Solving Agreement Problems with Weak Ordering Oracles[⋆]

Fernando Pedone[1,2], André Schiper[2], Péter Urbán[2], and David Cavin[2]

[1] Hewlett-Packard Laboratories
Software Technology Laboratory
Palo Alto, CA 94304, USA
`fernando_pedone@hp.com`

[2] Ecole Polytechnique Fédérale de Lausanne (EPFL)
Faculté Informatique & Communications
CH-1015 Lausanne, Switzerland
`{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. In this paper, we revisit the issue of non-CDB algorithms considering *ordering oracles*. Ordering oracles have a theoretical interest as well as a practical interest. 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.

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 [8, 12, 13, 4], partially synchronous models [9], asynchronous models with failure detectors [6, 1], timed asynchronous models [7], 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**

[⋆] Research partially supported by the CSEM – Swiss Center for Electronics and Microtechnology Inc., Neuchâtel, and the Swiss National Science Foundation NCCR MICS project.

..." There is a notable exception to the near universality of CDB algorithms: randomized consensus algorithms [5, 18], 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 [11]. 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 lead to false failure suspicions. False failure suspicions are problematic because they lead to actions (e.g., selecting 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 the need for failure detection from the algorithms, we eliminate this problem of tuning: non-CDB algorithms operate in the presence of failures just as quickly 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 [10]. To make agreement problems solvable, we extend the asynchronous system with *ordering oracles* (Section 2), which (1) receive queries to broadcast messages and (2) output these 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 the ordering property 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 [5] 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 [18] and requires $f < n/3$.[3] 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 $f < n/2$, where δ is the maximum message delay, while the consensus algorithm inspired by Rabin's algorithm has a time complexity of 2δ and $f < n/3$.

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

Our consensus algorithms can be compared to the leader-based consensus algorithms presented in [15]. Although partly similar in structure to ours[4], the consensus algorithms we propose in this paper have a better time complexity. This is because the approach in [15] 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 [6]. 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.

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, 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 the oracle had to broadcast). More formally, an oracle is a set of oracle histories that satisfy properties relating queries to outputs [2].[5] We introduce the *Weak Atomic Broadcast* oracle,

[4] Even though this is not mentioned in [15], similarly to ours, the algorithms in [15] follow the structure of the randomized algorithms proposed by Ben-Or [5] and Rabin [18].

[5] In [2] 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 [2] include failure detectors. We do not consider failure detectors here as our approach does not need them.

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 $\mathit{first}_p(r)$. A canonical sequence of queries, 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, p_3, executing the following queries to the oracle:

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

Assume the following prefix of sequences output by oracle at 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, q and for all i, $1 \leq i \leq k$, we have $\mathit{first}_p(r_i) = \mathit{first}_q(r_i)$.

If the oracle satisfies the eventual uniform k-order property, we 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*.
- **Weak Atomic Broadcast (WAB) Oracle:** A k-WAB oracle, where $k = \infty$.

To summarize, 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 (loop-back mode disabled).

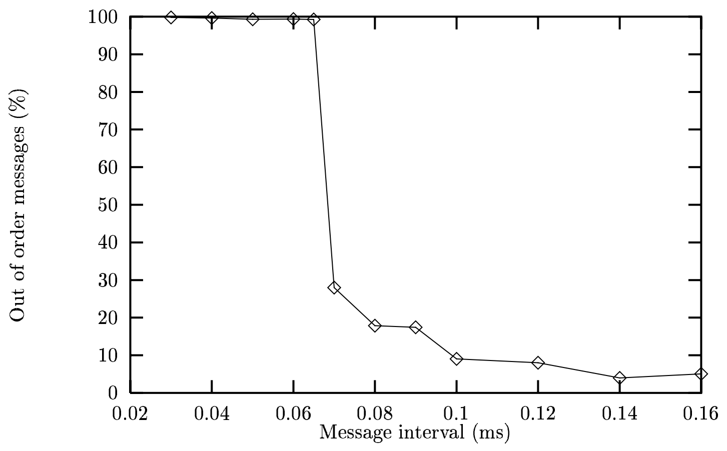


Fig. 1. Spontaneous total order property

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.[6] 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 [6]:

- **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 [5] and the second one, called R-Consensus algorithm, is inspired by Rabin's algorithm [18]. 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 [17], 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, where for clarity messages from a process to itself have been omitted). In the first stage of the round processes ask the 1-WAB oracle to broadcast their estimates to the other processes, and then wait for the first message of the current round output by the oracle.

The second stage is used to determine whether a majority of processes output the same estimate in the first stage. A process first 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

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

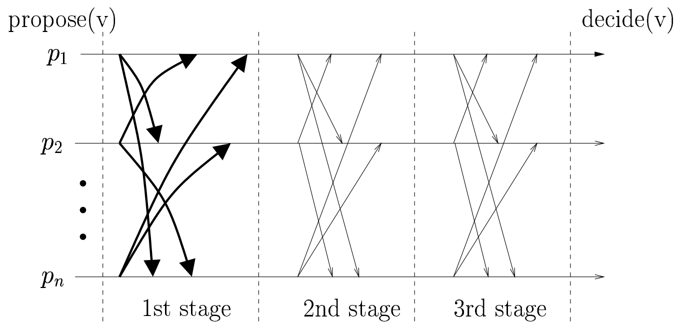


Fig. 2. One round of the B-Consensus algorithm

estimate value v, the process takes v as its estimate; otherwise it takes $\perp$ 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 $\perp$.

In the third stage, each process sends its estimate to the other processes and again waits for $n - f$ responses. If the same non-$\perp$ 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 decided. Stopping is discussed in Section 4.

B-Consensus in detail. Algorithm 1 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). After deciding, p continues the algorithm.

Compared to Ben-Or's algorithm, the coin toss has been replaced by an assignment of the initial value to $estimate_p$ (line 23). 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 in [17], together with the proof of termination.

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.

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 a majority of the values received are the same, the process adopts this value as its current estimate. If all values received by the process are the same, the process decides.

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

```
1:  To execute propose($initVal$):

2:    $estimate_p \leftarrow initVal$
3:    $decided_p \leftarrow false$
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 ($\text{FIRST}, r_p, v$) from $n - f$ processes
11:   if $\exists$ $v$ s.t. received ($\text{FIRST}, r_p, v$) 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 ($\text{SECOND}, r_p, v$) 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:   $r_p \leftarrow r_p + 1$
```

R-Consensus in detail. Algorithm 2 (page 9) 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).

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

```
1: To execute propose($initVal$):

2:    $estimate_p \leftarrow initVal$
3:    $decided_p \leftarrow false$
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 (FIRST, $r_p, estimate_p$) to all
10:   wait until received (FIRST, $r_p, v$) from $n - f$ processes
11:   if a majority of values received are equal to $\overline{v}$ then
12:      $estimate_p \leftarrow \overline{v}$

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

16:   $r_p \leftarrow r_p + 1$
```

Notice 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 in [17], together with the proof of termination.

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 processes do not fail and are not falsely suspected by other processes. Here we define a good run as a run in which, for all correct processes p, q, 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 crashes.

We measure the time complexity in terms of the maximum message delay δ [2]. 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 [19, 14], based on $\diamond\mathcal{S}$, have a time complexity of 2δ and assume $f < n/2$; however, the time complexity 3δ for B-Consensus and 2δ for 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 [6]. We consider here a different solution that closely integrates the ordering oracle with the atomic broadcast algorithm.[7] Our atomic broadcast algorithm is based on Algorithm 2 (considering Algorithm 1 instead leads to a similar solution), and 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 the atomic braodcast algorithm in [3], similarly to the algorithm hereafter, is based on prefix agreement. However, the structure of our algorithm is completely different: [3] 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.

[7] When reducing atomic broadcast to consensus, see [6], we get a solution in which the ordering oracle, used in the consensus algorithm, is decoupled from the atomic broadcast algorithm.

- **Concatenation** $s_1 \oplus s_2$ **:** The sequence $s \overset{\text{def}}{=} s_1 \oplus s_2$ 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 $s_2 \oplus s_1 = \langle m_0; m_1; m_4; m_2; m_3 \rangle$.
- **Prefix** $s_1 \otimes s_2$ **:** The sequence $s \overset{\text{def}}{=} s_1 \otimes s_2$ 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 $s \leq s'$, if and only if $s = s \otimes s'$. 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; we extend it in Section 4.4 with 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, process p concatenates the message with a sequence that it keeps locally, denoted *estimate*. Processes send their *estimate* sequence to other processes in the first stage of a round using the WAB oracle, and wait for the first sequence, with the current round number, output by the oracle. 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 A-delivers the messages in the common prefix not yet A-delivered. 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, is the process' new estimate. The process then starts the next round.

The algorithm in detail. Algorithm 3, page 12, 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 A-delivered by p, in the order in which they were A-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) ensures 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).

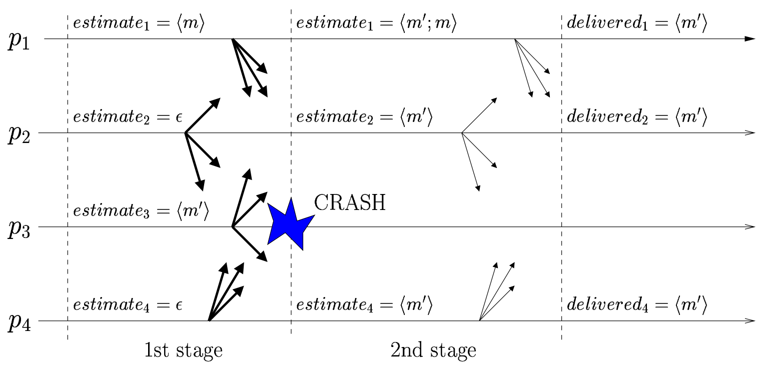
The proof of the algorithm is given in [17]. Correctness follows from the following invariants. Let p and q be two processes:

- If p terminates round r and q is correct, then q terminates round r.
- If p and q terminate round r, 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 $delivered_q^{r+1}$.

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

```
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 (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 $\otimes_{\{\text{majority of (FIRST},r_p,v) \text{ received}\}} v$
15:      $estimate_p \leftarrow majSeq \oplus estimate_p$

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

20:      $r_p \leftarrow r_p + 1$

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

Example. Figure 3 shows an execution of Algorithm 3. Processes p_1 and p_3 broadcast m and m', respectively, by appending them to their *estimate* sequence. All processes W-ABroadcast their sequences in the first stage, and p_3 crashes at the beginning of the second stage; nevertheless p_1, p_2 and p_4 W-ADeliver first the sequence $\langle m' \rangle$. The estimate of p_1 becomes $\langle m'; m \rangle$, and the estimates of p_2 and p_4 become $\langle m' \rangle$. In the second stage p_1, p_2, and p_4 exchange their estimates. Since all their sequences have m' as a common prefix, they A-deliver m'. In the next round (not shown in the figure), p_1 will W-ABroadcast m again.



p_1
$estimate_1 = \langle m \rangle$
$estimate_1 = \langle m'; m \rangle$
$delivered_1 = \langle m' \rangle$
p_2
$estimate_2 = \epsilon$
$estimate_2 = \langle m' \rangle$
$delivered_2 = \langle m' \rangle$
p_3
$estimate_3 = \langle m' \rangle$
CRASH
p_4
$estimate_4 = \epsilon$
$estimate_4 = \langle m' \rangle$
$delivered_4 = \langle m' \rangle$
1st stage
2nd stage

Fig. 3. Execution of the atomic broadcast algorithm

4.4 From WAB Oracles to Atomic Broadcast: Version 2

Algorithm 3 has two shortcomings. First, the $estimate_p$ sequence used by processes to store broadcast messages keeps growing: messages are never garbage collected. Second, processes never stop executing the *while loop* (lines 8–20) and, consequently, continue the execution even after all messages A-Broadcast have been A-delivered. To avoid wasting resources, processes should stop executing the *while loop* after all previously A-Broadcast messages have been A-delivered.

The two 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 remove messages from $estimate_p$, Algorithm 4 takes advantage of the following property of Algorithm 3: if the first process to A-deliver m does so at round r, then every process that terminates round $r+1$ also A-delivers m. So, at the end of round r, the messages A-delivered in rounds $r' \leq r-1$ can be discarded from $estimate_p$.

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

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

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

22:     $r_p \leftarrow r_p + 1$

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

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

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 and $f < n/2$. So we have the same "time complexity *vs.* resilience" trade-off as for consensus (Section 3.4).

5 Performance Evaluation

5.1 The Experiments

In order to evaluate our approach, we implemented version 2 of the atomic broadcast algorithm, and compared its performance to a Crash-Detection Based (CDB) algorithm. We chose the atomic broadcast algorithm proposed by Chandra and Toueg [6], along with the $\diamond\mathcal{S}$ consensus algorithm [6]. 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 model augmented with some additional assumptions: the 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 (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 executed the two algorithms with the minimal number of processes that could tolerate one crash, i.e., WABCast with $n = 4$ was compared to CT-ABCast with $n = 3$.

Processes communicate using 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, UDP/IP multicast datagrams are very much likely to arrive in the same order (see Section 2.3). 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 do not occur (message loss).[8]

In the experiment, messages are 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

[8] The algorithm does not need that these messages are reliably transmitted (Validity property of the WAB oracle, Section 2) because of line 12 of the algorithm.

A-Delivered in a time unit. We ran a lot of test runs with different throughput values, and determined the mean latency in each test run. Our results are plots representing the mean latency (and its 95% confidence interval) as a function of throughput.

The experiments were run on a cluster of 12 PCs running Red Hat Linux 7.2 (kernel 2.4.9). 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 [20]. 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.

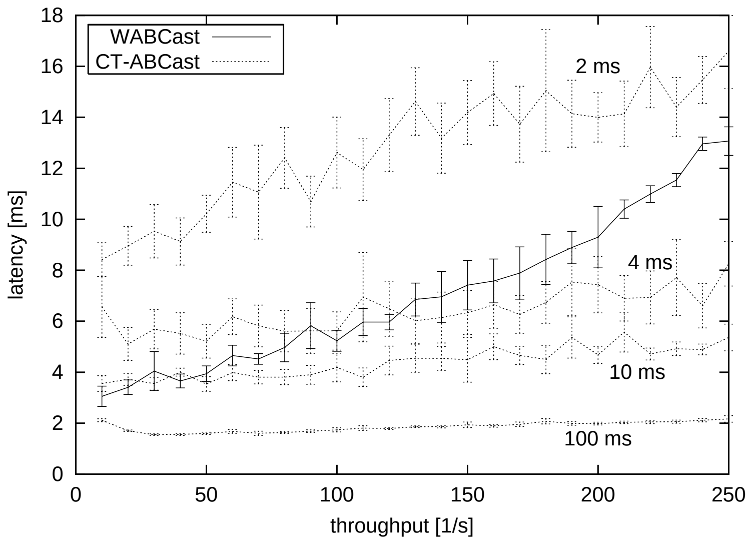


Fig. 4. WABCast vs. CT-ABCast. CT-ABCast was run with a variety of failure detection timeouts.

5.2 Results

Figure 4 depicts the results obtained. CT-ABCast was run with a variety of settings for failure detection: we used timeouts of 2, 4, 10 and 100 ms, respectively, to detect crashes. One can see that WABCast has a higher latency than CT-ABCast at high failure detection timeouts. However, note that this corresponds to "optimal" conditions for CT-ABCast: a case where failure detectors never make mistakes. Moreover, 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: with a timeout of 100 ms, it is possible that the algorithm is blocked for 100 ms after a crash. To achieve similar performances in the case of a crash, CT-ABCast requires an extremely aggressive failure detection mechanism (timeouts of 2 and 4 milliseconds and *"I am alive"* messages sent every few milliseconds). As the two latency curves on the top show, Such a failure detection mechanism significantly slows down the CT-ABCAST algorithm in the absence of failures, because (1) failure detection messages load the CPUs and the network, and (2) failure detectors often wrongly suspect correct processes, which increases the cost of the consensus algorithm that is part of CT-ABCAST. With such an aggressive failure detection mechanism, WABCast performs better (except at low throughputs when compared with CT-ABCast with a timeout of 4 ms).

The figure also shows that at high throughputs, the latency of WABCast increases faster than the latency of CT-ABCast as throughput increases. This is because the spontaneous ordering property, on which the oracle of WABCast is based, starts breaking down due to the high number of messages per time unit. The spontaneous total order property breaks down totally at around 400 requests/s, as predicted by Figure 1 (a request generates $\approx$16 messages altogether, i.e., $\approx$0.15 ms elapses between two messages).

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.

6 Conclusion

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 interesting characteristic. With CDB algorithms, in order to decide on timeout values, one is faced with the following dilemma: short fail-over time requires short timeouts; 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.

From a theoretical point of view, 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.

Finally, we are currently extending the atomic broadcast algorithm to efficiently solve generic broadcast [16] 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. 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.
2. 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.
3. 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.
4. H. Attiya and J. Welch. *Distributed Computing.* Mc Graw Hill, 1998.
5. 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.
6. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.
7. F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel & Distributed Systems*, 10(6):642–657, June 1999.
8. D.Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of ACM*, 34(1):77–97, January 1987.
9. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, April 1988.
10. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.
11. R. Guerraoui and A. Schiper. Consensus: the Big Misunderstanding. In *IEEE Proc of the Sixth Workshop on Future Trends of Distributed Computing Systems*, pages 183–188, October 1997.
12. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.
13. N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
14. 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.
15. A. Mostefaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11:95–107, 2001.
16. F. Pedone and A. Schiper. Generic Broadcast. In *13th. Intl. Symposium on Distributed Computing (DISC'99)*, pages 94–108. Springer Verlag, LNCS 1693, September 1999.
17. F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving Agreement Problems with Weak Ordering Oracles. TR IC/2002/010, EPFL, March 2002. Appears also as Technical Report HPL-2002-44, Hewlett-Packard Laboratories, March 2002.
18. M. Rabin. Randomized Byzantine Generals. In *Proc. 24th Annual ACM Symposium on Foundations of Computer Science*, pages 403–409, 1983.
19. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.
20. 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.