RZ 3754 (# 99764) 11/09/2009
Computer Science 20 pages

Research Report

Yet Another Visit to Paxos

Christian Cachin

IBM Research – Zurich
8803 Rüschlikon
Switzerland

E-mail: cca@zurich.ibm.com

Yet Another Visit to Paxos

Christian Cachin

IBM Research - Zurich
cca@zurich.ibm.com

4 November 2009

Abstract

This paper presents a modular introduction to crash-tolerant and Byzantine-tolerant protocols for reaching consensus that use the method introduced by the Paxos algorithm of Lamport and by the viewstamped replication algorithm of Oki and Liskov. The consensus protocol runs a sequence of *epoch* abstractions as governed by an *epoch-change* abstraction. Implementations of *epoch* and *epoch-change* that tolerate crash faults yield the consensus algorithm in Paxos and in viewstamped replication. Implementations of *epoch* and *epoch-change* that tolerate Byzantine faults yield the consensus algorithm in the PBFT protocol of Castro and Liskov.

1 Introduction

The Greek island of Paxos has become famous in distributed computing for its *consensus algorithm* [14], a leader-based protocol for reaching agreement among a set of processes in the presence of faults. It was only discovered about 20 years ago. Like many other Greek islands, Paxos is a nice location with sandy beaches and picturesque scenery. It is thus hardly surprising that many people have since traveled to Paxos to investigate the secrets of its consensus algorithm [15, 7, 16, 1, 2, 12, 17, 20].

The purpose of this trip is pedagogical: we give a modular introduction to the Paxos consensus protocol. The same algorithm also exists in the New World under the name of *viewstamped replication* [21], and our description covers it as well. Further variants of the Paxos protocol have been developed that tolerate not only crashes but arbitrary behavior of some processes, so-called *Byzantine faults*. The most prominent example is the *PBFT protocol* (for "Practical Byzantine Fault Tolerance") [5]. We explain how these two consensus protocols work in a unified language and identify common building blocks inherent in them. Our formulation relies on the findings of another recent excursion to Paxos [20], which uncovered the fundamental role of the so-called "weak interactive consistency" primitive for Paxos consensus tolerating Byzantine faults.

Our abstract consensus protocol relies on a dynamically changing leader process that is supposed to be a correct process eventually. The protocol is structured into a sequence of *epochs*, which are started by an *epoch-change* primitive. By implementing the epoch-change and epoch modules in the two failure models, tolerating crash faults and Byzantine faults, we instantiate abstract consensus to the Paxos protocol and to the PBFT protocol, respectively. As a side-effect, this description yields also the first formulation of Byzantine consensus relying on an abstract eventual leader-detector oracle (Ω).

This report focuses on using leader-based consensus algorithms for reaching agreement on a *single* question; in their original incarnations, the Paxos protocol, viewstamped replication, and the PBFT protocol address the more general atomic broadcast problem. Atomic broadcast corresponds to a sequence of such agreements and lets the processes implement a fault-tolerant service using replication [22].

2 Model and Assumptions

2.1 System Model

The system consists of n *processes* $p_1, \ldots, p_n$. Up to f of them may *fail* at some time by exhibiting crash faults or by behaving arbitrarily (so-called Byzantine faults). We assume that crashes are final and processes do not recover. We call a process *correct* until it fails and *faulty* afterwards; these terms are dynamic and depend on the time at which they apply. With crash faults, a process is either forever correct or correct until it fails, when it becomes faulty and no longer takes any steps. With Byzantine faults, it means that all processes are initially correct and that some of them become faulty at well-defined points during the execution, for example, as the result of being corrupted by an attacker; this corresponds to the notion of adaptive security from cryptography [10]. These notions differ from the standard terminology used in the literature on reliable distributed systems, but they are necessary for unifying our definitions covering crash faults and Byzantine faults.[1]

Every two correct processes can send messages to each other using a point-to-point channel abstraction (specified below). The system is partially synchronous [8] in the sense that there is no a priori bound on message delays and the processes have no synchronized clocks, but there is a time (not known to the processes) after which the system is *stable* in the sense that message delays and processing times are bounded. In other words, the system is *eventually synchronous*.

2.2 Definitions

The algorithms in this paper implement uniform consensus (with crash faults) and Byzantine consensus (with Byzantine faults).

Uniform consensus. The goal of a consensus protocol is to agree on a common value, despite the failure of some processes and the unbounded delays of the communication channels.

A *consensus protocol* is invoked at every process by a *propose*(v) event, containing an initial value v that the process proposes for the decision. The protocol terminates when it outputs an event *decide*(v) with a decision value v. In a system with crash faults, *uniform consensus* satisfies the following properties:

Validity: If a process decides v, then v was proposed by some process.

Agreement: No two correct processes decide differently.

Integrity: Every correct process decides at most once.

Termination: Every permanently correct process eventually decides some value.

Note how this actually defines uniform consensus [13] through our definition of "correct" processes.[2]

[1]The standard definitions [13] of "correct" and "faulty" processes are static, in the sense that a process that eventually crashes is called faulty from the outset. This leads to the distinction between non-uniform and uniform definitions. "Correct" and "faulty" processes according to the standard terminology are here called processes that remain correct forever and processes that eventually crash, respectively.

[2]If we use the standard notion of a "correct" process here, then this formulation of agreement does not define the standard notion of uniform consensus [13], as it allows a process to decide a different value and then to crash. Uniform consensus is therefore typically defined to require agreement for *any* two processes. But with Byzantine faults, a faulty process may decide arbitrarily and thus, no agreement condition referring to actions of all processes can cover crash faults and Byzantine faults simultaneously. Hence, our choice to define a "correct" process in a dynamic, time-dependent way.

Byzantine consensus. In a system with Byzantine faults the consensus problem is defined through the same events as with crash faults and has almost the same properties. The only difference lies in the validity condition because one cannot make any statement about the values that faulty processes propose. A protocol for *Byzantine consensus* satisfies the *agreement*, *integrity*, and *termination* conditions of uniform consensus and the following condition:

Weak validity: If all processes are correct and propose v, then a correct process may only decide v.

When some processes are faulty, weak validity allows Byzantine consensus to decide on a value originating from the faulty processes, which was never proposed by a correct process. Weak validity contrasts with *strong validity* in Byzantine consensus, which requires that if all correct processes propose v, then a correct process may only decide v.

2.3 Primitives

Basic communication primitives. The protocols use several standard abstractions that encapsulate elementary message exchange between the processes [13]. We first consider two primitives for systems with crash faults.

A *perfect point-to-point link (pp2p)* abstraction allows any process to send messages to any other process reliably, such that if both processes remain correct forever, every message is eventually delivered to the receiver.

A *best-effort broadcast (beb)* primitive enables a process to send a message in a one-shot operation to all processes in the system, including itself. Messages are only guaranteed to be delivered if the sender is correct forever and to those receivers that do not fail.

These two primitives have dual notions in the model with Byzantine faults. Since faulty processes may access messages sent over the network, modify them, or insert messages on the network, the basic communication abstractions additionally must guarantee message integrity. The interface of these abstractions contains an indication of the sender of for every delivered message.

More precisely, an *authenticated perfect point-to-point link (app2p)* abstraction is a perfect point-to-point link primitive that guarantees, in addition, that if a message m is delivered to a correct receiver process with indicated sender p_s, and p_s is correct up to this time, then p_s previously sent m.

Analogously, an *authenticated best-effort broadcast (abeb)* abstraction is a best-effort broadcast primitive that guarantees, in addition, that if a message m is delivered to a correct receiver process with indicated sender p_s, and p_s is correct up to this time, then p_s previously broadcast m.

The authenticated communication primitives can be implemented easily with a message-authentication code (MAC) [19], a symmetric cryptographic primitive that relies on a secret key shared by every pair of processes. We assume that these keys have been distributed by a trusted entity beforehand.

Finally, we introduce an *eventual leader detector (Ω)* abstraction, which applies to both failure models. Ω periodically outputs an event *trust*(j) at an arbitrary process, which indicates that p_j is trusted to be leader. We say that the process that receives the event henceforth *trusts* p_j (until the process receives the next *trust* event or forever). The eventual leader detector ensures (1) that there is a time after which every correct process trusts some process that remains correct forever, and (2) that there is a time after which no two correct processes trust a different process.

Weak interactive consistency. A protocol for *weak interactive consistency (WIC)* is invoked at every process by a *send*(m) event with an input message m; the protocol outputs a vector M with n entries through an event *deliver*(M) at every process, such that $M[i]$ is either a special symbol $\perp$ or corresponds to the input message of p_i. To differentiate the *send* and *deliver* events from others, we also say that a process *wic-sends* a message and *wic-delivers* a message vector. The protocol implementation may rely on a designated process p_ℓ, called the leader. It satisfies the following conditions:

Consistency: If the leader remains correct forever, then every correct process wic-delivers the same M, and this M contains at least $n - f$ messages different from $\perp$.

Integrity: If some correct process wic-delivers M with $M[j] \neq \perp$ and p_j is correct up to this time, then p_j has previously executed wic-send with message $M[j]$.

Termination: If the leader remains correct forever, then every permanently correct process eventually wic-delivers some M.

The weak interactive consistency primitive was defined by Milosevic, Hutle, and Schiper [20] with the goal of simplifying Byzantine consensus protocols.

A protocol that implements WIC with crash faults is straightforward. Every process simply uses the perfect point-to-point links to send m to the leader, the leader collects $n - f$ messages in M, and uses best-effort broadcast to disseminate M. This protocol involves only two rounds of messages.

With Byzantine faults, assuming the processes use a public-key digital signature scheme, the same two-round protocol implements WIC; every process signs its initial message to the leader and before wic-delivering M, a process verifies that $M[j]$ contains a valid signature by p_j.

Furthermore, a three-round implementation in the Byzantine model that uses only message authentication but no computationally expensive public-key signatures works as follows [5, 20].

1. In the first round, every process uses authenticated best-effort broadcast to send m to all processes. Every process collects $n - f$ such inputs in a vector E, such that $E[j] = m_j$ when it received input m_j from p_j.

2. In the second round, every process uses authenticated best-effort broadcast to send the vector E computed above to all processes. Every process waits until it receives $n - f$ such vectors such that E_k was broadcast by p_k. The leader p_ℓ then checks for each input m_j that it received from p_j in the first round if
$$\Big|\big\{k \,|\, E_k[j] = m_j\big\}\Big| \;>\; 2f.$$
If this is the case, then p_ℓ sets $M[j] \leftarrow m_j$, and otherwise, it sets $M[j] \leftarrow \perp$.

3. In the third round, only p_ℓ sends a message. It uses authenticated best-effort broadcast to send the vector M computed above to all processes. When a process p_i receives this message containing M, it revisits the vectors E_k received in the second round and verifies that for any j with $M[j] \neq \perp$, it holds
$$\Big|\big\{k \,|\, E_k[j] = M[j]\big\}\Big| \;>\; f.$$
If this is not the case, then p_i sets $M[j] \leftarrow \perp$. Finally, p_i outputs M.

We refer to the literature [20] for a proof that this protocol implements weak interactive consistency for $n > 3f$. The protocol has message complexity $O(n^2)$ and communication (bit) complexity $O(n^2 b)$, assuming that the length of the input messages is bounded by b. Using a collision-free hash function whose output is of size κ, the communication complexity can be reduced to $O(nb + n^2\kappa)$ [5].

Quorums. A *quorum* in a system of n processes is a set of more than $n/2$ processes. Every two quorums overlap in at least one process.

A *Byzantine quorum* in a system of n processes that tolerates f Byzantine faults is a set of more than $\frac{n+f}{2}$ processes. Every two Byzantine quorums overlap in at least one correct process. For $f = 0$, a 0-tolerant Byzantine quorum reduces to a quorum.

Consider a vector S of n entries, where each entry corresponds to a process and is either a tuple of the form $[\cdots, ts, v, \cdots]$ that contains at least a timestamp ts and a value v, or is equal to $\perp$. We define the *g-tolerant quorum maximum* of S, denoted by *quorum-max*$_g(S)$, as a timestamp/value pair (ts, v) such that

$$\Big|\{j \mid S[j] = [\cdots, ts', v', \cdots] \wedge (ts' < ts \vee (ts' = ts \wedge v' = v))\}\Big| \; > \; \frac{n+g}{2}.$$

If no such tuple exists, we define the quorum maximum to be $\perp$.

In other words, the 0-tolerant quorum maximum of S is the timestamp/value pair with the highest timestamp found in a quorum of non-$\perp$ entries of S; more generally, the g-tolerant quorum maximum of S is the timestamp/value pair with the highest timestamp found in a Byzantine quorum (tolerating g faults) of non-$\perp$ entries of S.

2.4 Notation for Protocols

Protocols are presented in a modular way using the notation of Guerraoui and Rodrigues [13]. Every primitive is defined by a protocol module that may be called by other protocol implementations. A protocol module can be instantiated statically; this usually happens only once and occurs implicitly when an algorithm includes the protocol module among the list of its *used* protocols. A protocol module can also be instantiated dynamically with an a-priori unknown number of instances. The initializations of dynamic instances are mentioned explicitly in the code of the algorithm that calls them. Dynamically instantiated protocol modules are named and addressed by an identifier. When an instance causes the basic communication primitives (pp2p, beb, app2p, abeb) to send and receive messages, this identifier is contained implicitly in all messages sent by the instance and only messages matching the identifier are received by the instance.

3 Abstract Leader-based Consensus

This section presents a leader-based consensus protocol, implemented using two abstract primitives defined next, *epoch-change* and *epoch*. The consensus protocol invokes a sequence of such epochs, governed by outputs from epoch-change.

The definitions of the two primitives are the same for crash faults and for Byzantine faults. The consensus protocol is also the same for both fault types, only the problem that it solves differs according to the implementation of the primitives.

3.1 Primitives

Epoch-change. The task of an *epoch-change* abstraction is to periodically output a *startepoch*(ts, ℓ) event. When this occurs, we say the process *starts epoch (ts, ℓ)*. The event contains two parameters, an *epoch timestamp ts* and a *leader index ℓ*, that serve to identify the starting epoch. We require that the timestamps in the sequence of epochs that are started at one process are monotonically increasing and that every process receives the same leader index for a given epoch timestamp. More precisely:

Monotonicity: If a correct process starts an epoch (ts, ℓ) and later starts an epoch (ts', ℓ'), then $ts' > ts$.

Consistency: If a correct process starts an epoch (ts, ℓ) and another correct process starts an epoch (ts', ℓ') with $ts = ts'$, then $\ell = \ell'$.

Eventual accuracy and agreement: There is a time after which every correct process has started some epoch and starts no further epoch, such that (1) the last epoch started at every correct process is epoch (ts, ℓ); and (2) p_ℓ remains forever correct.

When an epoch-change is initialized, the primitive makes available a default timestamp and leader index that are equal for all processes.

Epoch. An *epoch* is a primitive similar to consensus, where the processes propose a value and may decide a value. Every epoch is identified by an *epoch timestamp* and has a designated *leader*. As for consensus, the goal of an epoch is that all processes decide the same value. But an epoch is easier to implement than consensus because it only represents an attempt to reach consensus; an epoch may not terminate and can be aborted if it does not decide. As another simplification, only the leader proposes a value and the epoch is required to decide only when its leader is correct. Because an epoch may not decide, a protocol usually runs multiple epochs in a logical sequence such that later epochs depend on earlier ones.

More precisely, an *epoch* is initialized with a timestamp ts, a leader index ℓ, and some implementation-specific value $state$ at every process. To start the protocol, the leader p_ℓ must trigger a *propose*(v) event containing a value v. When this occurs, we say the leader *ep-proposes* v, to distinguish it from proposing a value for consensus. One way for an epoch to terminate is to output an event *decide*(v). When it occurs, we say the process *ep-decides* v.

An epoch may also terminate when the calling protocol locally triggers an *abort* event. After receiving this event, the epoch returns an event *aborted*$(state)$ to the caller, containing some internal state. The caller must use this state to initialize the next epoch in that it participates. Aborts are always triggered externally, an epoch does not abort on its own. Different processes may abort an epoch independently of each other at different times.

Every process runs at most one epoch at a time; the process may only initialize a new epoch after the previously active one has aborted or ep-decided. Moreover, a process must only initialize an epoch with a higher timestamp than that of all epochs that it initialized previously.

Under these assumptions about invoking multiple epochs, a single *epoch with timestamp* ts *and leader* p_ℓ satisfies the following properties:

Validity: If a correct process ep-decides v, then v was ep-proposed by the leader $p_{\ell'}$ of an epoch with timestamp $ts' \leq ts$ and leader $p_{\ell'}$.

Agreement: No two correct processes ep-decide differently.

Integrity: Every correct process ep-decides at most once.

Lock-in: If a correct process has ep-decided v in an epoch with timestamp $ts' < ts$, then no correct process ep-decides a value different from v.

Termination: If the leader p_ℓ is permanently correct, has ep-proposed a value, and no correct process aborts the epoch, then every permanently correct process eventually ep-decides some value.

Abort behavior: When a permanently correct process aborts the epoch, it eventually outputs *aborted*; moreover, a correct process outputs *aborted* only if the epoch has been aborted by some correct process.

The above validity condition only applies to the model of crash faults, as it refers to the actions of a leader $p_{\ell'}$, which poses a problem if $p_{\ell'}$ may suffer from a Byzantine fault. But our goal is to implement Byzantine consensus, which requires only weak validity, where all processes are correct. Thus, we can also use the above notion of validity for the Byzantine model under the restriction that all processes are correct.

3.2 Consensus

Algorithm 1 implements consensus from an epoch-change abstraction (ec) and multiple instances of an epoch abstraction (ep).

Intuitively, the value that is decided by the consensus algorithm is the value that is ep-decided by one of the underlying epochs. The protocol invokes a sequence of epoch instances, triggered by the *startepoch* events output by the epoch-change primitive. These events also contain timestamp and leader of the next epoch to start. To switch from one epoch to the next, the protocol aborts the running epoch, obtains its state, and initializes the next epoch with the state.

Algorithm 1 Abstract leader-based consensus: implements uniform consensus with crash faults and Byzantine consensus with Byzantine faults (code for p_i).

```
Implements:
    Consensus (c), either Uniform consensus or Byzantine consensus.

Uses:
    Epoch-change (ec);
    Epoch (ep).

upon ⟨init⟩ do
    $val \leftarrow \bot$
    $proposed \leftarrow$ FALSE; $decided \leftarrow$ FALSE
    Initialize an instance of Epoch-change and obtain its initial timestamp $ets_0$ and initial leader $\ell_0$
    Initialize a new instance of Epoch with timestamp $ets_0$, leader $\ell_0$, and state $(0, \bot)$
    $(ets, \ell) \leftarrow (ets_0, \ell_0)$

upon ⟨c.propose | $v$⟩ do
    $val \leftarrow v$

upon ⟨ec.startepoch | $newts, new\ell$⟩ do
    trigger ⟨ep.abort⟩ for instance with timestamp $ets$
    wait for ⟨ep.aborted | $state$⟩ from instance with timestamp $ets$
    $(ets, \ell) \leftarrow (newts, new\ell)$
    $proposed \leftarrow$ FALSE
    Initialize a new instance of Epoch with timestamp $ets$, leader $\ell$, and state $state$

upon $(\ell = i) \wedge (val \neq \bot) \wedge (proposed =$ FALSE$)$ do
    $proposed \leftarrow$ TRUE
    trigger ⟨ep.propose | $val$⟩ for instance with timestamp $ets$

upon ⟨ep.decide | $v$⟩ do
    if $decided =$ FALSE then
        $decided \leftarrow$ TRUE
        trigger ⟨c.decide | $v$⟩
```

When the process has received a value proposed for consensus from the caller and the process is also the leader of the current epoch, then it ep-proposes this value for the epoch. When the epoch ep-decides a value, the process also decides that value for consensus, but continues to participate in the consensus protocol forever, to help other processes decide.

The complexity of the leader-based consensus protocol depends entirely on the complexities of the underlying implementations of epoch-change and epoch, since the protocol does not directly send any messages using the point-to-point link abstraction or the best-effort broadcast abstraction.

Theorem 1. *Given implementations of epoch and epoch-change that tolerate crash faults, Algorithm 1 implements uniform consensus with crash faults. Moreover, assuming implementations of epoch and epoch-change that tolerate Byzantine faults, Algorithm 1 implements Byzantine consensus with Byzantine faults.*

Proof. Only the validity condition differs from uniform consensus to Byzantine consensus. We first establish the two notions of validity. The proof for the remaining conditions is the same for both types of faults.

Validity (for uniform consensus with crash faults). We show validity by induction on the sequence of epochs that have ever been started at any correct process, ordered by their timestamp. According to the protocol, a process decides v only when it has ep-decided v in the current epoch; hence, every decision can be attributed to a unique epoch. Let ts^* be the smallest timestamp of any epoch in which some process decides v. Then this process has ep-decided v in the epoch with timestamp ts^*. According to the validity of epoch, this means that v was ep-proposed by the leader of some epoch whose timestamp is at most ts^*, and because a process only ep-proposes val when val has been proposed for consensus, the condition holds.

Suppose now that validity (of consensus) holds for every process that decided in some epoch ts', and consider a correct process that decides in an epoch $ts > ts'$. According to the lock-in property of epoch, it may only decide v; hence, the validity property holds.

Weak validity (for Byzantine consensus with Byzantine faults). For the case of Byzantine faults, the validity condition of consensus follows essentially from the same argument as in the case of crash faults. We note that *any* value proposed for consensus may be decided because the protocol guarantees weak validity.

Agreement. According to the discussion in the proof of validity, every decision of consensus can be attributed to the decision of an epoch. Thus, if two correct processes decide when they are in the same epoch, then the agreement condition of an epoch ensures agreement; otherwise, if they decide in different epochs, the lock-in property of epochs establishes agreement.

Integrity. This property is straightforward to verify from the protocol, because the $decided$ flag prevents multiple decisions.

Termination. It is easy to see that the protocol satisfies the requirements on invoking a sequence of epochs, from the monotonicity and consistency of the epoch-change primitive and because the protocol only initializes a new epoch after the previous one, with a smaller timestamp, has aborted.

According to the eventual accuracy and agreement properties of epoch-change, there exists some epoch with timestamp ts and leader p_ℓ, such that no further epoch starts and p_ℓ remains correct forever. Observe that the protocol only aborts an epoch when the epoch-change primitive starts another epoch. Since this does not occur, the termination condition for epoch (ts, ℓ) now implies that every permanently correct process eventually ep-decides. □

4 Consensus (Paxos/Viewstamped Replication)

We give algorithms for the epoch-change and epoch primitives in a system subject to crash faults. When they are used in the abstract leader-based consensus, the resulting protocol closely resembles the mechanism to reach consensus in the Paxos protocol [14] and in viewstamped replication [21].

Implementing epoch-change. Algorithm 2 presents the protocol for epoch-change. The protocol relies on an eventual leader detector primitive.

The epoch-change protocol is quite simple. Process p_i maintains two timestamps: a timestamp $lastts$ of the last epoch that it started (i.e., for which it triggered a *startepoch* event), and the timestamp ts of the last epoch that it attempted to start with itself as leader (i.e., for which it broadcast a NEWEPOCH message, as described next). Initially, the process sets ts to its index i. Whenever the leader detector subsequently makes p_i trust itself, p_i adds n to ts and sends a NEWEPOCH message with ts. When process p_i receives a NEWEPOCH message with a parameter $newts > lastts$ from some p_ℓ *and* p_i most recently trusted p_ℓ, then the process triggers *startepoch* with parameters $newts$ and ℓ. Otherwise, the process informs p_ℓ using a NACK message that the new epoch could not be started. When a process receives a NACK message and still trusts itself, it increments ts by n and tries again to start an epoch by sending another NEWEPOCH message.

Algorithm 2 incurs a complexity of $O(n)$ messages and a communication complexity of $O(n)$ during stable periods.

Theorem 2. *Algorithm 2 implements epoch-change with f crash faults for $n > f$.*

Proof. Since process p_i locally maintains the timestamp $lastts$ of the most recently started epoch, the protocol ensures that p_i only starts further epochs with higher timestamps. This establishes the monotonicity property of epoch-change. Furthermore, the space of epoch timestamps is partitioned among to the n processes. Hence, no two distinct processes send a NEWEPOCH message with the same timestamp value, demonstrating the consistency property.

The eventual accuracy and agreement property is based on the properties of the leader detector. Let p_t be the process that is eventually trusted by all correct processes and that remains correct forever. At the last time when Ω causes p_t to trust itself, it broadcasts a NEWEPOCH message with a timestamp tts that should cause all processes to start an epoch with leader p_t and timestamp tts. Consider any correct process p_j that receives this message: p_j either last trusted p_t and tts is bigger than its variable $lastts$ and therefore p_j starts epoch (tts, t); or the condition does not hold and p_j sends a NACK message to p_t. In the latter case, this message causes p_t to increment its variable ts and to send another NEWEPOCH message. The properties of Ω ensure that eventually all correct processes trust p_t forever, therefore only p_t increments its ts variable and all other processes have stopped sending NEWEPOCH messages. Hence, p_t eventually sends a NEWEPOCH message with a timestamp bigger than the $lastts$ variable of p_j. Since p_j trusts p_t when it receives this message, p_j eventually starts some epoch with timestamp tts^* and leader p_t. And because p_t is correct and sends the NEWEPOCH message with timestamp tts^* to all processes, every correct process eventually starts this epoch and stops sending NACK messages.

Suppose that p_j above is the last process whose NACK message was delivered to p_t. Then, since p_t sends the NEWEPOCH message with timestamp tts^* to all processes, the epoch with timestamp tts^* is also the last epoch that every correct process starts. □

Implementing epoch. Algorithm 3 implements one instance of epoch. The protocol uses best-effort broadcast and perfect point-to-point links abstractions. In the description, the notation $[x]^n$ for any symbol x is an abbreviation for the n-vector $[x, \dots, x]$.

Algorithm 2 Implements epoch-change with f crash faults for $n > f$ (code for p_i).

```
Implements:
    Epoch-change (ec).

Uses:
    Eventual leader detector ($\Omega$);
    Best-effort broadcast (beb);
    Perfect point-to-point links (pp2p).

upon $\langle$init$\rangle$ do
    $trusted \leftarrow 0$
    $lastts \leftarrow 0$
    $ts \leftarrow i$
    Initialize an instance of Eventual leader detector

upon $\langle \Omega$.trust $| \ell\rangle$ do
    $trusted \leftarrow \ell$
    if $\ell = i$ then
        $ts \leftarrow ts + n$
        trigger $\langle$beb.broadcast $|$ [NEWEPOCH, $ts$]$\rangle$

upon $\langle$beb.deliver $| p_\ell$, [NEWEPOCH, $newts$]$\rangle$ do
    if $\ell = trusted \wedge newts > lastts$ then
        $lastts \leftarrow newts$
        trigger $\langle$ec.startepoch $| newts, \ell\rangle$
    else
        trigger $\langle$pp2p.send $| p_\ell$, [NACK]$\rangle$

upon $\langle$pp2p.deliver $| p_j$, [NACK]$\rangle$ do
    if $trusted = i$ then
        $ts \leftarrow ts + n$
        trigger $\langle$beb.broadcast $|$ [NEWEPOCH, $ts$]$\rangle$
```

Multiple instances of epoch may be executed at the same point in time on different processes, but when used in our leader-based consensus protocol (Algorithm 1), then every process only runs at most one epoch instance at a time. Different instances never interfere with each other according our assumption that every instance is identified by a unique epoch timestamp and because point-to-point messages and best-effort broadcast messages are only received from and delivered to other epoch instances with the same timestamp.

Intuitively, the protocol works as follows. Recall that an epoch instance is initialized with a $state$ value, which the previous epoch instance returned to the process when it was aborted. Passing $state$ to the next epoch only serves the *validity* and *lock-in* properties of an epoch, since these properties (and no others) link two epochs with different timestamps together.

The algorithm involves two rounds of message exchanges from the leader to all processes and uses quorums to ensure *lock-in* in a similar way as a replicated implementation of a read/write register. In the first round, the leader sends a READ message to obtain the state from a quorum of processes and determines whether there exists a value that may already have been ep-decided. Every process answers with a STATE message containing its locally stored value and the timestamp of the epoch during which the value was last written. The leader takes the value with the highest timestamp and writes it to a quorum of processes. The write succeeds when the leader receives an ACCEPT message from a quorum of processes, indicating that they have stored the value locally. It is now safe for the leader to ep-decide for the value just written. The leader announces this value in a DECIDED message to all processes and the processes that receive it ep-decide.

Remarks. The message complexity of an epoch is $O(n)$, and the communication complexity of the protocol is $O(nb)$, where b is a bound on the length of the proposal values.

When this protocol for epoch is used together with Algorithm 2 to implement consensus using the leader-based approach of Algorithm 1, the READ message can be omitted. Since the leader of the epoch sends READ immediately after it sends a NEWEPOCH message to all processes in Algorithm 2, every process could just send the STATE message upon initializing the epoch instance in reply to receiving the NEWEPOCH message.

One can transform the actions of sending the STATE message, choosing the highest timestamp/value pair, and sending the WRITE message into a form that uses the weak interactive consistency abstraction. In this variant, every process wic-sends its STATE message; after wic-delivering a vector $states$, the process chooses the highest timestamp/value pair from the vector and proceeds with that as if it arrived in a WRITE message. This is equivalent to Algorithm 3 in terms of implementing an epoch; but since every process needs to compute the maximum, it increases the communication (bit) complexity by a factor $\Theta(n)$.

Theorem 3. *Algorithm 3 implements epoch with timestamp ets and leader index ℓ with f crash faults for $n > 2f$.*

Proof. We first establish the *lock-in* property of an epoch. Suppose some process has ep-decided v in an epoch with timestamp $ts' < ts$. The process only ep-decided after receiving a DECIDED message with v from the leader $p_{\ell'}$ of epoch ts'. Before sending this message, $p_{\ell'}$ had broadcast a WRITE message containing v and had collected ACCEPT messages in response from a set $\mathcal{A}$ of more than $n/2$ distinct processes. According to the protocol, these processes set their variables val to v and $valts$ to ts'.

Consider the next epoch in which the leader sends a WRITE message, and let its timestamp be ts^* and its leader index be ℓ^*. This means that no process has changed its $valts$ and val variables in any epoch between ts' and ts^*. By the assumption on how a process passes the state of an epoch to the next one, every process in $\mathcal{A}$ starts epoch ts^* with state $(valts, val) = (ts', v)$. Hence, p_{ℓ^*} collects STATE messages whose 0-tolerant quorum maximum is (ts', v) and broadcasts a WRITE message containing v. This implies that a process can only ep-decide v and that the set of processes whose variable val is equal to v when they abort epoch ts^* is at least $\mathcal{A}$. Continuing this argument until epoch ts establishes the *lock-in* property.

To show *validity*, assume that a process ep-decides v. It is obvious from the protocol that a process only ep-decides for the value v received in a DECIDED message from p_ℓ; furthermore, every process stores in variable val only to the value received in a WRITE message from the leader. In both cases, this value comes from the variable $tmpval$ of the leader. But in any epoch the leader sets $tmpval$ only to the value that it ep-proposed or to some value that it received in a STATE message from another process. By backward induction in the sequence of epochs, this shows that v was ep-proposed by the leader in some epoch with timestamp $ts' \leq ts$.

Agreement follows easily from the protocol because p_ℓ sends the same value to all processes in the DECIDED message. Analogously, *integrity* follows from the protocol.

Termination is also easy to see because when p_ℓ is permanently correct and no process aborts the epoch, then every permanently correct process eventually receives a DECIDE message and ep-decides.

Finally, *abort behavior* is satisfied because the protocol returns *aborted* immediately and only if it has been aborted. □

Algorithm 3 Implements epoch with timestamp ets and leader ℓ with crash faults (Paxos/viewstamped replication), for $n > 2f$.

```
**Implements:**
    Epoch (ep).

**Uses:**
    Best-effort broadcast (beb);
    Perfect point-to-point links (pp2p).

upon $\langle$ init $\mid ets_0, \ell_0, state \rangle$ do
    $(ets, \ell) \leftarrow (ets_0, \ell_0)$
    $(valts, val) \leftarrow state$
    $tmpval \leftarrow \bot$
    $states \leftarrow [\bot]^n$; $accepted \leftarrow 0$

upon $\langle ep.propose \mid v \rangle$ do                                    // leader $p_\ell$ only
    $tmpval \leftarrow v$
    trigger $\langle beb.broadcast \mid [\text{READ}] \rangle$

upon $\langle beb.deliver \mid p_\ell, [\text{READ}] \rangle$ do
    trigger $\langle pp2p.send \mid p_\ell, [\text{STATE}, valts, val] \rangle$

upon $\langle pp2p.deliver \mid p_j, [\text{STATE}, ts, v] \rangle$ do          // leader $p_\ell$ only
    $states[j] \leftarrow [\text{STATE}, ts, v]$

upon $quorum\text{-}max_0(states) \neq \bot$ do                               // leader $p_\ell$ only
    $(ts, v) \leftarrow quorum\text{-}max_0(states)$
    if $v \neq \bot$ then $tmpval \leftarrow v$
    $states \leftarrow [\bot]^n$
    trigger $\langle beb.broadcast \mid [\text{WRITE}, tmpval] \rangle$

upon $\langle beb.deliver \mid p_\ell, [\text{WRITE}, v] \rangle$ do
    $(valts, val) \leftarrow (ets, v)$
    trigger $\langle pp2p.send \mid p_\ell, [\text{ACCEPT}] \rangle$

upon $\langle pp2p.deliver \mid p_j, [\text{ACCEPT}] \rangle$ do                // leader $p_\ell$ only
    $accepted \leftarrow accepted + 1$

upon $accepted > n/2$ do                                                  // leader $p_\ell$ only
    $accepted \leftarrow 0$
    trigger $\langle beb.broadcast \mid [\text{DECIDED}, tmpval] \rangle$

upon $\langle beb.deliver \mid p_j, [\text{DECIDED}, v] \rangle$
    trigger $\langle ep.decide \mid v \rangle$
    halt

upon $\langle ep.abort \rangle$ do
    trigger $\langle ep.aborted \mid (valts, val) \rangle$
    halt
```

5 Byzantine Consensus (PBFT)

This section presents the implementations of epoch-change and epoch in a system that is subject to Byzantine faults. When they are plugged into the leader-based consensus protocol, the result is very similar to the consensus protocol inherent in the PBFT protocol [5].

Implementing epoch-change. A protocol for epoch-change tolerating Byzantine faults is shown in Algorithm 4. Like Algorithm 2, which solves the same problem tolerating crash faults, it relies on an eventual leader detector primitive. But it is conceptually simpler than the epoch-change implementation with crash faults. Algorithm 4 outputs at all correct processes a sequence of timestamps that always increases by 1, and the leader of an epoch with timestamp ts is computed from ts by a function *leader*$(\cdot)$, defined by

$$\textit{leader}(ts) = \begin{cases} ts \bmod n & \text{if } ts \bmod n \neq 0 \\ n & \text{otherwise.} \end{cases}$$

Hence, the leader rotates in a round-robin fashion.

Algorithm 4 Implements epoch-change with Byzantine faults (code for p_i).

```
Implements:
    Epoch-change (ec).

Uses:
    Eventual leader detector (Ω);
    Auth. best-effort broadcast (abeb).

upon ⟨init⟩ do
    $ts \leftarrow 0$
    $nextts \leftarrow 0$
    $trusted \leftarrow 0$
    $newepoch \leftarrow [\text{FALSE}]^n$
    Initialize an instance of Eventual leader detector

upon ⟨Ω.trust | $\ell$⟩ do
    $trusted \leftarrow \ell$

upon $nextts = ts$ and $trusted \neq leader(ts)$ do
    $nextts \leftarrow ts + 1$
    trigger ⟨abeb.broadcast | [NEWEPOCH, $nextts$]⟩

upon ⟨abeb.deliver | $p_j$, [NEWEPOCH, $ts'$]⟩ such that $ts' = ts + 1$ do
    $newepoch[j] \leftarrow \text{TRUE}$

upon $nextts = ts$ and $|\{j \mid newepoch[j] = \text{TRUE}\}| > f$ do
    $nextts \leftarrow ts + 1$
    trigger ⟨abeb.broadcast | [NEWEPOCH, $nextts$]⟩

upon $nextts > ts$ and $|\{j \mid newepoch[j] = \text{TRUE}\}| > 2f$ do
    $ts \leftarrow nextts$
    $newepoch \leftarrow [\text{FALSE}]^n$
    trigger ⟨ec.startepoch | $ts, leader(ts)$⟩
```

The algorithm maintains a timestamp ts of the most recently started epoch and a timestamp $nextts$, which is equal to $ts + 1$ during the period when the process has sent a NEWEPOCH message but not yet started the epoch with timestamp $nextts$. Whenever the process observes that the leader of the current epoch is different from the process that it most recently trusted, the process begins to switch to the next epoch by sending a NEWEPOCH message. Alternatively, the process also begins to switch to the next epoch after receiving NEWEPOCH messages from $f + 1$ distinct processes. Once the process receives $2f + 1$ NEWEPOCH messages (from distinct processes) it starts the epoch.

Theorem 4. *Algorithm 4 implements epoch-change with f Byzantine faults for $n > 3f$.*

Proof. We first show monotonicity and consistency. It is obvious from the algorithm that the timestamps of two successive epochs started by a correct process increase by at least 1. Furthermore, the leader of an epoch is derived deterministically from its timestamp.

We call a process *good* when it remains correct forever. To show eventual accuracy, notice that every correct process sends a NEWEPOCH message for starting a new epoch whenever the leader of the current epoch is not the process that it trusts. Furthermore, there exists a time when Ω has caused every correct process to trust the same good process p_{ℓ^*} forever. Hence, eventually no good process sends any further NEWEPOCH messages. When all NEWEPOCH messages among correct processes have been delivered and the highest epoch started by a good process has timestamp ts^*, then this process has received at least $2f + 1$ NEWEPOCH messages with timestamp ts^*. Since at least $f + 1$ of those messages were sent by good processes, every good process has also sent a NEWEPOCH message with timestamp ts^* according to the protocol. Thus, every good process eventually receives at least $n - f > 2f$ NEWEPOCH messages with timestamp ts^* and starts the epoch with timestamp ts^* and no further epoch. □

Implementing epoch. Algorithm 5 implements one instance of epoch with timestamp ets and leader ℓ. As for the epoch protocol with crash faults, there may exist multiple instances with different timestamps in the system, but one process only runs one instance of epoch at a time in leader-based consensus. According to our protocol notation, this also means that ets is contained implicitly in every message sent by the epoch using the basic communication primitives.

Similar to Algorithm 3, an epoch consists of a *read phase* followed by a *write phase*. We say that a process *writes a value* v when it broadcasts a WRITE message containing v during the write phase.

The protocol is initialized with a $state$ value output by the epoch instance that the process ran previously. It contains a timestamp/value pair $(valts, val)$ with the value that the process received most recently in a Byzantine quorum of WRITE message in some epoch and a set $writeset$ of timestamp/value pairs with one entry for every value that this process has ever written and the timestamp of the most recent epoch where this occurred.

The read phase obtains the states from all processes to determine whether there exists a value that may already have been ep-decided. In the case of crash faults, it was sufficient that the leader alone computed this value and wrote it; here the leader might write a wrong value. Thus, every process must repeat the computation of the leader and write a value, to validate the choice of the leader.

The protocol starts by the leader broadcasting a READ message, which triggers every process to invoke a weak interactive consistency protocol and to wic-send a message $[\text{STATE}, valts, val, writeset]$ containing the state of the process.

The WIC primitive then wic-delivers a vector S of n entries, one for every process, in which an entry is either a STATE message or equal to $\perp$. Every non-$\perp$ entry contains a timestamp ts, a value v and a set of timestamp/value pairs, representing the writeset of the originating process.

Every process now determines whether a certain value v occurs in some writeset of an entry in S that originates from a correct process; since up to f entries in S may be from faulty processes, the process must filter them out first. When the writeset of more than f processes contains (ts, v) with timestamp ts or a higher timestamp than ts, then some correct process has written v in epoch ts or later. To capture this, we define a predicate *is-in-certified-writeset*(ts, v, S) to be TRUE whenever

$$\Big|\big\{j \mid S[j] = [\text{STATE}, \cdot, \cdot, ws'] \textbf{ and } \exists (ts', v') \in ws' \text{ such that } ts' \geq ts \wedge v' = v\big\}\Big| \; > \; f,$$

and FALSE otherwise.

Furthermore, every process needs to determine whether the STATE messages in S indicate that a certain timestamp/value pair (ts, v) occurs often enough among the non-$\perp$ entries of S so that a process may already have ep-decided v. Evidence for such a pair is given by an f-tolerant quorum maximum of S different from $\perp$. If such a pair exists, then this value must be written again during the epoch.

We say that S *binds* ts *to* v, denoted *binds*(S, ts, v), whenever S contains at least $n - f$ messages different from $\perp$ and satisfies

$$(ts, v) \; = \; \textit{quorum-max}_f(S) \;\; \textbf{and} \;\; \textit{is-in-certified-writeset}(ts, v, S) \; = \; \text{TRUE}.$$

If S contains at least $n - f$ messages different from $\perp$ but does not satisfy the condition, we say that S *binds* ts *to* $\perp$.

When S binds ts to some $v \neq \perp$, then the process must write v; otherwise, it writes the value from the leader p_ℓ, which it finds in $S[\ell]$. The process broadcasts a WRITE message with the value.

When a process has received more than $\frac{n+f}{2}$ WRITE messages from distinct processes containing the same value v, it sets its state to (ets, v) and broadcasts an ACCEPT message with v. When a process has received more than $\frac{n+f}{2}$ ACCEPT messages from distinct processes containing the same value v, it ep-decides v.

Remarks. The cost of an epoch when using the three-round implementation of weak interactive consistency described in Section 2.3 is as follows. The message complexity of an epoch is $O(n^2)$, and the communication complexity is $O(n^2bw)$, where b is a bound on the length of the proposal values and w is the maximal number of timestamp/value pairs that any correct process holds in the $writeset$ variable in its state.

When the epoch protocol is used together with Algorithm 4 to implement consensus using the leader-based approach of Algorithm 1, the protocol can be simplified in two ways.

First, the READ message can be omitted. Since the leader of the epoch sends READ immediately after it sends a NEWEPOCH message to all processes in Algorithm 4, every process could just send the STATE message upon initializing the epoch instance in reply to receiving the NEWEPOCH message.

Second, in the first epoch, with timestamp ets_0 and leader p_{ℓ_0}, the WIC primitive for reading the state of all processes may be skipped because all processes apart from the leader initially store the default state. Only p_{ℓ_0} needs to broadcast its state. One might think that it could use an authenticated best-effort broadcast for this purpose, but this does not work because it may be faulty and violate the integrity condition of WIC for the message from p_{ℓ_0}. Instead, the broadcast mechanism must at least ensure that when a correct process receives a value v from p_{ℓ_0}, then no correct process receives a value different from v. One can implement this with an extra round of echoing the value, as done in the broadcast primitive of Srikanth and Toueg [23], or any other primitive the implements consistent broadcast according to [4].

The protocol resulting from these two optimizations contains an initial message from the leader to all processes and two rounds of echoing the message among all processes. This communication pattern first appeared in Bracha's reliable broadcast protocol [3] and is also used during the "normal-case" operation of a view in the PBFT protocol.

Algorithm 5 Implements epoch with timestamp ets and leader ℓ with Byzantine faults (PBFT), for $n > 3f$.

```
**Implements:**
  Epoch (ep).

**Uses:**
  Auth. best-effort broadcast (abeb);
  Weak interactive consistency (wic);

**upon** $\langle \textit{init} \mid ets_0, \ell_0, state \rangle$ **do**
  $(ets, \ell) \leftarrow (ets_0, \ell_0)$
  $(valts, val, writeset) \leftarrow state$
  $written \leftarrow [\bot]^n$; $accepted \leftarrow [\bot]^n$

**upon** $\langle \textit{ep.propose} \mid v \rangle$ **do**                          // leader $p_\ell$ only
  **if** $val = \bot$ **then** $val \leftarrow v$
  **trigger** $\langle \textit{abeb.broadcast} \mid [\textsc{read}] \rangle$

**upon** $\langle \textit{abeb.deliver} \mid p_\ell, [\textsc{read}] \rangle$ **do**
  Initialize a new instance of Weak interactive consistency with leader $p_\ell$
  **trigger** $\langle \textit{wic.send} \mid [\textsc{state}, valts, val, writeset] \rangle$     // send / pre-prepare

**upon** $\langle \textit{wic.deliver} \mid states \rangle$
  $tmpval \leftarrow \bot$
  **if** $\exists v \neq \bot$ such that $\textit{binds}(states, ets, v)$ **then**
    $tmpval \leftarrow v$
  **else if** $\textit{binds}(states, ets, \bot)$ **and** $\exists v$ such that $states[\ell] = [\textsc{state}, \cdot, v, \cdot]$ **then**
    $tmpval \leftarrow v$
  **if** $tmpval \neq \bot$ **then**
    **if** $\exists ts$ such that $(ts, tmpval) \in writeset$ **then** $writeset \leftarrow writeset \setminus \{(ts, tmpval)\}$
    $writeset \leftarrow writeset \cup \{(ets, tmpval)\}$
    **trigger** $\langle \textit{abeb.broadcast} \mid [\textsc{write}, tmpval] \rangle$     // echo / prepare

**upon** $\langle \textit{abeb.deliver} \mid p_j, [\textsc{write}, v] \rangle$ **and** $written[j] = \bot$ **do**
  $written[j] \leftarrow v$

**upon** $\exists v$ such that $|\{j \mid written[j] = v\}| > \frac{n+f}{2}$ **do**
  $(valts, val) \leftarrow (ets, v)$
  $written \leftarrow [\bot]^n$
  **trigger** $\langle \textit{abeb.broadcast} \mid [\textsc{accept}, val] \rangle$     // ready / commit

**upon** $\langle \textit{abeb.deliver} \mid p_j, [\textsc{accept}, v] \rangle$ **and** $accepted[j] = \bot$ **do**
  $accepted[j] \leftarrow v$

**upon** $\exists v$ such that $|\{j \mid accepted[j] = v\}| > \frac{n+f}{2}$ **do**
  $accepted \leftarrow [\bot]^n$
  **trigger** $\langle \textit{ep.decide} \mid v \rangle$
  **halt**

**upon** $\langle \textit{ep.abort} \rangle$ **do**
  **trigger** $\langle \textit{ep.aborted} \mid (valts, val, writeset) \rangle$
  **halt**
```

Theorem 5. *Algorithm 5 implements epoch with timestamp ets and leader index ℓ with f Byzantine faults for $n > 3f$.*

Proof. We first establish the *lock-in* property of an epoch. Suppose some correct process has ep-decided v in an epoch with timestamp $ts' < ts$. The process only ep-decided after collecting ACCEPT messages containing v from more than $\frac{n+f}{2}$ processes; among the processes who sent those messages, there exists a set $\mathcal{A}$ of more than $\frac{n+f}{2} - f$ *correct* processes. According to the protocol, they all set their variables val to v and $valts$ to ts'.

The members of $\mathcal{A}$ only sent an ACCEPT message after collecting WRITE messages containing v from more than $\frac{n+f}{2}$ processes; among them, there exists a set $\mathcal{W}$ of more than $\frac{n+f}{2} - f$ *correct* processes. According to the protocol, they all added (ts', v) in their variable $writeset$.

Consider the next epoch with timestamp $ts^* > ets$, in which any correct process p_j wic-delivers $states$ such that *binds*$(states, ts^*, v^*)$ for some $v^* \neq \perp$. We claim that $v^* = v$. To see this, observe that no correct process has sent a WRITE message in any epoch between ts' and ts^*.

This means that no correct process has changed its $valts$, val, and $writeset$ variables. By the assumption on how a correct process passes the state of an epoch to the next one, every process in $\mathcal{A}$ starts epoch ts^* with state $(valts, val) = (ts', v)$. Furthermore, every process in $\mathcal{W}$ starts epoch ts^* with state variable $writeset$ that contains (ts', v). The integrity property of weak interactive consistency ensures that these state values are not modified in transit during WIC. Hence, the vector $states$ wic-delivered to p_j satisfies that (ts', v) is an f-tolerant quorum maximum of $states$ and that *is-in-certified-writeset*$(ts', v, states) =$ TRUE, as is easy to verify. Consequently, p_j writes v, and any other correct process who writes also writes v. This proves the above claim and implies that a correct process can only ep-decide v in epoch ts^*. Furthermore, the set of correct processes who set their variables val to v and $valts$ to a value at least as large as ts' when they abort epoch ts^* is now at least $\mathcal{A}$. Using the same reasoning, the set of correct processes whose $writeset$ variable contains (ts', v) is also at least $\mathcal{A}$. Continuing this argument until epoch ts establishes the *lock-in* property.

To show *validity*, assume that a correct process ep-decides v. It is obvious from the protocol that a correct process only ep-decides for the value v received in an ACCEPT message from a Byzantine quorum of processes and that any correct process only sends an ACCEPT message with v after receiving v in a WRITE message from a Byzantine quorum of processes. Moreover, any correct process only sends a WRITE message with v after receiving a vector $states$ that binds ts to v or after receiving $states$ that binds ts to $\perp$ and taking v from $states[\ell]$, which was wic-sent by p_ℓ. In the latter case, the validity property is satisfied. In the former case, we continue by backward induction in the sequence of epochs until we reach again an epoch where $states$ binds ts to $\perp$. This shows that v was ep-proposed by the leader in some epoch with timestamp $ts' \leq ts$.

For *agreement*, observe how any correct process that ep-decides v must have received more than $\frac{n+f}{2}$ ACCEPT messages with v. Since a correct process only sends one ACCEPT message in an epoch and since $n > 3f$, it is not possible that another correct process receives more than $\frac{n+f}{2}$ ACCEPT messages with a value different from v. The agreement property follows. *Integrity* is easy to see directly from the protocol.

To show *termination*, note that if p_ℓ is good, then the WIC primitive wic-delivers a vector with at least $n-t$ entries different from $\perp$ (recall that a process is *good* when it remains correct forever). Hence, every good process eventually assigns *tmpval* $\neq \perp$ and broadcasts a WRITE message containing some value. More precisely, all $n - f > \frac{n+f}{2}$ good processes write the same value v by the consistency of WIC. Hence, every good process eventually sends an ACCEPT message with v and every good process eventually decides because no further aborts occur.

Finally, *abort behavior* is satisfied because the protocol returns *aborted* immediately and only if it has been aborted. □

6 Related Work

Lamport's second description of Paxos [15] presents the protocol on a high level and in simple terms, without the formal overhead of the original paper [14].

De Prisco et al. [7] give a precise formulation of the crash-tolerant Paxos protocol in terms of the Clock General Timed Automaton model. They analyze the performance of Paxos both under normal behavior and when process crashes and timing failures occur.

Lampson [16] unifies the Paxos protocol, Disk Paxos [9], and the PBFT protocol with an abstract Paxos description. He structures the protocol into successive views, corresponding to the epochs of our work. Abstract Paxos is formulated with sometimes non-local predicates, which are instantiated by communication primitives corresponding to the three protocols. Due to the formal nature of the abstractions, this work is less suitable as an introduction to the Paxos family of protocols.

Boichat et al. [1, 2] were the first to identify a read/write register abstraction in Paxos, and to show how the Paxos protocol results from a combination of an *eventual register* primitive and an *eventual leader election* abstraction. Their register-based algorithm maintains the efficiency of the Paxos protocol.

Chockler and Malkhi [6] introduce the notion of a ranked register to solve consensus with the Paxos algorithm. They implement the abstraction with a collection of read-modify-write objects subject to faults, as they are available from disk storage devices, for example.

Malkhi et al. [18] further relax the conditions on the leader-election oracle Ω that is sufficient to implement Paxos. Their algorithm guarantees to elect a leader (and to reach agreement using Paxos) in absence of links that become eventually timely, only based on the assumption that one process can send messages and receives enough replies in time from a dynamic subset of the processes.

Li et al. [17] present the only other unified treatment for crash-tolerant and Byzantine-tolerant versions of Paxos, apart from this paper, that comes with actual implementations. Taking up the read/write register-based abstraction, their work extends it to the model with Byzantine faults and gives protocols to implement a *Paxos register* for both types of failures. Our model makes the epochs more prominently visible in contrast to their work; since epochs correspond to views in the PBFT protocol, our model is closer to the structure of PBFT and its sequence of alternating views and view changes.

The PBFT protocol exists in variations with and without public-key digital signatures. Our work does not rely on them for explaining the PBFT protocol, but the PBFT-like implementation based on the Paxos register uses signatures [17]. Since PBFT without signatures has become a landmark and has inspired many subsequent practical systems and protocols [11], all avoiding digital signatures for increased efficiency, our explanation covers their structure as well.

7 Conclusion

This paper describes Paxos-style consensus protocols tolerating crashes and Byzantine faults in a modular way. An abstract consensus protocol runs an epoch-change abstraction and a sequence of epoch abstractions, which can be implemented in both failure models.

Our formulation uses a deeper decomposition of the protocols compared to the previous works. We find it at the same time easier to understand and more faithful to the originals, because it illustrates the role of successive epochs, representing the numbered proposals [15] or ballots [14] in Paxos and the views in PBFT [5]. Furthermore, it uses a leader-election oracle also for the Byzantine case, which was not the case in previous works.

Acknowledgments

This work was done at the Distributed Programming Laboratory (LPD) at EPFL. I am grateful for their generous hospitality and their support.

References

[1] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, “Deconstructing Paxos,” *SIGACT News*, vol. 34, pp. 47–67, Mar. 2003.

[2] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, “Reconstructing Paxos,” *SIGACT News*, vol. 34, pp. 42–57, June 2003.

[3] G. Bracha, “Asynchronous Byzantine agreement protocols,” *Information and Computation*, vol. 75, pp. 130–143, 1987.

[4] C. Cachin, “State machine replication with Byzantine faults.” Available from `http://www.zurich.ibm.com/~cca/papers/byzrepl.pdf`, July 2009. A survey based on a talk given at the seminar *A 30-year perspective on replication*, held at Monte Verita, Switzerland (Nov. 2007).

[5] M. Castro and B. Liskov, “Practical Byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems*, vol. 20, pp. 398–461, Nov. 2002.

[6] G. Chockler and D. Malkhi, “Active disk Paxos with infinitely many processes,” *Distributed Computing*, vol. 18, no. 1, pp. 73–84, 2005.

[7] R. De Prisco, B. Lampson, and N. Lynch, “Revisiting the PAXOS algorithm,” *Theoretical Computer Science*, vol. 243, pp. 35–91, 2000.

[8] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[9] E. Gafni and L. Lamport, “Disk Paxos,” *Distributed Computing*, vol. 16, no. 1, pp. 1–20, 2003.

[10] O. Goldreich, *Foundations of Cryptography*, vol. I & II. Cambridge University Press, 2001–2004.

[11] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, “The next 700 BFT protocols,” Technical Report LPD-REPORT-2008-008, School of Computer & Communication Sciences, EPFL, 2009.

[12] R. Guerraoui and M. Raynal, “The Alpha of indulgent consensus,” *The Computer Journal*, vol. 50, no. 1, pp. 53–67, 2006.

[13] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer, 2006.

[14] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, May 1998.

[15] L. Lamport, “Paxos made simple,” *SIGACT News*, vol. 32, no. 4, pp. 51–58, 2001.

[16] B. Lampson, “The ABCD’s of Paxos,” in *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.

[17] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi, "The Paxos register," in *Proc. 26th Symposium on Reliable Distributed Systems (SRDS)*, pp. 114–126, 2007.

[18] D. Malkhi, F. Oprea, and L. Zhou, "Ω meets Paxos: Leader election and stability without eventual timely links," in *Proc. 19th International Conference on Distributed Computing (DISC)* (P. Fraigniaud, ed.), vol. 3724 of *Lecture Notes in Computer Science*, pp. 199–213, Springer, 2000.

[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1997.

[20] Z. Milosevic, M. Hutle, and A. Schiper, "Unifying Byzantine consensus algorithms with weak interactive consistency," in *Proc. OPODIS 2009 — 13th Intl. Conference on Principles of Distributed Systems*, Dec. 2009. To appear. Perliminary version available as Tech. Rep. LSR-REPORT-2008-007, EPFL, 2008.

[21] B. M. Oki and B. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[22] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.

[23] T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Distributed Computing*, vol. 2, pp. 80–94, 1987.