LUND UNIVERSITY

Distributed Algorithms (PhD course)
Consensus

SARDAR MUHAMMAD SULAMAN

Consensus

- The processes use consensus to agree on a common value out of values they initially propose
- Reaching consensus is one of the most fundamental problems in distributed computing
- Any algorithm that helps multiple processes maintain common state or to decide on a future action involves solving a consensus problem

LUND UNIVERSITY

Consensus Algorithms

- Regular consensus: (fail-stop model)
  - Flooding consensus algorithm
  - Hierarchical consensus algorithm
- Uniform consensus: (fail-stop model)
  - Flooding uniform consensus algorithm
  - Hierarchical uniform consensus
- Uniform consensus: (fail-noisy model)
  - Leader-Based epoch change
  - Epoch consensus
  - Leader-Driven consensus

LUND UNIVERSITY

Distributed System Models

- **Fail-Stop:**
  - Processes execute the deterministic algorithms assigned to them, unless they possibly crash, in which case they do not recover. Links are supposed to be perfect. Finally, the existence of a perfect failure detector
- **Fail-Noisy:**
  - Like fail-stop model together with perfect links. In addition, the existence of the eventually perfect failure detector

LUND UNIVERSITY

Regular consensus

- A consensus abstraction is specified in terms of two events:
  1. Propose (propose | v)
     - » Each process has an initial value v that it proposes for consensus through a propose request, in the form of triggering a propose event. All correct processes must initially propose a value
  2. Decide (Decide | v)
     - » All correct processes have to decide on the same value through a decide indication that carries a value **v**

  (The decided value has to be one of the proposed values)

Regular Consensus Properties

Module 5.1: Interface and properties of (regular) consensus

Module:

Name: Consensus, **instance** c.

Events:

Request: $\langle\ c, \mathit{Propose} \mid v\ \rangle$: Proposes value v for consensus.

Indication: $\langle\ c, \mathit{Decide} \mid v\ \rangle$: Outputs a decided value v of consensus.

Properties:

C1: *Termination:* Every correct process eventually decides some value.

C2: *Validity:* If a process decides v, then v was proposed by some process.

C3: *Integrity:* No process decides twice.

C4: *Agreement:* No two correct processes decide differently.

LUND UNIVERSITY

Contd.

- The ***termination*** and ***integrity*** properties together imply that every correct process decides exactly once
- The ***validity*** property ensures that the consensus primitive may not invent a decision value by itself
- The ***agreement*** property states the main feature of consensus, that every two correct processes that decide indeed decide the same value

Flooding Consensus Algorithm

- It uses a perfect failure-detector and a best-effort broadcast communication abstraction
- The processes execute sequential rounds. Each process maintains the set of proposed values that it has seen; this set initially consists of its own proposal
- The process typically extends this proposal set when it moves from one round to the next and new proposed values are encountered
- In each round, every process disseminates its set in a PROPOSAL message to all processes using the best-effort broadcast abstraction.

(Process floods the system with all proposals it has seen in previous rounds)

LUND UNIVERSITY

Contd.

- When a process receives a proposal set from another process, it merges this set with its own. In each round, the process computes the union of all proposal sets that it received so far.
- A process decides when it has reached a round during which it has gathered all proposals that will ever possibly be seen by any correct process. At the end of this round, the process decides a specific value in its proposal set.

Algorithm 5.1: Flooding Consensus

Implements:

Consensus, **instance** c.

Uses:

BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle\ c, \mathit{Init}\ \rangle$ **do**
$\mathit{correct} := \Pi$;
$\mathit{round} := 1$;
$\mathit{decision} := \bot$;
$\mathit{receivedfrom} := [\emptyset]^N$;
$\mathit{proposals} := [\emptyset]^N$;
$\mathit{receivedfrom}[0] := \Pi$;

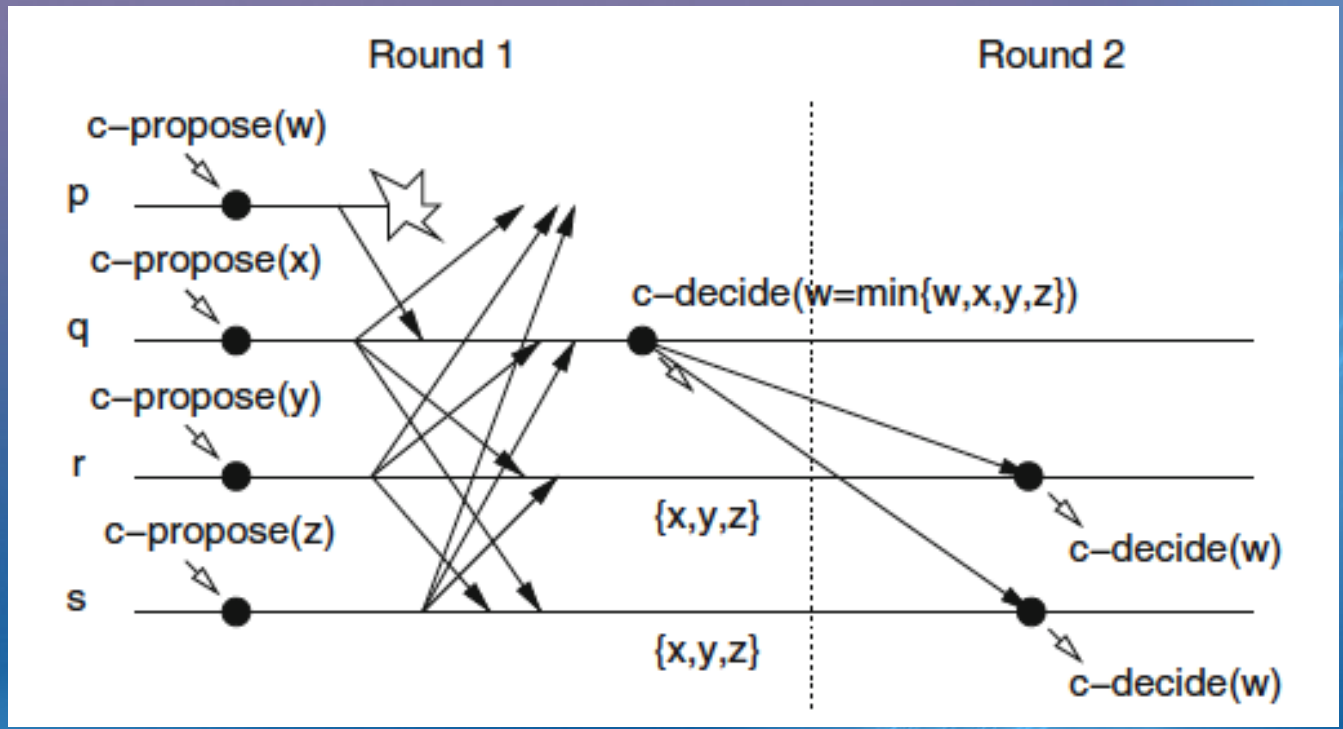
upon event $\langle\ \mathcal{P}, \mathit{Crash} \mid p\ \rangle$ **do**
$\mathit{correct} := \mathit{correct} \setminus \{p\}$;

upon event $\langle\ c, \mathit{Propose} \mid v\ \rangle$ **do**
$\mathit{proposals}[1] := \mathit{proposals}[1] \cup \{v\}$;
trigger $\langle\ \mathit{beb}, \mathit{Broadcast} \mid [\text{PROPOSAL}, 1, \mathit{proposals}[1]]\ \rangle$;

upon event $\langle\ \mathit{beb}, \mathit{Deliver} \mid p, [\text{PROPOSAL}, r, ps]\ \rangle$ **do**
$\mathit{receivedfrom}[r] := \mathit{receivedfrom}[r] \cup \{p\}$;
$\mathit{proposals}[r] := \mathit{proposals}[r] \cup ps$;

```
upon correct ⊆ receivedfrom[round] ∧ decision = ⊥ do
    if receivedfrom[round] = receivedfrom[round – 1] then
        decision := min(proposals[round]);
        trigger ⟨ beb, Broadcast | [DECIDED, decision] ⟩;
        trigger ⟨ c, Decide | decision ⟩;
    else
        round := round + 1;
        trigger ⟨ beb, Broadcast | [PROPOSAL, round, proposals[round – 1]] ⟩;

upon event ⟨ beb, Deliver | p, [DECIDED, v] ⟩ such that p ∈ correct ∧ decision = ⊥ do
    decision := v;
    trigger ⟨ beb, Broadcast | [DECIDED, decision] ⟩;
    trigger ⟨ c, Decide | decision ⟩;
```



Process p crashes during round 1 after broadcasting its proposal. Only process q sees that proposal. No other process crashes. As process q receives proposals in round 1 from all processes and this set is equal to the set of processes at the start of the algorithm in round 0, process q can decide. It selects the minimum value among the proposals and decides value w.

Contd.

- The **validity** and **integrity** properties follow from the algorithm and from the properties of the broadcast abstraction
- The **termination** property follows from the fact that in round **N** , at the latest, all processes decide. This is because:
  - Processes that do not decide keep moving from round to round due to the strong completeness property of the failure detector
  - At least one process needs to fail per round, in order to force the execution of a new round without decision
  - There are only **N** processes in the system

Hierarchical Consensus Algorithm

- It's an alternative way to implement regular consensus in the fail-stop model
- It is interesting because it uses fewer messages than our "Flooding Consensus" algorithm and enables one process to decide before exchanging any messages with the rest of the processes; this process has zero latency
- However, to reach a global decision, i.e., for all correct processes to decide, the algorithm requires N communication steps, even in situations where no failure occurs
- It exploits the ranking among the processes given by the rank**(.)** function. The rank is a unique number between 1 and N for every process
- The important ranks are low numbers, hence, the highest rank is 1 and the lowest rank is **N**

LUND UNIVERSITY

Contd.

- The “Hierarchical Consensus” algorithm works in rounds and relies on a **best effort broadcast** abstraction and **on a perfect failure detector**
- In round **i** , the process **p** with rank **i** decides its proposal and broadcasts it to all processes in a **DECIDED message**. All other processes that reach round **i** wait before taking any actions, until they deliver this message or until **P** detects the crash of **p**
- No other process than p broadcasts any message in round 1
- If the process **p** with rank 1 does not crash in the “Hierarchical Consensus” algorithm, it will impose its value on all other processes by broadcasting a **DECIDED message** and every correct process will decide the value proposed by **p**
- If **p** crashes immediately at the start of an execution and the process **q** with rank 2 is correct then the algorithm ensures that **the proposal of q** will be decided

LUND UNIVERSITY

Algorithm 5.2: Hierarchical Consensus

Implements:
Consensus, **instance** c.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle\ c, \mathit{Init}\ \rangle$ **do**
$\mathit{detectedranks} := \emptyset$;
$\mathit{round} := 1$;
$\mathit{proposal} := \bot$; $\mathit{proposer} := 0$;
$\mathit{delivered} := [\text{FALSE}]^N$;
$\mathit{broadcast} := \text{FALSE}$;

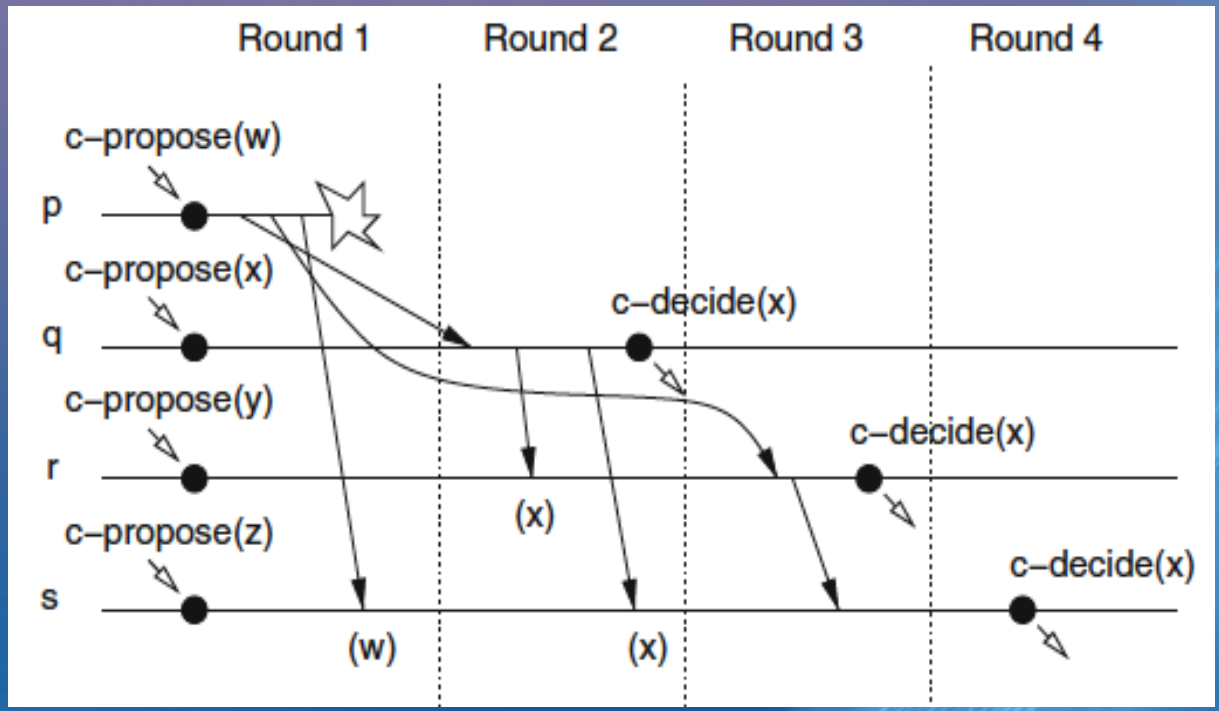
upon event $\langle\ \mathcal{P}, \mathit{Crash} \mid p\ \rangle$ **do**
$\mathit{detectedranks} := \mathit{detectedranks} \cup \{\mathit{rank}(p)\}$;

upon event $\langle\ c, \mathit{Propose} \mid v\ \rangle$ **such that** $\mathit{proposal} = \bot$ **do**
$\mathit{proposal} := v$;

upon $\mathit{round} = \mathit{rank}(\mathit{self}) \wedge \mathit{proposal} \neq \bot \wedge \mathit{broadcast} = \text{FALSE}$ **do**
$\mathit{broadcast} := \text{TRUE}$;
trigger $\langle\ \mathit{beb}, \mathit{Broadcast} \mid [\text{DECIDED}, \mathit{proposal}]\ \rangle$;
trigger $\langle\ c, \mathit{Decide} \mid \mathit{proposal}\ \rangle$;

upon $\mathit{round} \in \mathit{detectedranks} \vee \mathit{delivered}[\mathit{round}] = \text{TRUE}$ **do**
$\mathit{round} := \mathit{round} + 1$;

upon event $\langle\ \mathit{beb}, \mathit{Deliver} \mid p, [\text{DECIDED}, v]\ \rangle$ **do**
$r := \mathit{rank}(p)$;
if $r < \mathit{rank}(\mathit{self}) \wedge r > \mathit{proposer}$ **then**
$\mathit{proposal} := v$;
$\mathit{proposer} := r$;
$\mathit{delivered}[r] := \text{TRUE}$;



Process p decides w and broadcasts its proposal to all processes, but crashes. Processes q and r detect the crash before they deliver the proposal of p and advance to the next round. Process s delivers the message from p and changes its own proposal accordingly, i.e., s adopts the value w

In round **2** , process q decides its own proposal x and broadcasts this value. This causes s to change its proposal again and now to adopt the value x from q. From this point on, there are no further failures and the processes decide in sequence the same value, namely x, the proposal of q. Even if the message from p reaches process r much later, the process no longer adopts the value from p because it has already adopted a value from process with a less important rank.

Uniform Consensus

- Uniform consensus ensures that no two processes decide different values, whether they are correct or not
- Its **uniform agreement** property eliminates the restriction to the decisions of the correct processes and requires that every process, whether it later crashes or not, decides the same value.
- All other properties of uniform consensus are the **same** as in (regular) consensus

Contd.

Module 5.2: Interface and properties of uniform consensus

Module:

Name: UniformConsensus, **instance** *uc*.

Events:

Request: ⟨ *uc*, *Propose* | *v* ⟩: Proposes value *v* for consensus.

Indication: ⟨ *uc*, *Decide* | *v* ⟩: Outputs a decided value *v* of consensus.

Properties:

UC1–UC3: Same as properties C1–C3 in (regular) consensus (Module 5.1).

UC4: *Uniform agreement:* No two processes decide differently.

Flooding Uniform Consensus

- A process can no longer decide after receiving messages from the same set of processes in two consecutive rounds.
- Recall that a process might have decided and crashed before its proposal set or decision message reached any other process. (As this would violate the uniform agreement property)
- The “Flooding Uniform Consensus” algorithm always runs for N rounds and every process decides only in round N .
- Instead of a round-specific proposal set, only one global proposal set is maintained, and the variable ***receivedfrom*** contains only the set of processes from which the process has received a message in the current round

Algorithm 5.3: Flooding Uniform Consensus

Implements:
UniformConsensus, **instance** *uc*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *uc*, *Init* $\rangle$ **do**
correct := Π;
round := 1;
decision := $\bot$;
proposalset := $\emptyset$;
receivedfrom := $\emptyset$;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
correct := *correct* $\setminus$ $\{p\}$;

upon event $\langle$ *uc*, *Propose* | v $\rangle$ **do**
proposalset := *proposalset* $\cup$ $\{v\}$;
trigger $\langle$ *beb*, *Broadcast* | [PROPOSAL, 1, *proposalset*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [PROPOSAL, r, ps] $\rangle$ **such that** r = *round* **do**
receivedfrom := *receivedfrom* $\cup$ $\{p\}$;
proposalset := *proposalset* $\cup$ ps;

upon *correct* $\subseteq$ *receivedfrom* $\wedge$ *decision* = $\bot$ **do**
if *round* = N **then**
decision := min(*proposalset*);
trigger $\langle$ *uc*, *Decide* | *decision* $\rangle$;
else
round := *round* + 1;
receivedfrom := $\emptyset$;
trigger $\langle$ *beb*, *Broadcast* | [PROPOSAL, *round*, *proposalset*] $\rangle$;

Hierarchical Uniform Consensus

- The “Hierarchical Uniform Consensus” algorithm uses a **perfect failure-detector, a best-effort broadcast** to disseminate the proposal, **a perfect links abstraction** to acknowledge the receipt of a proposal, and **a reliable broadcast abstraction** to disseminate the decision
- Every process maintains a single proposal value that it broadcasts in the round corresponding to its **rank**. When it receives a proposal from a more importantly ranked process, it adopts the value
- In every round of the algorithm, the process whose **rank** corresponds to the **number of the round** is the leader, i.e., the most importantly ranked process is the leader of round 1

LUND UNIVERSITY

Contd.

- A round here consists **of two communication steps**: within the same round, the leader broadcasts a **PROPOSAL message** to all processes, trying to impose its value, and then expects to obtain an **acknowledgment** from all correct processes
- Processes that receive a proposal from the leader of the round adopt this proposal as their own and send an **acknowledgment** back to the leader of the round
- If the leader succeeds in collecting an **acknowledgment** from all processes except detected as crashed, the leader can decide. It **disseminates** the decided value using a reliable broadcast communication abstraction

Algorithm 5.4: Hierarchical Uniform Consensus

Implements:
UniformConsensus, **instance** *uc*.

Uses:
PerfectPointToPointLinks, **instance** *pl*;
BestEffortBroadcast, **instance** *beb*;
ReliableBroadcast, **instance** *rb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *uc*, *Init* $\rangle$ **do**
detectedranks := $\emptyset$;
ackranks := $\emptyset$;
round := 1;
proposal := $\perp$; *decision* := $\perp$;
proposed := $[\perp]^N$;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
detectedranks := *detectedranks* $\cup$ {*rank*(p)};

upon event $\langle$ *uc*, *Propose* | v $\rangle$ **such that** *proposal* = $\perp$ **do**
proposal := v;

upon *round* = *rank*(*self*) $\wedge$ *proposal* $\neq \perp \wedge$ *decision* = $\perp$ **do**
trigger $\langle$ *beb*, *Broadcast* | [PROPOSAL, *proposal*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [PROPOSAL, v] $\rangle$ **do**
proposed[*rank*(p)] := v;
if *rank*(p) $\geq$ *round* **then**
trigger $\langle$ *pl*, *Send* | p, [ACK] $\rangle$;

upon *round* $\in$ *detectedranks* **do**
if *proposed*[*round*] $\neq \perp$ **then**
proposal := *proposed*[*round*];
round := *round* + 1;

upon event $\langle$ *pl*, *Deliver* | q, [ACK] $\rangle$ **do**
ackranks := *ackranks* $\cup$ {*rank*(q)};

upon *detectedranks* $\cup$ *ackranks* = $\{1, \ldots, N\}$ **do**
trigger $\langle$ *rb*, *Broadcast* | [DECIDED, *proposal*] $\rangle$;

upon event $\langle$ *rb*, *Deliver* | p, [DECIDED, v] $\rangle$ **such that** *decision* = $\perp$ **do**
decision := v;
trigger $\langle$ *uc*, *Decide* | *decision* $\rangle$;

Uniform Consensus: (fail-noisy model)

- The consensus algorithms presented so far cannot be used in the fail-noisy model, where the failure detector is only eventually perfect and might make mistakes
- Fail-Noisy uniform consensus algorithm causes the processes to execute a sequence of epochs
- The epochs are identified with increasing timestamps; every epoch has a designated leader , whose task is to reach consensus among the processes
- If the leader is correct and no further epoch starts, then the leader succeeds in reaching consensus
- But if the next epoch in the sequence is triggered, the processes abort the current epoch and invoke the next one, even if some processes may already have decided in the current epoch

LUND UNIVERSITY

Contd.

- Introduces two new abstractions to build a fail-noisy consensus algorithm:
  - The first one is an **epoch-change** primitive that is responsible for triggering the sequence of epochs at all processes
  - The second one is an **epoch consensus** abstraction, whose goal is to reach consensus in a given epoch

Epoch-Change

- Epoch-change abstraction signals the start of a new epoch by triggering a (***StartEpoch*** **|** ***ts, l***) event, when a leader is suspected
- The event contains two parameters: an epoch timestamp ***ts*** and a leader process ***l*** that serve to identify the starting epoch. When this event occurs, we say the process starts epoch (***ts, l***)

LUND UNIVERSITY

Module 5.3: Interface and properties of epoch-change

Module:

Name: EpochChange, **instance** *ec*.

Events:

Indication: $\langle$ *ec*, *StartEpoch* $\mid ts, \ell \rangle$: 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') with $ts = ts'$, then $\ell = \ell'$.

EC3: *Eventual leadership:* There is a time after which every correct process has started some epoch and starts no further epoch, such that the last epoch started at every correct process is epoch (ts, ℓ) and process ℓ is correct.

Leader-Based Epoch-Change

- Every process p maintains two timestamps:
  - a timestamp ***lastts*** of the last epoch that it started (i.e., for which it triggered a StartEpoch event)
  - 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)

LUND UNIVERSITY

Contd.

- Initially, the process sets ***ts*** to its rank. Whenever the leader detector subsequently makes **p trust** itself, p adds ***N*** to ***ts*** and sends a **NEWEPOCH message** with ***ts***.
- When process p receives a **NEWEPOCH message** with a parameter ***newts > lastts*** from some process and p most recently trusted, then the process triggers a StartEpoch event with parameters ***newts*** and ***l***.
- Otherwise, the process informs the aspiring leader ***l*** with 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 5.5: Leader-Based Epoch-Change

Implements:
EpochChange, **instance** *ec*.

Uses:
PerfectPointToPointLinks, **instance** *pl*;
BestEffortBroadcast, **instance** *beb*;
EventualLeaderDetector, **instance** Ω.

upon event $\langle$ *ec*, *Init* $\rangle$ **do**
$trusted := \ell_0$;
$lastts := 0$;
$ts :=$ *rank*(*self*);

upon event $\langle$ Ω, *Trust* | p $\rangle$ **do**
$trusted := p$;
if $p =$ *self* **then**
$ts := ts + N$;
trigger $\langle$ *beb*, *Broadcast* | [NEWEPOCH, ts] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | ℓ, [NEWEPOCH, *newts*] $\rangle$ **do**
if $\ell = trusted \wedge newts > lastts$ **then**
$lastts := newts$;
trigger $\langle$ *ec*, *StartEpoch* | *newts*, ℓ $\rangle$;
else
trigger $\langle$ *pl*, *Send* | ℓ, [NACK] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | p, [NACK] $\rangle$ **do**
if *trusted* = *self* **then**
$ts := ts + N$;
trigger $\langle$ *beb*, *Broadcast* | [NEWEPOCH, ts] $\rangle$;

Epoch Consensus

- The properties of epoch consensus are closely related to those of uniform consensus. Its **uniform agreement and integrity properties are the same**
- The **termination condition** of epoch consensus is only **weakened by assuming the leader is correct**
- The **validity property** extends the possible decision values to those proposed in epochs with smaller timestamps, assuming a well-formed sequence of epochs
- Finally, the **lock-in property** is new and establishes an explicit link on the decision values across epochs: if some process has already ep-decided **v** in an earlier epoch of a well-formed sequence then only **v** may be ep -decided during this epoch

LUND UNIVERSITY

Module 5.4: Interface and properties of epoch consensus

Module:

Name: EpochConsensus, **instance** *ep*, with timestamp ts and leader process ℓ.

Events:

Request: $\langle$ *ep*, *Propose* | v $\rangle$: Proposes value v for epoch consensus. Executed only by the leader ℓ.

Request: $\langle$ *ep*, *Abort* $\rangle$: Aborts epoch consensus.

Indication: $\langle$ *ep*, *Decide* | v $\rangle$: Outputs a decided value v of epoch consensus.

Indication: $\langle$ *ep*, *Aborted* | *state* $\rangle$: Signals that epoch consensus has completed the abort and outputs internal state *state*.

Properties:

EP1: *Validity:* If a correct process *ep*-decides v, then v was *ep*-proposed by the leader ℓ' of some epoch consensus with timestamp $ts' \leq ts$ and leader ℓ'.

EP2: *Uniform agreement:* No two processes *ep*-decide differently.

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

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

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

EP6: *Abort behavior:* When a correct process aborts an epoch consensus, it eventually will have completed the abort; moreover, a correct process completes an abort only if the epoch consensus has been aborted by some correct process.

Read/Write Epoch Consensus

- The leader tries to **impose a decision value** on the processes.
- The algorithm involves **two rounds of message exchanges** from the leader to all processes
  1. **Propose** and **ACK**
  2. **Write** and **Accept**
- The goal is for the leader to write its **proposal value** to all processes, who store the epoch timestamp and the value in their state and **acknowledge** this to the leader
- When the leader receives **enough acknowledgments**, it will **ep–decide** this value

LUND UNIVERSITY

Contd.

- The leader 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
- The leader receives a **quorum of STATE messages** and choses the value that comes with the highest timestamp as its proposal value, if one exists. This step uses the function highest**(.)**
- The leader then writes the **chosen value** to all processes with a **WRITE message**. The write succeeds when the leader receives an **ACCEPT message** from a **quorum of processes**
- 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.

LUND UNIVERSITY

Algorithm 5.6: Read/Write Epoch Consensus

Implements:

EpochConsensus, **instance** *ep*, with timestamp *ets* and leader ℓ.

Uses:

PerfectPointToPointLinks, **instance** *pl*;
BestEffortBroadcast, **instance** *beb*.

upon event $\langle$ *ep*, *Init* | *state* $\rangle$ **do**
    $(valts, val) := state$;
    $tmpval := \bot$;
    $states := [\bot]^N$;
    $accepted := 0$;

upon event $\langle$ *ep*, *Propose* | v $\rangle$ **do** // only leader ℓ
    $tmpval := v$;
    trigger $\langle$ *beb*, *Broadcast* | [READ] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | ℓ, [READ] $\rangle$ **do**
    trigger $\langle$ *pl*, *Send* | ℓ, [STATE, *valts*, *val*] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [STATE, ts, v] $\rangle$ **do** // only leader ℓ
    $states[q] := (ts, v)$;

upon $\#(states) > N/2$ **do** // only leader ℓ
    $(ts, v) := highest(states)$;
    if $v \neq \bot$ **then**
        $tmpval := v$;
    $states := [\bot]^N$;
    trigger $\langle$ *beb*, *Broadcast* | [WRITE, *tmpval*] $\rangle$;

```
upon event ⟨ beb, Deliver | ℓ, [WRITE, v] ⟩ do
    (valts, val) := (ets, v);
    trigger ⟨ pl, Send | ℓ, [ACCEPT] ⟩;

upon event ⟨ pl, Deliver | q, [ACCEPT] ⟩ do                    // only leader ℓ
    accepted := accepted + 1;

upon accepted > N/2 do                                         // only leader ℓ
    accepted := 0;
    trigger ⟨ beb, Broadcast | [DECIDED, tmpval] ⟩;

upon event ⟨ beb, Deliver | ℓ, [DECIDED, v] ⟩ do
    trigger ⟨ ep, Decide | v ⟩;

upon event ⟨ ep, Abort ⟩ do
    trigger ⟨ ep, Aborted | (valts, val) ⟩;
    halt;                                                      // stop operating when aborted
```

LUND
UNIVERSITY