INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

# Optimal and Practical WAB-Based Consensus Algorithms

*Lasaro Camargos* *Fernando Pedone*
*Edmundo R. M. Madeira*

Technical Report - IC-05-07 - Relatório Técnico

April - 2005 - Abril

# Optimal and Practical WAB-Based Consensus Algorithms

Lasaro Camargos* Fernando Pedone† Edmundo R. M. Madeira‡

**Abstract**

In this paper we introduce two new WAB-based consensus algorithms for the crash-recovery model. The first one, B*-Consensus, is resilient to up to $f < n/2$ permanent faults, and can solve consensus in three communication steps. R*-Consensus, our second algorithm, is $f < n/3$ resilient, and can solve consensus in two communication steps. These algorithms are optimal with respect to the time complexity versus resilience tradeoff. We provide analytical and experimental evaluation of our algorithms, and compare them with Paxos [14], an optimal leader-election based consensus algorithm for the crash-recovery model.

## 1 Introduction

The consensus problem is recognized as a fundamental building block in fault-tolerant distributed systems. One of the most important results in the study of consensus states that no completely asynchronous distributed protocol can deterministically solve the problem if processes are subject to failures [11]. This result implies that any consensus algorithm requires extensions to the pure asynchronous model if at least one process may crash during the execution.

Motivated by this theoretical bound, several approaches have been proposed in the past years to solve consensus by strengthening asynchronous distributed systems. For example, Dolev et al. [8] and Dwork et al. [10] studied the minimum synchronization requirements needed by consensus. In [6], Chandra and Toueg introduced the concept of unreliable failure detectors. These are oracles that provide possibly incorrect information about process failures. Unreliable failure detector oracles encapsulate the synchronous assumptions needed to solve consensus and provide abstract properties to processes. The authors classified failure detectors in eight classes and showed that $\diamond\mathcal{S}$ encapsulates the minimal assumptions needed to solve consensus. Some proposals have also considered solving consensus using a leader election oracle $\Omega$ [9, 14]. Intuitively, a leader election oracle ensures that non-faulty processes should all eventually agree on the identity of some stable process, the leader. $\Omega$ oracles are equivalent to the $\diamond\mathcal{S}$ class of failure detectors [5].

Another way to circumvent the consensus impossibility result is to use randomization. The algorithms presented in [3, 19] use a random number generator to guarantee that with very high probability processes will reach a decision. Algorithms similar to those in [3, 19] were presented by Pedone et al. [18]. Instead of relying on randomization, however, progress is ensured using *weak ordering oracles*. Such oracles provide message ordering guarantees but, as unreliable failure detectors

*Institute of Computing, University of Campinas, 13081-970 Campinas SP, Brazil. Research supported in part by CNPq — Conselho Nacional de Desenvolvimento Científico e Tecnológico

†Faculty of Informatics, University of Lugano, CH-6900 Lugano, Switzerland

‡Institute of Computing, University of Campinas, 13081-970 Campinas SP, Brazil.

and $\Omega$, can make mistakes. More specifically, the algorithms in [18] use the *weak atomic broadcast (WAB)* oracle. WAB ensures that if processes keep exchanging broadcast messages in rounds, then in some rounds messages will be delivered in the same order by all nonfaulty processes—a more formal description of these oracles is given in Section 2. Weak ordering oracles are motivated by Ethernet broadcast, present in most clustered architectures.

Lower bounds on what consensus algorithms can achieve have been also considered. Lamport summarizes previous results (e.g., [7, 13]) and presents new ones in [15]. These bounds show a tradeoff between resilience and time complexity (i.e., the number of communication steps needed to solve consensus). Roughly, the results state that: (a) To ensure progress, at least a majority of processes needs to be nonfaulty. (b) To allow a decision to be made in two communication steps when different proposers propose, more than two-thirds of the processes should be nonfaulty.

Despite the great interest that consensus has attracted and the multitude of algorithms that have been proposed to solve it, most work has considered system models which are of more theoretical than practical interest. This is mainly reflected in two aspects: the failure behavior of processes and the reliability of communication links. From a practical perspective, processes should be capable of re-integrating the system after a crash. Moreover, algorithms capable of tolerating message losses can make better use of highly-efficient communication means (e.g., UDP messages). In these paper we call such algorithms *practical*.

Interest in practical consensus algorithms is not new. For example, algorithms relying on crash-recovery processes and omission-failure communication links were proposed in [14] and [1]. In this paper we introduce practical WAB-based consensus algorithms. Differently from those in [18], assume that processes can recover after failures and messages can be lost. The first one, B*-Consensus, is resilient to up to $f < n/2$ permanent failures; it solves consensus in three communication steps when the WAB oracle works. The second, R*-Consensus, is $f < n/3$ resilient and can solve consensus in two communication steps. Therefore, besides practical, our algorithms are also optimal regarding the time complexity versus reliability tradeoff.

We compare our algorithms both analytically and experimentally to Paxos, another optimal and practical consensus algorithm. Our experiments were conducted using Emulab's infrastructure [20]. We considered two network environments: one in which all processes are in the same sub-network – a configuration in which most likely messages are spontaneously ordered, and another in which processes are divided into two sub-networks. Our results show that in a single sub-network, R*-Consensus algorithm outperforms Paxos, and both of them outperform B*-Consensus. This was expected since R*-Consensus requires two communication steps while Paxos and B*-Consensus require three; furthermore, Paxos has a lower message complexity than B*-Consensus. More surprisingly, we show that under certain workloads, R*-Consensus performs better than Paxos even in environments with two sub-networks. We also point out situations in which our protocols become unstable, but Paxos remains stable, and discuss ways we are considering to address these problems in the future.

Our computational model and some definitions are presented in Section 2. In Section 4 we briefly describe Paxos. In Section 3 we present the B*-Consensus and the R*-Consensus algorithms. Analytical analysis and experimental evaluation are presented in Section 5. Related works are reviewed in Section 6. Section 7 concludes the paper. Proofs of correctness can be found in [4].

# 2 Model and definitions

## 2.1 Processes, communication, and failures

We consider an asynchronous system composed of a set $\Pi = \{p_1, \ldots, p_n\}$ of processes, $n \geq 3$. Processes communicate by message passing. Messages can be lost, however, we assume that they cannot be corrupted or duplicated. Processes can crash and recover an unlimited number of times but do not behave maliciously (i.e., no Byzantine failures). To ensure liveness we assume that eventually a subset of processes remains up forever. Such processes are called *stable*.

We are interested here in the consensus problem. A known result in distributed computing is that consensus cannot be solved in a purely asynchronous systems [11], and so cannot be solved in our model as described so far. In the following sections we first provide a definition of the consensus problem and then augment our asynchronous model with further assumptions.

## 2.2 The consensus problem

Processes executing consensus propose a value, interact to accept a single value, and then learn the decision. Some consensus definitions (e.g., [6, 1]) assume that processes play these roles indistinguishably: unless it crashes, each process should propose a value, cooperate to accept some value, and learn the decision. In this paper we consider a more general definition in which processes may play just a subset of these roles [15].

Characterizing processes as *proposers*, *acceptors*, and *learners* allows us to simplify the algorithm's presentation and evaluation. It also better models some real systems. For example, it adequately matches a system where application clients propose values to servers and then, without participating in the decision protocol themselves, learn the value accepted.

Using the decomposition of roles, consensus is defined as follows:

*Nontriviality:* only a proposed value may be learned.

*Consistency:* any two values that are learned must be equal.

*Progress:* for any proposer $p$ and learner $l$, if $p$, $l$ and $n - f$ acceptors are *stable*, and $p$ proposes a value, then $l$ must learn a value.

## 2.3 Oracle extensions

We consider two extensions to our asynchronous model. By augmenting our original model with each one of them at a time, consensus can be solved:

**Weak Ordering Oracles.** Weak ordering oracles [18] provide some message ordering guarantees. A WAB is a weak ordering oracle defined by the primitives w-broadcast$(k, m)$ and w-deliver$(k)$, where $k \in \mathbb{N}$ defines a w-broadcast instance, and $m$ is a message. The invocation of w-broadcast$(k, m)$ broadcasts message $m$ in instance $k$; w-deliver$(k, m)$ w-delivers a message $m$ w-broadcast in instance $k$. WAB satisfies the following property:

*Spontaneous Order.* If w-broadcast$(k, _)$ is invoked in an infinite number of instances $k$, then there are infinite $k'$ such that the first w-delivered message in instance $k'$ is the same for every process that w-delivers a message in $k'$.

Spontaneous order can be better understood by an example. Assume a system with three processes, $p_1$, $p_2$ and $p_3$. The following messages are w-broadcast: $(1, m_a)$, $(1, m_b)$, $(2, m_c)$, and $(2, m_d)$. If $p_1$ w-delivers messages $(1, m_a), (2, m_d), (1, m_b)$, and $(2, m_c)$, in this order; $p_2$ w-delivers $(1, m_b), (2, m_d)$, and $(2, m_c)$; and $p_3$ w-delivers $(1, m_a), (1, m_b)$ and $(2, m_d)$, then spontaneous order holds for $k = 2$, since all three processes w-deliver the same first message in instance 2, i.e., $(2, m_d)$.

WABs are motivated by empirical observation of the behavior of IP-multicast in some local-area networks (e.g.., Ethernet). In such environments, IP-multicast ensures that most broadcast messages are delivered in the same order to all network nodes.

**Leader election oracle ($\Omega$).** Intuitively, a leader election oracle ensures that stable processes should all eventually agree on the identity of some stable process. More precisely, assume that each process $p$ has a variable $leader_p$. The following property characterizes $\Omega$ [2]:

*Eventual leadership.* There exists some stable process $l$ and a time after which, for every stable process $p$, $leader_p = l$.

Hereafter we refer to an asynchronous system augmented with an $\Omega$ oracle as an AS+$\Omega$ model, and to an asynchronous system augmented with a WAB as an AS+WAB model.

### 2.4 Lower bounds on consensus

Lamport proved in [15] some lower bounds on the time complexity of consensus protocols. These bounds are valid in environments in which consensus can be eventually solved, that is, the bounds are derived assuming that the system passes through unfavorable periods in which no decision can be made. In an AS+WAB system, an unfavorable period happens in instances in which processes do not w-deliver the same first message. In a AS+$\Omega$ system, an unfavorable period happens when processes disagree on the identity of the leader.

The *Acceptor-Lower Bound* states that consensus cannot be solved in AS+WAB or AS+$\Omega$ if a majority of processes is not stable.[1]

*Acceptor Lower Bound.* For any natural number $f \leq n$ and any consensus algorithm, if every set of $n - f$ acceptors is enough to ensure eventual progress regardless of the proposer proposing, then $f < n/2$.

The *Fast-Learning Lower Bound* is of special interest here. This lower bound limits the size of $e$, the maximum number of faults that allows decision to be reached in two communication steps (*fast-learning*), and $f$, the maximum number of faults allowed to ensure progress.

*Fast-Learning Lower Bound.* For any natural numbers $e$ and $f$ with $f > 0$ and $e \leq f \leq n$, if every set of $n - f$ acceptors is enough to ensure eventual progress and if every set of $n - e$ acceptors allows proposals from at least two distinct proposers to be learned in two communications steps, then $n > 2e + f$.

[1] This lower bound and the one that follows it do not apply to two very specific cases, which are of no interest to our current work [15].

Minimizing $e$ leads to a resilience of $f < n/3$, when $e = f$. This two lower bounds together state that, in order to keep $e$ and $f$ in the minimum, i.e., $n/2$, consensus algorithms cannot allow fast-learning for more than one proposer. That is, there is only one proposer from which a proposal can be learned in two communication steps; other's proposals will take at least three communication steps to be learned.

In Section 3 we provide optimal algorithms for the AS+WAB model. We compare these algorithms with Paxos [14], an algorithm $f < n/2$ resilient, that allows fast learning for one proposer and that takes three communication steps for deciding on other proposer's proposals. Paxos is briefly explained in Section 4.

# 3 Optimal WAB-based algorithms

In this section we introduce two WAB-based consensus algorithms for the crash-recovery model: B*-Consensus and R*-Consensus. Our algorithms are inspired by those in [18], but differently from them, ours can tolerate an unbounded number of failures and cope with message losses. B*-Consensus requires $f < n/2$ for termination; R*-Consensus can reach decision faster than B*-Consensus but requires $f < n/3$ for termination. These are, to the best of our knowledge, the first WAB-based algorithms to consider crash-recovery failures and message loses. Furthermore, as we show in Section 5, they have optimal time complexity. We start by describing the behavior shared by both algorithms and then present the details of each one in separate sections.

Both B*-Consensus and R*-Consensus algorithms execute a sequence of rounds. In each round, proposers can propose a value, acceptors try to choose a value proposed in the round, and learners try to identify whether a decision has been made in the current round or if a new round must be started. For efficient recovery (from both crashes and message losses), our algorithms allow processes to jump to a round without performing the preceding ones: whenever a processor receives a message from a higher numbered round, it immediately joins it.

In a deciding round, (i) a proposer proposes a value by means of a WAB oracle; (ii) acceptors receive the proposed value, possibly interact to accept a value and notify the learners; and (iii) the learners, after receiving a reasonable number of acceptance messages, learn that a value was decided and tell the application. The main differences between the two algorithms are the way acceptors interact to accept a value (to increase the resilience from $f < n/3$ to $f < n/2$, acceptors must exchange additional messages among themselves before accepting a value), and the number of messages learners must receive to learn a decision.

The algorithms are divided in blocks: the *Initialization* block runs the first time the algorithm is started; when recovering from a crash, the *Recovery* block must run. The other blocks have clauses that are run when messages are received/w-delivered, according to the message's type. Each block runs until completion before another blocks can run.

In both algorithms, every process $p$ keeps a variable $r_p$ with the number of the highest-numbered round in which it took part, and a variable $prop_p$ that either has process' proposal for round $r_p$ or $\perp$, meaning that any value can be proposed. $prop_p$ and $r_p$ are always logged together (see Algorithms 1 and 2), ensuring that each process acts consistently when replaying a round after recovering from a crash.

**Skipping rounds.** When $p$ in round $r_p$ sees a message from round $r_q > r_p$, $p$ immediately jumps to $r_q$ without performing rounds $r_p + 1..r_q - 1$. This allows processes that were down for a long time to rapidly join the execution. Only processes that finished a round initially know which values can be proposed on the next one (after deciding rounds, for example, only the decided value can be proposed), but all processes joining a round must learn one of these values to act consistently with the other processes. This is accomplished by having the process' proposal attached to every message it sends (the last field of each message in the algorithm). The *Round Skipping Task*, presented in Algorithms 1 and 2, is responsible for skipping rounds. It runs on every message received/w-delivered before other clauses treat the message.

**Proposers.** A proposer's task is to receive a proposal request from the application for some value $v$ and try to reach decision on this value. As we consider unreliable communication channels and crash-recovery failures, a consensus instance may not terminate. To deal with this problem, our algorithms allow proposers to re-start a consensus instance if they believe that the previous attempt failed. To be able to learn that a round of the algorithm has terminated we assume that each proposer is also a learner. Thus, if a proposer does not learn the decision of the consensus it has initiated after some time, it re-starts its execution by proposing in its current round. Both Algorithms 1 and 2 use the same code fragment for proposers.

### 3.1 The B*-Consensus algorithm

Algorithm 1 presents the B*-Consensus algorithm.

**Acceptors.** After w-delivering the first proposal for a round, every acceptor $p$ determines whether it can accept it or not. A proposal is accepted if it is the first one w-delivered by $p$. In such a case, $p$ takes this proposal as its first estimative ($est1_p$), logs it together with the current round number $r_p$ and a valid proposal, so that no other first estimate will be accepted for this round. The proposer then exchanges estimatives with the other acceptors in CHECK messages. After collecting $\lceil\frac{n+1}{2}\rceil$ estimatives, $p$ checks whether they are equal. If so, $p$ accepts the value, that is, $p$ stores it in $est2_p$. If not, $p$ sets $est2_p$ to $\top$. In any case $p$ then logs $est2_p$ and $r_p$, and sends them to learners in SECOND messages.

**Learners.** Learners simply wait for $\lceil(n+1)/2\rceil$ decision messages. Once these messages are received, if all of them are for the same value $v$, a decision has been reached and $v$ is delivered to the application. If not, $p$ looks for at least one $v \neq \top$ in SECOND messages. In any case, $r_p$ is incremented and $prop_p$ is set to $v$ (or kept intact if such $v$ does not exist) so that any future rounds will only be able to decide $v$.

### 3.2 The R*-Consensus algorithm

Algorithm 2 presents the R*-Consensus algorithm.

**Acceptors.** As with B*-Consensus, every acceptor $p$ accepts the first proposal it w-delivers. The acceptor then sets $est1_p$ to the proposal, logs $est1_p$ together with $r_p$ and $prop_p$, so that these values will not be forgotten in case of crash, and sends them all to the learners in SECOND messages.

**Learners.** The learners' task is to wait for $\lceil(2n+1)/3\rceil$ SECOND messages and check whether they contain a decision value. Value $v$ is decided whenever all messages contain $v$. In such a case, $v$ is delivered to the application. If that is not the case, learners check whether a majority of messages

**Algorithm 1** The B*-Consensus Algorithm

Initialization:<br>
  $r_p \leftarrow 0, prop_p \leftarrow \perp, est1_p \leftarrow est2_p \leftarrow \perp, Cset \leftarrow Sset \leftarrow \emptyset$

Recovery:<br>
  $retrieve(r_p, prop_p, est1_p, est2_p)$<br>
  $Cset \leftarrow Sset \leftarrow \emptyset$

Round Skipping Task:<br>
  before receiving/w-delivering a message $m$ from round $r_q$<br>
    **if** $r_p > r_q$ **then**<br>
      send (SKIP,$r_p, prop_p$) to $q$<br>
    **if** $r_p < r_q$ **then**<br>
      $r_p \leftarrow r_q, prop_p \leftarrow prop_q, est1_p \leftarrow est2_p \leftarrow \perp, Cset \leftarrow Sset \leftarrow \emptyset$

To propose value $v_p$ proposers do as follows:<br>
  **if** $prop_p = \perp$ **then**<br>
    $prop_p \leftarrow v_p$<br>
  w-broadcast (First,$r_p, prop_p$) to acceptors

Acceptors execute as follows:<br>
  **upon** w-deliver (First, $r_q, prop_q$) **do**<br>
    **if** $est1_p = \perp$ **then**<br>
      $est1_p \leftarrow prop_q$<br>
      log ($est1_p, r_p, prop_p$)<br>
      send (Check, $r_p, est1_p, prop_q$) to all acceptors<br>
  **upon** receive (Check, $r_p, est1_q, prop_p$) **do**<br>
    $Cset \leftarrow Cset \cup \{($Check$, r_p, est1_q, prop_q)\}$<br>
    **if** $|Cset| = \lceil (n+1)/2 \rceil$ **then**<br>
      **if** $\forall($Check$, r_p, est1_q, -) \in Cset : est1_q = v$ **then**<br>
        $est2_p \leftarrow v$<br>
      **else**<br>
        $est2_p \leftarrow \top$<br>
      log ($est2_p, r_p, prop_p$)<br>
      send (Second, $r_p, est2_p, prop_p$) to all learners

Learners execute as follows:<br>
  **upon** receive (Second, $r_p, est2_q, v_q$) **do**<br>
    $Sset \leftarrow Sset \cup \{($Second$, r_p, est2_q, v_q)\}$<br>
    **if** $|Sset| = \lceil (n+1)/2 \rceil$ **then**<br>
      **if** $\forall($Second$, r_p, est2_q, _) \in Sset : est2_q = v \neq \top$ **then**<br>
        decide $v$<br>
      **if** $\exists($Second$, r_p, est2_q, _) \in Sset : est2_q = v \neq \top$ **then**<br>
        $prop_p \leftarrow v$<br>
      $r_p \leftarrow r_p + 1, est1_p \leftarrow est2_p \leftarrow \perp, Cset \leftarrow Sset \leftarrow \emptyset$

has the same value $v$. In any case, the learner passes to the next round, that is, $r_p$ is incremented, and $prop_p$ set to $v$, if it exists, so that any future round will only be able to decide $v$.

**Algorithm 2** The R*-Consensus Algorithm

<u>Initialization:</u>
  $r_p \leftarrow 0, prop_p \leftarrow, est1_p \leftarrow \bot, Sset \leftarrow \emptyset$

<u>Recovery:</u>
  $retrieve(r_p, prop_p, est1_p)$
  $Sset \leftarrow \emptyset$

<u>To be executed on each arriving message:</u>
  before receiving/w-delivering a message $m$ from round $r_q$
    **if** $r_p > r_q$ **then**
      send (SKIP,$r_p, prop_p$) to $q$
    **if** $r_p < r_q$ **then**
      $r_p \leftarrow r_q, prop_p \leftarrow prop_q, est1_p \leftarrow \bot, Sset \leftarrow \emptyset$

<u>To propose value $v_p$ proposers do as follows:</u>
  **if** $prop_p = \bot$ **then**
    $prop_p \leftarrow v_p$
  w-broadcast (<u>First</u>,$prop_p$) to acceptors

<u>Acceptors execute as follows:</u>
  **upon** deliver (<u>First</u>,$r_p, prop_q$) **do**
    **if** $est1_p = \bot$ **then**
      $est1_p \leftarrow prop_q$
      log $(est1_p, r_p, prop_p)$
      send (<u>Second</u>, $r_p, est1_p, prop_p$) to all learners

<u>Learners execute as follows:</u>
  **upon** deliver (<u>Second</u>,$r_p, est1_q, v_q$) **do**
    $Sset \leftarrow Sset \cup \{$(<u>Second</u>, $r_p, est1_q, v_q$)$\}$
    **if** $|Sset| = \lceil (2n+1)/3 \rceil$ **then**
      **if** $\forall$(<u>Second</u>, $r_p, est1_q, \_) \in Sset : est1_q = v$ **then**
        decide $v$
      **if** $\exists v_{maj} \ni$ for $\lceil (n+1)/2 \rceil$ (<u>Second</u>, $r_p, v, \_) \in Sset : v = v_{maj}$ **then**
        $prop_p \leftarrow v_{maj}$
      **else**
        $prop_p \leftarrow \bot$
      $r_p \leftarrow r_p + 1, est1_p \leftarrow \bot, Sset \leftarrow \emptyset$

# 4 Paxos

The Paxos algorithm relies on a leader to reach a decision. This is a process among the proposers, elected by the leader election oracle described in Section 2. The algorithm ensures that even in the presence of multiple simultaneous leaders, correctness is maintained. Termination is guaranteed when a single stable leader exists. To propose a value $v$ a proposer queries its leader election oracle and sends $v$ to the current leader.

Proposers use increasing timestamps, kept in the *tmp* variable, to tag messages they exchange with acceptors. Two different proposers can never use the same timestamp. Acceptors maintain in stable storage the greater timestamp they heard from a proposer, *g_tmp*, the last accepted value, *acc*, and the timestamp associated to this value, *t_acc*. The algorithm proceeds in phases:

- **Phase 1a.** The leader chooses a timestamp it believes to be bigger than any other used timestamp and sends it to all acceptors in a Phase 1a message $\langle 1a, tmp\rangle$.

- **Phase 1b.** If an acceptor receives an $\langle 1a, tmp\rangle$ message such that $tmp > g\_tmp$, it sets $g\_tmp$ to $tmp$, and sends a Phase 1b message $\langle 1b, tmp, acc, t\_acc\rangle$ to the proposer; i.e., it sends its last accepted value, if any, and the associated timestamp. Acceptors can also inform proposers that their timestamps were too small.

- **Phase 2a.** On receiving a majority of $1b$ messages, the leader checks if they carry any previous accepted value. If they do, the leader chooses the one with the bigger associated timestamp $t\_acc$ as its proposal $v$. If none of the messages carry previous accepted values, the leader sets $v$ to any value it chooses. A message $\langle 2a, tmp, v\rangle$ is then sent to the acceptors. If the leader does not hear from a majority of acceptors (or if it hears its timestamp was too small), it re-starts Phase 1 with a bigger timestamp.

- **Phase 2b.** Whenever an acceptor receives a $\langle 2a, tmp, v\rangle$ message with $tmp \geq g\_tmp$ it accepts it, sets $g\_tmp$ to $tmp, acc$ to $v$ and $t\_acc$ to $tmp$, and informs the learners by sending a message $\langle 2b, t\_acc, v\rangle$ to all of them.

- **Decision** When a learner receives a majority of $2b$ messages with the same $t\_acc$ it decides $v$.

This algorithm can be optimized so that when a process becomes the leader, it performs Phase 1 for future consensus instances before some value is actually proposed. Moreover, several Phase 1 messages can be grouped in single messages. This optimization allows a decision to be reached in two communication steps when the proposer is the leader, or in three communication steps when the proposer is another process.

# 5 Performance Evaluation

## 5.1 Analitical Evaluation

Table 1 compares Paxos, R*-Consensus and B*-Consensus algorithms in terms of communication steps (i.e., expected latency), number of messages, resilience, and oracle needed for termination. Latency is expressed in terms of $\delta$, the constant network delay assumed for the analysis of the algorithms. We consider both point-to-point and multicast communication, and assume that both point-to-point and multicast messages have the same cost, a reasonable assumption for a LAN.

From Table 1, the optimized version of Paxos takes the same number of communication steps as B*-Consensus but, due to its centralized nature, needs nearly half the messages. Two more communication steps are required when Paxos runs the first phase. Although the number of messages is half of B*-Consensus with point-to-point communication, it becomes almost the same when broadcast is available. Moreover, if the proposer is the current leader, then one communication

| Protocol | Expected Latency | Number of Messages | | Resilience | Oracle |
|---|---|---|---|---|---|
| | | Point-to-Point | Broadcast | | |
| B*-Consensus | $3\delta$ | $2n^2+n+1$ | $2n+1$ | $f<n/2$ | WAB |
| R*-Consensus | $2\delta$ | $n^2+n$ | $n+1$ | $f<n/3$ | WAB |
| Paxos (optimized) | $3\delta$ | $n^2+n$ | $n+2$ | $f<n/2$ | $\Omega$ |
| Paxos (normal) | $5\delta$ | $n^2+3n+1$ | $2n+3$ | $f<n/2$ | $\Omega$ |

Table 1: Analytical comparison

step and one message can be saved in Paxos. When compared to R*-Consensus, the optimized version of Paxos uses the same number of messages, and trades one communication step for better resilience: $f < n/2$ instead of $n < n/3$. Finally, notice that Paxos always degenerate to the normal case after the first try to achieve consensus fails using the optmized version of the protocol.

### 5.2 Experimental Evaluation

**Environment.** All algorithms were implemented in Java 1.4.2 in the form of a library. The library consists of a set of layers that can be put together in a stack to give the desired properties to the application programmer. Different algorithms were tested by simply changing the *Consensus Layer*. The library is currently tailored for LANs, relying on IP multicast, backed by Ethernet broadcast, for both communication and the WAB oracle. This minimizes the number of messages exchanged in the network as there is no need to send the same message to different processes individually.[2]

We used Emulab's network [20] as our testbed. All experiments were performed using 8, 16 or 32 nodes, either in the same network or divided into two sub-networks of the same size, connected through a node running the mrouted daemon. Each node in the system was a Pentium III 850MHz, 256 MB of RAM, running FreeBSD 4.10 on agent nodes and Red Hat 9.0 Linux on the routers. All nodes were connected to the same switches in all tests.

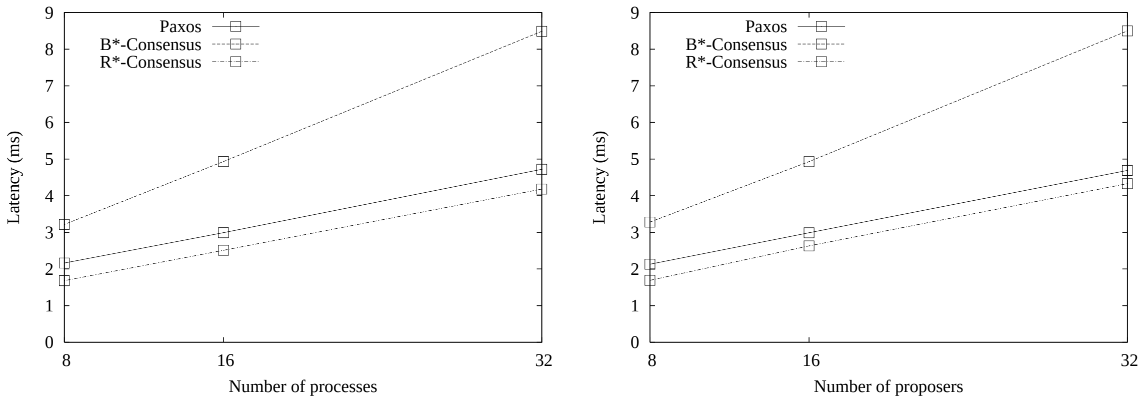
Our experiments consisted of four scenarios. In each one no process failed, but messages were lost. We considered a "perfect" $\Omega$ oracle that always returns the same process when queried. Actually, no real crash monitoring was performed and the leader process was "hardwired" in the oracle's implementation. This clearly benefited Paxos since no heartbeat messages were exchanged and no computation was performeed to elect a leader.

**First Scenario.** In this scenario there is a single proposer proposing values sequentially, that is, a value is only proposed after the decision of the previous proposal is learned by the process. All processes are acceptors and learners. Experiments were conducted with 8, 16 and 32 nodes in the same network (Figure 1(a)) and in two sub-networks, each one with 8 nodes (Figure 1(b)). Scenario 1 is expected to be the most common in practice. Our implementation of Paxos considers its optimized version, saving two communication steps in each instance. The proposer and the leader were in different nodes, forcing at least three communication steps in each instance. However, in the setup with two sub-networks, we placed the leader and the proposer in the same sub-network.

From Figures 1(a) and 1(b) we can see that latency increases linearly with the number of processes. This is not surprising since only one process starts consensus instances, and the messages generated did not flood the router. R*-Consensus has slightly better performance than Paxos due to the extra communication step needed by Paxos; this difference was constant with the number of

[2]All prototypes used in our evaluation can be downloaded from `http://www.inf.unisi.ch/sprint`.

processes. This can be explained by the fact that both algorithms introduce the same number of messages in the network. Proportionally to the number of processes, the advantage of R*-Consensus decreases as more processes are added. For example, in a single network, while the difference between R*-Consensus and Paxos is 22% with 8 processes, it is only 11% with 32 processes.




(a) Single network.

(b) Two networks

Figure 1: Sequential consensus instances (Scenario 1).

**Second Scenario.** In this experiment we used 8 nodes playing the acceptor and learner roles. The number of proposers increased from 1 to 8. Each proposer runs two threads: the first starts 50 or 100 consensus instances per second (CIPS); the second waits 10ms for a decision of each instance and then re-proposes, remaining in this cycle until a decision has been reached.

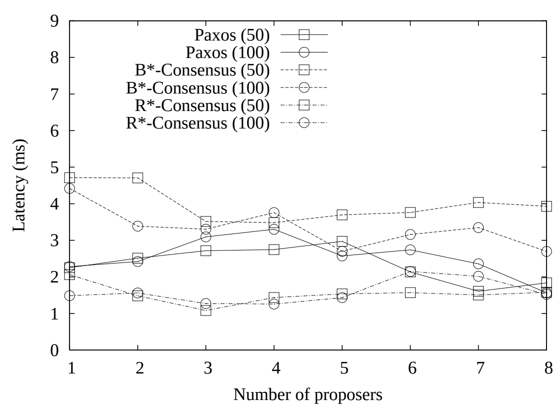



Figure 2: Scenario 2 (single network, varying the number of proposers and CIPS)

Figure 2 shows the mean latency of consensus instances. We can see that in general R*-Consensus has a better performance than Paxos that, in turn, has a better performance than B*-Consensus. This confirms the trend we found in Scenario 1. Moreover, for R*-Consensus and Paxos in this environment, there is little difference between 50 and 100 CIPS.

**Third Scenario.** This experiment is similar to the previous one but used 16 nodes instead of

8. The results for B*-Consensus were omitted from the graph due to its poor performance when compared to the other algorithms, as shown in Scenarios 1 and 2. For the other algorithms we also present the latency for 20 CIPS.

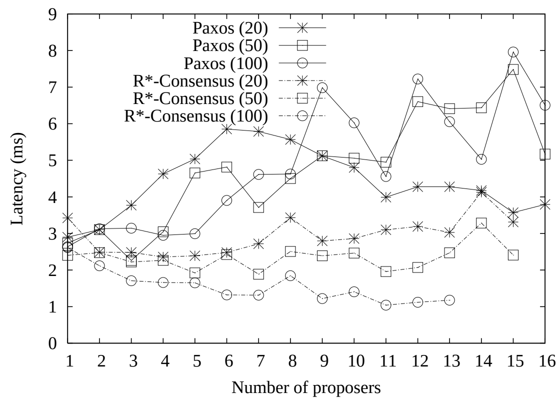



Figure 3: Scenario 3 (single network, varying the number of proposers and CIPS)

From Figure 3 we see that R*-Consensus performs better under high workload than under low. This seems to be related to our prototype implementation, as we now explain. As aforementioned, each process has two threads: the first one proposes in each instance and the second waits until it is notified by the first that an instance has started. For each consensus instance, an object is created after the first message for that instance arrives, or a decision is queried by the application. The more often the instance is queried by the application, the more likely the object for that instance will be allocated before a message arrives, and so be ready to answer the arriving messages. This fact also accounts for the decrease in the latency while increasing the number of proposers: with more processes initializing quickly each instance, quicker answers will be given to the protocol messages.

**Fourth Scenario.** This experiment used 16 nodes divided into two sub-networks connected by a router. We increased the number of proposers, adding one on each sub-network alternatively. We conducted experiments with both R*-Consensus and Paxos, both under 50 and 100 CIPS.

With two sub-networks and proposers in different ones, we expected R*-Consensus' performance to degrade since spontaneous order was unlikely to hold. Much to our surprise, under 100 CIPS and up to 4 concurrent proposers, R*-Consensus still outperforms Paxos (see Figure 4). This happens because for such workloads, the routers are not overloaded and most message still arrive in the same order in all processes. Thus, the two communication steps of R*-Consensus prevail. For other workloads, however, Paxos presents better performance than R*-Consensus. We also confirmed what we had found in our previous experiments: R*-Consensus performs better under 100 CIPS than 50 CIPS due to implementation details.

**Overall Conclusions.** Provided that spontaneous order holds, the number of messages and latency of the algorithms seem to determine their differences in performance. R*-Consensus is better than Paxos because the former needs two communication steps while the latter needs three. Both Paxos and B*-Consensus require three communication steps, and thus the former outperforms the latter. This does not seem to change with the number of processes and CIPS. The situation changes once we consider environments where spontaneous order is not expected to hold. Paxos,

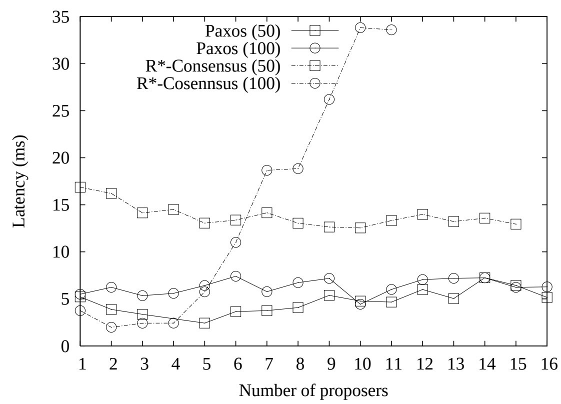



Figure 4: Scenario 4 (two networks, varying the number of proposers and CIPS)

which does not depend on spontaneous order, tends to behave better. However, in workloads with few proposers, few w-broadcast messages are w-delivered out of order and the number of communication steps prevails. Apparently, one inherent reason for Paxos performing well in such cases is that the leader provides a natural "load regulator". This happens because all messages should pass by the leader, and processes cannot overload the network. We are currently researching ways of introducing some load regulation in WAB-based protocols.

# 6 Related Work

WAB-based consensus algorithms were first studied in [18]. The algorithms studied assumed crash-stop failures and reliable channels. Here we presented WAB-based algorithms in the crash-recovery model in which messages can be lost.

The problem of consensus in the crash-recovery model was previously studied in [14, 16, 12, 17, 1]. These approaches considered either a leader-election oracle or unreliable failure detectors as extensions to the asynchronous model.

The Paxos algorithm [14, 16] runs on AS+$\Omega$ environments. Paxos is $f < n/2$ resilient and can decide in three communication steps. It is therefore optimal. The WAB-based algorithms we presented here are also optimal for $f < n/2$ and $f < n/3$. Moreover, processes executing Paxos log part of their state in disk to survive failures. This happens once per round in the optimized mode, and twice in the normal mode.

Hurfin et al [12] presented an algorithm that has the same message pattern as Paxos in optimized mode. Because it uses the rotating coordinator paradigm, decision can be delayed when coordinators, elected deterministically, crash. Processes log only once per round.

The algorithm presented by Oliveira et al. [17] is also based on the rotating coordinator paradigm. Processes can learn the proposal of the coordinator in 2 communication steps. Differently from ours, their algorithm assumes a stronger model in which memory contents survives processes crashes (i.e., stable main memory).

Aguilera et al. [1] showed that if there is a process that never crashes ("always-up process") and is always listened every time processes wait for $n - f$ messages then consensus is solvable without stable storage; without this assumption stable storage is needed. Algorithms that match both bounds are presented. The algorithm relying on stable storage is $f < n/2$ resilient. In best case runs, processes access stable storage twice in a round, exchange $4n$ messages, and reach decision within 3 communication steps.

## 7 Conclusion and Future Directions

In this paper we introduced B*-Consensus and R*-Consensus, two WAB-based algorithms that assume the crash-recovery model and tolerate message losses. Both algorithms can cope with any number of process failures without violating safety. B*-Consensus takes 3 communication steps to reach a decision and requires a majority of stable processes to ensure progress. R*-Consensus can decide in 2 communication steps, but requires more than two thirds of stable processes. Both algorithms are optimal in terms of communication steps for the resilience they provide.

We also compared our algorithms with Paxos, a well known $\Omega$-based optimal consensus algorithm. These results indicate that R*-Consensus performs better than Paxos on single sub-networks, having better latencies and being more stable when varying the workload. Paxos performs better than our algorithms in more complex networks for most workloads. The WAB-based algorithms we provided are completely decentralized. If on the one hand they provide very fast reaction to failures since they do not depend on any leader or coordinator election, on the other hand, they may flood the network with messages when several proposers exist.

As future work we plan to extend the scenarios of our experiments, exploring different network topologies (e.g., multiple routers connecting two sub-networks). We are also considering more sophisticated WAB implementations which would allow us to better control the execution flow.

## References


[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. of the 12th International Symposium on Distributed Computing*, Sept. 1998.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 108–122, London, UK, 2001. Springer-Verlag.

[3] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM Press, 1983.

[4] L. Camargos, F. Pedone, and E. Madeira. Optimal and practical wab-based consensus algorithms. Technical Report IC-05-07, Institute of Computing, State University of Campinas, Campinas, Brazil, Apr. 2005. Available at http://www.ic.unicamp.br/ic-tr/.

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

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

[7] B. Charron-Bost and A. Schiper. Uniform consensus is harder than consensus. *J. Algorithms*, 51(1):15–37, 2004.

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

[9] P. Dutta and R. Guerraoui. Fast indulgent consensus with zero degradation. *Lecture Notes in Computer Science*, 2485, 2002.

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

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[12] M. Hurfin, A. Mostefaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems, IEEE Comput.*, pages 280–286, Soc, Los Alamitos, CA, 1998.

[13] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *SIGACT News*, 32(2):45–63, 2001.

[14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[15] L. Lamport. Lower bounds for asynchronous consensus. Technical Report MSR-TR-2004-72, Microsoft Research, Switzerland, July 2004.

[16] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highlyavailable distributed systems. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, New York, NY, USA, 1988. ACM Press.

[17] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report TR-97/239, EPFL – Départment d'Informatique, Lausanne, Switzerland, Aug. 1997.

[18] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving agreement problems with weak ordering oracles. In *4th European Dependable Computing Conference (EDCC-4)*, Toulouse, France, 2002.

[19] M. O. Rabin. Randomized byzantine generals. In *Proc. of the 24th Annu. IEEE Symp. on Foundations of Computer Science*, pages 403–409, 1983.

[20] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

# Appendix

## $B^*$-Consensus' correctness proof

**Lemma 7.1** *No acceptor sends* <u>CHECK</u> *or* <u>SECOND</u> *messages in a given round $r$ of consensus with different estimatives $est1_a$ (or $est2_a$).*

**Proof:** Variables $est1_a$ and $est2_a$ are always logged after being changed and before being sent in <u>CHECK</u> or <u>SECOND</u> messages. Changes are done only once per round, and the round number $r_a$ is logged together with $est1_a$ and $est2_a$, ensuring that the same value will be used after recovering from a crash, when re-executing a round, and only when re-executing the correct round. Any resent message will carry this logged value. □

**Lemma 7.2** *If $v$ is decided in $r$, then any process that executes $r$ until the end will either decide $v$ or set $v$ as its proposal for the next round.*

**Proof:** Let $r$ be the round in which $v$ is decided, $l$ a learner that learned $v$ was decided in $r$, and $l'$ a learner that finishes round $r$. To learn $v$, $l$ received $\lceil (n+1)/2 \rceil$ messages (<u>SECOND</u>,$r, v, \_$), and there must be a set $Q$ of acceptors such that $|Q| = \lceil (n+1)/2 \rceil$ that accepted $v \neq \perp$. This implies that a message (<u>CHECK</u>,$r, v, \_$) was sent by a majority of acceptors and, moreover, that due to Lemma 7.1, for all acceptors $q \in \Pi, est2^r_q = \{v, \top\}$. It also follows that all messages (<u>SECOND</u>,$r, w, \_$) received by $l'$ have $w = \{v, \perp\}$, and at least one has $w = v$. From the algorithm, $l'$ learns $v$ or sets $prop_{l'} \leftarrow v$. □

**Property 7.1** *Consistency: any two values that are learned must be equal.*

**Proof:** Suppose $v$ is first decided in round $r$. By Lemma 7.2, no $v' \neq v$ caqn be decided in $r$. Moreover, any process that finishes $r$ and does not decide will set $v$ as its proposal on round $r+1$. Processes that join $r+1$ without finishing $r$ will do so after receiving a message sent in round $r+1$ that also carries $v$, which will be used as their proposal. Any process that finishes round $r+1$ and enters $r+2$ will set *prop* to $v$ at the end of $r+1$, as this is the only value being proposed at round $r+1$. By a simple induction on the round number we see that $v$ is the only proposal allowed in any round $r' > r$. It follows that any process that terminates decides $v$. □

**Property 7.2** *Nontriviality: only a proposed value may be learned.*

**Proof:** Immediate from the algorithm. □

**Property 7.3** *Progress: for any proposer $p$ and learner $l$, if $p$, $l$, and $n - f$ acceptors are stable, $f < n/2$, $p$ keeps proposing some value, and messages are not lost, then $l$ learns some value.*

**Proof:** By the *eventual order* property of WAB, eventually the same value will be w-delivered by all stable acceptors at a round $r$ and will be the only accepted value by them in this round. Thus, this value iis decided. From the algoriithm, a learner will learn this value in round $r' \geq r$. □

### R*-Consensus' correctness proof

**Lemma 7.3** *No acceptor sends* <u>SECOND</u> *messages in a given round* $r$ *of consensus with different estimatives* $est1_a$*.*

**Proof:** Variable $est1_a$ is always logged after being changed and before being sent in <u>SECOND</u> messages. Changes are done only once per round, and the round number $r_a$ is logged together with $est1_a$, ensuring that the same value will be used after recovering from a crash, when re-executing a round, and only when re-executing the correct round. Any resent message will carry this logged value. □

**Lemma 7.4** *If* $r$ *is a round in which some process decides, then any process that executes* $r$ *until the end will either decide* $v$ *or set* $v$ *to its proposal for the next round.*

**Proof:** Let $l$ be a learner that learned $v$ was decided on $r$, and $l'$ a learner that finishes round $r$. To learn $v$, $l$ received $\lceil (2n+1)/3 \rceil$ messages (<u>SECOND</u>,$r, v,$ _), and there must be a set $Q$ of acceptors such that $|Q| = \lceil (2n+1)/n \rceil$ that accepted $v$. This implies that, due to Lemma 7.3, at least $\lceil (n+1)/3 \rceil$ of all messages (<u>SECOND</u>,$r, w,$ _) received by $l'$ have $w = \{v\}$, i.e., a majority of the <u>SECOND</u> messages received. So, either all messages have $w = v$ and $l'$ learns $v$ and sets $prop_{l'}$ to $v$, or it just sets $prop_{l'}$ to $v$. □

**Property 7.4** *Consistency: any two values that are learned must be equal.*

**Proof:** Suppose $v$ is decided in round $r$. By Lemma 7.4, no $v' \neq v'$ can be decided in $r$. Moreover, any process that finishes $r$ and does not decide will set $v$ as its proposal on round $r+1$. Processes that join $r+1$ without finishing $r$ will do so after receiving a message sent in round $r+1$ that also carries $v$, which will be used as their proposal. Any process that finishes round $r+1$ and enters $r+2$ will set *prop* to $v$ at the end of $r+1$, as this is the only value being proposed at round $r+1$. By a simple induction on the round number we see that $v$ is the only proposal allowed in any round $r' > r$. It follows that any process that terminates decides $v$. □

**Property 7.5** *Nontriviality: only a proposed value may be learned.*

**Proof:** Immediate from the algorithm. □

**Property 7.6** *Progress: for any proposer* $p$ *and learner* $l$*, if* $p$*,* $l$*, and* $n - f$ *acceptors are stable,* $f < n/3$*,* $p$ *keeps proposing some value, and messages are not lost, then* $l$ *learns some value.*

**Proof:** By the *eventual order* property of WAB, eventually the same value will be w-delivered by all stable acceptors at a round $r$ and will be the only accepted value by them in this round. Eventually a learner will learn this value in round $r' \geq r$. □