Byzantine Consensus

Mohsen Lesani

(Strong) Byzantine Consensus

Module:

- ByzantineConsensus, instance bc.

Events:

- Request: < propose(v) >:
  Proposes value v.
- Indication: < decide(v) >:
  Outputs the decided value v.

(Strong) Byzantine Consensus

Properties

- **WBC1**: *Termination*:
  Every correct process eventually decides some value.
- **BC2: *Strong Validity*:**
  A correct process may only decide a value that was proposed by some **correct** process or the special value □. Further, if all correct processes propose the same value v, then no correct process decides a value different from v.
- **WBC3**: *Integrity*:
  No correct process decides twice.
- **WBC4**: *Agreement*:
  No two correct processes decide differently.

The only way to know that a value is from a correct process is that we show that it is from f+1 processes. That's why when we cannot, we have to decide the special □ value.

The total order broadcast abstraction that uses consensus first has a round where processes exchange their messages so that they propose the same value.

Weak Byzantine Consensus

Module:

- WeakByzantineConsensus, instance wbc.

Events:

- Request: < propose(v) >:
  Proposes value v.
- Indication: < decide(v) >:
  Outputs a decided value v.

Weak Byzantine Consensus

Properties

- **WBC1**: *Termination*:
  Every correct process eventually decides some value.
- **WBC2: *Weak validity*:**
  A correct process may only decide a value that was proposed by some process.
- **WBC3**: *Integrity*:
  No correct process decides twice.
- **WBC4**: *Agreement*:
  No two correct processes decide differently.

From Weak to Strong Consensus

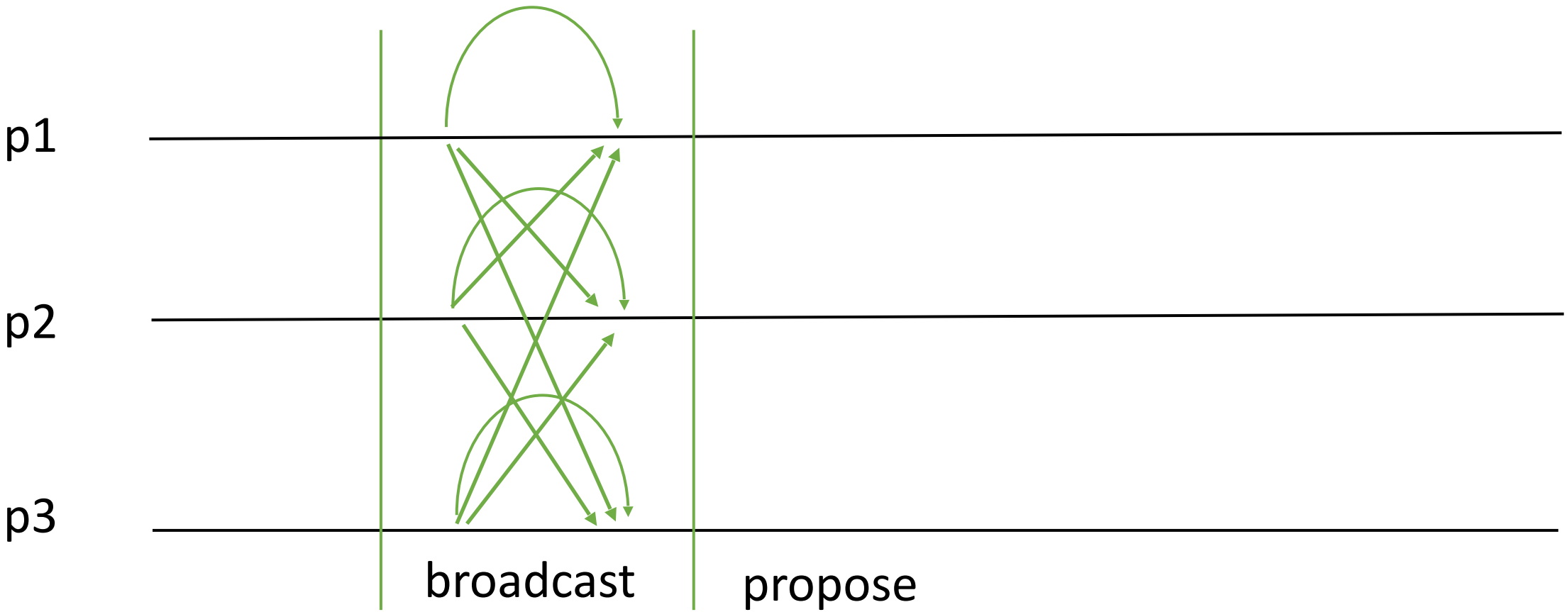
Idea:

We want only a value from a correct process to be decided. Weak consensus can decide a value proposed from a Byzantine process.

A process proposes a value to week consensus only if it can get the same value from more than f processes, and collect signatures from them. There is at least one correct process in that set. Otherwise, the process proposes the special □ value.

When weak consensus decides a value, we check that it has either f signatures, or it is the □ value.

From Weak to Strong Consensus



Collect a value v from more than f processes with signatures.

Validated Consensus: Check Signatures.

Validating the Proposal

A predicate on values.
Update the weak consensus protocol so that a process that receives a value that does not satisfy the predicate halts.

Here, a proposal is valid if it is signed by at least f processes or it is the special □ value.

```
function valid-proposal( (v, Σ) )
    if  #({p ∈ π | verify-sig(p, v, Σ[p]) = true}) > f  then
        return true;
    else if v = □ then
        return true
    else
        return false
```

From Weak to Strong Consensus

Implements:

ByzantineConsensus, instance bc.

Uses:

AuthPerfectPointToPointLinks, instance al

WeakByzantineConsensus (Validated), instance wc

upon event < init > **do**

proposals := $[\perp]^N$

$\Sigma := [\perp]^N$

From Weak to Strong Consensus

upon event < propose(v) > **do**
    σ := sign(self, v)
    forall q $\in \pi$ **do**
        trigger < al, send(q, Proposal(v, σ)) >

upon event < al, deliver(p, Proposal(v, σ)) > **do**
    if proposals[p] = $\perp \wedge$ verify-sig(p, v, σ) **then**
        proposals[p] := v
        Σ[p] := σ

From Weak to Strong Consensus

upon exists $v \neq \perp$ such that $\#(\{p \in \pi \mid proposals[p] = v\}) > f$ **do**
    proposals := $[\perp]^N$
    trigger < wc, propose((v, Σ)) >

upon ($\#proposals \geq N - f$) **do**
    proposals := $[\perp]^N$
    trigger < wc, propose(($\square$, Σ)) >

upon event < wc, decide((v', Σ')) > **do**
    trigger < decide(v') >

Collect signatures from more than f processes for the same value and propose it.

When there is no hope that it happens, propose the special $\square$ value. The remaining f processes may be Byzantine and may never respond.

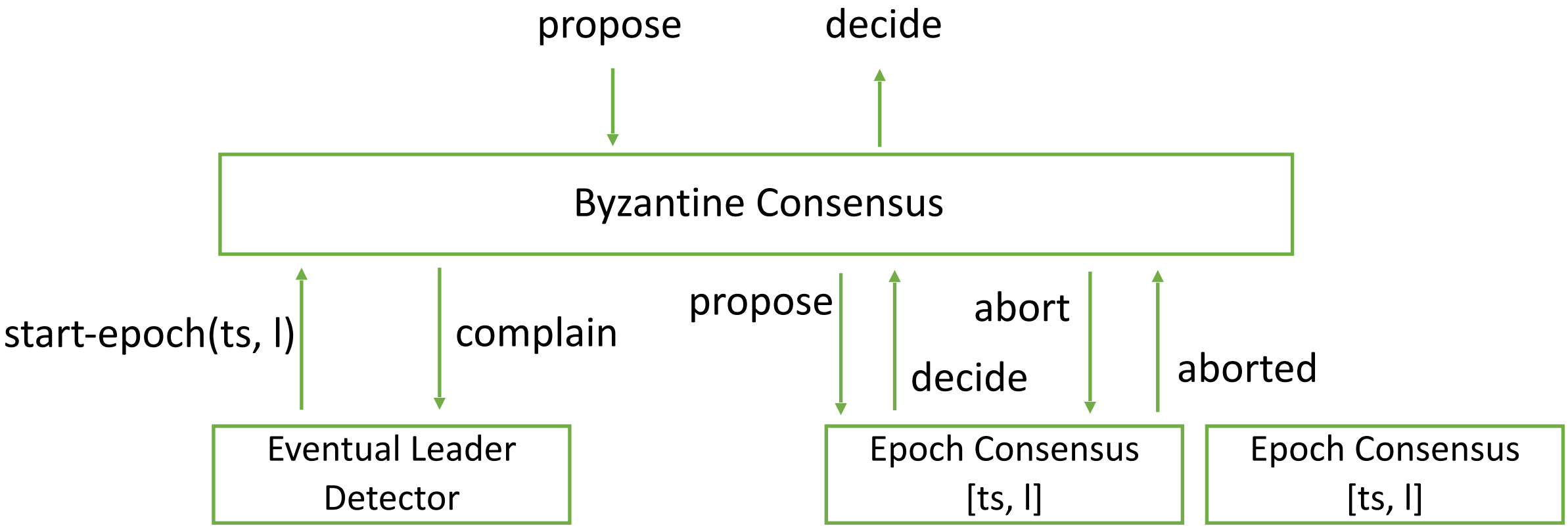
From Weak to Strong Validity

Because of validation of signatures from f+1 processes, if a **Byzantine process** wants to propose a value that is accepted, it must propose a value that was initially proposed by at least one correct process (or propose □).

By weak validity of weak consensus, a correct process may decide only a value proposed by some process.

Thus, a correct process may finally decide only a value that was proposed by some correct process or the special value □.

Leader-driven Consensus



Similar to the protocol for the fail-noisy environment

Byzantine Eventual Leader Detector

- We cannot rely on the timeliness of simple responses for detecting arbitrary faults. A Byzantine process can be responsive to a process and not the other.
- If the leader performs wrongly or exceeds the allocated time before reaching the goal, then other processes detect this and complain to their local leader detector component.
- The leader detector component gets input from the higher-level Byzantine consensus component as complain requests.

upon event (timeout)
  trigger <bld, complain(l) >

Eventual Byzantine Leader Detector

Module:

ByzantineLeaderDetector, instance bld.

Events:

- **Request**:
  < complain(p) >
  Receives a complaint about process p.
- **Indication**:
  < trust(p) >
  Indicates that process p is trusted to be leader.

Eventual Byzantine Leader Detector

Properties:

- **BLD1**: *Eventual succession*:
  If more than f correct processes (that trust some process p) complain about p, then every correct process eventually trusts a different process than p.

- **BLD2**: *Putsch resistance*:
  A correct process does not trust a new leader unless at least one correct process has complained against the current leader.

- **BLD3**: *Eventual agreement:*
  There is a time after which no two correct processes trust different processes.

Eventual Byzantine Leader Detector

- By properties 1 and 3, every correct process eventually trusts some process that appears to perform its task in the high-level component.
- In contrast to the crash model, one cannot require that every correct process eventually trusts a correct process because a Byzantine process may behave just like a correct process.

Rotating Byzantine Leader Detector

Idea:

- Round robin: the round number deterministically derives the leader of the round.
- A process considers complain about the current leader seriously only if it receives a complain from more than f processes. It then broadcasts a complain itself if it has not already. This amplifies the complain.
- A process moves to the next round and its leader only if it receives the complain from more than 2f processes. This ensures that after it moves to the next round, there are enough correct processes complaining that can push all correct processes to the next round.
- This is similar to the last phase of Bracha Byzantine broadcast: the complain message is like the ready message.
- It is assumed that $n > 3f$.

Protocol: Rotating Byzantine Leader Detector

Implements:
ByzantineLeaderDetector, instance bld.

Uses:
AuthPerfectPointToPointLinks, instance al.

upon event < init > **do**
round := 1
complains := $[\perp]^N$
complained := **false**
trigger < trust(leader(round)) >

upon event < complain(p) > such that
p = leader(round) and complained = **false do**
complained := **true**
forall $q \in \pi$ **do**
trigger < al, send(q, Complain(round)) >

A complain from the higher-level component.

Protocol: Rotating Byzantine leader detector

upon event < al, deliver(p, Complain(r)) > such that
    r = round and complains[p] = ⊥ **do**
  complains[p] := **true**
  if #(complains) > f ∧ complained = **false** **then**
    complained := **true**
    forall q ∈ π **do**
      trigger < al, send(q, Complain(round)) >
  else if #(complains) > 2f **then**
    round := round + 1
    complains := $[\bot]^N$
    complained := **false**
    trigger < trust(leader(round)) >

Byzantine Epoch-Change

Module:

- **Name**: ByzantineEpochChange, instance bec.

Events:

- **Indication**: < start-epoch(ts, l) >
  Starts the epoch identified by timestamp ts with leader l.

Properties:

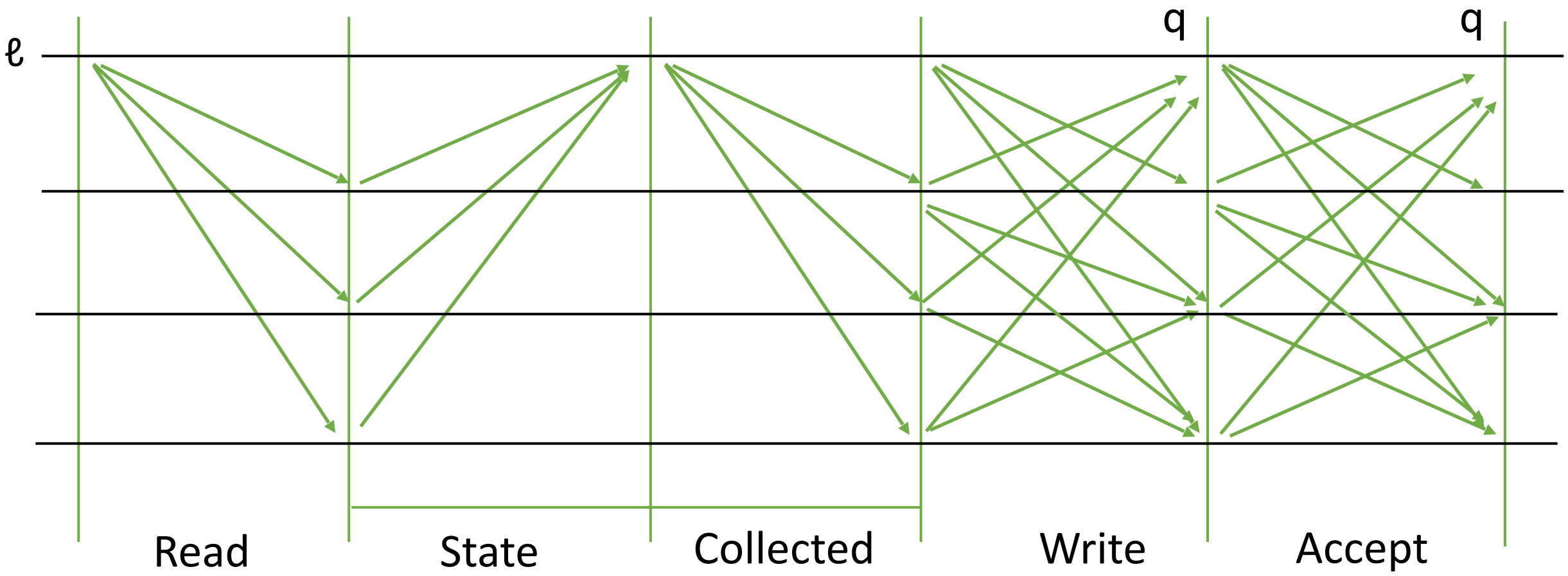
- **EC1:** Monotonicity:
  If a correct process starts an epoch (ts, l) and later starts an epoch (ts', l') then ts' > ts.
- **EC2**: Consistency:
  If a correct process starts an epoch (ts, l) and another correct process starts an epoch (ts, l') then l = l'.
- **EC3**: Eventual Leadership:
  There is a time after which every correct process has started some epoch (ts, l) and starts no further epoch, and the process l is correct.

It does not lead to f+1 complaints

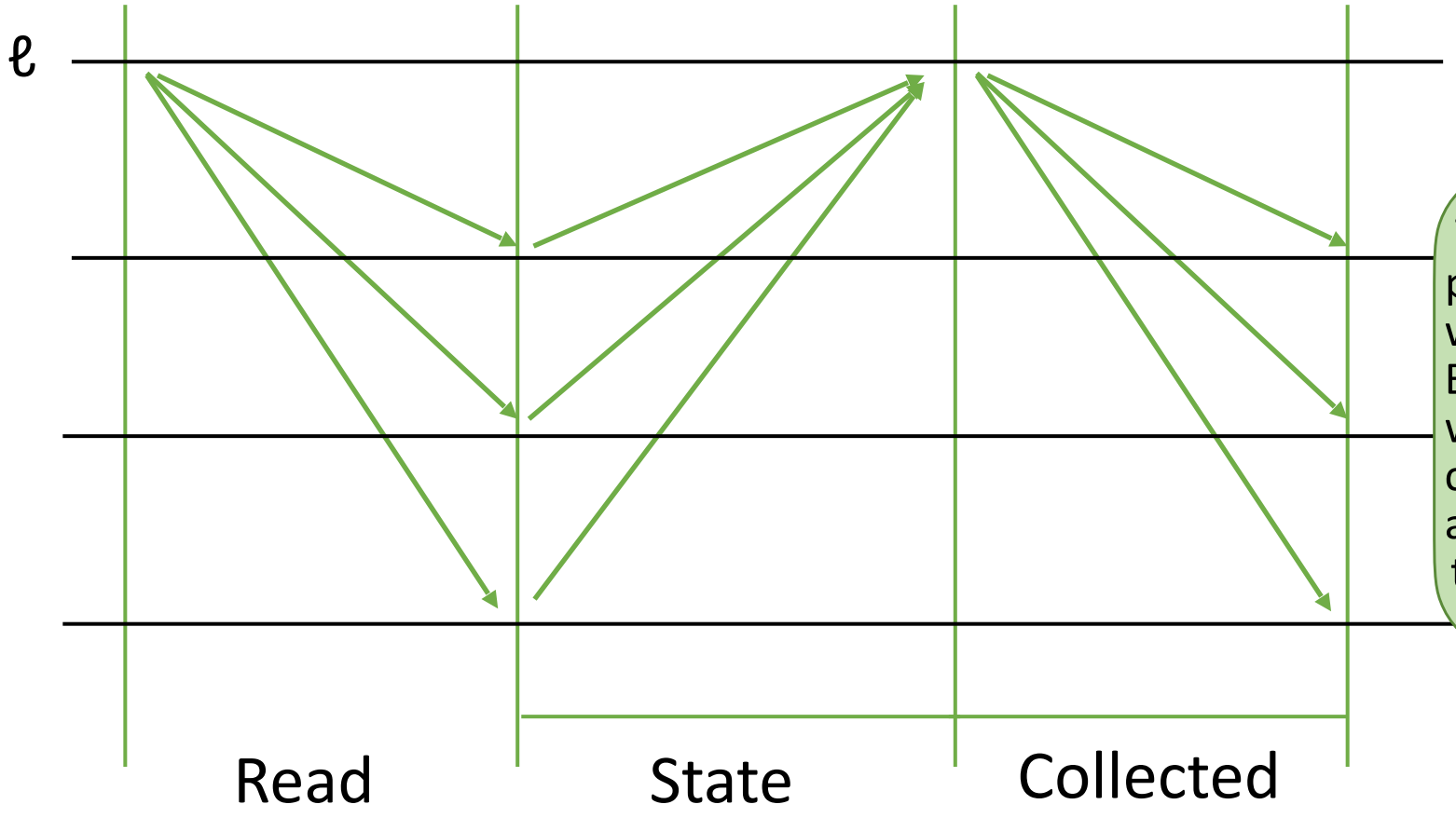
Protocol: Byzantine Epoch-Change

The same as the previous protocol. Just output the round number as the timestamp in addition to the leader.

Byzantine Epoch Consensus



Conditional Collect



Conditional Collect

The Leader should read the state of all processes and determine the value to be written, and send it to all processes to write. But processes cannot trust a single sender with the value that it sends. Thus, the leader collects signed values and forwards them to all processes. They themselves determine the value from the collection.

Conditional Collect

A leader collects messages from others such that the collection satisfies a predicate.

Module:

ConditionalCollect, **instance** cc, with leader *l* and output predicate C.

Events:

- **Request**:
  < input(m) >
  Inputs a message m.
- **Indication**:
  < collected(M) >:
  Outputs a vector M of collected messages.

Protocol: Signed ConditionalCollect

Implements:

ConditionalCollect, instance cc, with leader *l* and output predicate C.

Uses:

AuthPerfectPointToPointLinks, instance al.

upon event < init > **do**

states := $[\perp]^N$; $\Sigma := [\perp]^N$

collected := **false**

upon event < input(s) > **do**

σ := sign(self, s)

trigger < al, send(*l*, State(s, σ)) >

upon event < al, deliver(p, State(s, σ)) > **do** // only leader *l*

if verify-sig(p, s, σ) **then**

states[p] := s; Σ[p] := σ

To prevent storing wrong messages in the leader, the signature is checked although the checks are done in followers as well.

Protocol: Signed Conditional Collect

upon C(states) $\vee$ #(states) $\geq$ N-f **do** // only leader *l*

  forall q $\in \pi$ **do**

    trigger <al, send(q, Collected(states, Σ)) >

  states := $[\perp]^N$; $\Sigma := [\perp]^N$

upon event < al, deliver(*l*, Collected(S, Σ)) > **do**

  if collected = **false** $\wedge$ C(S) $\wedge$

    $\forall p \in \pi$. M[p] $\neq \perp \Rightarrow$ verify-sig(p, S[p], Σ[p]) **then**

  collected := **true**;

  trigger < collected(S) >

Byzantine Epoch Consensus

Module:

ByzantineEpochConsensus, instance bep, with timestamp ts and leader *l*.

Events:

- **Request**: < propose(v) >
  Proposes value v. Executed only by the leader *l*.
- **Indication**: < decide(v) >:
  Outputs the decided value v.
- **Request**: < abort >:
  Aborts epoch consensus.
- **Indication**: < aborted(state) >:
  Signals that epoch consensus has completed abortion, and outputs the internal state.

Protocol State

State:

(valts, val, cert)

The current timestamp and value pair, and

a certificate that shows they are the result of a valid write.

Retrieving Previously Decided Values

In a collection S with a Byzantine quorum (more than 2f processes) that have valid certs,

the consensus is either

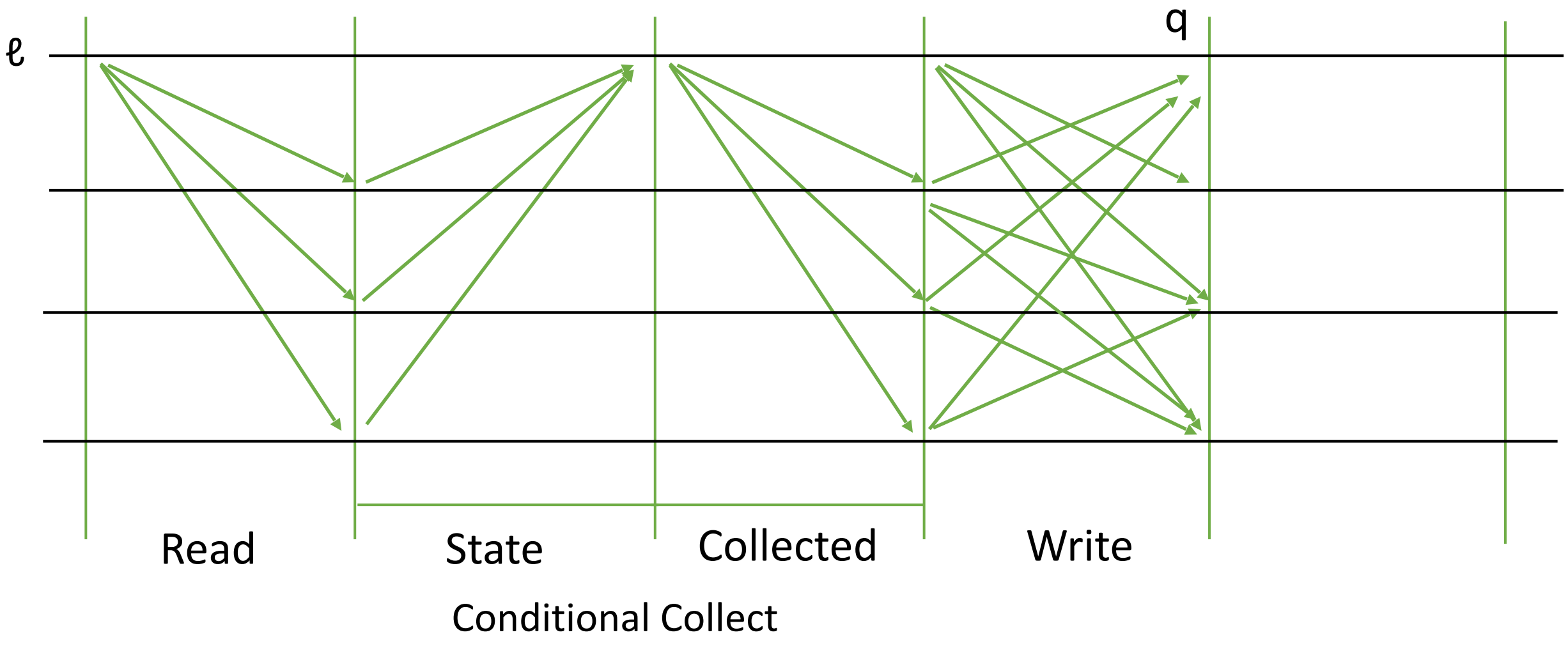
- **unbound**, unbound(S):
  - The largest timestamp is the initial 0.
  - Adopt the proposal of leader.
- **bound** to a value v, binds(v, S)
  - The largest timestamp is positive.
  - The value v of that timestamp is already locked-in. Adopt it.

The predicate for Conditional Collect is that

S is at least a quorum, and

cert is valid for each entry of S.

Write Phase



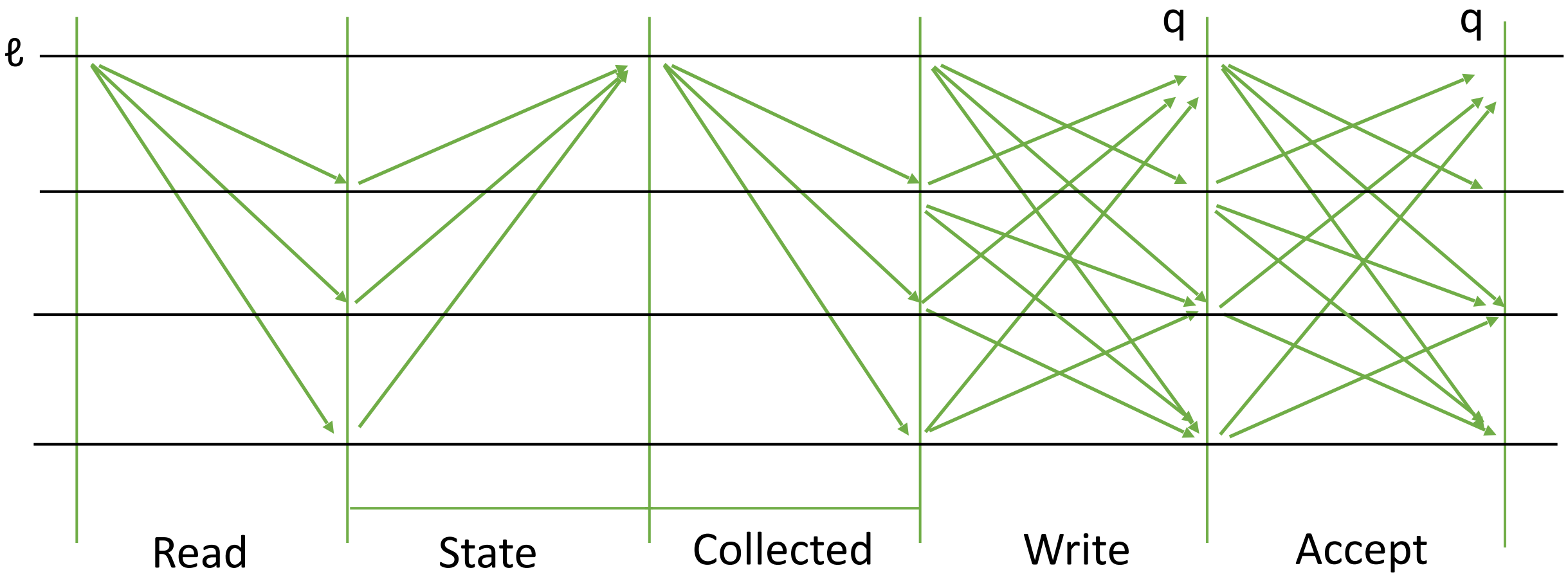
Write Phase

A Byzantine leader may send different values as his own proposal.

Thus, processes should receive the same value from a Byzantine quorum.

To make sure that no two correct processes decide different values (agreement), this round needs an all-to-all communication, similar to Byzantine consistent broadcast.

Accept Phase



Accept Phase

- The accept round makes sure that before deciding a value, it is already written to a Byzantine quorum.
- It ensures the lock-in property.

Protocol: Byzantine Epoch Consensus

Implements ep,

with timestamp ets and leader L

Uses al, abeb, cc (N = 3f + 1)

upon < init (ts, v) > **do**
    written := $[\perp]^N$
    accepted := $[\perp]^N$
    (valts, val) := (ts, v)
    cert := $\perp$

upon < propose (v) > **do** ▷ At the leader L
    if val = $\perp$ **then** val := v
    trigger <abeb, broadcast(Read)>

When val = $\perp$, ts = 0. Thus, the proposal of the leader, val, does not make the collection bound. The value val is passed in the collection to be used when the collection is unbound.

Protocol: Byzantine Epoch Consensus

upon < abeb, deliver(L, Read) > **do**
  trigger <cc, input(State(valts, val, cert))>

upon < cc, collected(S) > do
  if ∃v. binds(v, S) **then**
    cv := v
  else if unbound(S) ∧ S[L]=State(_, v, _) **then**
    cv := v
  trigger <abeb, broadcast(Write(ets, cv))>

For each p:
S[p] = State(ts, v, cert) or ⊥

Protocol: Byzantine Epoch Consensus

upon < abeb, deliver(p, Write(ts, v)$^{\sigma}$) > where ts = ets **do**
    written[p] := v$^{\sigma}$
    if ∃v. #{p | written[p] = v} ≥ 2f+1 **then**
        (valts, val) := (ets, v)
        cert := written
        written := [⊥]^{N}
        trigger <abeb, broadcast(Accept(v))>

Superscript is the signature.

upon < abeb, deliver(p, Accept(v)$^{\sigma}$) > **do**
    accepted[p] := v$^{\sigma}$
    if ∃v. #{p | accepted[p] = v} ≥ f+1 **then**
        (valts, val) := (ets, v)
        cert := accepted
        trigger <abeb, broadcast(Accept(v))>
    if ∃v. #{p | accepted[p] = v} ≥ 2f+1 **then**
        accepted := [⊥]^{N}
        trigger < decide(v) >

Amplification

upon < abort > **do**
    trigger < aborted(valts, val, cert) >

Amplification and Termination

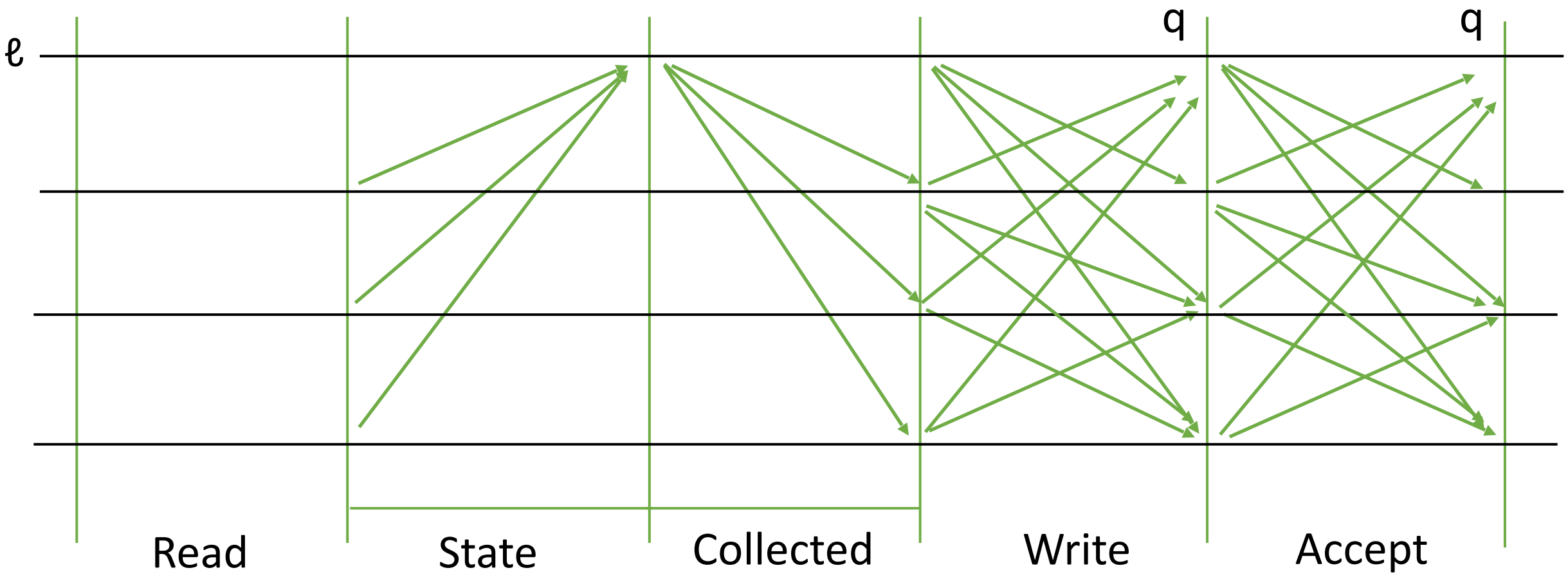
Without amplification, consider the following scenario:

The leader is Byzantine. The f byzantine processes communicate in the Write and Accept rounds with only f+1 out of 2f+1 correct processes. The communicating processes f + f + 1 = 2f + 1 are enough to make each other decide. However, the remaining f correct processes are left undecided. Since their number is less than f, their complaints cannot start a new epoch.

Optimizations

First, the Read message may be omitted. Upon initializing the new epoch consensus instance, every process simply invokes conditional collect with its State message.

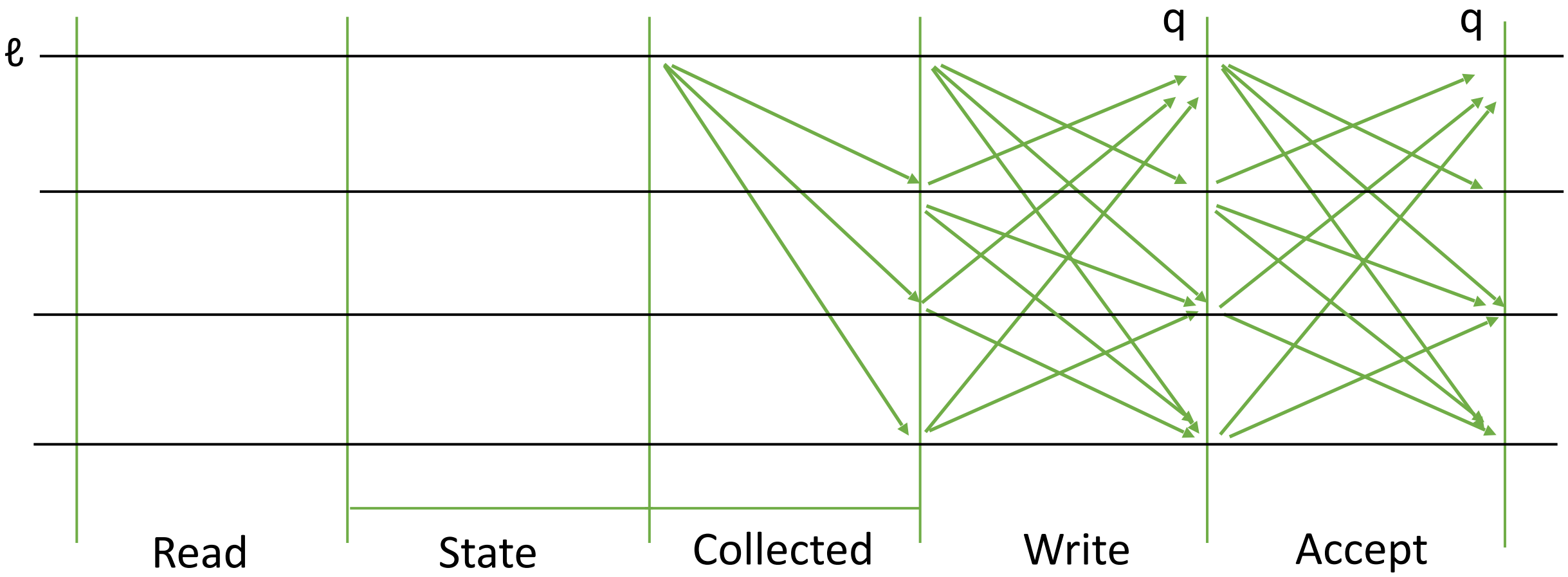
Accept Phase



Optimizations

Second, in the first epoch consensus instance, the conditional collect primitive for reading the state of all processes may be skipped because all processes store the default state initially.
Therefore, the algorithm involves an initial message from the leader to all processes and two rounds of echoing the message among all processes. This is the same communication pattern as first used in the Byzantine reliable broadcast algorithm of Bracha, and it is also used during the normal-case operation of a view in the PBFT algorithm.

Accept Phase



Byzantine epoch consensus

Properties:

- **EP1**: *Validity*:
  If a correct process decides v, then v was proposed by the leader ℓ' of some epoch consensus with timestamp ts' ≤ ts and leader ℓ'.
- **BEP2**: *Agreement*:
  No two correct processes decide differently.
- **EP3**: *Integrity*:
  Every correct process decides at most once.
- **EP4**: *Lock-in*:
  If a correct process has decided v in an epoch consensus with timestamp ts' < ts, then no correct process decides a value different from v.
- **EP5**: *Termination*:
  If the leader ℓ is correct, has proposed a value, and no correct process aborts this epoch consensus, then every correct process eventually decides some value.
- **EP6**: *Abort behavior*:
  When a correct process requests abort, it will eventually receive an aborted response; moreover, a correct process receives an aborted response only if some correct process has requested abort.

Correctness

EP1: *Validity*:
If a correct process decides v, then v was proposed by the leader ℓ’ of some epoch consensus with timestamp ts’ ≤ ts and leader ℓ’.

The decided value is either bound or unbound.

- If it is unbound, it is proposed by the leader of this round.
- If it is bound, by induction, it is proposed by the leader of a previous round.

Correctness

BEP2: *Agreement*:
No two correct processes decide differently.

Immediate from a quorum of Accept messages.
Every two 2f+1 processes intersect in a correct process.

Correctness

EP3: *Integrity*:
Every correct process decides at most once.

Before issuing decide, the accepted array is nullified, and cannot be populated by a quorum again.

Correctness

EP4: *Lock-in*:
If a correct process has decided v in an epoch consensus with timestamp ts’ < ts, then no correct process decides a value different from v.

A quorum (more than 2f processes) stored v before sending an Accept message in the previous epoch ts' < ts.

Processes passed it in state to subsequent epochs.

In later epochs, every quorum (more than 2f processes) in the collection intersects with the write quorum above, and retrieves the value v.

Correctness

EP5: *Termination*:
If the leader ℓ is correct, has proposed a value, and no correct process aborts this epoch consensus, then every correct process eventually decides some value.

Progress in Send and Collected rounds by the termination of conditional collect.

Progress in Write and Accept rounds as the 2f+1 correct processes can help each other.

Correctness

EP6: *Abort behavior*:
When a correct process requests abort, it will eventually receive an aborted response; moreover, a correct process receives an aborted response only if some correct process has requested abort.

Immediate from algorithm.