LUND UNIVERSITY

Distributed Algorithms (PhD course)
Consensus

SARDAR MUHAMMAD SULAMAN

Consensus (Recapitulation)

- 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)

LUND UNIVERSITY

Consensus Algorithms (Last Week)

- 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

Consensus Algorithms

- Randomized consensus: (fail-silent model)
  - Randomized Binary Consensus
  - Randomized Consensus with Large Domain
- Byzantine Consensus
  - Byzantine Epoch-Change (fail-noisy arbitrary model)
  - Byzantine Epoch Consensus (fail-arbitrary model)
  - Byzantine Read/Write Epoch Consensus (fail-arbitrary model)
- Byzantine Randomized Consensus (fail-arbitrary model)
  - Byzantine Randomized Binary Consensus

LUND UNIVERSITY

D. System Models 2/2

- **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**
- **Fail-Silent:**
  - Fail-stop model together with **perfect links**. In addition, **no failure detector**

D. System Models 2/2

- **Fail-arbitrary:**
  - **It** uses the **fail-arbitrary (or Byzantine)** process abstraction and the **authenticated perfect links** abstraction**. No failure detector**.
  - This model could also be called the **fail silent-arbitrary model**
- **Fail-noisy-arbitrary model**:
  - **Fail-arbitrary (or Byzantine)** process abstraction are considered together with **authenticated perfect links** and in combination with the **Byzantine eventual leader-detector abstraction**

LUND UNIVERSITY

Randomized consensus: (fail-silent)

- Any algorithm for consensus must either rely on a **failure-detector abstraction** (i.e., use a fail-stop or a fail-noisy model) or **it must be probabilistic**
- Deterministic consensus algorithm in a fail-silent model has executions that **do not terminate**
- No deterministic algorithm solves consensus in **asynchronous systems**
- It uses the same events to propose a value and to decide a value, and all correct processes must **initially propose a value**

LUND UNIVERSITY

Properties (Same as regular consensus)

Module 5.8: Interface and properties of randomized consensus

Module:

Name: RandomizedConsensus, **instance** *rc*.

Events:

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

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

Properties:

RC1: *Probabilistic termination:* With probability 1, every correct process eventually decides some value.

RC2–RC4: Same as properties C2–C4 in (regular) consensus (Module 5.1).

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

Common Coin

- Common coin is a primitive that is invoked by triggering an event **Release** at every process; a process releases the coin because the coin's value is unpredictable before the first process invokes the coin
- The value **c** of the coin is output to every process through an event **Output | c**
- It is assumed that every correct process releases its coin initially
- Common coin has an **output domain B** and is characterized by four properties

Properties

Module 5.9: Interface and properties of a common coin

Module:

Name: CommonCoin, **instance** *coin*, with domain $\mathcal{B}$.

Events:

Request: $\langle$ *coin*, *Release* $\rangle$: Releases the coin.

Indication: $\langle$ *coin*, *Output* $\mid b \rangle$: Outputs the coin value $b \in \mathcal{B}$.

Properties:

COIN1: *Termination:* Every correct process eventually outputs a coin value.

COIN2: *Unpredictability:* Unless at least one correct process has released the coin, no process has any information about the coin output by a correct process.

COIN3: *Matching:* With probability at least δ, every correct process outputs the same coin value.

COIN4: *No bias:* In the event that all correct processes output the same coin value, the distribution of the coin is uniform over $\mathcal{B}$ (i.e., a matching coin outputs any value in $\mathcal{B}$ with probability $\frac{1}{\#(\mathcal{B})}$).

LUND UNIVERSITY

Properties Contd.

- The first property ensures **termination**. The second property keeps the coin value **secret** until the first process releases the coin
- The third and fourth properties specify the **probability distribution** of the coin output. In particular, we require that with probability at least **δ > 0**, the outputs of all correct processes match because they are equal; we call such a coin δ – matching:
  - If the coin outputs match always, i.e., when δ **= 1**, we say the coin matches perfectly
- Furthermore, given that all coin outputs actually match, the distribution of the coin must be unbiased , **that is, uniform over B**

LUND UNIVERSITY

Randomized Binary Consensus

- It relies on a **majority of correct processes to make progress** and on a **common coin** abstraction for **terminating and reaching agreement**
- The algorithm operates in **sequential rounds**, where the processes try to ensure that the **same value is proposed** by a majority of the processes in each round
- If there is no such value, the processes **resort to the coin** abstraction and let it select a value to propose in the next round

LUND UNIVERSITY

Contd.

- Each round of the “Randomized Binary Consensus” algorithm consists of **two phases**
- In the **first phase**, every correct process **proposes** a value by sending it to all processes with a **best-effort broadcast** primitive
- Then it **receives** proposals from a **quorum** of processes. If a process observes that all responses contain the **same phase-one proposal value v*** then it proposes that value for the **second phase**
- If a process does not obtain a **unanimous set of proposals** in the first phase, the process simply **proposes ⊥ for the second phase**

Contd.

- The purpose of the **second phase is to verify if v∗** was also observed by enough other processes. After a process receives **N − f phase-two messages**, it checks if **more than f phase-two proposals are equal to v∗**, and may decide this value if there are enough of them
- A process that receives **v∗** in the second phase, but is unable to collect enough **v∗** values to decide, **starts a new round with v∗ as its proposal**
- If a process does not receive **v∗** in the second phase. In this case, the process starts a new round, with a new proposal that **it sets to the value output by the common coin abstraction**

Contd.

- Then, it distributes a **DECIDED message** with the decision value using a **reliable broadcast** abstraction. Every process **decides upon receiving this message**

Algorithm 5.12: Randomized Binary Consensus (phase 1)

Implements:

RandomizedConsensus, **instance** *rc*, with domain $\{0, 1\}$.

Uses:

BestEffortBroadcast, **instance** *beb*;
ReliableBroadcast, **instance** *rb*;
CommonCoin (multiple instances).

upon event $\langle$ *rc*, *Init* $\rangle$ **do**
    $round := 0$; $phase := 0$;
    $proposal := \bot$;
    $decision := \bot$;
    $val := [\bot]^N$;

upon event $\langle$ *rc*, *Propose* | v $\rangle$ **do**
    $proposal := v$;
    $round := 1$; $phase := 1$;
    trigger $\langle$ *beb*, *Broadcast* | [PHASE-1, *round*, *proposal*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [PHASE-1, r, v] $\rangle$ **such that** $phase = 1 \wedge r = round$ **do**
    $val[p] := v$;

upon $\#(val) > N/2 \wedge phase = 1 \wedge decision = \bot$ **do**
    if exists $v \neq \bot$ such that $\#(\{p \in \Pi \mid val[p] = v\}) > N/2$ **then**
        $proposal := v$;
    else
        $proposal := \bot$;
    $val := [\bot]^N$;
    $phase := 2$;
    trigger $\langle$ *beb*, *Broadcast* | [PHASE-2, *round*, *proposal*] $\rangle$;

Algorithm 5.13: Randomized Binary Consensus (phase 2)

upon event $\langle$ *beb*, *Deliver* | p, [PHASE-2, r, v] $\rangle$ **such that** $phase = 2 \wedge r = round$ **do**
    $val[p] := v$;

upon $\#(val) \geq N - f \wedge phase = 2 \wedge decision = \perp$ **do**
    $phase := 0$;
    Initialize a new instance *coin.round* of CommonCoin with domain $\{0, 1\}$;
    trigger $\langle$ *coin.round*, *Release* $\rangle$;

upon event $\langle$ *coin.round*, *Output* | c $\rangle$ **do**
    if exists $v \neq \perp$ such that $\#(\{p \in \Pi \mid val[p] = v\}) > f$ **then**
        $decision := v$;
        trigger $\langle$ *rb*, *Broadcast* | [DECIDED, *decision*] $\rangle$;
    else
        if exists $p \in \Pi$, $w \neq \perp$ such that $val[p] = w$ **then**
            $proposal := w$;
        else
            $proposal := c$;
        $val := [\perp]^N$;
        $round := round + 1$; $phase := 1$;
        trigger $\langle$ *beb*, *Broadcast* | [PHASE-1, *round*, *proposal*] $\rangle$;

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

Randomized Consensus with Large Domain

- The “Randomized Binary Consensus” algorithm can only decide on **one-bit values**. This restriction has been introduced because the processes sometimes set their proposal values to **an output of the common coin** and the **coin outputs only one bit**
- A solution is somewhat **relaxed common coin** abstraction, which does not require that all processes invoke the common coin **with the same domain**
- Every process simply uses the set of proposed values that it is aware of. This set **grows** and should eventually become **stable**, in the sense that every correct process invokes the **common coin with the same set**

LUND UNIVERSITY

Contd.

- Every process additionally **disseminates** its initial proposal with **reliable broadcast**, and every process collects the received proposals in a variable **values**
- A process then initializes the common coin with domain **values**. This ensures that the coin always outputs a value that has been proposed
- ***Termination property***, observe that eventually, all correct processes have rb-delivered the same **PROPOSAL messages**, and therefore, their **values** variables are equal

Algorithm 5.14: Randomized Consensus with Large Domain (extends Algorithm 5.12–5.13)

Implements:

RandomizedConsensus, **instance** *rc*.

Uses:

BestEffortBroadcast, **instance** *beb*;
ReliableBroadcast, **instance** *rb*;
CommonCoin (multiple instances).

// Except for the event handlers below with the extensions mentioned here, it is
// the same as Algorithm 5.12–5.13.

upon event $\langle$ *rc*, *Init* $\rangle$ **do**

...
values := $\emptyset$;

upon event $\langle$ *rc*, *Propose* | *v* $\rangle$ **do**

...
values := *values* $\cup$ {*v*};
trigger $\langle$ *rb*, *Broadcast* | [PROPOSAL, *v*] $\rangle$;

upon event $\langle$ *rb*, *Deliver* | *p*, [PROPOSAL, *v*] $\rangle$**do**

values := *values* $\cup$ {*v*};

upon #(*val*) > *N*/2 $\wedge$ *phase* = 2 $\wedge$ *decision* = $\perp$ **do**

Initialize a new instance *coin.round* of CommonCoin with domain *values*;

...

Byzantine Consensus

- A consensus primitive for **arbitrary-fault** or Byzantine process abstractions should allow all processes to reach a common decision despite the presence of faulty ones, in order for the correct processes to coordinate their actions. There are two differences, however
  - A first difference lies in the behavior of Byzantine processes: the abstraction cannot require anything from them. Therefore, **restrict all its properties to correct processes**
  - The second difference is that the validity property of consensus requires that every value decided by a (correct) process has been proposed by some process (**For Byzantine Consensus**)

LUND UNIVERSITY

Contd.

- But because a **faulty and potentially malicious process** can pretend to have proposed **arbitrary values**, we must formulate validity in another way (Weak and Strong)
- **Weak** validity:
  - The **weak** validity property maintains this guarantee only for executions in which **all processes are correct** and none of them is Byzantine. It considers the case that all processes propose the **same value** and requires that an algorithm only decides the proposed value in this case
- Moreover, the algorithm must decide a value that was actually proposed and not invented out of thin air

LUND UNIVERSITY

Weak Byzantine Consensus

Module 5.10: Interface and properties of weak Byzantine consensus

Module:

Name: WeakByzantineConsensus, **instance** *wbc*.

Events:

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

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

Properties:

WBC1: *Termination:* Every correct process eventually decides some value.

WBC2: *Weak validity:* If all processes are correct and propose the same value v, then no correct process decides a value different from v; furthermore, if all processes are correct and some process decides v, then v was proposed by some process.

WBC3: *Integrity:* No correct process decides twice.

WBC4: *Agreement:* No two correct processes decide differently.

Strong Byzantine Consensus

- The **strong validity** for Byzantine consensus tolerates **arbitrary-fault processes** and instead requires the decision value to be the value proposed by the **correct processes**
- If not all of them propose the **same value**, the decision value must still be a value proposed by a correct process or may be some special symbol **ø**. *The latter* **ø** *denotes a default value that indicates no valid decision was found*
- In other words, if all correct processes propose the same value then Byzantine consensus decides this value, and otherwise, it may decide some value proposed by a correct process or **ø**
- Importantly, the decision value cannot originate only from the Byzantine processes

Contd.

Module 5.11: Interface and properties of (strong) Byzantine consensus

Module:

Name: ByzantineConsensus, **instance** *bc*.

Events:

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

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

Properties:

BC1 and BC3–BC4: Same as properties WBC1 and WBC3–WBC4 in weak Byzantine consensus (Module 5.10).

BC2: *Strong validity:* If all correct processes propose the same value *v*, then no correct process decides a value different from *v*; otherwise, a correct process may only decide a value that was proposed by some correct process or the special value □.

LUND UNIVERSITY

Byzantine Epoch-Change (fail-noisy)

- The **epoch-change** primitive in the Byzantine model has the same interface and satisfies the same properties as the epoch-change primitive with **crash-stop processes**
- It relies on an **eventual leader detector**
- The leader of an epoch with timestamp **ts** is computed deterministically from **ts**, using the function leader**(.)**
  - The value of **leader(ts)** is process whose rank is **ts**, if **ts mod N = 0**,

    or the process with rank **N**, if **ts mod N = 0**
  - Hence, the leader rotates in a round-robin fashion

LUND UNIVERSITY

Contd.

- It maintains a timestamp **lastts** of the most recently started epoch and a timestamp **nextts**, which is equal to **lastts + 1** during the period when the process has broadcast 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 broadcasting a **NEWEPOCH message** to all processes
- Alternatively, the process also begins to switch to the next epoch after receiving **NEWEPOCH messages** from more than **f** distinct processes
- Once the process receives more than **2f NEWEPOCH messages** (from distinct processes) it **starts the epoch**

LUND UNIVERSITY

Module 5.3: Interface and properties of epoch-change

Module:

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

Events:

Indication: $\langle\ ec, \mathit{StartEpoch} \mid ts, \ell\ \rangle$: 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.

Algorithm 5.15: Byzantine Leader-Based Epoch-Change

Implements:
ByzantineEpochChange, **instance** *bec*.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*;
ByzantineLeaderDetector, **instance** *bld*.

upon event $\langle$ *bec*, *Init* $\rangle$ **do**
lastts := 0; *nextts* := 0;
trusted := 0;
newepoch := $[\perp]^n$;

upon event $\langle$ *bld*, *Trust* | p $\rangle$ **do**
trusted := p;

upon *nextts* = *lastts* $\wedge$ *trusted* $\neq$ *leader*(*lastts*) **do**
nextts := *lastts* + 1;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [NEWEPOCH, *nextts*] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [NEWEPOCH, ts] $\rangle$ **such that** ts = *lastts* + 1 **do**
newepoch[p] := NEWEPOCH;

upon #(*newepoch*) > $f \wedge$ *nextts* = *lastts* **do**
nextts := *lastts* + 1;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [NEWEPOCH, *nextts*] $\rangle$;

upon #(*newepoch*) > $2f \wedge$ *nextts* > *lastts* **do**
lastts := *nextts*;
newepoch := $[\perp]^n$;
trigger $\langle$ *bec*, *StartEpoch* | *lastts*, *leader*(*lastts*) $\rangle$;

Byzantine Epoch Consensus

- Epoch consensus abstraction in the Byzantine model has the same interface and satisfies almost the same properties as the (uniform) epoch consensus abstraction for crash-stop processes
- Only its **agreement property** differs in a minor way as it only refers to decisions of correct processes
  - **Agreement: No two correct processes ep-decide differently**
- It also uses **conditional collect** primitive

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.

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 event **(Input | m)** with an input message m
- It outputs a **vector M** with **n entries** indexed by processes, through an event **(Collected | M)** at every process, such that **M[p]** is either equal to **UNDEFINED** or corresponds to the input message of process **p**
- A conditional collect primitive is parameterized by an **output predicate C(.)**, defined on an **N**-vector of messages, and it should only output a collected vector that **satisfies the predicate**

LUND UNIVERSITY

Module 5.14: Interface and properties of conditional collect

Module:

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

Events:

Request: $\langle\, cc, \mathit{Input} \mid m \,\rangle$: Inputs a message m.

Indication: $\langle\, cc, \mathit{Collected} \mid M \,\rangle$: Outputs a vector M of collected messages.

Properties:

CC1: *Consistency:* If the leader is correct, then every correct process collects the same M, and this M contains at least $N - f$ messages different from UNDEFINED.

CC2: *Integrity:* If some correct process collects M with $M[p] \neq$ UNDEFINED for some process p and p is correct, then p has input message $M[p]$.

CC3: *Termination:* If all correct processes input compliant messages and the leader is correct, then every correct process eventually collects some M such that $C(M) =$ TRUE.

Signed Conditional Collect (Fail-arbitrary)

- It uses **two communication rounds** and assumes a **digital signature** scheme
- In the **first round**, every process signs its input message and sends it together with the signature to the **leader** over a **point-to-point** link. The leader collects enough messages (at least **N − f**) such that they satisfy the **output predicate**
- In the **second round**, the leader sends the collected and signed messages to all processes, using **authenticated point-to-point** links. Processes **verify** the source for every entry, the value **Σ[p]** represents a valid signature from process **p**

Algorithm 5.16: Signed Conditional Collect

Implements:

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

Uses:

AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle$ *cc*, *Init* $\rangle$ **do**
  $messages := [\text{UNDEFINED}]^N$; $\Sigma := [\bot]^N$;
  collected := FALSE;

upon event $\langle$ *cc*, *Input* | m $\rangle$ **do**
  $\sigma := sign(self, cc\|self\|\text{INPUT}\|m)$;
  trigger $\langle$ *al*, *Send* | ℓ, $[\text{SEND}, m, \sigma]$ $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, $[\text{SEND}, m, \sigma]$ $\rangle$ **do** // only leader ℓ
  if $verifysig(p, cc\|p\|\text{INPUT}\|m, \sigma)$ **then**
    $messages[p] := m$; $\Sigma[p] := \sigma$;

upon $\#(messages) \geq N - f \wedge C(messages)$ **do** // only leader ℓ
  forall $q \in \Pi$ **do**
    trigger $\langle$ *al*, *Send* | q, $[\text{COLLECTED}, messages, \Sigma]$ $\rangle$;
  $messages := [\text{UNDEFINED}]^N$; $\Sigma := [\bot]^N$;

upon event $\langle$ *al*, *Deliver* | ℓ, $[\text{COLLECTED}, M, \Sigma]$ $\rangle$ **do**
  if $collected = \text{FALSE} \wedge \#(M) \geq N - f \wedge C(M) \wedge$
      (forall $p \in \Pi$ such that $M[p] \neq \text{UNDEFINED}$, it holds
       $verifysig(p, cc\|p\|\text{INPUT}\|M[p], \Sigma[p])$) **then**
    collected := TRUE;
    trigger $\langle$ *cc*, *Collected* | M $\rangle$;

Byzantine Read/Write Epoch Consensus

- The algorithm starts by the leader sending a **READ message** to all processes, which triggers every process to invoke a **conditional collect primitive**. Every process inputs a **message[STATE, valts, val, writeset]** containing its state. The leader in **conditional collect** is the leader 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

Contd.

- It introduces a predicate **sound(S)** on an **N**-vector **S** of **STATE messages**, to be used in the **conditional collect** primitive
- An entry of **S** 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 ***I*** is correct, at least **N −f** entries in the collected **S** are **defined**; otherwise, more than **f** entries may be **undefined**

Algorithm 5.17: Byzantine Read/Write Epoch Consensus (part 1, read phase)

Implements:
ByzantineEpochConsensus, **instance** *bep*, with timestamp ets and leader process ℓ.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*;
ConditionalCollect, **instance** *cc*, with leader ℓ and predicate $sound(\cdot)$.

upon event $\langle$ *bep*, *Init* | *epochstate* $\rangle$ **do**
  $(valts, val, writeset) := epochstate$;
  $written := [\bot]^N$; $accepted := [\bot]^N$;

upon event $\langle$ *bep*, *Propose* | v $\rangle$ **do** // only leader ℓ
  if $val = \bot$ **then** $val := v$;
  forall $q \in \Pi$ **do**
    trigger $\langle$ *al*, *Send* | q, [READ] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [READ] $\rangle$ **such that** $p = \ell$ **do**
  trigger $\langle$ *cc*, *Input* | [STATE, *valts*, *val*, *writeset*] $\rangle$;

upon event $\langle$ *cc*, *Collected* | *states* $\rangle$ **do**
  // $states[p] = [\text{STATE}, ts, v, ws]$ or $states[p] = \text{UNDEFINED}$
  $tmpval := \bot$;
  if exists $ts \geq 0, v \neq \bot$ from S such that $binds(ts, v, states)$ **then**
    $tmpval := v$;
  else if exists $v \neq \bot$ such that $unbound(states) \wedge states[\ell] = [\text{STATE}, \cdot, v, \cdot]$ **then**
    $tmpval := v$;
  if $tmpval \neq \bot$ **then**
    if exists ts such that $(ts, tmpval) \in writeset$ **then**
      $writeset := writeset \setminus \{(ts, tmpval)\}$;
    $writeset := writeset \cup \{(ets, tmpval)\}$;
    forall $q \in \Pi$ **do**
      trigger $\langle$ *al*, *Send* | q, [WRITE, *tmpval*] $\rangle$;

Detail on next slide

Contd.

- When the **leader** is correct, conditional collect outputs a vector **S** that satisfies **sound(S) = TRUE**
- If **S** binds **ts** to some **v not equal** ⊥ then the process must write **v**; otherwise, **S** is **unbound** and the process writes the value from the **leader**, which it finds in S**[/].** The process sends a **WRITE message** to all processes with the value
- In case **sound(S) = FALSE**, the leader must be faulty and the process halts
- When a process has received more than **(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** over the **authenticated point-to-point** links
- When a process has received more than **(N + f) / 2 ACCEPT messages** from distinct processes containing the same **value v** , it

  bep -decides **v**

Byzantine Randomized Consensus

- It proceeds in **global rounds** and every round consists of **two phases**
  - In phase one, the processes exchange their proposals
  - In phase two, they determine if enough processes proposed the same value
- If one process observes a large number (**more than 2f**) of phase-two messages with the same proposal then this process may decide
- If a process observes enough phase-two messages with the same value **v** to be sure that **v** is the proposal of a correct process (the value occurs more than than **f** times) then the process adopts **v** as its own proposal
- All processes then **access a common coin** and if they have not yet **decided or adopted** a value in this round, they use the output from the coin as their proposal for the next round

LUND UNIVERSITY

Algorithm 5.20: Byzantine Randomized Binary Consensus (phase 1)

Implements:

ByzantineRandomizedConsensus, **instance** *brc*, with domain $\{0, 1\}$.

Uses:

AuthPerfectPointToPointLinks, **instance** *al*;
ByzantineConsistentBroadcast (multiple instances);
CommonCoin (multiple instances).

upon event $\langle$ *brc*, *Init* $\rangle$ **do**
    $round := 0$; $phase := 0$;
    $proposal := \perp$;
    $decision := \perp$;
    $val := [\perp]^N$;

upon event $\langle$ *brc*, *Propose* | v $\rangle$ **do**
    $proposal := v$;
    $round := 1$; $phase := 1$;
    forall $q \in \Pi$ **do**
        trigger $\langle$ *al*, *Send* | q, [PHASE-1, *round*, *proposal*] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [PHASE-1, r, v] $\rangle$ **such that** $phase = 1 \wedge r = round$ **do**
    $val[p] := v$;

upon $\#(val) \geq N - f \wedge phase = 1$ **do**
    if exists $v \neq \perp$ such that $\#(\{p \in \Pi \mid val[p] = v\}) > \frac{N+f}{2}$ **then**
        $proposal := v$;
    else
        $proposal := \perp$;
    $val := [\perp]^N$;
    $phase := 2$;
    forall $p \in \Pi$ **do**
        Initialize a new instance *bcb.round.p* of ByzantineConsistentBroadcast;
    trigger $\langle$ *bcb.round.self*, *Broadcast* | [PHASE-2, *round*, *proposal*] $\rangle$;

Algorithm 5.21: Byzantine Randomized Binary Consensus (phase 2)

upon event $\langle$ *bcb.round.p*, *Deliver* | p, [PHASE-2, r, v] $\rangle$ **such that**
$phase = 2 \wedge r = round$ **do**
$val[p] := v$;

upon $\#(val) \geq N - f \wedge phase = 2$ **do**
$phase := 0$;
Initialize a new instance *coin.round* of CommonCoin with domain $\{0, 1\}$;
trigger $\langle$ *coin.round*, *Release* $\rangle$;

upon event $\langle$ *coin.round*, *Output* | c $\rangle$ **do**
if exists $v \neq \perp$ such that $\#(\{p \in \Pi \mid val[p] = v\}) > 2f$ **then**
if $decision = \perp$ **then**
$decision := v$;
trigger $\langle$ *brc*, *Decide* | *decision* $\rangle$;
$proposal := v$;
else if exists $w \neq \perp$ such that $\#(\{p \in \Pi \mid val[p] = w\}) > f$ **then**
$proposal := w$;
else
$proposal := c$;
$val := [\perp]^N$;
$round := round + 1$; $phase := 1$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [PHASE-1, *round*, *proposal*] $\rangle$;

- Every correct process advances through the rounds of the algorithm because it waits for more than **(N + f) / 2** PHASE-1 messages and for **N − f PHASE-2** broadcasts, but all **N − f** correct processes eventually broadcast such messages
- A correct process decides as soon as it receives the majority value of the round from more than **2f** processes
- At the end of every round, the correct processes either set their proposal to the majority value **or** to the common coin output
- At the latest when the coin values at all correct processes match, the algorithm is guaranteed to decide in the next round and therefore satisfies termination

LUND UNIVERSITY

LUND
UNIVERSITY