

Yet Another Visit to Paxos

Christian Cachin



Abstract—This paper presents a modular decomposition of 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* [1], a leader-based protocol for reaching agreement among a set of processes in the presence of faults. It was only discovered some 20 years ago. Meanwhile the algorithm has been deployed as a critical piece in several of the world’s biggest information systems [2], [3], [4]. 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 [5], [6], [7], [8], [9], [10], [11], [12], [13].

The purpose of this trip is pedagogical: we give a modular decomposition of the Paxos consensus protocol. The same algorithm also exists in the New World under the name of *viewstamped replication* [14], 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”) [15]. 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 [13], 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. Our description also yields a formulation of Byzantine consensus relying on an abstract eventual leader-detector oracle (Ω), and we exhibit two slightly different notions of Ω for the two failure models.

This text 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 [16], [17].

In summary, our contributions are:

- 1) A precise and modular description of Paxos-style consensus protocols tolerating crash faults and Byzantine processes.
- 2) An extension of the PBFT consensus protocol, in which the leader does not simply rotate over all processes, but is determined by an abstract leader-detector oracle.
- 3) The formulation of abstract epoch-change and epoch primitives, from which the Paxos and the PBFT consensus protocols can be derived.

The paper continues in the next section by introducing the model and prerequisites. Section 3 describes the two main abstractions and a consensus protocol based on them. Sections 4 and 5 show how to instantiate the abstractions to obtain the Paxos and PBFT protocols, respectively. Section 6 discusses related work.

2 MODEL AND ASSUMPTIONS

2.1 System Model

The system consists of n processes p_1, \dots, p_n . Up to f of them may *fail* at some time by exhibiting crash faults or by behaving arbitrarily (so-called *Byzantine* [18] 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

This is a revised version, dated 7 April 2011, of IBM Research Report RZ 3754.

Address: IBM Research - Zurich, Säumerstr. 4, CH-8803 Rüschlikon, Switzerland. Email: cca@zurich.ibm.com.

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 [19]. These notions differ slightly from the standard terminology used in the literature, but the change allows for a unified definition of consensus for 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 [21] 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).

2.2.1 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 this defines uniform consensus [20] through our notion of “correct” processes.²

2.2.2 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

1. The standard notion [20] of “correct” and “faulty” processes is static, i.e., a process that eventually crashes is never correct. This leads to the distinction between non-uniform and uniform definitions.

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 [20], 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.

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 .

2.2.3 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.

These are the standard majority quorums, but all our algorithms generalize to arbitrary quorum systems [22], [23].

2.3 Primitives

2.3.1 Basic abstractions

The protocols use several standard abstractions that encapsulate elementary message exchange between the processes, which are characterized in terms of *broadcast* and *deliver* events [20]. In the model with crash faults, a *perfect point-to-point link* (*pl*) 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.

We consider a dual notion in the model with Byzantine faults. Since faulty processes may tamper with messages sent over the network, the basic communication abstractions additionally guarantee message integrity. The interface of these abstractions contains an indication of the sender for every delivered message. Message integrity guarantees 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 .

An *authenticated best-effort broadcast* (*abeb*) abstraction now is a best-effort broadcast primitive that also guarantees message integrity. Authenticated communication can be implemented easily from point-to-point links with a message-authentication code (MAC) [24], 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 [25]. Ω periodically outputs an event $trust(j)$ at every process, which indicates that p_j is trusted to be leader. When a process receives this event, it henceforth trusts p_j (until the next $trust$ event or forever). In a model with crash faults, 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.

With Byzantine faults, Ω satisfies the same basic properties, but cannot be implemented alone from the timeliness of the responses by remote processes. A Byzantine leader, for instance, might answer promptly but wrongly; the notion of Byzantine leader detection is application-specific [26]. But we can exploit the following: If the leader should perform some action according to the application within known time bounds and fails to do so, then other processes may detect this as a failure. In an eventually synchronous system that has become stable, this means the process is indeed faulty. Therefore, Ω with Byzantine faults takes some input from the application (unlike Ω with crash faults). Specifically, every process may *complain* against the currently trusted leader through a local *complain* request. The primitive satisfies property (2) of Ω and instead of property (1), that (1a) it eventually elects a new leader after more than f correct processes have complained against the leader, and that (1b) it does not elect a new leader unless at least one *correct* process has complained against the leader.

2.4 Notation for Protocols

Protocols are presented in a modular way following [20]. Every primitive is defined by a protocol module that may be called by other protocol implementations. Modules interact asynchronously by exchanging events: A process may *trigger* a new event and react to an event *upon* being notified. 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 protocols that it *uses*. 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 by an identifier. Events are qualified by the respective module identifier. When an instance causes the basic communication primitives (pl , beb , $abeb$) 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 called

epoch-change and *epoch*, which are introduced here. 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

3.1.1 Epoch-change

The task of an *epoch-change* abstraction is to periodically output a $startepoch(ts, \ell)$ 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. Eventually the primitive must cease to start new epochs, and the last epoch started at every correct process must be the same; furthermore, the leader of this last epoch must remain correct. 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 leadership:** 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 correct forever.

When an *epoch-change* is initialized, the primitive locally makes available a default timestamp/leader-index pair, which is equal at all processes.

3.1.2 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 *eproposes* 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 which 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, and it must use the state of the most recently aborted epoch to initialize the next epoch instance.

Under these assumptions about invoking multiple epochs, an $epoch$ 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 $ts' \leq ts$.
- **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

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

Uses:

- Epoch-change (ec)
- Epoch (ep)
- Eventual leader detector (Ω)

upon $\langle init \rangle$ **do**

$val \leftarrow \perp$

$proposed \leftarrow \text{FALSE}$

$decided \leftarrow \text{FALSE}$

Initialize an instance of Epoch-change and obtain initial timestamp ets_0 and leader p_{ℓ_0}

Initialize a new instance of Epoch with timestamp ets_0 , leader p_{ℓ_0} , and state $(0, \perp)$

Start timer T

$(ets, \ell) \leftarrow (ets_0, \ell_0)$

upon $\langle c.propose \mid v \rangle$ **do**

$val \leftarrow v$

upon $\langle ec.startepoch \mid newts, new\ell \rangle$ **do**

trigger $\langle ep.abort \rangle$ for instance ets

wait for $\langle ep.aborted \mid state \rangle$ from instance ets

$(ets, \ell) \leftarrow (newts, new\ell)$

$proposed \leftarrow \text{FALSE}$

Initialize a new instance of Epoch with timestamp ets , leader p_ℓ , and state $state$

Start timer T

upon $(\ell = i) \wedge (val \neq \perp) \wedge (proposed = \text{FALSE})$ **do**

$proposed \leftarrow \text{TRUE}$

trigger $\langle ep.propose \mid val \rangle$ for instance ets

upon $\langle ep.decide \mid v \rangle$ **do**

if $decided = \text{FALSE}$ **then**

$decided \leftarrow \text{TRUE}$

trigger $\langle c.decide \mid v \rangle$

upon $\langle T.timeout \rangle$ **such that** $decided = \text{FALSE}$ **do**

trigger $\langle \Omega.complain \rangle$

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.

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 algorithm also starts a timer T whenever a new epoch begins. If the leader is not successful and the epoch does not ep-decide before the timer T expires, then the process $complains$ against the leader at the underlying leader detector Ω .

The timer is set to an estimate of how long the epoch should take before ep-deciding, which is increased after every unsuccessful epoch. (This mechanism is actually only necessary for the Byzantine model; we assume that Ω in the crash model simply ignores complaints.)

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 a point-to-point link abstraction or a 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: All conditions except for validity are the same for uniform consensus and for Byzantine consensus. But because we use the *weak validity* notion of Byzantine consensus, which assumes that all process are correct, we can also apply a common argument for validity in both failure models.

Validity. 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 condition of the epoch abstraction, 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 for a process that decides in epoch with timestamp ts^* .

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. This establishes the validity condition for decisions in all epochs.

Agreement. According to the discussion in the proof of validity, every decision of consensus can be attributed to the decision of a particular 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, which has a smaller timestamp, has aborted.

According to the eventual leadership property 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.

The complaint mechanism satisfies the requirements of Ω in the Byzantine model because if an epoch has started at more than f correct processes but takes longer than the timeout to ep-decide, then more than f correct processes eventually complain. This causes the election of a new leader. Furthermore, the Byzantine processes alone cannot cause Ω to elect a new leader once the system is stable, because no further correct process will complain. \square

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 [1] and in viewstamped replication [14].

4.1 Implementing epoch-change

The epoch-change protocol in Algorithm 2 relies on an eventual leader detector primitive and 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 broadcasts 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 (bit) complexity of $O(n)$ during stable

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

Uses:

- Eventual leader detector (Ω)
- Best-effort broadcast (beb)
- Perfect point-to-point links (pl)

```

upon  $\langle \text{init} \rangle$  do
   $trusted \leftarrow 0$ 
   $lastts \leftarrow 0$ 
   $ts \leftarrow i$ 
  Initialize Eventual leader detector  $\Omega$ 

upon  $\langle \Omega.trust \mid \ell \rangle$  do
   $trusted \leftarrow \ell$ 
  if  $\ell = i$  then
     $ts \leftarrow ts + n$ 
    trigger  $\langle \text{beb.broadcast} \mid [\text{NEWPOCH}, ts] \rangle$ 

upon  $\langle \text{beb.deliver} \mid p_\ell, [\text{NEWPOCH}, newts] \rangle$  do
  if  $\ell = trusted \wedge newts > lastts$  then
     $lastts \leftarrow newts$ 
    trigger  $\langle \text{ec.startepoch} \mid newts, \ell \rangle$ 
  else
    trigger  $\langle \text{pl.send} \mid p_\ell, [\text{NACK}] \rangle$ 

upon  $\langle \text{pl.deliver} \mid p_j, [\text{NACK}] \rangle$  do
  if  $trusted = i$  then
     $ts \leftarrow ts + n$ 
    trigger  $\langle \text{beb.broadcast} \mid [\text{NEWPOCH}, ts] \rangle$ 

```

periods. For this calculation and later ones, we assume that timestamps have constant length.

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 and compares it to the timestamp in every NEWPOCH message, 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 the n processes. Hence, no two distinct processes send a NEWPOCH message with the same timestamp value, demonstrating the consistency property.

The eventual leadership 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 NEWPOCH 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 NEWPOCH 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 NEWPOCH messages. Hence, p_t eventually sends a NEWPOCH 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 NEWPOCH 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 NEWPOCH message with timestamp tts^* to all processes, the epoch with timestamp tts^* is also the last epoch that every correct process starts. Note that NACK messages are crucial for establishing the eventual leadership condition. \square

4.2 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 $[x, \dots, x]$.

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 to 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. The leader tries to impose a decision value on the processes. The other processes witness the actions of the leader, should the leader fail, and they also witness actions of leaders in earlier epochs. Recall that an epoch instance is initialized with local state (in *state*), which the previous epoch instance returned to the process when it was aborted. The state contains a timestamp and a value. Passing this 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 p_ℓ to all processes. The goal is for p_ℓ to write its proposal value to all processes, who store the epoch timestamp and the value in their state and acknowledge this to p_ℓ . When p_ℓ receives enough acknowledgments, it ep-decides this value. But it may be that the leader of some previous epoch already ep-decided some value. To prevent that the epoch violates *lock-in*, p_ℓ must write the same value again. Hence, p_ℓ

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

Uses:
 Best-effort broadcast (beb)
 Perfect point-to-point links (pl)

upon $\langle \text{init} \mid \text{state} \rangle$ **do**
 $(valts, val) \leftarrow \text{state}$
 $tmpval \leftarrow \perp$
 $states \leftarrow [\perp]^n$
 $accepted \leftarrow 0$

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

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

upon $\langle \text{pl.deliver} \mid p_j, [\text{STATE}, ts, v] \rangle$ **do** // only p_ℓ
 $states[j] \leftarrow (ts, v)$

upon $|\{j \mid S[j] \neq \perp\}| > n/2$ **do** // only p_ℓ
 $(ts, v) \leftarrow \text{highest}(states)$
if $v \neq \perp$ **then**
 $tmpval \leftarrow v$
 $states \leftarrow [\perp]^n$
trigger $\langle \text{beb.broadcast} \mid [\text{WRITE}, tmpval] \rangle$

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

upon $\langle \text{pl.deliver} \mid p_j, [\text{ACCEPT}] \rangle$ **do** // only p_ℓ
 $accepted \leftarrow accepted + 1$

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

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

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

first reads the state of the processes by sending a READ message. 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. Process p_ℓ receives a quorum of STATE messages and chooses as its proposal value the value that comes with the highest timestamp, if one exists. This choice is encapsulated in a function *highest*. Then p_ℓ writes the chosen value to all processes with a WRITE message. The write succeeds when p_ℓ receives an ACCEPT message from a quorum of processes, indicating that they have stored the value locally. The leader now ep-decides the chosen

value and announces this in a DECIDED message to all processes; the processes that receive this ep-decide as well.

Since every two quorums overlap in one process, the leader is sure to read any value that may have been ep-decided in a previous epoch and to write it again. Quorums play a similar role as in replicated implementations of read/write registers.

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 Algorithm 3 is used together with Algorithm 2 to implement consensus using the leader-based approach according to Algorithm 1, the READ message can be omitted by making the following trivial modification. Every process just sends the STATE message after initializing the epoch instance, in reply to receiving the NEWEPOCH message. This works because the leader of the epoch sends READ immediately after it sends a NEWEPOCH message to all processes in Algorithm 2.

The resulting protocol closely resembles the consensus algorithm inherent in Paxos [1], [5]. The following table shows how its messages correspond to those in Lamport's *Paxos Synod* [1] and *Simple Paxos* [5] algorithms.

This protocol	Paxos Synod	Simple Paxos
NACK	–	inform-higher
NEWEPOCH	NextBallot	prepare-request
STATE	LastVote	prepare-response
WRITE	BeginBallot	accept-request
ACCEPT	Voted	accept-response
DECIDED	Success	inform-chosen

Here the READ message is subsumed by the NEWEPOCH message as explained, in Simple Paxos the *inform-higher* message sent by acceptor to a proposer informs about a higher proposal, and the *inform-chosen* message is sent by a distinguished learner with the chosen value.

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 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 $(val_{ts}, val) = (ts', v)$. 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 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 p_{ℓ} receives enough ACCEPT messages and broadcasts a DECIDED message. Every permanently correct process eventually receives this message and ep-decides.

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

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 [15]. In order to simplify the implementation of the epoch abstraction, we introduce also a *conditional collect* primitive.

5.1 Implementing epoch-change

We describe a protocol for epoch-change tolerating Byzantine faults. 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 index of the leader of an epoch with timestamp ts is computed from ts by a function $leader$, defined by

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

Hence, the leader rotates according to a predefined pattern.

The algorithm maintains the 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.

A correct process waits for NEWEPOCH messages from $f + 1$ processes before switching to a new epoch because the Byzantine processes alone must not be able to trigger an epoch-change without cause. At least one correct process that no longer trusts the leader of the current epoch is also needed for switching.

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

Uses:

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

upon $\langle init \rangle$ **do**

$ts \leftarrow 0$
 $nextts \leftarrow 0$
 $trusted \leftarrow 0$
 $newepoch \leftarrow [\perp]^n$
Initialize Eventual leader detector Ω

upon $\langle \Omega.trust \mid \ell \rangle$ **do**

$trusted \leftarrow \ell$

upon $nextts = ts \wedge trusted \neq leader(ts)$ **do**

$nextts \leftarrow ts + 1$
trigger $\langle abeb.broadcast \mid [NEWPOCH, nextts] \rangle$

upon $\langle abeb.deliver \mid p_j, [NEWPOCH, ts'] \rangle$

such that $ts' = ts + 1$ **do**
 $newepoch[j] \leftarrow \text{TRUE}$

upon $nextts = ts \wedge |\{j \mid newepoch[j] \neq \perp\}| > f$ **do**

$nextts \leftarrow ts + 1$
trigger $\langle abeb.broadcast \mid [NEWPOCH, nextts] \rangle$

upon $nextts > ts \wedge |\{j \mid newepoch[j] \neq \perp\}| > 2f$ **do**

$ts \leftarrow nextts$
 $newepoch \leftarrow [\perp]^n$
trigger $\langle ec.startepoch \mid ts, leader(ts) \rangle$

Remark. The specification of epoch-change permits more flexible implementations also in the Byzantine model, i.e., where the leader would not follow predefined pattern. We are not aware of any such solution, however.

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.

To show eventual leadership, and more precisely its first condition, 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. Call a process *good* when it remains correct forever. Then 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 more than $2f$ NEWEPOCH messages with timestamp ts^* by the protocol.

Since more than f 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 also $n - f > 2f$ NEWEPOCH messages with timestamp ts^* and starts the epoch with timestamp ts^* and no further epoch. The second condition in eventual leadership is evident because the index of the leader is computed by the *leader* function from the timestamp value. \square

5.2 Conditional collect

The purpose of a primitive for *conditional collect* (CC) is to collect information in the system, in the form of messages from all processes, in a consistent way. The abstraction is invoked at every process by an *input*(m) event with an input message m ; it outputs a vector M with n entries through an event *collected*(M) at every process, such that $M[i]$ is either equal to UNDEFINED or corresponds to the input message of p_i .

A conditional collect primitive is parametrized by a condition \mathcal{C} , defined on an n -vector of messages, and it should only output a collected vector that satisfies \mathcal{C} . The condition must be specified at the time when the primitive is initialized, in terms of an efficiently computable predicate on a vector of messages; the predicate is fixed and cannot be changed afterwards (for instance, it must not access variables that may concurrently be updated). Furthermore, every correct process must specify the same predicate.

The goal of a conditional collect primitive is to collect the same vector of messages at every correct process such that this vector satisfies \mathcal{C} . Naturally the correct processes must all input messages that will satisfy \mathcal{C} ; otherwise, this goal cannot be achieved. More precisely, we say that the correct processes input *compliant* messages when every correct process p_i inputs a message m_i such

that any n -vector M with $M[i] = m_i$ for every correct process p_i satisfies $\mathcal{C}(M)$.

Since the above goal may be difficult to reach in general, the primitive additionally identifies one special process p_{ℓ} , called the leader, and must achieve this goal only when the leader is correct. Specifically, conditional collect with condition \mathcal{C} satisfies:

- **Consistency:** If the leader remains correct forever, then every correct process collects the same M , and this M contains at least $n - f$ messages different from UNDEFINED.
- **Integrity:** If some correct process collects M with $M[j] \neq \text{UNDEFINED}$ for some j and p_j is correct up to this time, then p_j has input message $M[j]$.
- **Termination:** If all correct processes input compliant messages and the leader remains correct forever, then every permanently correct process eventually collects some M such that $\mathcal{C}(M)$ holds.

Regardless of whether the leader is correct, the integrity property demands that the primitive must not modify the input messages of the correct processes in transit. Note that with a faulty leader, some inputs may be omitted in the output vector of a correct process, and the outputs of two correct processes may differ.

The conditional collect primitive is inspired by the weak interactive consistency (WIC) primitive defined by Milosevic, Huttle, and Schiper [13], who used it to simplify Byzantine consensus protocols. Compared to WIC, conditional collect introduces a leader into the specification purely for ensuring termination. Although the leader does not directly appear in the interface of the primitive, the abstraction serves our goal of explaining multiple leader-based consensus protocols in a modular way.

Assuming the processes have access to a public-key digital signature scheme [19], a two-round protocol implements conditional in a straightforward way: (1) Every process sends its input m using a (suitably defined) authenticated point-to-point links primitive to the leader, (2) the leader collects at least $n - f$ messages with valid signatures in M such that they satisfy \mathcal{C} , (3) uses authenticated best-effort broadcast to disseminate M , and (4) every process verifies that all $M[j] \neq \text{UNDEFINED}$ contain a valid signature by p_j before outputting the collected M . This protocol involves only two rounds of messages.

Furthermore, a three-round implementation in the Byzantine model that uses only message authentication works as follows [15], [13].

- 1) The protocol starts when a process inputs m . The process then broadcasts a SEND message containing m to all processes using authenticated best-effort broadcast.
- 2) Whenever a process delivers over authenticated best-effort broadcast a SEND message with m_j from sender p_j (for the first time), it broadcasts a message $[\text{ECHO}, j, m_j]$ to all processes using authenticated best-effort broadcast.

- 3) Until the end of the protocol, every process continuously counts how many ECHO messages that it delivers (counting only those from distinct senders) with respect to every other process.
- 4) The leader p_ℓ initializes every entry in an n -vector M to UNDEFINED. As soon as p_ℓ delivers more than $2f$ messages $[\text{ECHO}, j, m_j]$ with the same m_j for some j from distinct senders, it sets $M[j] \leftarrow m_j$. The leader continues to collect messages like this until the vector M contains at least $n - f$ entries different from UNDEFINED and $\mathcal{C}(M)$ holds. Once this is the case, p_ℓ broadcasts the message $[\text{CHOICE}, M]$ to all processes using authenticated best-effort broadcast.
- 5) Finally, when a process p_i receives a message $[\text{CHOICE}, M_\ell]$ from p_ℓ and for every j such that $M_\ell[j] = m_j \neq \text{UNDEFINED}$, process p_i receives also more than f messages $[\text{ECHO}, j, m_j]$, then p_i outputs the collected vector M_ℓ . Note that this involves waiting for multiple messages that were sent in the second step (by all processes) and in the fourth step (by the leader).

It is easy to verify that this protocol implements conditional collect for $n > 3f$. The integrity property follows because for every collected M with $M[j] \neq \text{UNDEFINED}$ where p_j is correct, at least one correct process has sent an ECHO message containing $M[j]$ and thus, also p_j has input $M[j]$. Furthermore, if p_ℓ is correct forever, even when no faulty process sends any ECHO message, there are $n - f > 2f$ correct processes that do send ECHO messages. Since the correct processes input compliant messages, p_ℓ eventually broadcasts a CHOICE message with M that satisfies $\mathcal{C}(M)$ and with at least $n - f$ entries different from UNDEFINED. Every permanently correct process that receives this message is also guaranteed to receive f ECHO messages for every entry as required to make progress; hence, it collects M , as required by termination and consistency.

The protocol has message complexity $O(n^2)$ and communication complexity $O(n^2b)$, 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 $O(nb + n^2\kappa)$ [15].

5.3 Implementing epoch

Algorithm 5 implements one instance of epoch with timestamp ets and leader p_ℓ in the Byzantine model. It invokes a conditional collect primitive. 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 convention, 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 algorithm is initialized with a value $state$, output by the epoch consensus instance that the process ran previously. This value determines its local state and contains (1) a timestamp/value pair $(valts, val)$ with the value that the process received most recently in a Byzantine quorum of WRITE messages, during the epoch with timestamp $valts$, and (2) a set $writeset$ of timestamp/value pairs with one entry for every value that this process has ever written (where the timestamp denotes the most recent epoch in which the value was written).

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; with Byzantine processes the leader might write a wrong value. Thus, every process must repeat the computation of the leader and write a value, in order to validate the choice of the leader.

The algorithm starts by the leader broadcasting a READ message to all processes, which triggers every process to invoke conditional collect. Every process inputs a message $[\text{STATE}, valts, val, writeset]$ containing its state. The leader in conditional collect is the leader p_ℓ of the epoch.

The conditional collect primitive determines whether there exists a value (from an earlier epoch) that must be written during the write phase; if such a value exists, the read phase must identify it or conclude that no such value exists. To this end, we introduce a predicate $sound(M)$ on an n -vector M of STATE messages, to be used in the conditional collect primitive. An entry of M may be *defined* and contain a STATE message or may be *undefined* and contain UNDEFINED. In every defined entry, there is a timestamp ts , a value v , and a set of timestamp/value pairs, representing the writeset of the originating process. When process p_ℓ is correct, at least $n - f$ entries in the collected M are defined; otherwise, more than f entries may be undefined.

The predicate $sound$ is formulated in terms of two conditions. Together they determine whether the STATE messages in M indicate that a certain timestamp/value pair (ts, v) occurs often enough among the defined entries of M so that a process may already have ep-decided v in an earlier epoch. If yes, value v must be written.

The first condition gathers evidence for such a value, which consists of a Byzantine quorum of defined entries in M . A suitable timestamp/value pair is one with the *highest* timestamp among some Byzantine quorum of defined entries in M . If such a pair exists, then its value is a candidate for being written again in this epoch consensus instance. (However, it could also be a value from a forged entry of a Byzantine process; the second condition will filter out such values.)

Formally, we introduce a predicate

$quorumhighest(ts, v, M)$, which returns TRUE whenever the timestamp/value pair (ts, v) appears in an entry of M and contains the largest timestamp among some Byzantine quorum of entries in M . In other words, $quorumhighest(ts, v, M)$ is TRUE whenever $M[j] = [STATE, ts, v, ws]$ for some j and some ws and

$$|\{j \mid M[j] = [STATE, ts', v', ws'] \wedge (ts' < ts \vee (ts', v') = (ts, v))\}| > \frac{n+f}{2},$$

and FALSE otherwise.

The second condition determines whether a value v occurs in some writeset of an entry in M that originates from a correct process; since up to f entries in M may be from faulty processes, these must be filtered out. When the writeset of more than f processes contains (ts, v) with timestamp ts or a higher timestamp than ts , then v is *certified* and some correct process has written v in epoch ts or later. To capture this, we define a predicate $certifiedvalue(ts, v, M)$ to be TRUE whenever

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

and FALSE otherwise. As will become clear, a process never writes \perp ; therefore, the value \perp is never certified for any timestamp.

For a timestamp/value pair (ts, s) occurring in M , we say that M binds ts to v whenever $|\{j \mid M[j] \neq \perp\}| \geq n-f$ and

$$quorumhighest(ts, v, M) \wedge certifiedvalue(ts, v, M).$$

We abbreviate this by writing $binds(ts, v, M)$.

When $|\{j \mid M[j] \neq \perp\}| \geq n-f$ and the timestamps from a Byzantine quorum of entries in M are equal to 0 (the initial timestamp value), then we say that M is *unbound* and abbreviate this by writing $unbound(M)$.

Finally, the predicate $sound(M)$ is TRUE if and only if there exists a timestamp/value pair (ts, v) in M such that M binds ts to v or M is unbound, that is,

$$sound(M) \equiv (\exists (ts, v) \text{ such that } binds(ts, v, M)) \vee unbound(M).$$

Every correct process initializes the conditional collect primitive with this predicate $sound$.

We now return to the algorithm, where conditional collect outputs a vector $states$. As shown later, the inputs of the correct processes are compliant when they all input a STATE message to the conditional collect primitive. Thus, when the leader is correct, the primitive outputs a vector $states$ that satisfies $sound(states) = \text{TRUE}$. If $states$ binds ts to some $v \neq \perp$, then the process must write v ; otherwise, $states$ is unbound and the process writes the value from the leader p_ℓ , which it finds in $states[\ell]$. The process sends a WRITE message to all processes with the value. In case $sound(states) = \text{FALSE}$, the leader must be faulty and the process does not take any further action in this epoch.

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 .

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

Uses:

Auth. best-effort broadcast (abeb)
Conditional collect (cc)

upon $\langle init \mid state \rangle$ **do**

$(valts, val, writeset) \leftarrow state$
 $written \leftarrow [\perp]^n$
 $accepted \leftarrow [\perp]^n$

upon $\langle ep.propose \mid v \rangle$ **do**

// only p_ℓ

if $val = \perp$ **then** $val \leftarrow v$

trigger $\langle abeb.broadcast \mid [READ] \rangle$

upon $\langle abeb.deliver \mid p_j, [READ] \rangle$ **such that** $j = \ell$ **do**

Initialize a new instance of Conditional collect
with leader p_ℓ and predicate $sound$
trigger $\langle cc.input \mid [STATE, valts, val, writeset] \rangle$

upon $\langle cc.collected \mid states \rangle$ **do**

// note that $states[j] = [STATE, ts, v, ws]$ or
 $states[j] = \text{UNDEFINED}$

$tmpval \leftarrow \perp$

if $\exists ts \geq 0, v \neq \perp$ such that $binds(ts, v, states)$ **then**
 $tmpval \leftarrow v$

else if $\exists v \neq \perp$ such that $unbound(states)$

$\wedge states[\ell] = [STATE, \cdot, v, \cdot]$ **then**

$tmpval \leftarrow v$

if $tmpval \neq \perp$ **then**

if $\exists ts$ such that $(ts, tmpval) \in writeset$ **then**

$writeset \leftarrow writeset \setminus \{(ts, tmpval)\}$

$writeset \leftarrow writeset \cup \{(ets, tmpval)\}$

trigger $\langle abeb.broadcast \mid [WRITE, tmpval] \rangle$

upon $\langle abeb.deliver \mid p_j, [WRITE, v] \rangle$ **do**

$written[j] \leftarrow v$

upon $\exists v$ such that $|\{j \mid written[j] = v\}| > \frac{n+f}{2}$ **do**

$(valts, val) \leftarrow (ets, v)$

$written \leftarrow [\perp]^n$

trigger $\langle abeb.broadcast \mid [ACCEPT, val] \rangle$

upon $\langle abeb.deliver \mid p_j, [ACCEPT, v] \rangle$ **do**

$accepted[j] \leftarrow v$

upon $\exists v$ such that $|\{j \mid accepted[j] = v\}| > \frac{n+f}{2}$ **do**

$accepted \leftarrow [\perp]^n$

trigger $\langle ep.decide \mid v \rangle$

halt

upon $\langle ep.abort \rangle$ **do**

trigger $\langle ep.aborted \mid (valts, val, writeset) \rangle$

halt

Remarks. The cost of an epoch when using the three-round implementation of conditional collect described previously 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 simply inputs the STATE message to conditional collect after initializing the epoch instance, in reply to receiving the NEWEPOCH message.

Second, in the first epoch the conditional collect primitive for reading the state of all processes may be skipped because all processes apart from the leader initially store the default state. Only the initial leader needs to send its state using authenticated best-effort broadcast.

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 [27] and is also used during the “normal-case” operation of a view in the PBFT protocol.

The resulting protocol closely resembles the Byzantine consensus algorithm inherent in PBFT [15]. The following table shows how its messages correspond to those in Bracha’s broadcast and PBFT. (The READ message is again subsumed by the NEWEPOCH message. The *view-change-ack* and *new-view* messages of PBFT are subsumed by our CC primitive.)

This protocol	Bracha	PBFT
NEWEPOCH	–	view-change
STATE	initial	pre-prepare
WRITE	echo	prepare
ACCEPT	ready	commit

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 that sent those messages, there exists a set \mathcal{A} of more than $\frac{n+f}{2} - f > f$ 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 these processes, there exists a set \mathcal{W} of more than $\frac{n+f}{2} - f > f$ correct processes. According to the protocol, they all added (ts', v) to their variable *writeset*.

Consider the next epoch instance with timestamp $ts^* > ets$, in which any correct process p_j collects *states* from conditional collect (CC) 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 about how a correct process passes the state from one epoch to the next, every process in \mathcal{A} starts epoch ts^* with its state containing $(valts, val) = (ts', v)$. Furthermore, every process in \mathcal{W} starts epoch ts^* with a variable *writeset* that contains (ts', v) . The integrity property of CC ensures that these state values are not modified in transit by the primitive. Hence, the vector *states* output to p_j satisfies that $quorumhighest(ts', v, states) = \text{TRUE}$ because the state from at least one member of \mathcal{A} is contained in every Byzantine quorum. Furthermore, $certifiedvalue(ts', v, states) = \text{TRUE}$ because the *writesets* of all processes in \mathcal{W} include (ts', v) .

Consequently, p_j writes v , and any other correct process that 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 that set *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 in two cases: either (1) after collecting a vector *states* that binds ts to v or (2) after collecting *states* that is unbound and taking v from $states[\ell]$, which was input by p_ℓ . In case (2), the validity property is satisfied. In case (1), we continue by applying the same argument inductively, backward in the sequence of epochs, until we reach an epoch where *states* is unbound; in that epoch, the reasoning for case (2) applies. This shows that v was ep-proposed by the leader in some epoch with timestamp $ts' \leq ts$.

For the *agreement* condition, 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 as $n >$

$3f$, 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. The *integrity* property is easy to see directly from the protocol.

To show *termination*, note that all correct processes input compliant STATE messages to the CC primitive, and therefore $\text{sound}(M) = \text{TRUE}$ when at least $n - f$ entries of M are those STATE messages. To see this, suppose the state of a correct process contains a pair (ts, v) ; the pair is either $(0, \perp)$ or was assigned after receiving a Byzantine quorum of WRITE messages in epoch ts . In the latter case, no output vector M containing this pair could be unbound. Since $\frac{n+f}{2} > 2f$, it follows from the protocol that v is also in the writeset of more than f correct processes. Consider now the pair (ts', v') with the largest timestamp $ts' \geq ts$ held by a correct process. This pair has the maximum timestamp in the state of a Byzantine quorum of correct processes. In addition, the writesets of more than f correct processes contain (ts', v') . Hence, conditional collect with a correct leader may obtain a vector M such that $\text{binds}(ts', v', M)$. Alternatively, if a Byzantine quorum of correct processes input a state with timestamp 0, the algorithm can find an unbound M . Hence, conditional collect eventually outputs M such that $\text{sound}(M)$ holds.

If p_ℓ is good, every correct process collects the same vector $states$ with at least $n - t$ entries different from UNDEFINED by the consistency property of CC (recall that a process is *good* when it remains correct forever). Hence, every good process eventually assigns $\text{tmpval} \neq \perp$ and broadcasts a WRITE message containing some value tmpval . More precisely, all $n - f > \frac{n+f}{2}$ good processes write the same tmpval by the consistency property of CC. Hence, every good process eventually broadcasts an ACCEPT message with tmpval and every good process eventually ep-decides because no further aborts occur.

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

6 RELATED WORK

Lamport's second description of Paxos [5] presents the protocol on a high level and in simple terms, without the formal overhead of the original paper [1]. This version, called *Simple Paxos*, further separates the processes by their role and by the message(s) they send: a process can be a proposer, an acceptor, or a learner. In the previous descriptions, these roles are implemented by the same global set of n processes. For ease of exposition, we maintain a global set of processes here.

De Prisco et al. [6] 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 [7] unifies the Paxos protocol, Disk Paxos [28], and the PBFT protocol through a set of

abstract conditions on Paxos algorithms. He structures the protocol into successive views, corresponding to the epochs of our work. The conditions are formulated through predicates, which are sometimes not local. These are then instantiated by communication primitives corresponding to the three protocols. Due to the formal and non-constructive nature of the work, it is less suitable as an introduction to the Paxos family of protocols.

Boichat et al. [8], [9] 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. Guerraoui and Raynal [11] extend this deconstruction and apply it to other models. Our work is based on a version of this algorithm [20], which uses a notion of abortable consensus.

Several versions of the Paxos protocol and related algorithms permit that correct processes crash and recover again later. Through logging critical data in stable storage, they may resume and participate again in the protocol. For practical deployment this feature is very important. But we have left it out in order to better expose the algorithmic mechanism. Allowing processes to crash and recover can still be added to our formulations through generic methods [20].

Chockler and Malkhi [29] 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. [30] 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 the 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. [12] present the first unified treatment (before this paper) of crash-tolerant and Byzantine-tolerant versions of Paxos, which 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 [12]. Since PBFT without signatures has become a landmark and has inspired many subsequent practical systems and protocols [31], all avoiding digital signatures for increased efficiency, our explanation cov-

ers their structure as well.

A recent paper of Rütli, Milosevic, and Schiper [32] shares our goal of presenting multiple consensus algorithms in one formalism. It covers not only the Paxos algorithm but also its “fast” variants in both failure models. Since the paper uses a round model of communication, the formulation of the algorithms is different. The assumptions required for the algorithms seem stronger: for instance, some critical rounds of the corresponding protocols in [32] have to satisfy a predicate (\mathcal{P}_{good}) that ensures all correct processes to receive every message sent by a correct process. No such assumption is used here (and neither in the original Paxos and PBFT protocols). The formulation in the round-model appears to be more abstract and further away from the original Paxos and PBFT descriptions.

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 [5] or ballots [1] in Paxos and the views in PBFT [15]. Furthermore, it uses a leader-election oracle also for the Byzantine case, which was not the case in previous works.

ACKNOWLEDGMENTS

I thank Hagit Attiya, Dan Dobre, Seth Gilbert, Rachid Guerraoui, Thomas Locher, Zarko Milosevic, André Schiper, Marco Serafini, and Marko Vukolić for interesting discussions and helpful comments. This work was done at the Distributed Programming Laboratory (LPD) at EPFL. I am grateful for the generous hospitality and the support that I enjoyed.

REFERENCES

- [1] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.
- [2] M. Burrows, “The Chubby lock service for loosely-coupled distributed systems,” in *Proc. 7th Symp. Operating Systems Design and Implementation (OSDI)*, 2006.
- [3] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: An engineering perspective,” in *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, 2007, pp. 398–407.
- [4] M. Isard, “Autopilot: Automatic data center management,” *Operating Systems Review*, vol. 41, pp. 60–67, Apr. 2007.
- [5] L. Lamport, “Paxos made simple,” *SIGACT News*, vol. 32, no. 4, pp. 51–58, 2001.
- [6] R. De Prisco, B. Lampson, and N. Lynch, “Revisiting the PAXOS algorithm,” *Theoretical Computer Science*, vol. 243, pp. 35–91, 2000.
- [7] B. Lampson, “The ABCD’s of Paxos,” in *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.
- [8] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, “Deconstructing Paxos,” *SIGACT News*, vol. 34, no. 1, pp. 47–67, Mar. 2003.
- [9] —, “Reconstructing Paxos,” *SIGACT News*, vol. 34, no. 2, pp. 42–57, Jun. 2003.
- [10] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, “Byzantine disk Paxos: Optimal resilience with Byzantine shared memory,” *Distributed Computing*, vol. 18, no. 5, pp. 387–408, 2006.
- [11] R. Guerraoui and M. Raynal, “The Alpha of indulgent consensus,” *The Computer Journal*, vol. 50, no. 1, pp. 53–67, 2006.
- [12] H. C. Li, A. Clement, A. S. Aiyer, and L. Alvisi, “The Paxos register,” in *Proc. 26th Symposium on Reliable Distributed Systems (SRDS)*, 2007, pp. 114–126.
- [13] Z. Milosevic, M. Hutle, and A. Schiper, “Unifying Byzantine consensus algorithms with weak interactive consistency,” in *Proc. 13th Conference on Principles of Distributed Systems (OPODIS)*, ser. Lecture Notes in Computer Science, T. F. Abdelzaher, M. Raynal, and N. Santoro, Eds., vol. 5923. Springer, 2009, pp. 300–314.
- [14] 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, pp. 8–17.
- [15] M. Castro and B. Liskov, “Practical Byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.
- [16] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.
- [17] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication: Theory and Practice*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 5959.
- [18] M. Pease, R. Shostak, and L. Lamport, “Reaching agreement in the presence of faults,” *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.
- [19] O. Goldreich, *Foundations of Cryptography*. Cambridge University Press, 2001–2004, vol. I & II.
- [20] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer, 2006.
- [21] 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.
- [22] M. Naor and A. Wool, “The load, capacity and availability of quorum systems,” *SIAM Journal on Computing*, vol. 27, no. 2, pp. 423–447, 1998.
- [23] D. Malkhi and M. K. Reiter, “Byzantine quorum systems,” *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.
- [24] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1997.
- [25] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.
- [26] A. Doudou, B. Garbinato, and R. Guerraoui, “Tolerating arbitrary failures with state machine replication,” in *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, H. B. Diab and A. Y. Zomaya, Eds. Wiley, 2005.
- [27] G. Bracha, “Asynchronous Byzantine agreement protocols,” *Information and Computation*, vol. 75, pp. 130–143, 1987.
- [28] E. Gafni and L. Lamport, “Disk Paxos,” *Distributed Computing*, vol. 16, no. 1, pp. 1–20, 2003.
- [29] G. Chockler and D. Malkhi, “Active disk Paxos with infinitely many processes,” *Distributed Computing*, vol. 18, no. 1, pp. 73–84, 2005.
- [30] 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)*, ser. Lecture Notes in Computer Science, P. Fraigniaud, Ed., vol. 3724. Springer, 2000, pp. 199–213.
- [31] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, “The next 700 BFT protocols,” in *Proc. 5th European Conference on Computer Systems (EuroSys)*, 2010, pp. 363–376.
- [32] O. Rütli, Z. Milosevic, and A. Schiper, “Generic construction of consensus algorithms for benign and Byzantine faults,” in *Proc. 40th International Conference on Dependable Systems and Networks (DSN-DCCS)*, 2010.