\text{def}}{=} s_1 \oplus s_2$ is defined as s_1 followed by $s_2 \setminus s_1$, that is, all the messages in s_1 followed by all the messages in s_2 that are not in s_1 (in the same order as they appear in s_2). For example, let $s_1 = \langle m_0; m_1; m_2; m_3; \rangle$, and $s_2 = \langle m_0; m_1; m_4 \rangle$. We have $s_1 \oplus s_2 = \langle m_0; m_1; m_2; m_3; m_4 \rangle$, and $s_2 \oplus s_1 = \langle m_0; m_1; m_4; m_2; m_3 \rangle$.

[5] When reducing atomic broadcast to consensus, see [9], we get a solution in which the ordering oracle, used in the consensus algorithm, is decoupled from the atomic broadcast algorithm.

- **Prefix** $s_1 \otimes s_2$ **:** The sequence $s \stackrel{\text{def}}{=} s_1 \otimes s_2$ is defined as the longest common prefix of s_1 and s_2. The $\otimes$ operator is commutative and associative. For example, taking s_1 and s_2 as defined above, $s_1 \otimes s_2 = s_2 \otimes s_1 = \langle m_0; m_1 \rangle$. We say that a sequence s is a prefix of another sequence s', denoted $s \leq s'$, iff $s = s \otimes s'$. Notice that the empty sequence ϵ is a prefix of every sequence.

4.3 From WAB Oracles to Atomic Broadcast (Version 1)

In this section we give a simple version of our atomic broadcast algorithm; in the next section we extend it to include some optimizations.

Overview of the algorithm. The structure of our atomic broadcast algorithm is close to the structure of the R-Consensus algorithm (Section 3.3) and also assumes $f < n/3$. The main difference is that the atomic broadcast algorithm uses sequences of messages instead of single messages. The execution proceeds in rounds; to broadcast a message, a process concatenates it with a sequence that it keeps locally, denoted *estimate*. Processes constantly send their *estimate* sequences to other processes in the first stage of a round using the WAB oracle and wait for the first sequence output by the oracle in the current round. In the second stage, processes exchange the estimate sequences output by the oracle in the first stage (possibly with some other messages appended). Each process waits for $n - f$ messages. If all sequences received have a common non-empty prefix, the process can A-deliver all such messages if it has not A-delivered them yet (in previous rounds). Then, the process determines the longest prefix among a majority of the sequences received; this prefix, followed by any other messages the process may have received, will be the process' new estimate. The process then starts the next round.

The algorithm in detail. Algorithm 3, page 13, is the first version of our atomic broadcast algorithm. Tasks 1, 2 and 3 execute concurrently. Variable r_p (line 2) is the current round number, $estimate_p$ (line 3) contains a sequence of messages broadcast by p or by any other process, and $delivered_p$ (line 4) contains the sequence of messages delivered by p, in the order in which they were delivered.

To broadcast a message m, process p appends m to $estimate_p$ (line 6, Task 1). The main algorithm and actual broadcasting of messages is performed by Task 2 (lines 8–20). Task 3 (lines 21–22) is related to the validity property of atomic broadcast. The variable $estimate_p$ is concurrently accessed by Task 1, Task 2, and Task 3; we implicitly assume that it is accessed in mutual exclusion (e.g., using semaphores).

Algorithm 3 Atomic Broadcast with the WAB oracle ($f < n/3$)—version 1

Initialization

  $r_p \leftarrow 1$
  $estimate_p \leftarrow \epsilon$
  $delivered_p \leftarrow \epsilon$

To execute A-broadcast(m): {*Task 1*}

  $estimate_p \leftarrow estimate_p \oplus \langle m \rangle$

A-deliver($-$) *occurs as follows*: {*Task 2*}

  while *true* **do**
    W-ABroadcast($r_p, estimate_p$)
    wait until W-ADeliver of the first message (r_p, v)
    $estimate_p \leftarrow v \oplus estimate_p$

    send $(\text{FIRST}, r_p, estimate_p)$ to all
    wait until received (FIRST, r_p, v) from $n - f$ processes
    $majSeq \leftarrow$ the longest sequence $\otimes_{\{\text{majority of } (\text{FIRST}, r_p, v) \text{ received}\}}\, v$
    $estimate_p \leftarrow majSeq \oplus estimate_p$

    $allSeq \leftarrow \otimes_{\{\text{all } (\text{FIRST}, r_p, v) \text{ received}\}}\, v$
    for each $m \in (allSeq \setminus delivered_p)$ **do**
      A-deliver m
    $delivered_p \leftarrow allSeq$

    $r_p \leftarrow r_p + 1$

  when W-ADeliver($-, v$) of the second and next messages of any round {*Task 3*}
    $estimate_p \leftarrow estimate_p \oplus v$

The proof that the algorithm correctly implements atomic broadcast is given in the Appendix C. Correctness follows from some invariants about rounds. Let p and q be two processes:

- If p executes round r until the end and q is correct, then q executes round r until the end.
- If p and q execute round r until the end, either $delivered_p^r$ is a prefix of $delivered_q^r$ or $delivered_q^r$ is a prefix of $delivered_p^r$.
- If p executes round r until the end, and q executes round $r+1$ until the end, then $delivered_p^r$ is a prefix of $delivered_q^{r+1}$.

Example. Figure 3 shows an execution of our atomic broadcast algorithm. Processes p_1 and p_3 broadcast messages m and m', respectively, by appending them to their *estimate* sequence. All processes propagate their sequences in the first stage and p_3 crashes at the beginning of the second stage; p_1 and p_2 receive, however, sequence $\langle m' \rangle$ from p_3. Since p_1's sequence started with m, at the beginning of the second stage it becomes $\langle m'; m \rangle$ (notice that processes include the messages output by the oracle *at the first position* in their *estimate* sequences). Process p_4's oracle outputs first the sequence sent by p_1. In the second stage p_1, p_2, and p_4 exchange the sequences received and since two such sequences (the ones from p_1 and p_2) have m' as a common non-empty prefix, processes deliver message m'. In the next round (not shown in the figure), p_1 will use its oracle to propagate m again.

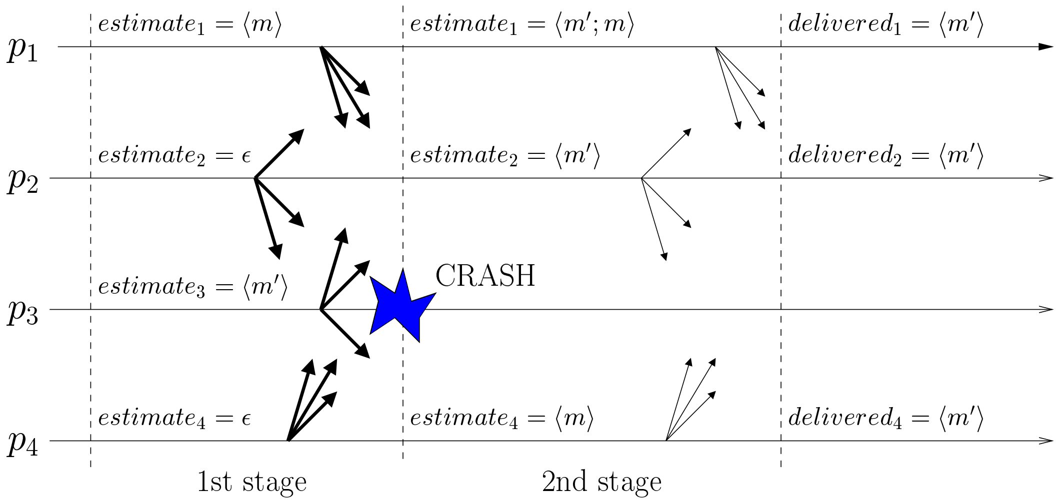

Figure 3: Execution of the atomic broadcast algorithm

4.4 From WAB Oracles to Atomic Broadcast (Version 2)

Algorithm 3 has two shortcomings. First, the *estimate* sequence used by processes to store broadcast messages keeps growing throughout the execution—that is, messages are never garbage collected. Second, processes never stop executing the *while loop* (lines 8–20) and, consequently, are exchanging messages, even after all broadcast messages have been delivered. To save resources, if messages are not broadcast for long periods of time, processes should stop executing the *while loop* after all previously broadcast messages have been delivered.

These problems can be solved with small modifications to Algorithm 3. Algorithm 4 is similar to Algorithm 3, but for the underlined lines (14, 16, 19, 21, 23, and 24). To garbage collect messages from *estimate*, Algorithm 4 takes advantage of the following property of Algorithm 3: if the first process to deliver m does so at round r, then every process that executes round $r+1$ until the end also delivers m. Therefore, at the end of round r, the information about the messages delivered in rounds $r' \leq r-1$ can be discarded.

Algorithm 4 Atomic Broadcast with the WAB oracle ($f < n/3$)—version 2

1: *Initialization*

2: $r_p \leftarrow 1$
3: $estimate_p \leftarrow \epsilon$
4: $delivered_p \leftarrow \epsilon$

5: *To execute* A-broadcast(m): {*Task 1*}

6: $estimate_p \leftarrow estimate_p \oplus \langle\, m \,\rangle$

7: A-deliver($-$) *occurs as follows*: {*Task 2*}

8: **while** *true* **do**
9: W-ABroadcast($r_p, estimate_p$)
10: **wait until** W-ADeliver of the first message (r_p, v)
11: $estimate_p \leftarrow v \oplus estimate_p$

12: send $(\text{FIRST}, r_p, estimate_p)$ to all
13: **wait until** received (FIRST, r_p, v) from $n - f$ processes
14: $majSeq \leftarrow$ the longest sequence $\otimes_{\{\text{majority of } (\text{FIRST}, r_p, v) \text{ received}\}}\ delivered_p \oplus v$
15: $estimate_p \leftarrow majSeq \oplus estimate_p$

16: $allSeq \leftarrow \otimes_{\{\text{all } (\text{FIRST}, r_p, v) \text{ received}\}}\ delivered_p \oplus v$
17: **for each** $m \in (allSeq \setminus delivered_p)$ **do**
18: A-deliver m
19: $m.round \leftarrow r_p$
20: $delivered_p \leftarrow allSeq$
21: $estimate_p \leftarrow estimate_p \setminus \{m \mid m \in delivered_p \textbf{ and } m.round < r_p\}$

22: $r_p \leftarrow r_p + 1$

23: **if** $estimate_p = \epsilon$ **then**
24: **wait until** W-ADeliver of the first message (r_p, v) **or** $estimate_p \neq \epsilon$

25: **when** W-ADeliver($-, v$) of the second and next messages of any round {*Task 3*}
26: $estimate_p \leftarrow estimate_p \oplus v$

To address the second shortcoming of Algorithm 3 described above, whenever *estimate* is empty at the end of some round r at process p, p stops executing the *while* loop (line 8) and waits until either (a) p W-ADelivers some message for round $r+1$, or (b) some message is included in $estimate_p$ — which may happen if p itself broadcasts a message (line 6) or p W-ADelivers the second or next message for any round at line 25. Notice that if p exits the *wait* statement at line 24 because it W-ADelivered the first message of some round r_p, p will W-ABroadcast message $(r_p, estimate_p)$ (line 9) and then since p has already W-ADelivered the first message of round r_p, p will not be blocked at line 10.

4.5 Time Complexity *vs.* Resilience

If we define time complexity as in Section 3.4, we get the following result. In good runs, our atomic broadcast algorithms deliver messages within 2δ and require $f < n/3$. This result is for an atomic broadcast algorithm inspired by Rabin's algorithm. Similarly, we could have derived an atomic broadcast algorithm from Ben-Or's algorithm, which would have led to a time complexity of 3δ for the delivery of messages and $f < n/2$. So we have the same "time complexity *vs.* resilience" trade-off as for consensus, see Section 3.4.

5 Performance Evaluation

5.1 The Experiments

In order to evaluate our approach, we implemented the atomic broadcast algorithm with the WAB oracle, version 2 (see Section 4.4) and compared its performance to the performance of a crash detection based (CDB) algorithm. We chose the atomic broadcast algorithm proposed by Chandra and Toueg [9], along with the consensus algorithm in the same paper. In the rest of this section, we refer to these algorithms as WABCast and CT ABCast.

We chose to compare WABCast to CT ABCast because (a) both algorithms are proved correct in the asynchronous communication model augmented with some additional assumptions: the existence of a WAB oracle (for WABCast) and a $\diamond\mathcal{S}$ failure detector (for CT ABCast); and (b) in both algorithms, each process proceeds in a sequence of asynchronous rounds, i.e., not all processes necessarily execute the same round at a given time. The algorithms differ with respect to the number of crashes they tolerate: WABCast tolerates $f < n/3$ crashes and CT ABCast $f < n/2$ crashes. In the experiments, we compared the two algorithms with the minimal number n of processes that could tolerate one crash, i.e., WABCast with $n = 4$ was compared to CT ABCast with $n = 3$ (we have also evaluated CT ABCast with $n = 4$).

Processes communicate through message passing implemented with TCP/IP connections. The WAB oracle is implemented as follows. W-ABroadcast(r, m) results in a UDP/IP multicast of (r, m) to all participants of the algorithm, and the receipt of (r, m) corresponds to W-ADeliver(r, m). In a local area network, several UDP/IP multicast datagrams are very much likely to arrive in the same order. Notice that WABCast only uses the first W-ADeliver event of a given round r, and works even if the other W-ADeliver events of round r are lost.[6]

Let m be an arbitrary message, typically around 100 bytes. We define the *latency* of an atomic broadcast algorithm as the time between the A-Broadcast(m) event and the first A-Deliver(m) event (these events do not necessarily occur on the same process). In each of our test runs, messages are A-Broadcast by all n processes. The A-Broadcast events follow a Poisson arrival distribution with the same fixed rate on each process. We call the overall rate of A-Broadcast events "throughput". Throughput, given in s^{-1}, is also the average number of messages A-Delivered in a time unit. We ran a lot of test runs with different throughput values, and determined the average latency in each test run. Our results are plots representing the average latency as a function of throughput.

The experiments were run on a cluster of 12 PCs running Red Hat Linux 7.0 (kernel 2.2.19). The hosts have Intel Pentium III 766 MHz processors with 128 MB of RAM. They are interconnected by a simplex 100 Base-TX Ethernet hub. The algorithms were implemented in Java (Sun's JDK 1.4.0 beta 2) on top of the Neko development framework [21]. In our environment, we could synchronize the clocks of processes up to a precision of $50\,\mu$s. This enabled us to determine the latency of the algorithms ($\gg 50\,\mu$s) rather precisely.

5.2 Results

Figure 4 depicts the results obtained in our evaluation. For high throughput, WABCast has a latency of a few milliseconds (2-3) higher than the latency of CT ABCast for the same number of processes. However, note that CT ABCast has been evaluated under "optimal" conditions, that is, in cases where failure detectors never make mistakes. Furthermore, during the executions of CT ABCast, processes do not exchange *I am alive* messages, necessary to implement failure detection. The big advantage of WABCast over CT ABCast is that the latency of the algorithm does not increase in case of a crash — which is not the case with CT ABCast! To achieve similar performances in the case of a crash, CT ABCast would require an extremely aggressive

[6] The algorithm does not need messages to be reliably transmitted (Validity property of the WAB oracle; see Section 2) because line 12 of the algorithm ensures this.

failure detection mechanism (*I am alive* messages sent approx. every 2-4 milliseconds). Such a frequency would significantly slow down the CT ABCAST algorithm in the absence of failures, because of (1) the CPU and network load of the failure detection messages, and (2) the probably frequent false failure detections (which increase the cost of the consensus algorithm that is part of CT ABCAST). The higher latency of WABCast is probably related to the message complexity: ($O(n)$ for CT ABCast *vs.* $O(n^2)$ for WABCast), which seems to prevail over time complexity (4δ *vs.* 2δ).

We believe that the performances of the WABCast algorithm may further be improved, e.g., by using UDP/IP multicast for the *send to all* of line 12 in Algorithm 4. We hope being able, with the WABCast algorithm, to achieve performances at least as good as those of the CT ABCast algorithm.

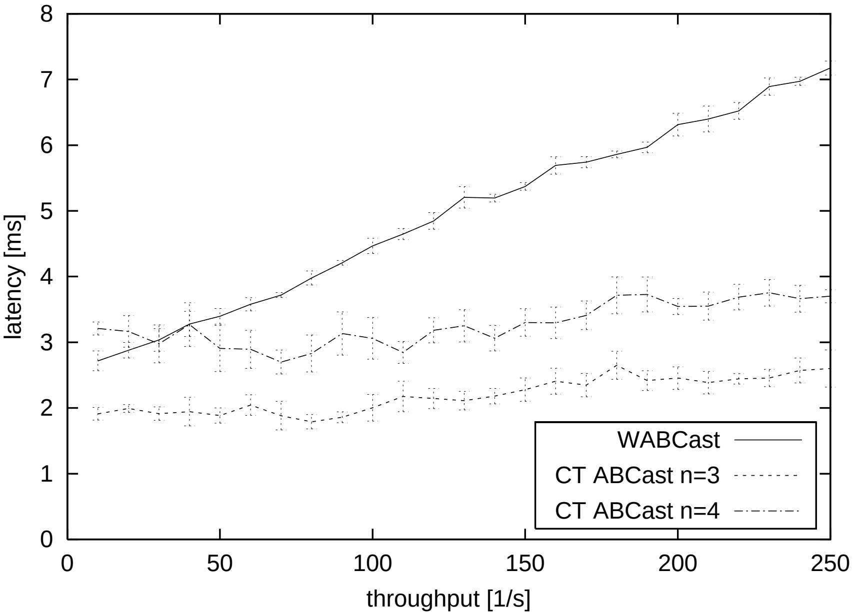

Figure 4: WABCast vs. CT ABCast

6 Conclusion

This paper addressed the issue of solving agreement problems using weak-ordering oracles. Weak-ordering oracles have a theoretical as well as a practical interest. From a theoretical viewpoint, weak-ordering oracles help extending the class of non-CDB algorithms beyond the well-known randomized algorithms. Furthermore, weak-ordering oracles are an alternative to circumvent the FLP-impossibility result, which states that consensus cannot be solved in asynchronous systems. Previous solutions to this problem have been based on strengthening the synchrony assumptions about the system [16], randomization [7, 19], and failure detection [9].

From a practical viewpoint, algorithms based on weak-ordering oracles do not have to deal with the tradeoffs involved in tuning timeouts. This is a quite powerful characteristic of algorithms based on weak-ordering oracles. To decide on timeout values, one is faced with the following dilemma: to have a short fail-over time, timeouts should be short; to prevent false failure suspicions, timeouts should be long. The "ideal" timeout value is somewhere between the two extremes, and the problem is not only finding it, but also constantly re-adapting to the environment changes that make this ideal value sway back and forth.

On a different issue, all our algorithms derived from Rabin's algorithm have in good runs a time complexity of 2δ and require $f < n/3$, while the corresponding algorithms derived from Ben-Or's algorithm have in good runs a time complexity of 3δ and require $f < n/2$. It would be interesting to understand this trade-off from a more general perspective. Currently, we are investigating whether weak-ordering oracles could be implemented in environments other than local-area networks, and how to efficiently solve other agreement problems (e.g., generic broadcast [18]) using weak-ordering oracles.

Acknowledgments

We thank Bernadette Charron-Bost for early discussions about randomized consensus algorithms and ordering oracles, and Matthias Wiesmann for providing us with Figure 1.

References

[1] Mostefaoui Achour and Michel Raynal. Leader-based consensus. *Parallel Processing Letters*, 11:95–107, 2001.

[2] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing*, pages 231–245, September 1998.

[3] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

[4] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'2000)*, October 2000.

[5] E. Anceaume. A Lightweight Solution to Uniform Atomic Broadcast for Asynchronous Systems. In *IEEE 27th Int Symp on Fault-Tolerant Computing (FTCS-27)*, pages 292–301, June 1997.

[6] H. Attiya and J. Welch. *Distributed Computing*. Mc Graw Hill, 1998.

[7] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *proc. 2nd annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

[8] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of ACM*, 43(4):685–722, 1996.

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

[10] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel & Distributed Systems*, 10(6):642–657, June 1999.

[11] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of ACM*, 34(1):77–97, January 1987.

[12] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, April 1988.

[13] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.

[14] R. Guerraoui and A. Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, Tunis, Tunisia, October 1997. IEEE Computer Society Press.

[15] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.

[16] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[17] A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: a synthetic approach. In *13th. Intl. Symposium on Distributed Computing (DISC'99).* Springer Verlag, LNCS 1693, September 1999.

[18] F. Pedone and A. Schiper. Generic broadcast. In *13th. Intl. Symposium on Distributed Computing (DISC'99).* Springer Verlag, LNCS 1693, September 1999.

[19] M. Rabin. Randomized Byzantine generals. In *Proc. 24th Annual ACM Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[20] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[21] Péter Urbán, Xavier Défago, and André Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Proc. of the 15th Int'l Conf. on Information Networking (ICOIN-15)*, Beppu City, Japan, February 2001.

A Appendix: Randomized Consensus Algorithms

We give here the two classical randomized consensus algorithms: Ben-Or's algorithm [7], Algorithm 5 (page 21), and Rabin's algorithm [19], Algorithm 6 (page 22),

Algorithm 5 Ben-Or binary consensus algorithm

Consensus ($initVal$):

  $estimate_p \leftarrow initVal$
  $decided \leftarrow false$
  $r_p \leftarrow 0$

while $true$ **do**

  send $(\textsc{first}, r_p, estimate_p)$ to all
  wait until received $(\textsc{first}, r_p, v)$ from $n-f$ processes
  if $\exists\ v$ s.t. received $(\textsc{first}, r_p, v)$ from $n-f$ processes **then**
    $estimate_p \leftarrow v$
  else
    $estimate_p \leftarrow \perp$

  send $(\textsc{second}, r_p, estimate_p)$ to all
  wait until received $(\textsc{second}, r_p, v)$ from $n-f$ processes
  if not $decided_p$ and ($\exists\ v \neq \perp$ s.t. received $(second,\ r_p, v)$ from $f+1$ processes) **then**
    decide v {*continue the algorithm after the decision*}
    $decided_p \leftarrow true$
  if $\exists\ v \neq \perp$ s.t. received $(\textsc{second}, r_p, v)$ **then**
    $estimate_p \leftarrow v$
  else
    $estimate_p \leftarrow coin()$ {*toss the coin*}

  $r_p \leftarrow r_p + 1$

B Appendix: Proof of Correctness – Consensus

B.1 B-Consensus algorithm

The line numbers in the proofs refer to Algorithm 1 (page 8).

Proposition B.1 (Uniform Agreement) *If $f < n/2$, no two processes decide differently.*

Proof: Consider round r, and some process p that sets $estimate_p$ to $\overline{v}$ at line 12. So, p has received $n-f$ messages $(\textsc{first}, r_p, \overline{v})$ at line 10, i.e., $n-f$ processes have sent a message $(\textsc{first}, r_p, estimate)$ with $estimate = \overline{v}$ at line 9. As $f < n/2$, no process can set $estimate$ to

Algorithm 6 Rabin binary consensus algorithm

1: Consensus ($initVal$):

2: $\quad estimate_p \leftarrow initVal$
3: $\quad decided \leftarrow false$
4: $\quad r_p \leftarrow 0$

5: **while** *true* **do**

6: $\quad$ send $(\textsc{first}, r_p, estimate_p)$ to all
7: $\quad$ **wait until** received $(\textsc{first}, r_p, v)$ from $n - f$ processes
8: $\quad$ let $\overline{v}$ be the majority of values v received
9: $\quad estimate_p \leftarrow \overline{v}$
10: $\quad$ **if not** $decided_p$ and (all values received are $\overline{v}$) **then**
11: $\qquad$ decide $\overline{v}$ $\quad$ {*continue the algorithm after the decision*}
12: $\qquad decided_p \leftarrow true$

13: $\quad$ send $(\textsc{second}, r_p, estimate_p)$ to all
14: $\quad$ **wait until** received $(\textsc{second}, r_p, v)$ from $n - f$ processes
15: $\quad$ **if** all values v received are the same **then**
16: $\qquad estimate_p \leftarrow v$
17: $\quad$ **else**
18: $\qquad estimate_p \leftarrow common\text{-}coin()$ $\quad$ {*toss the common coin*}

19: $\quad r_p \leftarrow r_p + 1$

a value different from $\overline{v}$ at line 12. So, at line 15 of each round r, there exists $\overline{v}$ such that for every process p we have $estimate_p \in \{\overline{v}, \perp\}$.

Therefore, if a process p sends at line 15 of round r the message $(\textsc{second}, r, estimate_p)$ with $estimate_p \neq \perp$, then all processes send at line 15 of round r, either $(\textsc{second}, r, \overline{v})$ or $(\textsc{second}, r, \perp)$. Let r be the smallest round in which some process p decides $\overline{v}$. If p decides at round r, all processes that also decide at round r necessarily decide on the same value. It remains to prove that the processes deciding in some round $r' > r$ decide $\overline{v}$.

If p decides at line 18 of round r, it must have received $f + 1$ messages $(\textsc{second}, r, \overline{v})$ at line 16. As $f < n/2$, each process that receives $n - f$ messages at line 16, receives at least one message $(\textsc{second}, r, \overline{v})$. So all processes set their *estimate* value to $\overline{v}$ at line 21 and start round $r + 1$ with $estimate = \overline{v}$. It is easy to see that the only possible decision in round $r' > r$ is $\overline{v}$. $\square$

Proposition B.2 (Termination) *With 1-WAB oracles every correct process eventually decides.*

Proof: We initially claim that no process waits forever at the *wait* statements (lines 7, 10 and 16). This follows from a simple induction on k. We present next only the inductive step.

From validity of 1-WAB and the fact that all correct processes start round r and query their oracles, no correct process remains blocked at line 7. Thus, $n-f$ correct processes send messages with the tag FIRST at line 9, and none of them blocks at line 10. From a similar argument, it follows that no process blocks at line 16—concluding the proof of the claim.

By the order property of the 1-WAB oracle, there exists a round r such that for all processes p and q, we have $first_p(r) = first_q(r) \stackrel{\text{def}}{=} \overline{v}$. At round r each process sets *estimate* to $\overline{v}$, and sends $(\text{FIRST}, r, \overline{v})$ to all at line 9. Every process evaluates the condition of line 11 to *true*, sets *estimate* to $\overline{v}$ at line 12, and sends $(\text{SECOND}, r, \overline{v})$ to all at line 16. So every process receives $f+1$ values $\overline{v}$ at line 16, and decides at line 18 (if it has not done so yet). □

B.2 R-Consensus algorithm

The line numbers in the proofs refer to Algorithm 2 (page 10).

Proposition B.3 (Uniform Agreement) *If $f < n/3$, no two processes decide differently.*

PROOF: Let r be the smallest round in which some process p, decides $\overline{v}$ (at line 14). So, p has received $n-f$ messages $(\text{FIRST}, r, \overline{v})$ at line 10, i.e., $n-f$ processes have sent a message $(\text{FIRST}, r, \overline{v})$ at line 9. As $f < n/3$, no process can receive at line 10 of round r $n-f$ values v different from $\overline{v}$, i.e., no process can decide at line 14 of round r a value different from $\overline{v}$.

We prove now that no process can decide a value different from $\overline{v}$ in some round $r' > r$. As $n-f$ processes have sent a message with $estimate_p = \overline{v}$ at line 9, and because $f < n/3$, all processes that do not crash set their *estimate* to $\overline{v}$ at line 12. It follows that all processes q that start round $r+1$, do so with $estimate_q = \overline{v}$. So the only possible decision in round $r' > r$ is $\overline{v}$. □

Proposition B.4 (Termination) *With 1-WAB oracles every correct process eventually decides.*

PROOF: From an argument similar to the one presented in Proposition 3.2, no process waits forever at the *wait* statements (line 7 and 10). By the order property of the 1-WAB oracle, there exits a round r such that for all processes p and q that do not crash, we have $first_p(r) = first_q(r) \stackrel{\text{def}}{=} \overline{v}$. At line 8 of round r all process set their estimate value to $\overline{v}$, and send $\overline{v}$ to all at line 9. So, all value received at line 10 are equal to $\overline{v}$, and all processes that have not decided yet, decide at line 14 of round r. □

C Appendix: Proof of Correctness — Atomic Broadcast

We initially present the proofs for version 1 of our atomic broadcast algorithm. All the lemma statements presented for version 1 are also valid for version 2, and only some of the proofs have to be changed, so, for version 2, we keep the lemma statements and present the new proofs, when necessary.

Lemma C.1 (Lemma 4.1 of Section 4) *For all $r > 0$, every process p, and every correct process q, if p executes round r until the end, then q executes round r until the end.*

PROOF (SKETCH): This follows from a simple induction on r. We only proof the inductive step: assume that the lemma holds for $r-1$, and that p executes round $r > 1$ until the end; we show that every correct process executes round r until the end. From the inductive hypothesis, all correct processes execute round $r-1$ until the end, and so, execute W-ABroadcast$(r, -)$ in round r. From validity of the ordering oracles, all correct processes eventually execute W-ADeliver$(r, -)$. It also follows that since there are $n - f$ correct processes that execute send(FIRST$, r, -)$ at line 12, no correct process remains blocked forever at the *wait* statement at line 13, and executes round r until the end, concluding the proof. □

Lemma C.2 (Lemma 4.2 of Section 4) *For all $r > 0$, every process p that executes round r until the end, and every process q that executes round $r+1$ until the end, $delivered_p^r$ is a prefix of $delivered_q^{r+1}$.*

PROOF (SKETCH): Assume p executed round r until the end. Then, p received at line 13 $n - f$ messages of the type (FIRST$, r, v$), and from lines 16 and 19, $allSeq_p$ and $delivered_p^r$ are prefixes of v. Since there are $n - f$ processes that execute send(FIRST$, r, v$), and $f < n/3$, for every process u that executes lines 14–15, we have that $allSeq_p$ and $delivered_p^r$ are prefixes of $estimate_u^r$, where $estimate_u^r$ is the value of $estimate_u$ right after process u executes line 14–15.

Let q be a process that executes line 13 of round $r + 1$. Then q receives $n - f$ messages of the type (FIRST$, r + 1, v'$), where $v' = estimate_u^r$, and so, $allSeq_p$ and $delivered_p^r$ are prefixes of v'. Therefore, $allSeq_p$ is a prefix of $allSeq_q$ and $delivered_q^{r+1}$, and we conclude that $delivered_p^r$ is a prefix of $delivered_q^{r+1}$. □

Lemma C.3 *Let s, σ_1, and σ_2 be sequences of messages. If σ_1 and σ_2 are prefixes of s, then either (a) σ_1 is a prefix of σ_2, or (b) σ_2 is a prefix of σ_1.*

PROOF (SKETCH): Since σ_1 and σ_2 are prefixes of s, there exist sequences s_1 and s_2 such that $s = \sigma_1 \oplus s_1$ and $s = \sigma_2 \oplus s_2$. Thus, $\sigma_1 \oplus s_1 = \sigma_2 \oplus s_2$. Assume $|\sigma_1| \geq |\sigma_2|$, and $\sigma_1 = \sigma_1^1 \oplus \sigma_1^2$, such that $|\sigma_1^1| = |\sigma_2|$. Therefore, $\sigma_1^1 \oplus \sigma_1^2 \oplus s_1 = \sigma_2 \oplus s_2$, and from the definition of $\oplus$, it has to be that $\sigma_1^1 = \sigma_2$. We conclude that σ_2 is a prefix of σ_1. $\square$

Lemma C.4 (Lemma 4.3 of Section 4) *For all $r > 0$, and every process p and q that execute round r until the end, either $delivered_p^r$ is a prefix of $delivered_q^r$ or $delivered_q^r$ is a prefix of $delivered_p^r$.*

PROOF (SKETCH): The lemma is trivially true if $delivered_p^r$ or $delivered_q^r$ is the empty sequence. So, assume that $delivered_p^r \neq \epsilon$ and $delivered_q^r \neq \epsilon$. We have that $delivered_p^r = allSeq_p$ and $delivered_q^r = allSeq_q$. Since p and q received $n - f$ sequences and $f < n/3$, there is at least one process u whose sequence v_u was taken into account by both p and q to compute $allSeq_p$ and $allSeq_q$. Therefore, $allSeq_p$ and $allSeq_q$ are both prefixes of v_u, and from Lemma C.3, we conclude that either $delivered_p^r$ is a prefix of $delivered_q^r$ or $delivered_q^r$ is a prefix of $delivered_p^r$. $\square$

Proposition C.5 *(Uniform Agreement.) If a process p A-delivers m, then every correct process q eventually A-delivers m.*

PROOF (SKETCH): Assume p A-delivers m in round r. From Lemma C.1, every correct process executes round r until the end, and so, start round $r+1$. Thus, it follows that q starts round $r+1$ and executes it until the end. Since p A-delivers m in round r, by Algorithm 3, $m \in delivered_p^r$, and from Lemma C.2, $delivered_p^r$ is a prefix of $delivered_q^{r+1}$. Therefore, $m \in delivered_q^{r+1}$, and we conclude that q A-delivers m. $\square$

Proposition C.6 *(Uniform Total Order.) If two processes p and q both A-deliver the messages m and m', then p A-delivers m before m' if and only if q A-delivers m before m'.*

PROOF (SKETCH): The proof follows from Lemma C.4. $\square$

Proposition C.7 *(Uniform Integrity.) Every message is A-delivered at most once, and only if it was previously A-broadcast.*

PROOF (SKETCH): Immediate from Algorithm 3. $\square$

Proposition C.8 *(Validity.) If a correct process A-broadcasts message m, it A-delivers m.*

Proof (sketch): By contradiction. From Algorithms 3, once a process includes a message in its *estimate* sequence, the message is either never removed from it, although the message may change its rank in *estimate*. Let p be a correct process that A-broadcasts m. So, p includes m in $estimate_p$ (line 6) and W-ABroadcasts $estimate_p$ (line 9). Since m is not A-delivered—by the contradiction hypothesis, not all processes W-ADeliver $(-, estimate_p)$ as the first message, but from the validity of the ordering oracle, all correct processes W-ADeliver $(-, estimate_p)$. Thus, there is a round after which for every correct processes q, $m \in estimate_q$. Since faulty processes eventually crash, there is a round that no faulty process executes—that is, all faulty processes crash before that round. Thus, there is a round r that only correct processes execute and, for each correct process q, m is in $estimate_q$. Let $r' > r$ be a round where the k-WAB property holds. It follows that at r', m is A-delivered. □

We now prove the correctness of Algorithm 4. Algorithm 4 modifies Algorithm 3 in two ways, and each one of these modifications leads to changes in the proofs as presented next.

- In Algorithm 4, if there is a time after which messages are not broadcast, processes eventually stop exchanging messages (lines 23–24). As for the proofs, this means that the proof presented for Lemma C.1 no longer holds. We restate Lemma C.1 next as Lemma C.9 and prove it correct.

- Processes executing Algorithm 4 do not always have to send all the messages they already have delivered. Therefore, processes reduce their *estimate* sequences by removing messages they know the other processes have already delivered (lines 19, and 21). But to keep Algorithm 4 as similar as possible to Algorithm 3, sequences are "rebuilt" when they are received by a process (lines 14 and 16). We prove in Lemma C.10 that all the messages removed by a process p before sending its *estimate* are reintroduced back by any other process q that receives it. Therefore, we indirectly show that proofs for Lemmas C.2 and C.4 are still valid.

Proofs for Lemmas C.3, C.5, C.6, C.7, and C.8 also hold for Algorithm 4.

Lemma C.9 *For all $r > 0$, every process p, and every correct process q, if p executes round r until the end, then q executes round r until the end.*

Proof (sketch): As for Lemma C.1, the proof is by induction on r. Assume the lemma holds for $r - 1$, and that p executes round $r > 1$ until the end. Thus, p received $n - f$ messages of the type $(\textsc{first}, r, -)$ at line 13, and W-Adelivered message (r, v) at line 10. Since $f < n/3$, there is at least one correct process u that execute send$(\textsc{first}, r, -)$ at line 12, and before doing that, u executed W-ABroadcast$(-, estimate)$. From the inductive hypothesis, all correct processes terminate round $r-1$, and so, they are either blocked at the *wait* statement at line 10 or 24.

In the former case, the proof continues with an argument similar to the one presented in the proof of Lemma C.1. In the latter case, from the validity of WAB oracles, all correct processes will eventually execute W-ADeliver(r, v_u) and send message $(\textsc{first}, r, -)$ to all processes. It follows that all correct processes terminate round r. □

For the following proof, we consider that $estimate_p^r$ is the value of sequence *estimate* at process p right after p executes line 15 of round $(r-1)$—that is, $estimate_p^r$ will be the sequence sent by p in round r, if p executes round r.

Lemma C.10 *For all $r > 0$, and every process p that receives a message with $estimate_q^r$ from process q in round r, we have $delivered_p^{r-1} \oplus estimate_q^r = delivered_q^{r-1} \oplus estimate_q^r$.*

Proof (sketch): When process q sets the value of $estimate_q^r$ in round $r-1$, it follows that for every process u, $delivered_u^{r-1}$ is a prefix of $estimate_q^r$, and so, $delivered_p^{r-1}$ and $delivered_q^{r-1}$ are prefixes of $estimate_q^r$. Thus, from the definition of $\oplus$, $delivered_p^{r-1} \oplus estimate_q^r = estimate_q^r$ and $delivered_q^{r-1} \oplus estimate_q^r = estimate_q^r$, and we conclude that $delivered_p^{r-1} \oplus estimate_q^r = delivered_q^{r-1} \oplus estimate_q^r$. □