Technical Report

UCAM-CL-TR-582
ISSN 1476-2986

Number 582

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Latency-optimal Uniform Atomic Broadcast algorithm

Piotr Zieliński

February 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Series editor: Markus Kuhn

ISSN 1476-2986

Latency-optimal Uniform Atomic Broadcast algorithm

Piotr Zieliński
University of Cambridge
Computer Laboratory
`piotr.zielinski@cl.cam.ac.uk`

Abstract

We present a new asynchronous Uniform Atomic Broadcast algorithm with a delivery latency of two communication steps in optimistic settings, which is faster than any other known algorithm and has been shown to be the lower bound. It also has the weakest possible liveness requirements (the Ω failure detector and a majority of correct processes) and achieves three new lower bounds presented in this paper. Finally, we introduce a new notation and several new abstractions, which are used to construct and present the algorithm in a clear and modular way.

1 Introduction

State machine replication [11] allows computer systems to achieve high fault-tolerance. Several computers perform the same sequence of operations so that the system still provides service when some of its components become unavailable. The main difficulty with this approach is to ensure that all participants perceive events in the same order. The solution, *Atomic Broadcast* [11], has been studied extensively and many algorithms have been proposed (see [7] for a survey).

In this paper, we consider Uniform Atomic Broadcast, in which safety properties hold at all processes (including the faulty ones). Our goal is to maximize the performance of the system by minimizing the delivery latency in failure-free runs. We present an algorithm that is faster, in this respect, than any previously proposed one.

The definition of delivery latency in this context can be a source of confusion. In this paper, we define it as the time between the atomic broadcast of a message and its atomic delivery. Note that many papers (e.g., [8, 17]) ignore the first step, in which the sender physically broadcasts the message to other processes; in that case one communication step must be added to the reported delivery latency.

The algorithm presented in this paper is always safe and achieves several lower bounds. Firstly, it requires a majority of correct processes and an Ω failure detector for liveness [4]. Secondly, in failure-free runs, it delivers messages sent by any process in two communication steps [5, 13]. Thirdly, the new lower bounds presented in Section 5 imply that the conditions under which it achieves the two-step delivery latency cannot be weakened. Finally, our algorithm is *quiet*, that is, no network messages are sent unless something is actually atomically broadcast.

An important contribution of this paper is the presentation of several new abstractions that are used to construct the final algorithm in a modular way. In particular, we introduce weaker versions of Interactive Consistency [18] and Atomic Broadcast. Also, we use a new guard-based notation to make the presentation clearer.

This paper is structured in the following way. Section 2 introduces the system model, and Section 3 introduces our notation. Section 4 is the core of the paper; it introduces the definitions and implementations of several new abstractions and also presents our algorithm. Section 5 presents the new lower bounds, Section 4.6 discusses possible optimizations, and Section 6 concludes the paper.

1.1 Related work

A number of Uniform Atomic Broadcast protocols have been proposed in the literature (see [7] for a comprehensive survey). All of them must ensure uniform agreement on the order of message delivery, which requires two communication steps [5, 13]. Moreover, all non-communication-history algorithms [7] need at least one additional step before the agreement process can start. (This is necessary for acquiring the token, or sending the message to the sequencer or to all destinations.) Therefore, all these protocols have the delivery latency of at least three communication steps. Examples include Paxos [14], Chandra-Toueg [4], and others [8, 9]. The optimal latency of two steps reported in [8, 17] results merely from not counting the first step.

One of our lower bounds states that no delivery latency below three communication steps is possible if physical time is not used. The only communication history algorithm in [7] that satisfies this criterion is HAS [6]. However, HAS bases its safety on timing assumptions, which can lead to the violation of Total Order if they do not hold [7].

To the best of our knowledge, the method in [19] is the only Uniform Atomic Broadcast algorithm proposed so far to break the barrier of three communication steps for delivery latency in failure-free runs. There, all processes are equipped with perfectly synchronized clocks and send messages every δ units of time even if nothing is being atomically broadcast. The algorithm has a latency of $\min\{2 + \delta, 3\}$. In comparison, our algorithm achieves a latency of 2, and does not require processes to send empty messages.

2 System model and problem statement

We consider a distributed system that consists of n processes p_1, p_2, $\ldots$, p_n. For safety properties, we assume that processes can fail only by crashing and communicate through asynchronous channels that cannot create or modify messages.

For liveness properties, we additionally assume correct-restricted reliable channels[1] [10], an Ω failure detector [4], and a majority of correct processes. We say that Ω has *stabilized* with process p if its output is p at all processes that have not crashed, and will never change. Process p is called the *eventual leader*.

For all performance properties, we assume *stable runs*: all processes are correct, equipped with perfectly synchronized clocks, and all network messages reach their destination within 1 time unit.

[1]These are channels that are reliable only if both the sender and the receiver are correct processes

Action	Guard	Body
$\mathbf{init}_p$	$\mathit{init}_p(\mathit{leader})$, idemp $\mathit{propose}_p$, idemp decide_p	—
$\mathbf{leader}_p$	*once* and $\mathit{propose}_q(v)$ and $p = \mathit{leader}$	$\mathit{decide}_p(v)$
$\mathbf{decide}_p$	$\mathit{decide}_q(v)$	$\mathit{decide}_p(v)$

Figure 1: Simple Consensus: code for every process p.

Uniform Atomic Broadcast allows processes to broadcast and deliver messages, with the following restrictions [7, 11]:

Uniform Agreement If a process delivers a message m, then all correct processes will eventually deliver m.

Uniform Validity For any message m, every process delivers m at most once and only if some process has broadcast m.

Uniform Total Order If a process delivers a message m' after a message m then a process that delivers m' has previously delivered m.

Termination If a correct process broadcasts a message m, it every correct process will eventually deliver m.

3 Notation

To allow the implementations of our algorithm maximum flexibility, we avoid the use of a sequential pseudocode. Instead, we use a guard-based notation similar to that in [16]. This decision, we believe, improves clarity of presentation, makes formal proofs easier, and in some cases also saves space. We will introduce the notation using the simple fault-intolerant Consensus protocol from Figure 1 as an example. There, every process p proposes a value v_p by calling $\mathit{propose}(v_p)$ and the unique leader decides on one of those values by calling $\mathit{decide}(v)$. If a process notices that another process has decided, it decides on the same value.

We specify the system as a collection of actions of the form $(\mathit{name}, \mathit{guard}, \mathit{body})$. Here, *name* is the action name, *guard* is a boolean expression, and *body* is an atomic operation, which is performed when the action is *executed*. We assume the following about action execution. For safety properties: an action can be executed only if it is enabled (i.e., *guard* is true). For liveness properties: an action enabled forever will eventually be executed *at least* once. For performance properties: every enabled action that has not been executed yet will be executed immediately. In our example, $\mathbf{leader}_p$ can be executed only if it has not been executed before, only if some process has proposed v, and only by the leader. If those three conditions are met, $\mathbf{leader}_p$ will eventually be executed.

Actions communicate with each other through local variables or correct-restricted reliable channels. Processes use channels mainly internally as function calls and our notation reflects this (*propose*, *decide*, and *once* are all channels). For every channel *ch*, the operation $\mathbf{put}\,\mathit{ch}(m)$ inserts m into *ch*, and the predicate $\mathbf{get}\,\mathit{ch}(m)$ is true if m can be extracted from *ch*. Whenever $\mathbf{put}\,\mathit{ch}(m)$ is executed, we say that $\mathit{ch}(m)$ is *invoked.*

For clarity, we drop the name of the operation (**put** or **get**) if it is obvious from the context: $ch(m)$ denotes $\mathbf{get}\, ch(m)$ in a guard and $\mathbf{put}\, ch(m)$ in an action body. The type of x in $ch(x)$ depends on the channel; in particular, for *anonymous channels*, x can only take one value: *tick*. We say that an anonymous channel *ch ticks* if $\mathbf{get}\, ch$ becomes true. Note that *propose* can be viewed either as a channel accepting values or as a collection of anonymous channels indexed by those values.

There are two types of channels: *idempotent* and *regular*. Idempotent ones (*propose* and *decide* in our example) do not distinguish identical elements. Those channels are basically sets with "delayed insertion": $\mathbf{put}\, ch(x)$ ensures that x will be eventually added to the set, whereas $\mathbf{get}\, ch(x)$ tests for membership in this set. As a result, the number of executions of $\mathbf{put}\, ch(x)$ does not matter and $\mathbf{get}\, ch(x)$ does not have any side-effects. Only idempotent channels are used for interprocess communication in this paper.

Regular channels do distinguish identical elements by assigning a unique instance id to each object. When an action that contains $\mathbf{get}\, ch(x)$ in its guard is executed, one instance of x is removed from ch[2]. We use two special anonymous regular channels: *once* and *occ* (for "occasional"), which allow a process to eventually extract, respectively, one and infinitely many *tick* objects. Finally, FIFO channels form a subtype of regular channels in which the order of insertion and extraction is the same.

Actions and channels are identified by their classes and instances. For actions, the class name consists of the action name and the process at which it is executed (e.g., $\mathbf{leader}_p$). The instance of a given action class is identified by the values of all free variables and object instances extracted from all channels in the guard (in our case, q, v, and an instance of *tick* from *once*). Each forever enabled action instance is executed at least once.

For channels, the class name consists of the channel name, which includes the owner (e.g., $decide_p$). An instance of a given channel class is identified by the action class in which the channel appears. (All instances of the same action class share the same instance of any channel class.) The $\mathbf{get}\, ch$ operation acts only on one instance of ch. On the other hand, $\mathbf{put}\, ch$ acts simultaneously on all instances of class ch but can be invoked only by the owner.

For example, adding *once* to the guard of $\mathbf{decide}_p$ would ensure that each process could execute $\mathbf{decide}_p$ at most once. This is because (for fixed p and variable v) all instances of $\mathbf{decide}_p$ belong to the same class and share a single instance of the *once* channel. Since each action class gets an independent instance of *once*, adding *once* to the guard of $\mathbf{decide}_p$ would neither mean that at most one process p would execute $\mathbf{decide}_p$ nor affect $\mathbf{leader}_p$.

To save space, we assume that channel instances are created on demand. For any channel class ch, the owner creates instances for all action classes whose guards contain $\mathbf{get}\, ch$. In Figure 1, every process creates an instance of *propose* for $\mathbf{decide}_l$ at the leader l, and an instance of *decide* for $\mathbf{decide}_p$ at every process p.

We use the following conventions. All free variables in guards are single-letter and all others are longer, except for p, which denotes the current process. We use p, q, r for process names, v, w for proposed values, s, t for times, and m for messages. We subscript all actions, channels, and variables with the name p of the process but we usually drop it if p is obvious from the context (e.g., if p is the current process). Superscripts have no semantic meaning, they are only used to refer to a particular occurrence of the name.

[2]For a formal treatment of predicates with side-effects see the notion of "action" in TLA+ [15].

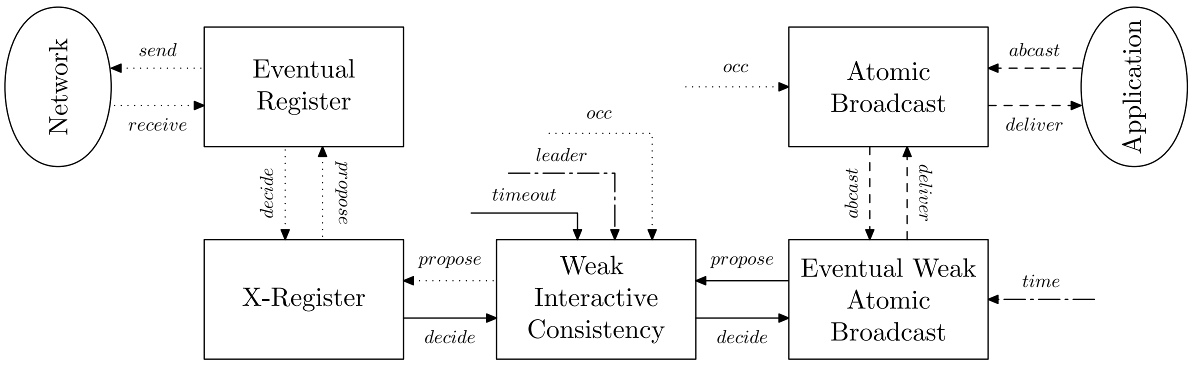


Figure 2: General structure of the algorithm.

Each object has a special **init** action, which is executed only once and before all other actions. Its *guard* and *body* fields specify the types of channels used by the object and initialize its local variables. For each parameter of the *init* channel, a local constant with the same name and value is created. In our example, both *propose* and *decide* are idempotent channels, and the *leader* is a constant with the value from the *init* channel.

4 Algorithm

Our algorithm is constructed in a modular way shown in Figure 2. Arrows represent variables (—·—▸) and channels: idempotent (——▸), regular (⋯⋯▸), and regular FIFO (- - -▸). The main effort is put into *Eventually Weak Atomic Broadcast*, which is implemented using a sequence of *Eventually Weak Interactive Consistency* objects. These two new abstractions are "eventually weak" because some of their properties hold only for the eventual leader. X-Register is a variant of the $\Diamond$Register from [2, 3] which is particularly efficient for writing a special symbol X to it.

4.1 Eventual Register

The Eventual Register is a shared "write-and-lock" register, which is initially undefined, but once a write operation has succeeded, the contents of the register cannot be changed any more. It is a slightly modified version of $\Diamond$Register [2, 3] so that it has the *owner* – a process whose writes are particularly fast.

The register can be accessed through two regular channels: *propose* and *decide*. To propose (write) a value v, the caller invokes *propose*(v). When a value w has been successfully written, the register invokes *decide*(w).

We say that a process has *proposed* or *decided* v if it has invoked *propose*(v) or *decide*(v), respectively. All our abstractions assume that no process proposes two different values. In exchange, they guarantee that no two processes will decide on different values. Formally, the Eventual Register satisfies the following properties:

Validity If $decide_p(v)$ has been invoked, then $propose_q(v)$ has been invoked for some q.

Agreement If both $decide_p(v)$ and $decide_q(w)$ have been invoked, then $v = w$.

Action	Guard	Body
init	$init(owner)$, regular *propose*, idemp *decide*	$ereg.init(owner)$
propose	$propose(v)$ and $p = owner \vee v = X$	$ereg.propose(v)$
decide	$ereg.decide(v)$	$decide^1(v)$
fast-decide	$propose_{owner}(X)$	$decide^2(X)$

Figure 3: X-Register: code for process p.

Termination If eventually a *single* correct process p keeps invoking $propose_p$ infinitely many times, then eventually $decide_q$ will be invoked at all correct processes q.

Performance If the owner invokes $propose(v)$ at time t and no other process ever invokes *propose*, then all processes will execute *decide* by time $t + 2$.

Validity ensures that the value stored in the register could not have appeared there out of the blue; it must have been proposed by some process. Agreement states that the contents of the register perceived by different processes is the same. Termination says that if a *single* correct process (e.g., the leader) keeps writing, it will eventually succeed.

Validity, Agreement, and Termination are the same as in $\Diamond$Register [2, 3]. The Performance property is technically new, but the implementation of $\Diamond$Register in Figure 5 in [2] can be easily changed to accommodate that requirement. There, without loss of generality we can assume that p_1 is the owner of the register. Then, the first invocation of $propose(v)$ by p_1 corresponds to the first round of Paxos. Thus, as shown in FastPaxos [1] and $\Diamond$RegisterII [2], the *read* phase of this round can be omitted. This is because the purpose of the *read* phase is to ensure that no value different than the proposed one has been written by one of the previous rounds. Since for the first round no previous rounds exist, omitting the *read* phase is safe.

4.2 X-Register

The X-Register is a version of the Eventual Register in which the special symbol X is the only value can be proposed by processes other than the owner. As a result, a process can decide on X as soon as it has learnt that the owner has invoked $propose(X)$. This is safe because as no value other than X could have been proposed, Validity implies that X is the only possible decision value.

The implementation of X-Register shown in Figure 3 uses an Eventual Register object *ereg*, a regular channel *propose*, and an idempotent *decide*. The guard of **propose** requires $v = X$ or the caller to be the owner. As a consequence, $decide^2(X)$ can be invoked as soon as p knows that the owner has invoked $propose(X)$, without violating Agreement. Thus, the following additional performance property holds:

X-Performance. If the owner invokes $propose(X)$ at time t, then all processes will have invoked $decide(X)$ by time $t + 1$.

The Validity, Termination, and Performance properties of an Eventual Register trivially hold for an X-Register. Moreover, as Figure 3 shows, if the (correct) owner proposes X, the decision X will be reached by **fast-decide** without using *ereg*. Therefore, to reduce network usage, **propose** should not be executed if $p = owner$ and $v = X$.

Action	Guard	Body
init	*init*, idemp *propose*, *decide*, *timeout*	$xreg[i].init(p_i)$ for all p_i
propose	*once* and $p = p_i$ and $propose(v)$	$xreg[i].propose^1(v)$
leader	*occ* and $p = leader = p_i$ and $propose(v)$	$xreg[i].propose^2(v)$
timeout	*occ* and $p = leader \neq p_i$ and *timeout*	$xreg[i].propose^3(X)$
decide	$xreg[i].decide(v_i)$ for all p_i	$decide([v_1, \ldots, v_n])$

Figure 4: Eventually Weak Interactive Consistency: code for process p.

4.3 Eventually Weak Interactive Consistency

The Eventually Weak Interactive Consistency (EWIC) abstraction is based on Interactive Consistency [18]. There, every process i invokes $propose(v_i)$ and $decide([w_1, w_2, \ldots, w_n])$ with the following properties: (i) $w_i \in \{v_i, X\}$ for all p_i, and (ii) $w_i = v_i$ for all correct p_i. In the EWIC abstraction, the latter condition is satisfied only in stable runs or if the Ω failure detector has stabilized with p_i.

The implementation shown in Figure 4 uses one X-Register $xreg[i]$ for each process p_i, the variable *leader* (which is the output of Ω), and three idempotent channels: *propose*, *decide*, and *timeout*. Process p_i proposes v by invoking $xreg[i].propose(v)$. The final decision vector is composed from the decisions made by individual X-Registers.

To ensure termination even in runs with failures, we employ two mechanisms. Firstly, the leader periodically repeats its proposal (**leader**). Secondly, if its *timeout* has ticked, the leader starts to keep proposing X to X-Registers of other processes (**timeout**). Intuitively, *timeout* ticks when the decision would have been made if the run was stable. In a typical implementation, *timeout* ticks Δ units of time after *propose* was invoked, for some $\Delta > 0$. We assume that Δ is finite, so that if a process has proposed, its *timeout* will eventually tick.

If process p_i has invoked $propose(v)$, then only v and X can be proposed to $xreg[i]$. Note that $xreg[i].propose^3(X)$ can be invoked only if *timeout* has ticked and p_i is not the leader, otherwise v is the only possible decision. The decision will eventually be reached because the eventual leader will invoke $xreg[i].propose^2$ or $xreg[i].propose^3$ infinitely many times and $xreg[i].propose^1$ will be invoked at most once.

Formally, Eventually Weak Interactive Consistency has the following properties:

Validity Invoking $decide_q([v_1, \ldots, v_n])$ implies that $propose(v_i)$ must have been invoked by process p_i if (i) $v_i \neq X$ or (ii) before *timeout* ticked at any process either: (a) $decide_q([v_1, \ldots, v_n])$ was invoked, or (b) Ω stabilized with p_i.

Agreement If $decide_p([v_1, \ldots, v_n])$ and $decide_q([w_1, \ldots, w_n])$ have been invoked, then $[v_1, \ldots, v_n] = [w_1, \ldots, w_n]$.

Termination If the eventual leader has proposed, then every correct process will eventually decide.

Performance If (i) all processes proposed by $t+1$, and (ii) all processes p_i with proposals $v_i \neq X$ proposed by t, then by time $t + 2$ (i) all processes decided, or (ii) *timeout* ticked at at least one process.

Action	Guard	Body
init	*init*, FIFO *abcast*, FIFO *deliver*	$ewic[t].init$ for all $t \in \mathcal{T}$; $list \leftarrow X$
abcast	$abcast(m)$	add m to *list*
propose	$times(t)$ and $time \geq t$	$ewic[t].propose(list)$; $list \leftarrow X$
deliver	$times(t)$ and $ewic[t].decide([v_1, \ldots, v_n])$	$deliver(m)$ for all $m \in v_1, v_2, \ldots, v_n$

Figure 5: Simple Eventual Weak Atomic Broadcast: code for process p.

4.4 Eventually Weak Atomic Broadcast

Eventually Weak Atomic Broadcast (EWAB) is a version of Atomic Broadcast, where in non-stable runs only messages sent by the eventual leader are guaranteed to be delivered.

Let $\mathcal{T}$ be a subset set of all possible readings of the real-time clock *time*, for example, $\mathcal{T} = \{0, \delta, 2\delta, 3\delta, \ldots\}$ for some small $\delta > 0$. The implementation shown in Figure 5 uses two regular FIFO channels *abcast* and *deliver*. It also uses two copies of an extract-only regular FIFO channel *times*, which contains all elements of $\mathcal{T}$ in increasing order. For each $t \in \mathcal{T}$, we use a separate EWIC object $ewic[t]$.

Every message to be broadcast is added to the initially empty variable *list*. (We assume that X is equivalent to the empty set $\emptyset$.) When the current time *time* passes the time t extracted from *times*, the current value of *list* is proposed using $ewic[t]$ and emptied. The decisions of $ewic[t]$ for successive t's become the delivered messages[3]. In stable runs, all processes propose at the same time, so taking any $\Delta > 2$ for the *timeout* implementation in EWIC will guarantee that all messages will be delivered.

The algorithm in Figure 5 satisfies Uniform Agreement, Uniform Validity, and Uniform Total Order from Section 2, and

Termination Eventually, any message broadcast by the eventual leader will eventually be delivered by every correct process.

Performance A message broadcast at time t will be delivered by $t + 2 + \delta$.

The Performance property states that the algorithm achieves the same delivery latency of $2 + \delta$ as the one in [19].

4.4.1 Quiet version

The implementation of EWAB from Figure 5 uses one EWIC object per element of $\mathcal{T}$. This results not only in memory waste but also in sending network messages even if no *abcast* is ever invoked. In this section, we present a variant of EWAB, in which the number of EWIC object depends on the number of broadcast messages, not on the size of $\mathcal{T}$. As a result, we can assume that $\mathcal{T} = \mathbb{R}$ and thus achieve the delivery latency of 2.

The implementation in Figure 6 uses two new variables: *lastp* and *lastd* with the following properties: $ewic[t].propose$ has been invoked iff $t \leq lastp$, and all messages $m \in v_1, \ldots, v_n$ where $ewic[t].decide([v_1, \ldots, v_n])$ have been delivered iff $t \leq lastd$.

Splitting **propose** into **active** and **passive** defers invoking $ewic[t].propose$ until necessary. Now, it is invoked only if the current process has something to send (**active**) or it

[3]we assume that **deliver** always delivers the messages from $[v_1, \ldots, v_n]$ in the same order.

Action	Guard	Body
init	*init*, FIFO *abcast*, FIFO *deliver*, idemp *active*	$ewic[t].init$ for all $t \in \mathcal{T}$; $list \leftarrow X$; $lastp \leftarrow lastd \leftarrow -\infty$
abcast	$abcast(m)$	add m to *list*
active	$list \neq X$ and $t > lastp$ and $t = time$	$ewic[s].propose(X)$ for $s \in (lastp, t)$ $ewic[t].propose(list)$; $list \leftarrow X$ $lastp \leftarrow t$ $active(t)$
passive	$list = X$ and $t > lastp$ and $active_q(t)$	$ewic[s].propose(X)$ for $s \in (lastp, t)$ $ewic[t].propose(X)$ $lastp \leftarrow t$
deliver	$ewic[s].decide([X, \ldots, X])$ for all $s \in (lastd, t)$, $ewic[t].decide([v_1, \ldots, v_n])$ and some $v_i \neq X$	$deliver(m)$ for all $m \in v_1, v_2, \ldots, v_n$ $lastd \leftarrow t$

Figure 6: Quiet Eventual Weak Atomic Broadcast: code for process p.

has already been invoked by another process (**passive**). The idempotent channel *active* is used to inform other processes that time t is *active*, that is, there is a process that has invoked $ewic[t].propose(v \neq X)$. Note that since **active** clears *list*, the number A of active times cannot exceed the number M of abcast messages.

Consider a stable run in which some processes execute **active** at time $T = t$, which invokes $ewic[t].propose$. By $T+1$, all other processes will have invoked $ewic[t].propose(X)$ by executing **active** or **passive** with $t \geq T$. Performance of $ewic[t]$ implies that, by time $T+2$, all processes will have invoked $ewic[t].decide$ and thus delivered the messages sent at T (see the Appendix for the proofs).

To reduce the number of EWIC objects, notice that $ewic[t].propose$ is invoked only for an active t or for all $t \in (s, s')$ where both s and s' are active. As a result, if s and s' are consecutive active times, all $ewic[t]$ objects with $t \in (s, s')$ can be simulated by one object denoted by $ewic(s, s')$. Let $t_1, \ldots, t_A$ be the ordered sequence of all active times. Then, the algorithm needs only $2A+1$ EWIC objects corresponding to the following intervals: $(-\infty, t_1)$, $[t_1]$, (t_1, t_2), $[t_2]$, $\ldots$, $[t_A]$, (t_A, ∞), where $[t_i] = \{t_i\}$.

Since the values of $t_1, \ldots, t_A$ are not known a priori, we have to use a *divide and clone* technique. Each process starts with one object $ewic(-\infty, \infty)$, which will be divided into smaller intervals as requests arrive. At any time, each process maintains a list $\mathcal{I}$ of intervals of the form (s_1, s_2), $[s_2]$, (s_2, s_3), $[s_3]$, $\ldots$, $[s_{k-1}]$, (s_{k-1}, s_k) with $s_1 = -\infty$ and $s_k = \infty$. If a request for all $ewic[t]$ with $t \in J$ arrives, some intervals in $\mathcal{I}$ are split so that every member of $\mathcal{I}$ will be either a subinterval of J or disjoint with it. When an interval is being split, the local state of the corresponding EWIC object is cloned. Finally, the request is applied to all elements of $\mathcal{I}$ that are subintervals of J. Note that the interval lists $\mathcal{I}$ at different processes do not need to be the same.

As an example, assume that $\mathcal{I} = (-\infty, 4), [4], (4, \infty)$ and $J = (3, 7)$. In this case, $\mathcal{I}$ becomes $(-\infty, 3)$, $[3]$, $(3, 4)$, $[4]$, $(4, 7)$, $[7]$, $(7, \infty)$. Object $ewic(-\infty, 4)$ is replaced by $ewic(-\infty, 3)$, $ewic[3]$, and $ewic(3, 4)$, all with the same local state. Similarly, $ewic(4, \infty)$ is transformed into $ewic(4, 7)$, $ewic[7]$, and $ewic(7, -\infty)$. The request is applied to

Action	Guard	Body
init	*init*, FIFO *abcast*, FIFO *deliver*, idemp *active*	$ewic[t].init$ for all $t \in \mathcal{T}$; $list \leftarrow X$; $lastp \leftarrow lastd \leftarrow -\infty$
abcast	$abcast(m)$	add m to *list*
active	$list \neq X$ and $t > lastp$ and $t = time$	$ewic[s].propose(X)$ for $s \in (lastp, t)$ $ewic[t].propose(list)$; $list \leftarrow X$ $lastp \leftarrow t$ $active(t)$
passive	$list = X$ and $t > lastp$ and $active(t)$	$ewic[s].propose(X)$ for $s \in (lastp, t)$ $ewic[t].propose(X)$ $lastp \leftarrow t$
relay	$active_q(t)$	$active(t)$
decide	$ewic[t].decide([v_1, \dots, v_n])$ and $v_i \neq X$	$active(t)$
deliver	$ewic[s].decide([X, \dots, X])$ for all $s \in (lastd, t)$, $ewic[t].decide([v_1, \dots, v_n])$ with some $v_i \neq X$, and $active_q(t)$ for a majority of processes q	$deliver(m)$ for all $m \in v_1, v_2, \dots, v_n$ $lastd \leftarrow t$

Figure 7: Pingless Quiet Eventual Weak Atomic Broadcast: code for process p.

$ewic(3, 4)$, $ewic[4]$, and $ewic(4, 7)$.

In order to achieve Uniform Agreement, the algorithm assumes that one of the correct processes atomically broadcasts infinitely many messages: occasional pings, for example. In comparison to the algorithm from Figure 5, the one in Figure 6 has a smaller delivery latency in stable runs. Moreover, since $ewic[s].propose$ is invoked only if there is an active $t \geq s$, the algorithm is also quiet:

Quietness If no process invokes $abcast(m)$, then no network messages are sent.

Performance A message broadcast at time t will be delivered by $t + 2$.

4.4.2 Pingless quiet version

The EWAB from Figure 6 requires at least one correct process to broadcast infinitely many times in order to achieve Uniform Agreement. In this section, we show how to waive this requirement.

Figure 7 shows the improved version. Additional actions **relay** and **decide** ensure that $active(t)$ is invoked if another process has invoked it or $ewic[t]$ has decided on $[v_1, \dots, v_n] \neq [X, \dots, X]$. Action **deliver** can be now executed executed only if a majority of processes has invoked $active(t)$. This ensures that at least one correct process have invoked $active(t)$, and eventually all correct processes will do so.

Consider a stable run. If time t is active, then at least one process invokes $active(t)$ at time t, which implies that all processes will invoke $active(t)$ in **relay** by time $t + 1$. Therefore, by $t + 2$ the $active_q(t)$ condition in the guard of **deliver** will hold, so all messages sent at t will be delivered by $t + 2$. Moreover, since all processes invoke $active(t)$ by time $t + 1$, the invocation of $active(t)$ in **decide**, which is usually executed at $t + 2$, does not send any messages.

Action	Guard	Body
init	*init*, FIFO *abcast*, FIFO *deliver*	*ewab.init*; $\quad$ *blist* $\leftarrow \emptyset$
abcast	*abcast*(m)	*ewab.abcast*$(\{m\})$
store	*abcast*$_q(m)$	add m to *blist*
liveness	*occ* and *blist* $\neq \emptyset$	*ewab.abcast*(*blist*)
deliver	*ewab.deliver*(v)	*deliver*(m) for every $m \in v$

Figure 8: Atomic Broadcast: code for process p.

4.5 Atomic Broadcast

Figure 8 shows how to use EWAB from Section 4.4 to implement standard Atomic Broadcast, in which a message sent by *any* correct process is eventually delivered. Whenever a correct process p atomically broadcasts m, it just invokes *ewab.abcast*$(\{m\})$, which is sufficient in stable runs. To guarantee eventual delivery of m in any run, every process keeps a sequence *blist* of all messages broadcast in the system and periodically invokes *ewab.abcast*(*blist*). Action **store** ensures that m will eventually belong to *blist*$_l$ at the eventual leader l. Since l periodically broadcasts its *blist* (action **liveness**), the Termination property of EWAB implies that m will eventually be delivered. To avoid delivering the same message twice we assume that the *deliver* channel automatically eliminates duplicates. In addition to all properties of EWAB, this algorithm satisfies the Termination property from Section 2:

Termination If a correct process broadcasts a message m, then every correct process will eventually deliver m.

4.6 Optimizations

A number of standard optimization techniques can be applied to reduce the message and memory complexity of our algorithm. First, when a decision is reached, all propose-and-decide objects should broadcast it to all participants, destroy all their subobjects, and stop operating. Similarly, once **deliver** in EWAB has been executed for a given t, all *ewic*$[s]$ for $s \leq t$ can be destroyed (provided that all delivered messages have been FIFO-broadcast to other processes). In stable runs, this will limit the number of EWIC objects by the number of messages broadcast in the last two communication steps. Finally, in Figure 8, delivered messages can be removed from *blist* (duplicates can still be detected using sequence numbers).

To maximize the performance, the ticking frequency of various *occ* channels can be (adaptively) adjusted to the current conditions. Also, executing *ewic*$[t]$.**timeout** if $p_i \neq$ *leader* was suspected to have crashed at any time after $t - \Delta$ (for some Δ) will limit the influence of crashed process on the performance.

5 Optimality results

The algorithm presented in Section 4 achieves the two-communication-step lower bound for Uniform Consensus in failure-free runs [5, 13]. Since Atomic Broadcast can be solved using Consensus with no additional message delay [4], the same lower bound applies to Atomic Broadcast. In this section, we will prove three new lower bounds for Atomic Broadcast that are achieved by our algorithm.

Firstly, in order to guarantee atomic delivery in two steps, our algorithm assumes that all processes are correct. This condition cannot be weakened, even if we assume synchronous settings and consider only scenarios in which one, non-leader process can fail. This should be contrasted with three-step Atomic Broadcast protocols such as [4,14], which require only a majority of correct processes.

Secondly, even if only one of the non-leader processes crashes, our algorithm would not be live in purely asynchronous settings (recall that we use a *timeout* failure detector, which is accurate in stable runs and always complete). We have shown that no two-communication-step Atomic Broadcast protocol can guarantee liveness in such circumstances. This bound does not hold for synchronous settings, where a perfectly complete and accurate timeout mechanism can be implemented.

Finally, two-step delivery latency cannot be achieved without perfectly synchronized clocks. Any asynchronous Atomic Broadcast algorithm that does not use physical time must base the delivery order on causal dependencies only. We have shown that this implies that there are stable runs where message delivery takes $3 - \varepsilon$ time for any $\varepsilon > 0$.

5.1 Correct leader

In many Atomic Broadcast and Consensus protocols (esp. asynchronous ones), a single distinguished process called the *leader* or coordinator plays an important role [4,14]. As a result, a failure this process has much more severe consequences for the performance of the algorithm than a failure of any other process. In this section, we will consider Atomic Broadcast algorithms that achieve a two-step delivery latency in failure-free runs. We will investigate their performance in runs where at most one non-leader process fails.

We define Exclusive Consensus as a variant of Uniform Consensus in which the leader does not propose. Note that as long as there is at least one correct non-leader process, Exclusive Consensus can be easily solved using Atomic Broadcast with no additional message delay; all processes except for the leader atomically broadcast their proposals and the first delivered message is adopted as the decision. Therefore, all latency bounds on Exclusive Consensus also hold for Atomic Broadcast.

5.1.1 Synchronous settings

In this section, we will investigate Exclusive Consensus in synchronous settings. Consider an algorithm that decides in two steps in all scenarios in which (i) the leader does not crash, and (ii) at most one process crashes. We will show that such an algorithm does not exist.

Consider a system consisting of n processes $1, \dots, n$. We use the layering technique and notation from [12]. Action $(i, [k])$, where $1 \leq i \leq n$ and $0 \leq k \leq n$, defines a round in which process i permanently fails and all messages sent by i to processes $1, \dots, k$ are lost. Action $(0, [0])$ models a round without failures. For a given (global) state x, we define $L'(x)$ to be the set of all states obtained from x by applying one action of the form

$(i, [k])$, where process 1 (the leader) does not crash. Formally,

$$L'(x) = \{\, x \cdot (i, [k]) : i \neq 1 \text{ and } (i, [k]) \text{ is applicable to } x \,\}.$$

Lemma 5.1. *Let Init be the set of possible initial states. Then, $L'(Init)$ is similarity-connected.*

Proof. Set *Init* can be easily proved to be similarity-connected [13]. To prove the assertion, we can use the proof of Lemma 2.3 from [12] provided that we do not consider states in which the leader has crashed. The first part of that proof, which shows that $L'(x)$ is similarity-connected for any $x \in Init$, works without modifications. Consider the second part, which proves that that for any $x \sim x'$ (for $x \neq x$) there are similar states $y \in L'(x)$ and $y' \in L'(x')$. Here, no modifications are necessary either because x and x' cannot differ in the state of the leader. This is because the leader has no state at the beginning (it does not propose anything). □

Since $L'(Init)$ is similarity-connected (Lemma 5.1), Lemma 2.2 from [12] implies that there is at least one run in which the algorithm does not decide in two communication steps.

5.1.2 Asynchronous settings

In asynchronous settings with correct-restricted reliable channels [10], the result from Section 5.1.1 can be strengthened. Consider any Exclusive Consensus algorithm that decides in two steps in failure free runs. We will show that there is a scenario with only one (non-leader) process failure, in which the algorithm never decides. This result should be contrasted with three-step Atomic Broadcast protocols such as [4, 14]. There, each message is first broadcast to the leader, who can atomically broadcast it if at least half of the processes are correct.

Definition 5.2. *A* run r *is an execution of the algorithm with a given* latency function L_r. *The value of $L_r(i, j, t) > 0$ specifies the latency of the message sent by process i to process j at time $t \geq 0$. (Infinite latency means that the message will never reach its destination.)*

To precisely define “two communication rounds” in asynchronous settings we use the notion of *communication level.*

Definition 5.3. *The* communication level *of process i at time t in run r is a non-negative integer denoted by $C_r(i, t)$. All processes start at level 0, and progress to the next level once they could have received current-level messages from all processes. In other words, $C_r(i, t) \geq 0$ and*

$$C_r(i, t) \geq k + 1 \overset{\text{def}}{\iff} \forall j : \exists s : C_r(j, s) \geq k \wedge L_r(j, i, s) + s \leq t \quad \text{for all } k \geq 0. \tag{1}$$

The communication level $C_r(i, t)$ is defined as the smallest number allowed by (1).

Note that the communication level is defined in terms of messages that *could* be sent. For example, if $L_r(i,j,t) \equiv 1$, then at time 1 all processes will have the communication level of 1, even if no messages are actually sent. This way, algorithms cannot keep the communication level artificially low by not sending messages.

For the rest of the section assume that A is an Exclusive Consensus algorithm with the following properties. Firstly, all processes decide as soon as they reach the communication level of 2. Secondly, if the leader does not crash, all correct processes will eventually decide. Finally, if a decision has been made by any process, then a global observer will always be able to deduce its value from local states of all *correct* processes[4]. (It does not have to be able to determine *whether* a decision has been made or not.) This assumption implies the following lemma.

Lemma 5.4. *Let r and r' be runs, in which process i decides. If r and r' are indistinguishable to processes other than i, then the decision in both runs must be the same.*

Definition 5.5. *A run r* dominates *run r', which we denote as $r \succeq r'$, if all message latencies in r are not higher than those in r'. Formally,*

$$r \succeq r' \overset{\text{def}}{\iff} \forall\, i,j,t : L_r(i,j,t) \leq L_{r'}(i,j,t)$$

Lemma 5.6. *If $r \succeq r'$, then $C_{r'}(i,t) \geq k \implies C_r(i,t) \geq k$ for all i, t, and k.*

Proof. By induction on k. Since $C_r(i,t) \geq 0$, the base case is obvious. Assuming the inequality holds for k, we will prove it for $k+1$:

$$\begin{aligned} C_{r'}(i,t) \geq k+1 \implies \forall\, j : \exists\, s : C_{r'}(j,s) \geq k \wedge L_{r'}(j,i,s) + s \leq t \implies \\ \forall\, j : \exists\, s : C_r(j,s) \geq k \wedge L_r(j,i,s) + s \leq t \implies C_r(i,t) \geq k+1. \quad \square \end{aligned}$$

Corollary 5.7. *If $r \succeq r'$, then $C_r(i,t) \geq C_{r'}(i,t)$ for all i and t.*

Definition 5.8. *For any run r, let $r' = r \oplus \mathsf{lost}(j,t)$ be a run identical to r except that all messages sent by process j to* other *processes after or at time t are lost. Formally $L_{r'} = L_r$ except that $L_{r'}(j,k,s) = \infty$ for $j \neq k$ and $s \geq t$.*

Lemma 5.9. *Let d be the lower bound for message latency. For any run r and process i, run $r \oplus \mathsf{lost}(i,t)$ is indistinguishable from r to i before time $t + 2d$.*

Proof. Run r and $r \oplus \mathsf{lost}(i,t)$ are identical before time t. Since any message takes at least d time to reach its destination, these runs are indistinguishable to processes other than i before time $t+d$. For the same reason, r and $r \oplus \mathsf{lost}(i,t)$ are indistinguishable to i before $t + 2d$. $\square$

[4]Without this assumption the following algorithm would work. All processes send their proposals to the leader, which decides on the first proposal it has received, and broadcasts it. Unfortunately, if the leader crashes, even a global observer cannot ensure Uniform Agreement and Termination.

Consider the following family of runs. For any process i, let $R_i(j,t)$ be a run in which all messages travel two units of time except for those sent by process i to *other* processes. These require $t \geq 2$ units to reach processes $k > j$ and $t+1$ to reach processes $k \leq j$. In other words,

$$L_{R_i(j,t)}(i',j',t') = \begin{cases} 2 & \text{if } i' \neq i \text{ or } i = j \\ t & \text{if } i' = i \neq j \text{ and } j' > j, \\ t+1 & \text{if } i' = i \neq j \text{ and } j' \leq j. \end{cases}$$

Lemma 5.10. *Consider $R_i(n,t) \oplus \mathsf{lost}(j,1)$ for $i \neq j$. Then, the following holds:*

1. $C(i,2) \geq 1$,
2. $C(k,t+1) \geq 1$ *for all processes* k,
3. $C(j,t+3) \geq 2$.

Proof. 1. Follows from the fact that $L(l,i,0) = 2$ for all l.

2. Follows from the fact that $L(l,k,0) \leq t+1$ for all l.

3. Follows from

   (a) $C(i,2) \geq 1$ and $L(i,j,2) \leq t+1$, and

   (b) $C(k,t+1) \geq 1$ and $L(k,j,2) = 2$ for all $k \neq i$. □

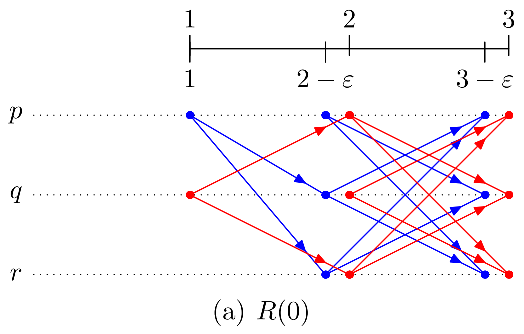
Theorem 5.11. *The algorithm for Exclusive Consensus with the desired properties does not exist.*

Proof. For sake of contradiction assume that the algorithm exists. Then, if i is not the leader, there will be a decision in $R_i(j,t)$ for any j and t. We will first show that that decisions in $R_i(j-1,t)$ and $R_i(j,t)$ are identical.

If $j = i$, then $R_i(j-1,t) = R_i(j,t)$, so we will assume $i \neq j$. Since $R_i(j,t) \succeq R_i(n,t) \oplus \mathsf{lost}(j,1)$, Lemma 5.10 implies that j will decide by $t+3$ on some value, say d. Runs $R_i(j,t)$ and $R_i(j,t) \oplus \mathsf{lost}(j,t)$ are indistinguishable to j until $t+4$ (Lemma 5.9), so j will decide by $t+3$ on d in $R_i(j,t) \oplus \mathsf{lost}(j,t)$ too. By the same argument, in runs $R_i(j-1)$ and $R_i(j-1,t) \oplus \mathsf{lost}(j,t)$, process j will reach will reach same decision d' by time $t+3$. Now, runs $R_i(j,t) \oplus \mathsf{lost}(j,t)$ and $R_i(j-1,t) \oplus \mathsf{lost}(j,t)$ are indistinguishable to processes other than i, so Lemma 5.4 implies $d = d'$.

It is now easy to prove by induction on j that the decisions $R_i(0,t)$ and $R_i(n,t) = R_i(0,t+1)$ will be the same. By induction on t, we conclude that the decision in $R_i(j,t)$ does not depend on j or t.

Let R be $R_i(0,2)$, which is the same for all i. Let every process propose a different value and let i be the process whose proposal was adopted as the decision in $R = R_i(0,2)$. Since i is not the leader (who cannot propose anything), a decision will eventually be made even in run $R_i(0,2) \oplus \mathsf{lost}(i,0)$, in which i does not send any messages. Assume that the leader decided at time t. Notice two facts: (i) $R_i(0,2) \oplus \mathsf{lost}(i,0)$ and $R_i(0,t+1)$ are indistinguishable to the leader before time $t+1$, and (ii) the decision in $R_i(0,t+1)$



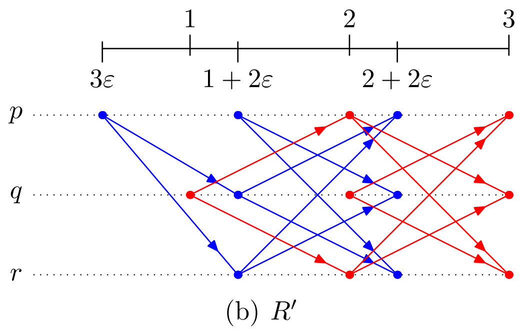


Figure 9: Runs $R(0)$ and R'

and $R_i(0,2)$ is the same. Therefore, the decision in $R_i(0,2) \oplus \mathsf{lost}(i,0)$ is the same as in $R = R_i(0,2)$. On the other hand, since in $R_i(0,2) \oplus \mathsf{lost}(i,0)$ all messages from i are lost, the leader has no way of knowing i's proposal, which is the decision in R. This is a contradiction. □

5.2 Physical clocks

Consider *stable runs* in which no process fails and all network messages are delivered within one unit of time. Assume that there is an Atomic Broadcast algorithm that does not use physical clocks yet in stable runs atomically delivers all messages within $M = 3-6\varepsilon$ units of time. We will show that such an assumption leads to a contradiction.

In our model, each message carries the complete state of the sender. We assume that processes can send messages only if their state changes because of an external action (such as a reception of a network message or an atomic broadcast request). This does not restrict the generality of our proof because all internal actions can be inferred by other processes, which know the complete state of the sender.

Consider a simple latency function L, in which all messages sent no later than 2 take 1 unit of time to reach their destinations, and all the others are lost. Formally,

$$L(i,j,t) = \begin{cases} 1 & \text{if } t \le 2, \\ \infty & \text{if } t > 2. \end{cases}$$

Consider a family of runs $R(k)$, in which processes p and q broadcast two messages m_p and m_q, respectively, at time 1, and no other process broadcasts anything. The latency function $L_{R(k)}$ is the same as L, except for all $t \le 1$ we have

$$L_{R(k)}(i,j,t) = \begin{cases} 1-\varepsilon & \text{for } i = p \text{ and } j > k, \\ 1-\varepsilon & \text{for } i = q \text{ and } j \le k. \end{cases}$$

We will prove that, in $R(k)$ and $R(k-1)$, messages m_p and m_q are delivered in the same order . Lemma 5.9 implies that $R(k)$ and $R(k) \oplus \mathsf{lost}(k, 2-\varepsilon)$ are indistinguishable to k until $4-3\varepsilon > 1+M$. Therefore, in both runs, k delivers the same message m first. Similarly, k delivers the same message m' first in both $R(k-1)$ and $R(k-1) \oplus \mathsf{lost}(k, 2-\varepsilon)$. Since $R(k) \oplus \mathsf{lost}(k, 2-\varepsilon)$ and $R(k-1) \oplus \mathsf{lost}(k, 2-\varepsilon)$, are indistinguishable for processes other than k, Lemma 5.4 implies that $m = m'$.

It follows that the same message is delivered first in $R(0)$ and $R(n)$. Since those runs are symmetrical with respect to p and q, without loss of generality we may assume that m_q is delivered first in all those runs. Consider a run R' with the same latency function as $R(0)$, in which p send m_p at 3ε instead of 1. Note that processes do not have access to physical time clocks and all causal dependencies in R' and $R(0)$ are the same (see Figure 9), so the same message (m_q) is delivered first in both runs. However, m_q cannot be delivered faster than in two communication steps [5, 13], that is, before $3 - 2\varepsilon$. Since m_p is delivered after m_q, its delivery latency will be at least $3 - 5\varepsilon > M$. Finally, note that to no process is R' distinguishable from a stable run before time 3, which proves the assertion.

Since the above argument works for any $\varepsilon > 0$, it follows that no protocol can guarantee a delivery latency smaller than three communication steps unless it uses physical clocks.

6 Conclusion

In this paper, we introduced several new distributed agreement abstractions and showed how to use them to implement Atomic Broadcast efficiently. Our implementation uses Eventually Weak Atomic Broadcast, which is based on a sequence of Eventually Weak Interactive Consistency objects, which, in turn, use one X-Register per process.

Processes use their own X-Registers to propose messages to be atomically broadcast. These registers decide in two communication steps. One step after the messages are broadcast, all other processes learn of this and propose a special symbol X. Recall that for this symbol, X-Registers take only one step to decide. Therefore, all X-Registers will decide two steps after the broadcast, and the messages will be delivered then.

This algorithm is quiet and achieves the optimum [5, 13] delivery latency of two communication steps. It also achieves three new lower bounds presented in this paper. In the future, we plan to design other two-step Atomic Broadcast algorithms which gracefully handle non-leader process failures when no two processes broadcast at the same time. This assumption allows one to circumvent the second impossibility result from Section 5. Finally, we would like to formalize our notation as a TLA+ module [15].

References

[1] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Deconstructing Paxos. Technical Report 200232, École Polytechnique Fédérale de Lausanne, January 2001.

[2] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Deconstructing paxos. *ACM SIGACT News*, 34(1):47–67, 2003.

[3] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Reconstructing Paxos. *ACM SIGACT News*, 34, 2003.

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

[5] Bernadette Charron-Bost and André Schiper. Uniform Consensus is harder than Consensus. Technical Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[6] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, Ann Arbor, MI, USA, 1985. IEEE Computer Society Press.

[7] Xavier Défago, André Schiper, and Péter Urbán. Totally ordered broadcast and multicast algorithms: A comprehensive survey. Technical Report DSC/2000/036, École Polytechnique Fédérale de Lausanne, Switzerland, September 2000.

[8] Paul Ezhilchelvan, Doug Palmer, and Michel Raynal. An optimal atomic broadcast protocol and an implementation framework. In *Proceedings of the Eithth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, pages 32–41, January 2003.

[9] André Schiper Fernando Pedone. Optimistic atomic broadcast. In *Proceedings of the* 12^{th} *International Symposium on Distributed Computing (DISC'98, formerly WDAG)*, September 1998.

[10] Rachid Guerraoui, Riucarlos Oliveira, and André Schiper. Stubborn communication channels. Technical Report 98/272, Ecole Polytechnique Federale Lausanne, Switzerland, 1998.

[11] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcast and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. ACM Press, New York, 2nd edition, 1993.

[12] I. Keidar and S. Rajsbaum. A simple proof of the uniform consensus synchronous lower bound, 2002.

[13] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant Consensus when there are no faults. *ACM SIGACT News*, 32, 2001.

[14] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[15] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Professional, 2002.

[16] Butler Lampson. The ABCD of Paxos. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing.* ACM Press, 2001.

[17] Achour Mostefaoui and Michel Raynal. Low cost Consensus-based atomic broadcast. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing.* IEEE Computer Society, 2000.

[18] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. In C. J. Walter, M. M. Hugue, and Neeraj Suri, editors, *Advances in Ultra-Dependable Distributed Systems*. IEEE Computer Society, January 1995.

[19] Pedro Vicente and Luís Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of 21st Symposium on Reliable Distributed Systems (SRDS 2002), 13-16 October 2002, Osaka, Japan*. IEEE Computer Society, 2002.

A Proofs

A.2 X-Register

Theorem A.1 (Agreement). *If both $decide_p(v)$ and $decide_q(w)$ have been invoked, then $v = w$.*

Proof. Without loss of generality we can assume $v \neq X$, so process p has invoked $decide^1(v)$. Therefore, the owner must have invoked $propose(v \neq X)$, so $decide^2_q$ has not been invoked. Therefore, q has invoked $decide^1_q(w)$, in which case the assertion follows from the Agreement property of *ereg*. □

Theorem A.2 (X-Performance). *If the owner invokes $propose(X)$ at time t, then all processes will have invoked $decide(X)$ by time $t + 1$.*

Proof. The assumption implies that all processes will have **get** $propose_{owner}(X)$ by time $t + 1$ and thus invoke $decide^2(X)$. □

Validity, Termination, and Performance follow from the analogous properties of the Eventual Register.

A.3 Eventually Weak Interactive Consistency

Theorem A.3 (Well-formedness). *A given process p invokes $xreg[i].propose(v)$ for at most one v.*

Proof. If $p = p_i$, then $propose(v)$ has been invoked. If $p \neq p_i$, then $v = X$. □

Theorem A.4 (Validity). *Invoking $decide_q([v_1, \dots, v_n])$ implies that $propose(v_i)$ has been invoked by process p_i if (i) $v_i \neq X$ or (ii) before timeout ticked at any process either: (a) $decide_q([v_1, \dots, v_n])$ was invoked, or (b) Ω stabilized with p_i.*

Proof. If q has invoked $xreg[i].decide(v_i)$, then Validity of $xreg[i]$ implies that some process p must have invoked $xreg[i].propose^k(v_i)$ for some $k \in \{1, 2, 3\}$. To conclude that $p = p_i$ we must prove that $k \neq 3$. This is implied by each of the following conditions: (i) $v_i \neq X$, (ii) (a) $decide_q$ was invoked before $timeout_p$, so $xreg[i].propose^k_p(v_i)$ was invoked before $timeout_p$, (b) $timeout_p$ and $leader_p \neq p_i$ are never true at the same time. □

Theorem A.5 (Agreement). *If $decide_p([v_1, \dots, v_n])$ and $decide_q([w_1, \dots, w_n])$ have been invoked, then $[v_1, \dots, v_n] = [w_1, \dots, w_n]$.*

Proof. Follows from Agreement of $xreg[i]$ for all p_i. □

Theorem A.6 (Termination). *If the eventual leader has proposed, then every correct process will eventually decide.*

Proof. Assume that all $\mathbf{propose}_p$ actions are disabled and will remain disabled forever (this must eventually happen). Also, assume that Ω has stabilized with l. Then, for any i, process l is the only one able to invoke $xreg[i].propose$. We need to prove that l will keep invoking $xreg[i].propose$ forever for every p_i. If $p_i = l$, then since l has proposed, $\mathbf{leader}_l$ will be executed infinitely many times. Similarly, if $p_i \neq l$, then $timeout_l$ will eventually tick and $\mathbf{timeout}_l$ will be executed infinitely many times. □

Theorem A.7 (Performance). *If (i) all processes proposed by* $t + 1$*, and (ii) all processes with proposals* $v \neq X$ *proposed by* t*, then by time* $t + 2$ *(i) all processes decided, or (ii)* timeout *ticked at at least one process.*

Proof (Performance). Consider the execution of the system until time $t + 2$ and assume *timeout* does not tick at any process. Then, for no p_i is $\mathbf{timeout}_{p_i}$ ever executed, so $xreg[i].propose$ is only invoked by p_i. Assume that p_i has proposed v_i. If $v_i \neq X$, then Performance of $xreg[i]$ implies that $xreg[i].decide$ will be invoked at all processes by $t+2$. On the other hand, if $v_i = X$, then X-Performance of $xreg[i]$ implies that $xreg[i].decide$ will be also invoked at all processes by $(t + 1) + 1 = t + 2$. □

A.4 Eventually Weak Atomic Broadcast

We use $\mathbf{action}(v)$ to denote the instance of action **action** where the value of the free variable "v" in the guard of **action** is v. For example, $\mathbf{deliver}_p(t)$ denotes the instance of $\mathbf{deliver}_p$ with the specified value of t.

We make two technical assumptions about $ewic[t].timeout$. First, at any time, there is t such that for no $s > t$ has $ewic[s].timeout$ ticked at any process. Second, in stable runs, $ewic[t].timeout_q$ does not tick earlier than $\Delta > 2$ units of time after $ewic[t].propose$ was invoked. We also assume that the value *time* of the current time is increased after each action execution.

Theorem A.8 (Well-formedness). *For any process* p *and time* t*,* $ewic[t].propose_p$ *is invoked at most once.*

Proof. Follows from the fact that $\mathbf{get}_p\, times(t)$ can be executed only once. □

Lemma A.9. *Action* $\mathbf{deliver}(t)$ *will eventually be executed for every* t *at every correct process* p*.*

Proof. Assume that Ω has stabilized with l, which will (eventually) invoke $ewic[t].propose$. Then, Termination of $ewic[t]$ ensures that $ewic[t].decide_p$ will eventually be invoked, which implies the assertion. □

Theorem A.10 (Termination). *Let* l *be the eventual leader. Eventually, every message sent by* l *will eventually be delivered by every correct process.*

Proof. Assume Ω stabilized at time s. Let $T > s$ be a time for which no $ewic[T'].timeout$ with $T' > T$ ticked before time s at any process. Assume that l invoked $abcast(m)$ after time T. Then, m will eventually be added to $list_l$ and eventually $ewic[t].propose_l(list_l \ni m)$ will be invoked for some $t > T$. Note that at no process did $ewic[t].timeout$ tick before time s. Therefore, Validity of $ewic[t]$ implies that any $ewic[t].decide([v_1, \ldots, v_n])$ will have $m \in v_l$. Lemma A.9 implies that $\mathbf{deliver}(t)$ will eventually be executed, so m will be delivered. □

Theorem A.11 (Uniform Agreement). *If a process delivers a message m, then all correct processes will eventually deliver m.*

Proof. The assumption implies that there is t such that $\mathit{ewic}[t].\mathit{decide}([v_1, \ldots, v_n])$ has been invoked with some $v_i \ni m$. Agreement of $\mathit{ewic}[t]$ implies that any invocation of $\mathit{ewic}[t].\mathit{decide}([w_1, \ldots, w_n])$ will have $w_i = v_i \ni m$. Lemma A.9 implies the assertion. □

Theorem A.12 (Performance). *A message broadcast at time t will be delivered to all processes by $t + 2 + \delta$.*

Proof (Performance). For any time s, all processes invoke $\mathit{ewic}[s].\mathit{propose}$ at time s. Therefore, assuming $\Delta > 2$ in the $\mathit{ewic}[s].\mathit{timeout}$ implementation, Performance of $\mathit{ewic}[s]$ implies all processes will invoke $\mathit{ewic}[s].\mathit{decide}$ by $s + 2$. Therefore, all processes will execute **deliver**(s) by $s + 2$. If process p invokes $\mathit{abcast}(m)$ at time t, then it invokes $\mathit{ewic}[s].\mathit{propose}(\mathit{list} \ni m)$ at $s < t + \delta$. As a result, m will be delivered by all processes by $s + 2 < t + 2 + \delta$. □

For the proof of Uniform Validity and Uniform Total Order see Appendix A.4.1.

A.4.1 Quiet version

Lemma A.13. *Assume that $\mathit{lastd} \geq t$ and $\mathit{ewic}[t].\mathit{decide}([v_1, \ldots, v_n])$ with some $v_i \neq X$ has been invoked, then* **deliver**(t) *has been executed.*

Proof. Consider the instance of **deliver**(t'') that increased lastd from $t' < t$ to $t'' \geq t$. Since $\mathit{ewic}[s].\mathit{decide}([X, \ldots, X])$ for all $s \in (t', t'')$, we have $t \notin (t', t'')$. Since $t \in (t', t'']$, we have $t = t''$. □

Theorem A.14 (Uniform Validity). *For any message m, every process delivers m at most once and only if some process has broadcast m.*

Proof. Assume $\mathit{abcast}(m)$ is called at most once, say by process p. Then m is added to list_p at most once, and never to list_q for processes $q \neq p$. Since **active** empties list, there is at most one pair (p, t) such that $\mathit{ewic}[t].\mathit{propose}_p(\mathit{list} \ni m)$ is invoked. Therefore, by Validity of EWIC, only for this (p, t) can $\mathit{ewic}[t].\mathit{decide}([v_1, \ldots, v_n])$ be invoked with $v_p \ni m$. Similarly, if no process has ever broadcast m, then there will be no (p, t) with that property. □

Lemma A.15. *If a correct process p invokes $\mathit{active}(t)$ at time t', then any correct process q will eventually (by $t' + 1$ in stable runs) invoke $\mathit{ewic}[s].\mathit{propose}$ for all $s \leq t$.*

Proof. Eventually (by $t'+1$ in stable runs), process q will have $\mathit{time}_q > t$ and $\mathbf{get}_q\, \mathit{active}_p(t)$. If $\mathit{list}_q = X$, then after a potential execution of **passive**, we will have $t \leq \mathit{lastp}_q$. If $\mathit{list} \neq X$, then a potential execution of **active** will also result in $t < \mathit{time}_q \leq \mathit{lastp}_q$. The assertion follows from the meaning of lastp_q. □

Lemma A.16. *If $\mathit{ewic}[t].\mathit{decide}([v_1, \ldots, v_n])$ with some $v_i \neq X$ is invoked, then t is active.*

Proof. By Validity of $ewic[t]$, process p_i has invoked $ewic[t].propose(v_i \neq X)$. □

Since for any time t, there are only a finite number of active times earlier than t, the following lemma allows us to use induction on *lastp* and *lastd*.

Lemma A.17. *If not equal to $-\infty$, variables lastp and lastd are active times.*

Proof. The former follows from the body of **active** and the guard of **passive**. The latter is implied by Lemma A.16. □

Lemma A.18. *Let t be a time. If a correct process p invokes $ewic[s].decide([v_{s1}, \ldots, v_{sn}])$ for all $s \leq t$ and $v_{ti} \neq X$ for some p_i, then (eventually in non-stable runs) $lastd \geq t$.*

Proof. For the sake of contradiction assume that time $s < t$ is the maximum value *lastd* ever achieves. Also let $t' > s$ be the earliest time for which process p has invoked $ewic[t'].decide([v_1, \ldots, v_n])$ with some $v_i \neq X$. Obviously, $t' \leq t$ so **deliver**(t') will (eventually in non-stable runs) be executed. Therefore, the value of *lastd* will be increased from s to t', which contradicts the assumption. □

Lemma A.19. *Assume that, for some time t, message m, and process p_i, all invocations of $ewic[t].decide([v_1, \ldots, v_n])$ have $m \in v_i$. If a correct process p invokes $active(t')$ with $t' \geq t$, then eventually every correct process q delivers m.*

Proof. Assume that Ω has stabilized with l. Lemma A.15 implies that l will (eventually) invoke $ewic[s].propose$ for all $s \leq t \leq t'$. Then, Termination $ewic[s]$ implies that $ewic[s].decide_q([v_{s1}, \ldots, v_{sn}])$ will eventually be invoked. From the assumption, we have $m \in v_{ti}$, so Lemma A.18 implies that eventually $lastd_q \geq t$. Therefore, Lemma A.13 proves the assertion. □

Theorem A.20 (Termination). *Eventually, any message broadcast by the eventual leader l will eventually be delivered by every correct process q.*

Proof. Assume Ω stabilized at time s. Let $T > s$ be such that no $ewic[T'].timeout$ with $T' > T$ ticked before time s at any process. Assume that l invoked $abcast(m)$ after T. Then, m will eventually be added to $list_l$. Therefore, **active**$_l$ for some $t > T$ will eventually be executed, resulting in invoking $active_l(t)$ and $ewic[t].propose_l(list_l \ni m)$. Note that at no process did $ewic[t].timeout$ tick before time s. Therefore, Validity of $ewic[t]$ implies that any $ewic[t].decide([v_1, \ldots, v_n])$ will have $m \in v_l$. Lemma A.19 proves the assertion. □

Lemma A.21. *If a process has executed* **deliver**(t)*, then eventually a correct process will invoke $active(t')$ with $t' \geq t$.*

Proof. Follows from the fact that at least one correct process invokes *abcast* infinitely many times. (The assumption about **deliver**(t) is not necessary in this version of EWAB.) □

Theorem A.22 (Uniform Agreement). *If a process p delivers a message m, then every correct process q will eventually deliver m.*

Proof. The assumption implies that process p has executed **deliver**(t) and also invoked $\mathit{ewic}[t].\mathit{decide}([v_1, \ldots, v_n])$ with some $v_i \ni m$. Lemma A.21 ensures that eventually a correct process will invoke $\mathit{active}(t')$ with $t' \geq t$. Agreement of $\mathit{ewic}[t]$ and Lemma A.19 prove the assertion. □

Theorem A.23 (Uniform Total Order). *If a process p delivers a message m' after a message m then a process q that delivers m' has previously delivered m.*

Proof. For $r \in \{p, q\}$, let t_r be the earliest time, for which action $\mathbf{deliver}_r(t_r)$ delivers m'. Assume $t_p > t_q$, so eventually $\mathit{lastd}_p \geq t_p > t_q$. Then, Agreement of $\mathit{ewic}[t_q]$ and Lemma A.13 imply that $\mathbf{deliver}_p(t_q)$ must have been executed and m' delivered, which contradicts the definition of t_p. Therefore, $t_p \leq t_q$, so eventually $t_p \leq t_q \leq \mathit{lastd}_q$.

Assume m is first delivered at p by **deliver**(t) for $t \leq t_p \leq \mathit{lastd}_q$. By Agreement of $\mathit{ewic}[t]$ and Lemma A.13, action **deliver**(t) is also executed at q and delivers m. This implies the assertion because $t \leq t_p \leq t_q$ (if $t = t_p = t_q$, then note that **deliver**(t) delivers messages m and m' in same order at p and q). □

Lemma A.24. *In stable runs, $\mathit{lastp} \leq \mathit{time}$.*

Proof. The variable *lastp* gets updated by **active** and **passive**. The former obviously preserves the invariant. Since we assume synchronized clocks, $\mathit{active}_q(t)$ in the guard of **passive** cannot become true before $\mathit{active}(t)$ is invoked at q, which happens at time t. Therefore, $t \leq \mathit{time}$, so the invariant is preserved as well. □

Lemma A.25. *Let s be a time and t be the earliest active time not earlier than s. In stable runs, at no process p will $\mathit{ewic}[s].\mathit{timeout}$ tick before $t + 2$.*

Proof. Consider the moment when $\mathit{ewic}[s].\mathit{propose}$ was invoked at p by **active** or **passive**. After the execution of the action we have $\mathit{lastp} \geq s$, so $\mathit{lastp} \geq t$ because of Lemma A.17. Lemma A.24 implies that $\mathit{time} \geq \mathit{lastp} \geq t$, which means that $\mathit{ewic}[s].\mathit{propose}$ could not have been invoked before time t. Therefore, assuming that $\Delta > 2$ in the *ewic* timeout implementation, $\mathit{ewic}[s].\mathit{timeout}$ will not tick before $t + 2$. □

Lemma A.26. *Let s be a time and t be the earliest active time not earlier than s. In stable runs, all processes invoke $\mathit{ewic}[s].\mathit{decide}$ by time $t + 2$.*

Proof. Action **active** is the only place where $\mathit{ewic}[s].\mathit{propose}(v \neq X)$ can be invoked, so all $\mathit{ewic}[s].\mathit{propose}(v \neq X)$ were invoked at time $s \leq t$. Since t is active, at least one process invoked $\mathit{ewic}[t].\mathit{propose}(v \neq X)$ and $\mathit{active}(t)$, so Lemma A.15 implies that by $t+1$ all processes have invoked $\mathit{ewic}[s].\mathit{propose}$. Therefore, Lemma A.25 and the Performance property of $\mathit{ewic}[s]$ imply the assertion. □

Theorem A.27 (Performance). *A message broadcast at time t will be delivered at all processes by time $t + 2$.*

Proof. Since *time* is increased after each action execution, Lemma A.24 implies that **active** is enabled whenever $list \neq X$. Assume p_i invokes $abcast(m)$ at time t, which adds m to *list* and enables **active**, which is immediately executed. As a consequence, $ewic[t].propose(list \ni m)$ is invoked.

Let $s \leq t$ be a time and t' be the earliest active time not earlier than s. Since time t is active, we have $t' < t$ and Lemma A.26 implies that all processes will invoke $ewic[s].decide$ by $t' + 2 \leq t + 2$. In particular, Lemma A.25 and the Validity condition of $ewic[t]$ imply $ewic[t].decide([v_1, \ldots, v_n])$ will have $m \in v_i$. Therefore, Lemmata A.18 and A.13 imply that **deliver**(t) will be executed by time $t + 2$ and deliver m. □

A.4.2 Pingless version

Lemma A.28. *If active*(t) *has been invoked, then* t *is active.*

Proof. If invoked in **active**, then obviously true. If in **decide**, it follows from Lemma A.16. Action **relay** is executed only if $active(t)$ has been invoked before. □

Lemma A.29. *If a correct process* p *invokes* $ewic[t].decide([v_1, \ldots, v_n])$ *with some* $v_i \neq X$*, then each correct process will eventually invoke* $active(t)$*.*

Proof. Process p will execute **decide**(t), which will invoke $active(t)$, so eventually all correct processes will execute **relay**(t). □

Lemma A.30. *Let* t *be a time. If a correct process* p *invokes* $ewic[s].decide([v_{s1}, \ldots, v_{sn}])$ *for all* $s \leq t$ *and there is* $v_{ti} \neq X$*, then (eventually in non-stable runs)* $lastd \geq t$*. (Lemma A.18)*

Proof. For the sake of contradiction assume that time $s < t$ is the maximum value *lastd* ever achieves. Also, let $t' > s$ be the earliest time for which p has invoked $ewic[t'].decide([v_1, \ldots, v_n])$ with some $v_i \neq X$. Obviously, we have $t' \leq t$ and from Lemma A.29 used for t' we can conclude that **deliver**(t') will eventually be executed. Therefore, the value of *lastd* will be increased s to t', which contradicts the assumption. □

Lemma A.31. *If a process has executed deliver*(t)*, then eventually a correct process will invoke* $active(t')$ *with* $t' \geq t$*. (Lemma A.21)*

Proof. The assumption implies that a majority of processes have invoked $active(t)$, which includes at least one correct process. □

Theorem A.32 (Performance). *A message broadcast at time* t *will be delivered at all processes by* $t + 2$*.*

Proof. The assumption implies that $active(t)$ is invoked by the sender at time t. Therefore, all processes will invoke $active(t)$ at time $t + 1$ (action **relay**). Therefore, at time $t + 2$ every process will know that $active(t)$ has been invoked at every process, so Theorem A.27 implies the assertion. □

All other proofs from Appendix A.4.1 can be used for the pingless version of EWAB.

A.5 Atomic Broadcast

Theorem A.33 (Uniform Validity). *For any message m, every process delivers m at most once and only if some process has broadcast m.*

Proof. Follows from fact that *deliver* eliminates duplicates and Uniform Validity of EWAB. ☐

Theorem A.34 (Termination). *If a correct process p broadcasts a message m, every correct process q will eventually deliver m.*

Proof. Assume that l is the eventual leader. If process p invokes $abcast(m)$, then eventually m is permanently added to $blist_l$. Therefore, l will invoke $ewab.abcast(blist \ni m)$ infinitely many times. Hence, by Termination of *ewab*, action **deliver** with $v \ni m$ will eventually be executed at every correct process and m will be delivered. ☐

Uniform Agreement, Uniform Total Order, Performance, and Quietness follow from the analogous properties of EWAB.