Algorithms and Distributed Systems 2023/2024 (Lecture Five)

MIEI - Integrated Master in Computer Science and Informatics
MEI – Master in Computer Science and Informatics
Specialization block

Nuno Preguiça (nmp@fct.unl.pt)
Alex Davidson (a.davidson@fct.unl.pt)

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Acknowledgments

- Slides mostly from João Leitão.

Lecture structure:

- State Machine Replication
- Consensus Problem
- Consensus in Synchronous Systems

Up to now in ASD…

- We have discussed multiple replication strategies.
- Shared Register Replication.
- Quorum Systems (Multiple configurations)
    - Read/Write operations.

Up to now in ASD…

- We have discussed multiple replication strategies.

- Shared Register Replication.

- Quorum Systems (Multiple configurations)
  - Read/Write operations.

- What happens if operations are more complex than a read or a write of a value?

Think about replicating an integer…

- You can now execute two operations:
  - ADD (X) where you add a value X to the current value of your integer (modifies the value) and returns the value.
  - MULTIPLY (X) where you multiply the current value of your integer by X (modifies the value) and returns the value.

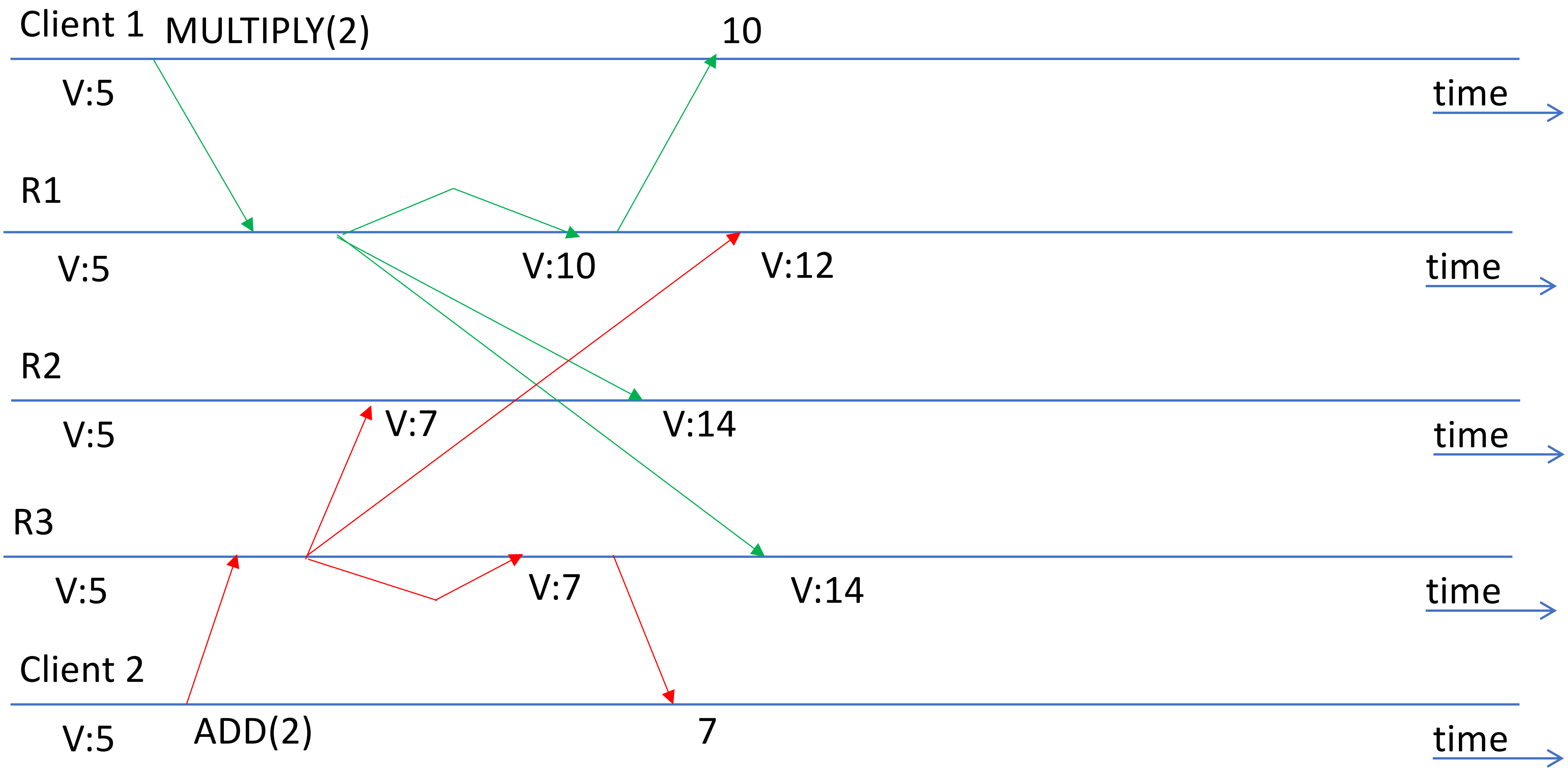
- How can we do this?

Replicating an Integer.

- Two possible solutions that you can think of:
- Broadcast operations to all replicas (Reliable Broadcast).
- Use a Quorum System (Majority Quorum System).

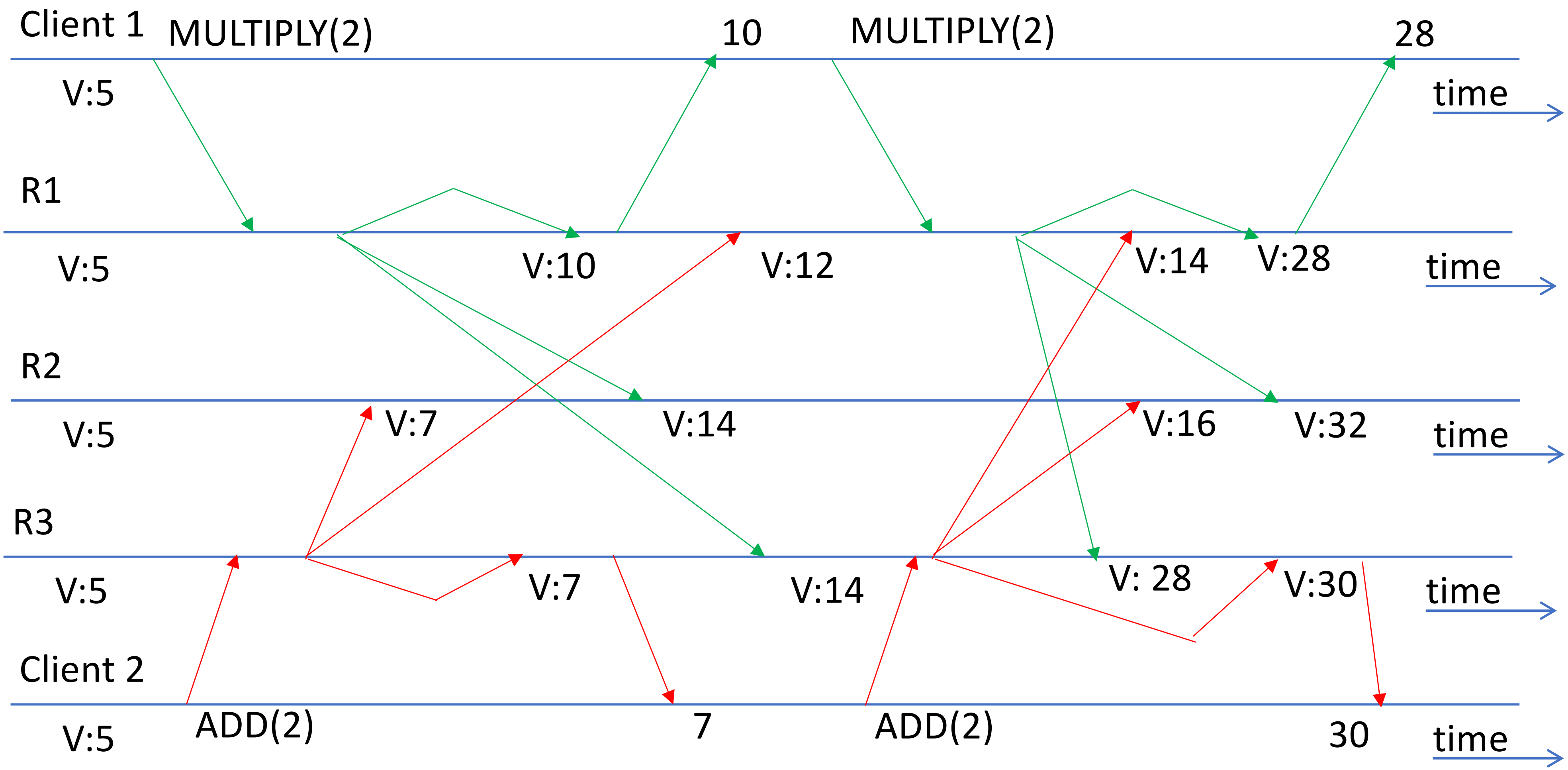
Replicating an Integer

- Use Reliable Broadcast



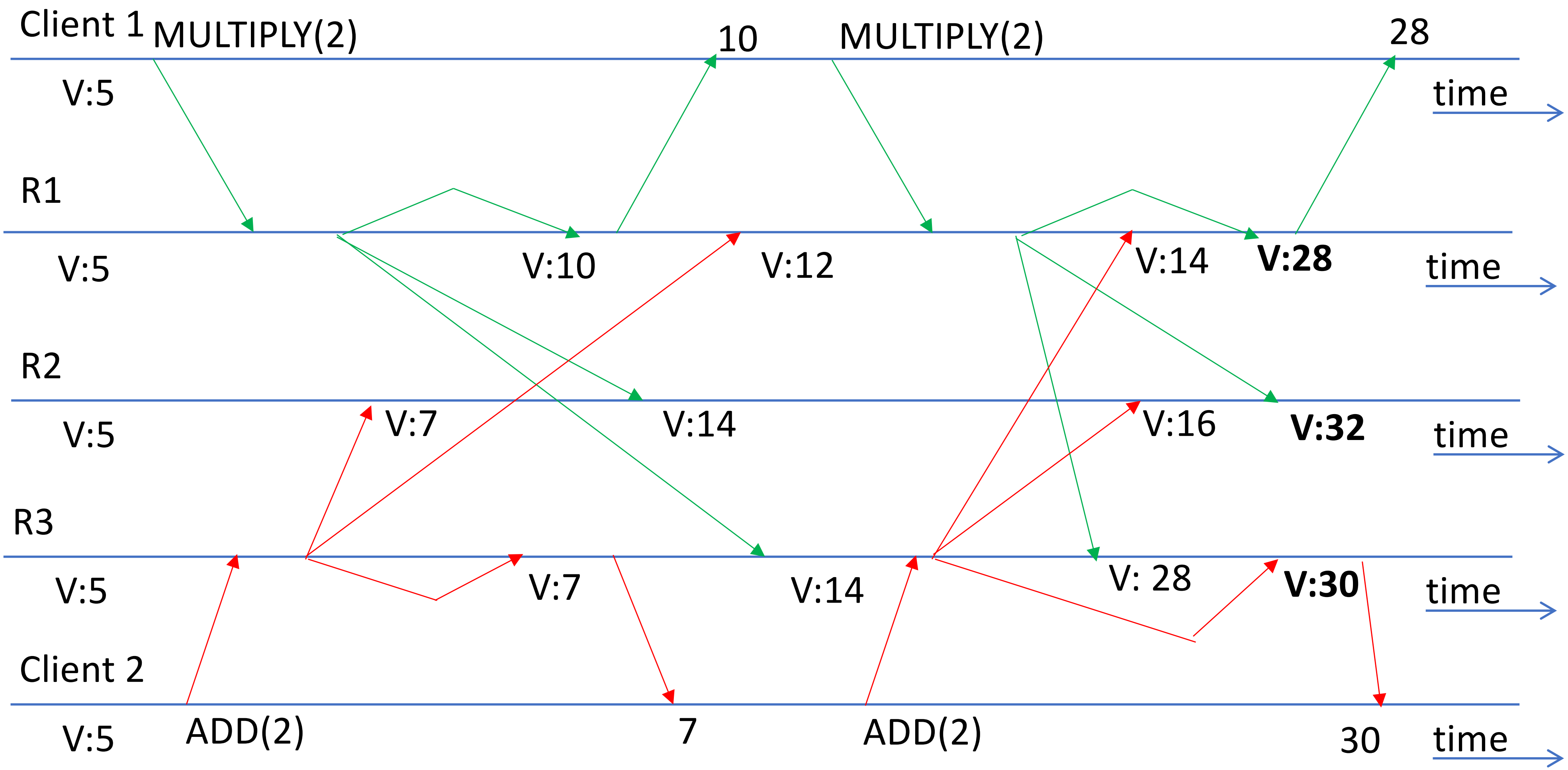
Replicating an Integer

- Use Reliable Broadcast



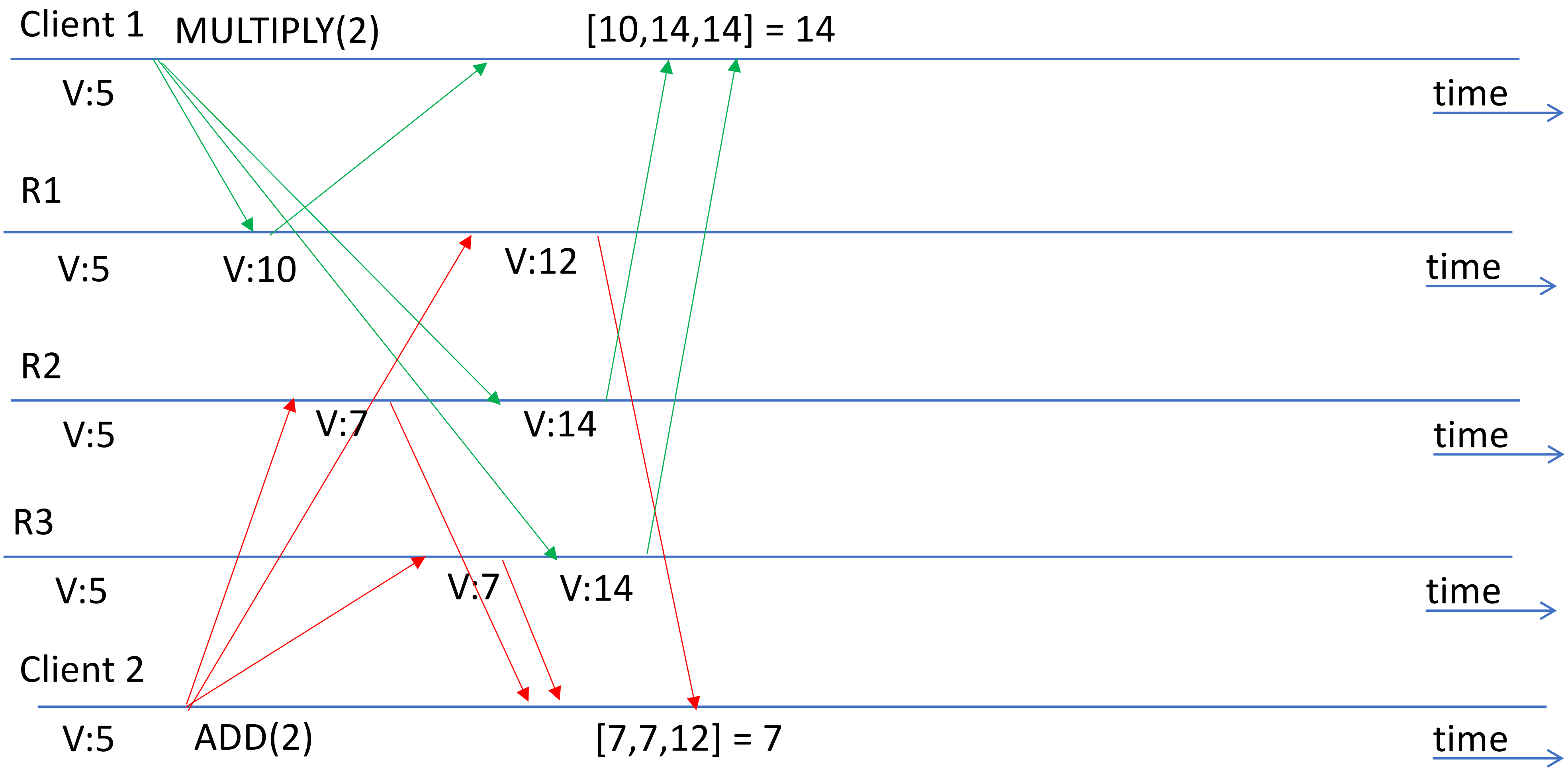
Replicating an Integer

- Use Reliable Broadcast
  - **Does not work… now we have full divergence!!**



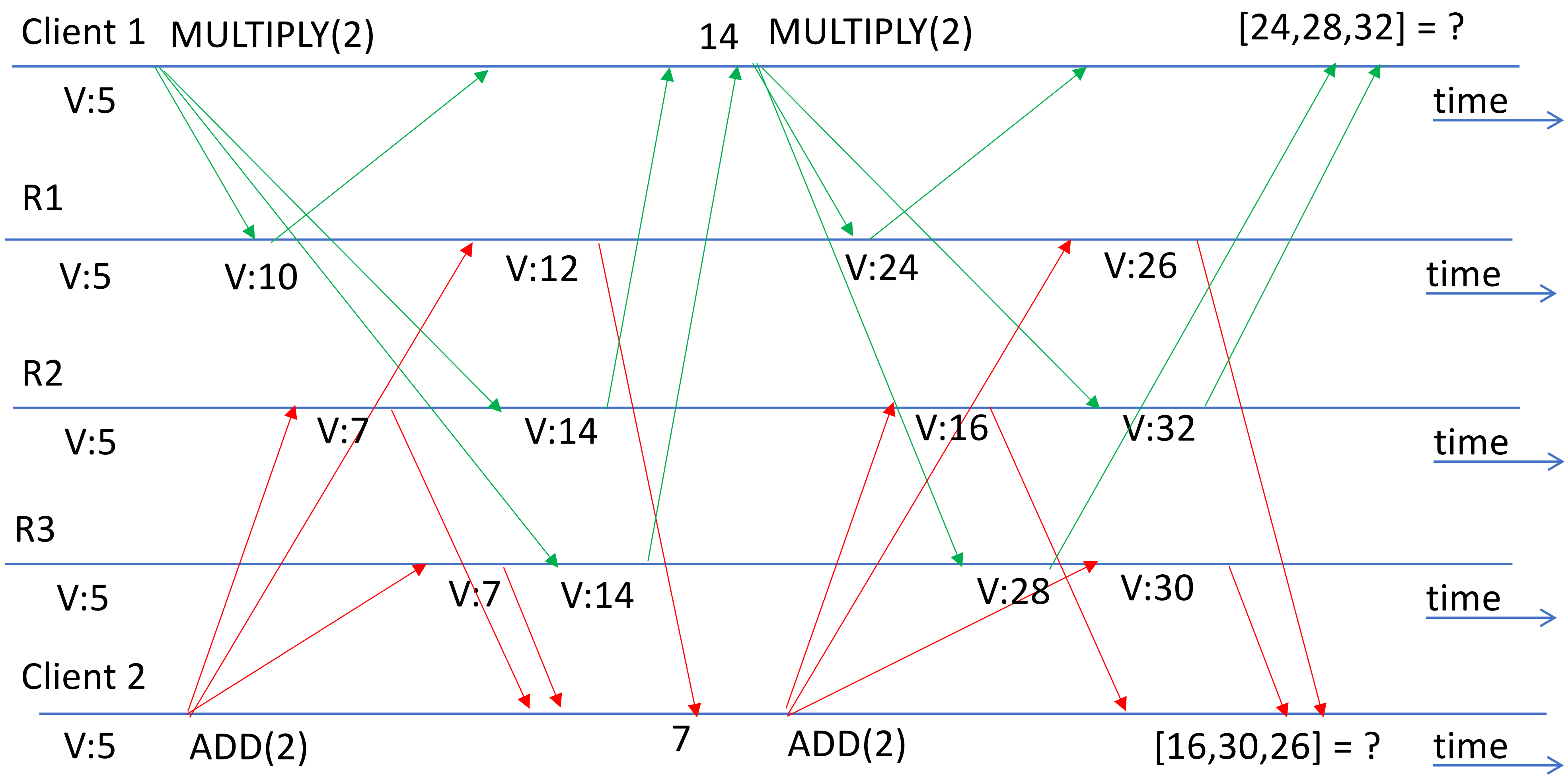
Replicating an Integer

- Use Majority Based Quorum



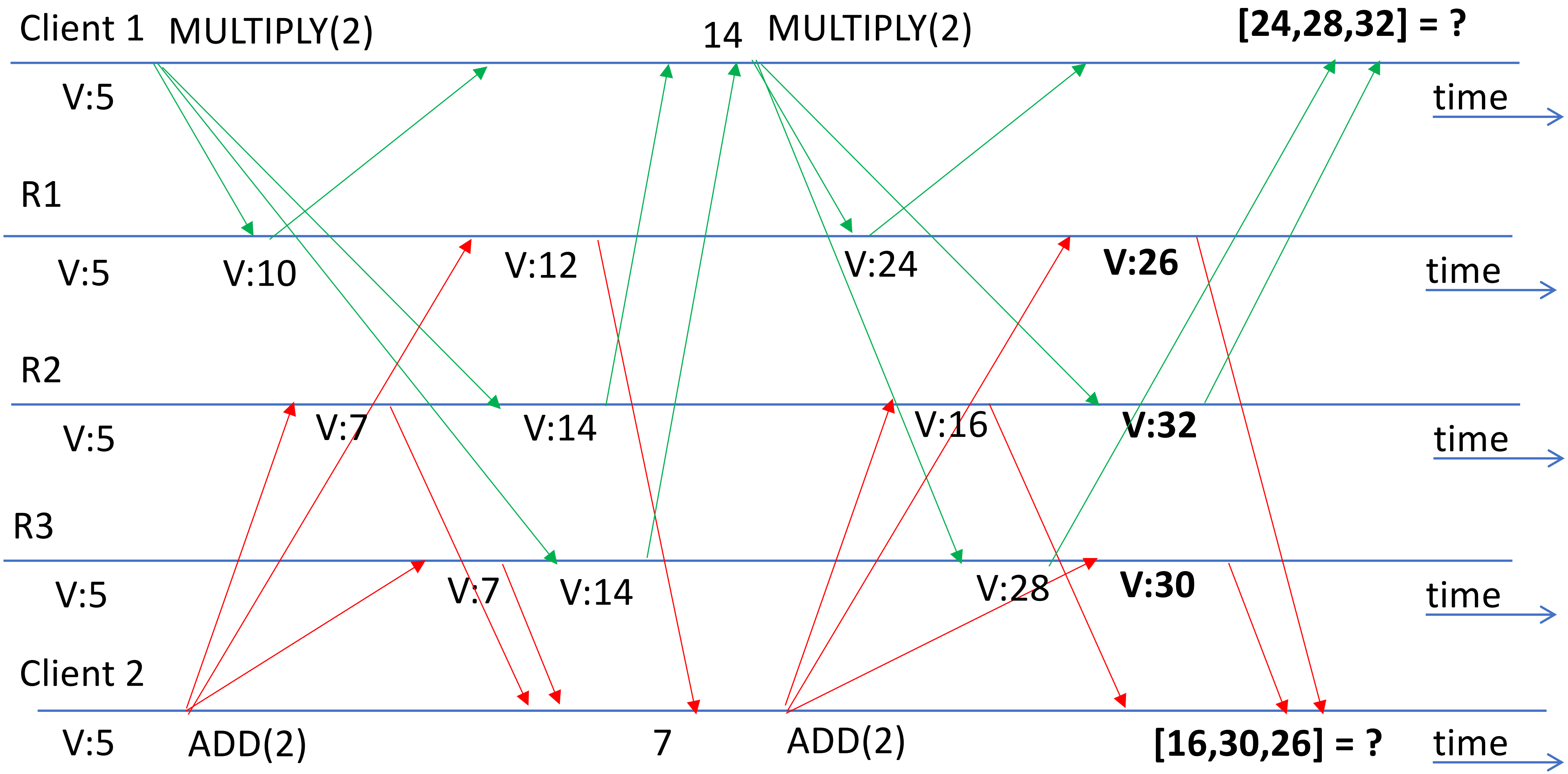
Replicating an Integer

- Use Majority Based Quorum



Replicating an Integer

- Use Majority Based Quorum
- **Does not work… now we have full divergence!!**



Replicating an Integer

- Why did these solutions fail?

Replicating an Integer

- Why did these solutions fail?

- Operations were not commutative…
- Which implies that the evolution of the state (in this case the integer) depends on the order in which operations are executed...
- What do we need to solve this?

Replicating an Integer

- Why did these solutions fail?

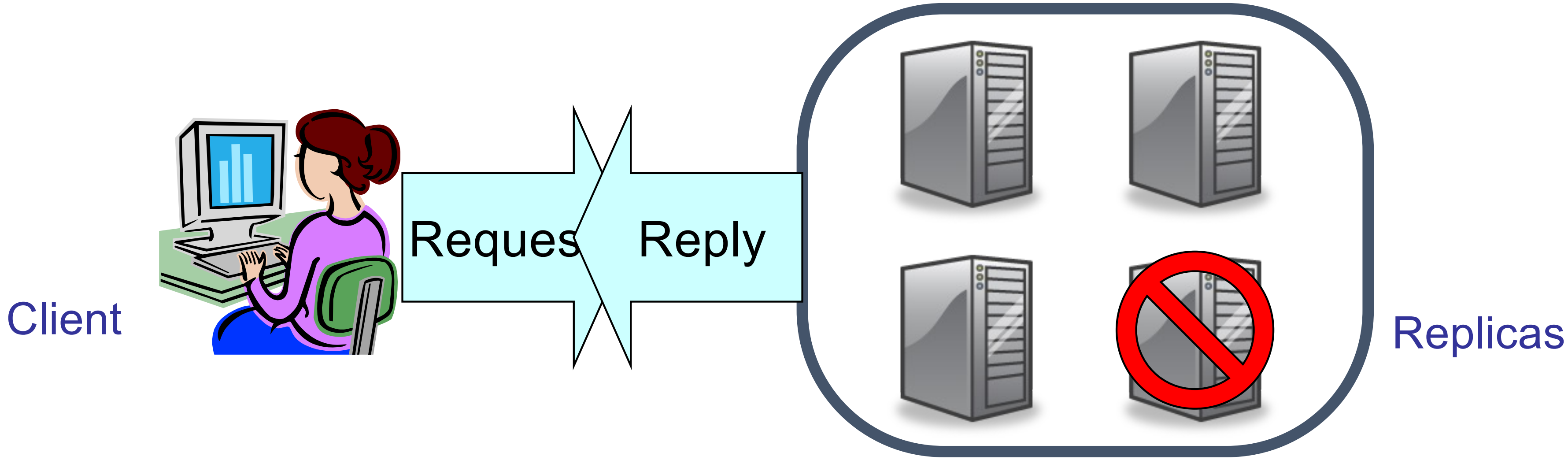
- Operations were not commutative…
- Which implies that the evolution of the state (in this case the integer) depends on the order in which operations are executed...
- What do we need to solve this?
- **State Machine Replication**

State Machine Replication: Intuition

- Used in practice to replicate the state of servers.
- A server replica has:
  - A copy of the service running and a state **S**.
  - Exports a set of operations **O**.
  - Each operation in **O** has a set of arguments (*inputs*):
    - Makes the server transit to a new state **S'**.
    - Produces a reply to the client (*output*)

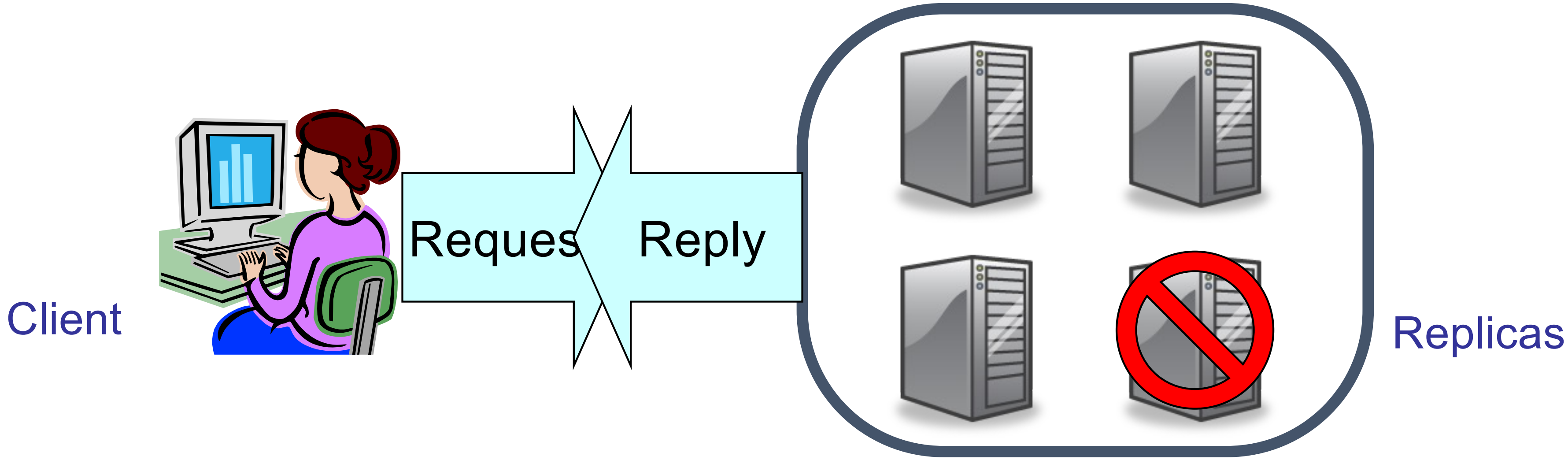
State Machine Replication

- Each server replica is seen as a state machine.
- Each server is made deterministic
  (i.e., all operations must be deterministic).
- Replicate the server.
- Ensure that all correct replicas follow the same sequence of state transitions.
- Use majority on outputs to tolerate failures.



State Machine Replication

- Each server replica is seen as a state machine.
- Each server is made deterministic
  (i.e., all operations must be deterministic).
- Replicate the server.
- **Ensure that all correct replicas follow the same sequence of state transitions.**
- Use majority on outputs to tolerate failures.



Main requirement to achieve State Machine Replication

- All correct replicas must receive and execute operations in the same order.

- Therefore, abstractions such as reliable broadcast or quorum system replication will not work.
- None of these abstractions provide guarantees related with the ordering of operations

  (of different clients).

Main requirement to achieve State Machine Replication

- Since we must consider majorities of replies, we must have a number of replicas ***n*** that depends on the total number of faults that are tolerated ***f***.
- $n = 2f+1$ (strictly speaking $n >= 2f+1$)
- n will depend on the service being replicated, and the underlying fault and system models.

Enforcing Ordering

- How can one enforce ordering among the operations executed by each server (i.e., each state machine)?

- This is a classical problem in distributed systems that was first captured by the General's story.

The Generals Problems

- Two Roman Generals must decide to either attack an enemy in a coordinated fashion or retreat.
- They can only communicate with each other through messengers that have to cross a valley full of dangers.

The Generals Problems

- Each General has an initial preference (either attack or retreat) and must agree in what to do.
- If both have the same initial preference, then that is the only valid decision.
- Messengers can either be killed or get arbitrarily delayed]0,+∞[in their path.
- A general might die at any moment along side all his troops.
  - In which case she will not send any messenger.
- General that do not die must eventually decide.

The Generals Problems (the pseudo-modern version)

- Each General has an initial preference (either attack or retreat) and must agree in what to do.
- If both have the same initial preference, then that is the only valid decision.
- Messengers can either be killed or get arbitrarily delayed]0,+∞[in their path.
- A general might die at any moment along side all his troops.
  - In which case she will not send any messenger.
- General that do not die must eventually decide.

The Generals Problems
(the pseudo-modern version)

NOT CONSIDERING THAT A GENERAL CAN BE A TRAITOR

That will be addressed in Confiabilidade de Sistemas Distribuídos (CSD).

- Eac ... st agr
- If b ... sion.
- Me ... r pat
- A g
    - in which case she will not send any messenger.
- General that do not die must eventually decide.

Why is this related with State Machine Replication?

Why is this related with State Machine Replication?

- To build a correct state machine replication solution replicas must **agree** on some (common) order in which to execute operations.
- Which is fundamentally the same as the two generals exchanging messages to **decide** if they should attack or retreat (should I deliver operation A or B first).
- The **Generals Problem** was the intuition used to introduce the **Consensus Problem** by Leslie Lamport.

The Consensus Problem (Regular Variant)

- Each process has an initial value v that he proposes.
- Each correct process eventually decides a value.
- Properties

C1 Termination: Every correct process eventually decides a 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.

The Consensus Problem (Regular Variant)

- Let's start slow (for this week)…
- Assume:
  - Synchronous System.
  - Fail-Stop Model.
  - Assume just that $n > f$ (i.e., should work with $n = f + 1$)

The Consensus Problem (Regular Variant)

- Let's start slow (for this week)…
- Assume:
  - Synchronous System.
  - Fail-Stop Model.
  - Assume just that $n > f$ (i.e., should work with $n = f + 1$)

- Suggestion:
  - Think about using best effort broadcast.

Regular Consensus (bcast consensus): Intuition

- Algorithm progresses in rounds.
- Each process knowns:
  - in which round it is (variable *round*).
  - if he already decided (variable *decision*).
  - the set of correct processes in previous rounds (initially, at round zero, all processes are considered correct).
  - the set of all proposals known in each round (initially at round zero, a process only knowns its proposal)

Regular Consensus (bcast consensus): Intuition

- Algorithm starts at round 1.
- In each round, each process broadcasts the set of proposals that it got in the previous round.
- A round terminates when a process gets a message from every correct process in that round.
- A decision is made when a message is obtained from all (correct) processes and no new crashed process is detected during a round (i.e., when the set of correct processes in round r == set of correct processes in round r-1).
- The decision is obtained by applying a deterministic function (e.g., min) over the set of all proposals.

Regular Consensus (bcast consensus): Intuition

- Why is this correct?
- The high-level intuition is that in a round with no failures, all processes will get the same set of messages, and hence will know the same set of proposals.
- If all correct processes known the same set of proposals, then, any deterministic function that picks one of these proposals will yield the same value (for all correct processes).

Regular Consensus (bcast consensus)

- Interface:
- Request:
  - cPropose(V): Used to propose a value v for consensus
- Indication:
  - cDecide(V): Used to indicate the decided value v for consensus

Regular Consensus (bcast consensus)

Algorithm 1: Regular Consensus Algorithm (Fail-Stop/Synchronous)

```
Interface:
    Requests:
        cPropose( v )
    Indications:
        cDecide ( v )

State:
    correct //set of correct processes
    correct-this-round //map associating for
                //each round number the set of correct processes
    round //current round number
    decided //decided value (if any)
    proposal-set //map associating for each round number
        the set of proposals received in that round

    Upon Init do:
        correct ⟵ Π
        correct-this-round[0] ⟵ Π
        round ⟵ 1
        decided ⟵ ⊥
        for all i ∈ 1 to #Π do
            correct-this-round[i] ⟵ {}
            proposal-set[i] ⟵ {}

    Upon crash( p ) do:
        correct ⟵ correct \ p
        if correct ⊊ correct-this-round[round] ∧ decided = ⊥ then
            Call CheckRoundTermination( )

    Upon cPropose ( v ) do:
        proposal-set[1] ⟵ proposal-set[1] ∪ {v}
        trigger bebBroadcast( MYSET, 1, proposal-set[1])

    Upon bebDeliver( p, MYSET, r, set) do:
        correct-this-round[r] ⟵ correct-this-round[r] ∪ {p}
        proposal-set[r] ⟵ proposal-set[r] ∪set
        if r = round ∧ correct ⊊ correct-this-round[r] ∧ decided = ⊥ then
            Call CheckRoundTermination( )

    Procedure CheckRoundTermination( )
        if correct-this-round[r] = correct-this-round[r − 1] then
            decided ⟵ min(proposal-set[r])
            trigger cDecide(decided)
            trigger bebBroadcast( DECISION, decided)
        else
            round ⟵ round +1
            trigger bebBroadcast( MYSET, round, proposal-set[round−1])

    Upon bebDeliver( p, DECISION, value ) do:
        if p ∈ correct ∧ decided = ⊥ then
            decided ⟵ value
            trigger cDecide(decided)
            trigger bebBroadcast( DECISION, decided)
```

Regular Consensus (State and Init)

Algorithm 2: Regular Consensus Algorithm (Fail-Stop/Synchronous) - Part I/II

Interface:
  Requests:
    cPropose(v)
  Indications:
    cDecide (v)

State:
  correct `//set of correct processes`
  correct-this-round `//map associating for`
                    `//each round number the set of correct processes`
  round `//current round number`
  decided `//decided value (if any)`
  proposal-set `//map associating for each round number`
      `the set of proposals received in that round`

  Upon Init do:
    correct $\longleftarrow \Pi$
    correct-this-round[0] $\longleftarrow \Pi$
    round $\longleftarrow 1$
    decided $\longleftarrow \perp$
    for all $i \in 1$ to $\#\Pi$ **do**
      correct-this-round[i] $\longleftarrow \{\}$
      proposal-set[i] $\longleftarrow \{\}$

Regular Consensus (Main Algorithm)

Algorithm 3: Regular Consensus Algorithm (Fail-Stop/Synchronous) - Part II/II

Upon crash(p **) do:**
  correct $\longleftarrow$ correct $\setminus p$
  if correct $\subseteq$ correct-this-round$[round]$ $\wedge$ decided $= \perp$ **then**
    Call `CheckRoundTermination( )`

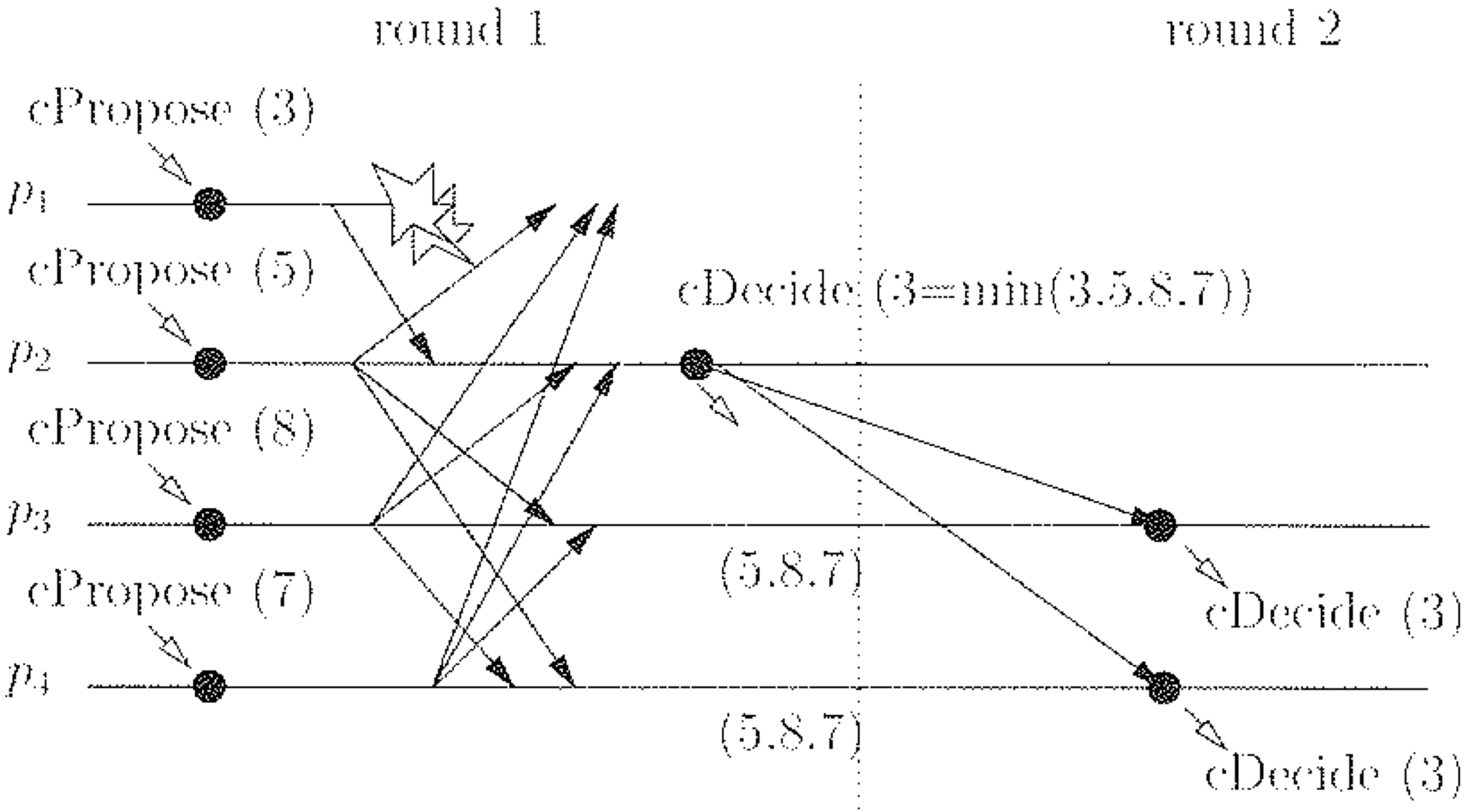
Upon cPropose (v **) do:**
  proposal-set[1] $\longleftarrow$ proposal-set[1] $\cup \{v\}$
  trigger bebBroadcast(MYSET, 1, proposal-set[1])

Upon bebDeliver(p**, MYSET,** r**,** set**) do:**
  correct-this-round$[r]$ $\longleftarrow$ correct-this-round$[r]$ $\cup \{p\}$
  proposal-set$[r]$ $\longleftarrow$ proposal-set$[r]$ $\cup set$
  if $r =$ round $\wedge$ correct $\subseteq$ correct-this-round$[r]$ $\wedge$ decided $= \perp$ **then**
    Call `CheckRoundTermination( )`

Procedure `CheckRoundTermination( )`
  if correct-this-round$[r]$ = correct-this-round$[r-1]$ **then**
    decided $\longleftarrow$ min(proposal-set$[r]$)
    trigger cDecide(decided)
    trigger bebBroadcast(DECISION, decided)
  else
    round $\longleftarrow$ round $+1$
    trigger bebBroadcast(MYSET, round, proposal-set[round-1])

Upon bebDeliver(p**, DECISION,** $value$ **) do:**
  if $p \in$ correct $\wedge$ decided $= \perp$ **then**
    decided $\longleftarrow$ $value$
    trigger cDecide(decided)
    trigger bebBroadcast(DECISION, decided)

Regular Consensus (bcast consensus): Execution Example



Regular Consensus (bcast consensus): Correctness argument

C1 Termination: Every correct process eventually decides a 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.

- Termination: At most in round n, all processes decide:
    1. processes that does not fail will keep moving from round to round (failures will be detected for sure);
    2. in the worst case a processes fails in each round;
    3. there are n processes in the system and only f processes can fail with f < n.

Regular Consensus (bcast consensus): Correctness argument

C1 Termination: Every correct process eventually decides a 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.

- Validity: Follows from the algorithm.
  1. To decide a value, that value has to be in the set of known proposals.
  2. For the value to be in that set, then some process broadcasted that value in its set.
  3. The only way to add a value to the set of proposals is by either receiving it from another process or if the value was proposed by that same process.

Regular Consensus (bcast consensus): Correctness argument

C1 Termination: Every correct process eventually decides a 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.

- Integrity: Follows from the algorithm.
    1. A decision can only be made if *decided* has a bottom value (decided is always tested before emitting a decision).
    2. When a process decides, its *decided* state becomes the decided value (and hence no longer bottom).

Regular Consensus (bcast consensus): Correctness argument

C1 Termination: Every correct process eventually decides a 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.

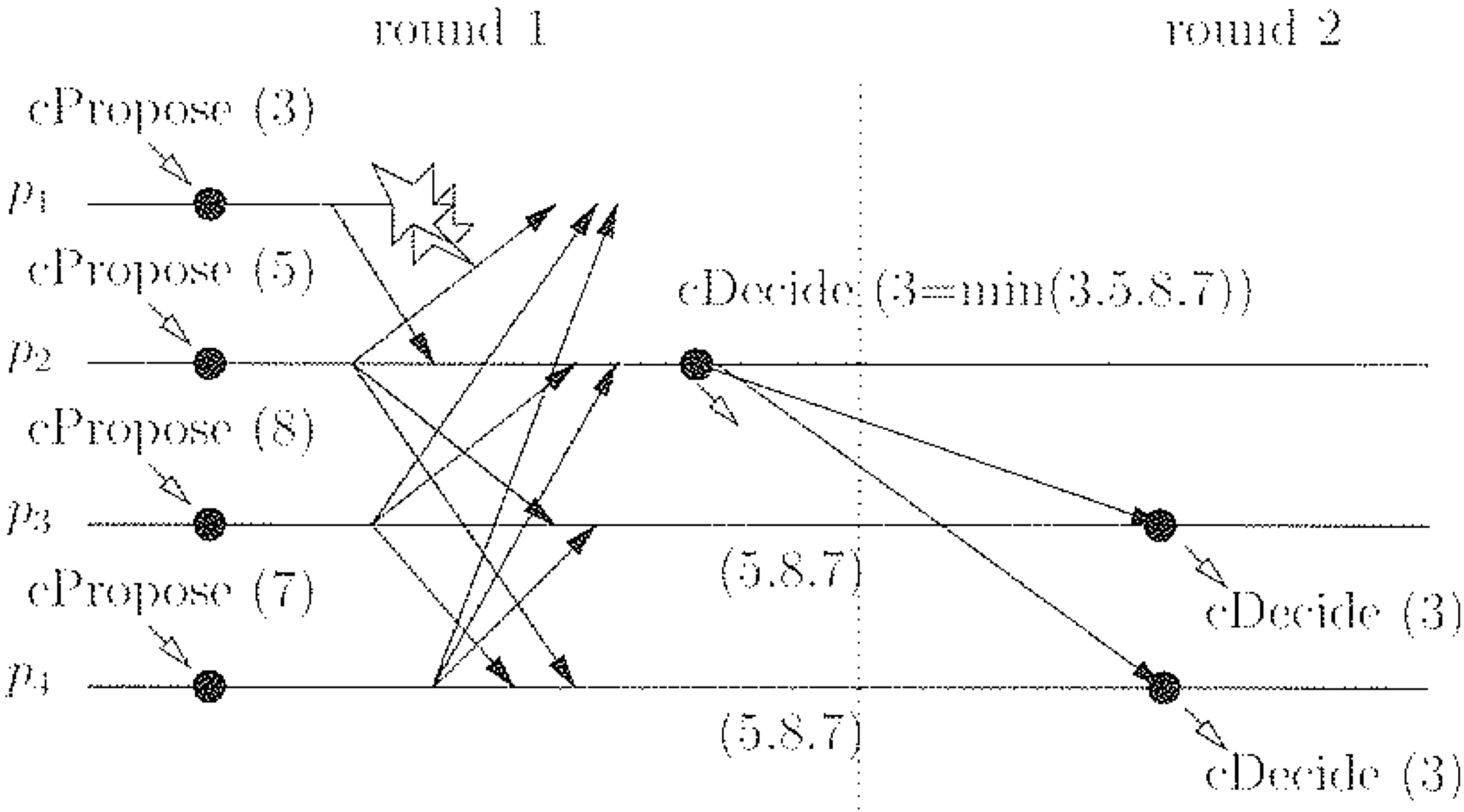
- Agreement: Assume that r is the smallest round in which some correct process p decides v, there are two cases:
  1. If p decided because it got a value v from a correct process, then all other processes (eventually) receive that value, in the worst case if other process j detects (another) failure in round r, it will decide v in round r+1 because it receives a decided message from p.
  2. If p decided because it got a value v from a process that crashes afterwards, then some other process j might not have received the value. However, j will decide v in round r+1 because it receives a decided from p.

The Consensus Problem (Uniform Variant)

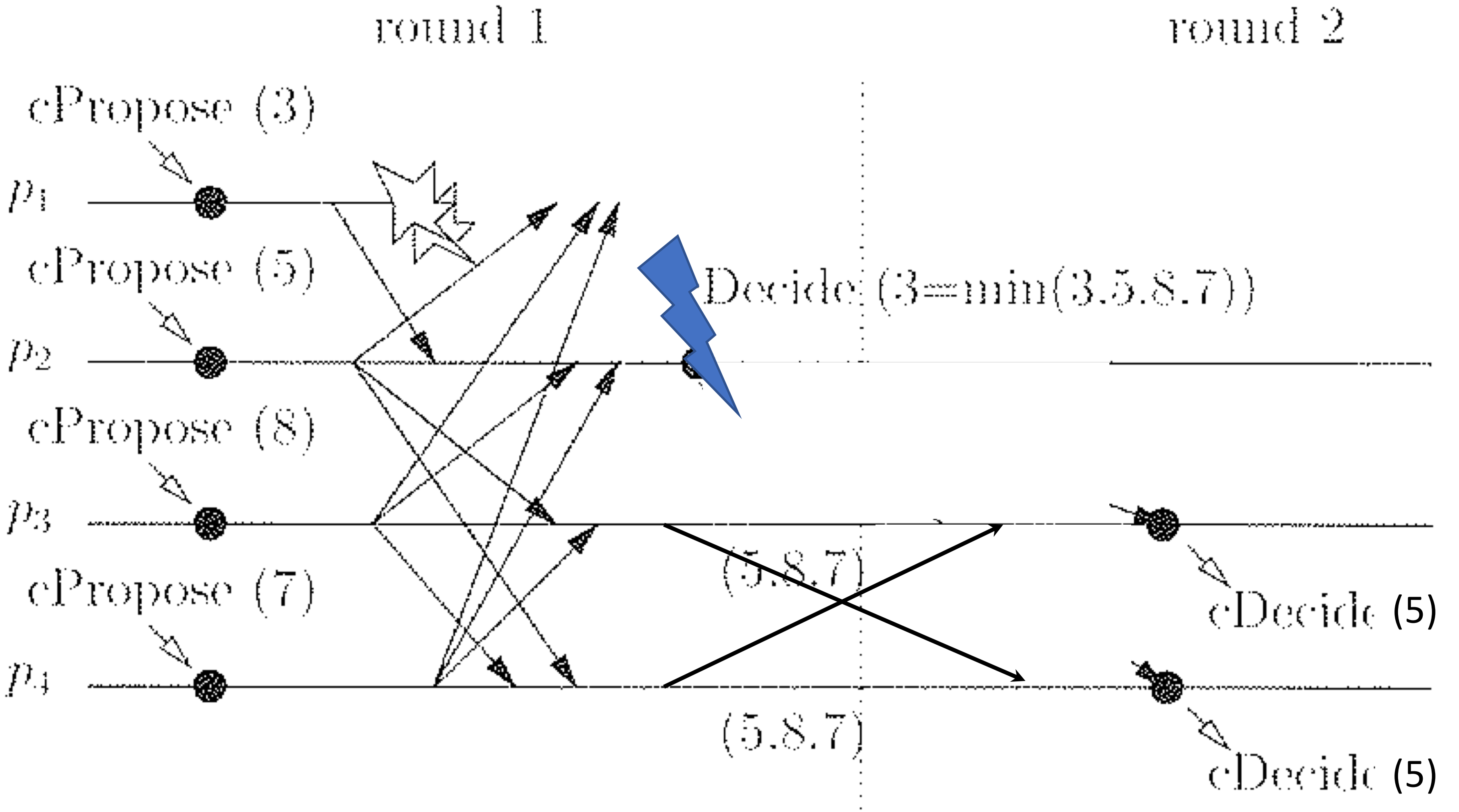
- Each process has an initial value v that he proposes.
- Each correct process eventually decides a value.
- Properties
- *C1 Termination*: Every correct process eventually decides a value.
- *C2 Validity*: If a process decides v, then v was proposed by some process.
- *C3 Integrity*: No process decides twice.
- *C4* ***Uniform*** *Agreement*: No two ~~**correct**~~ processes decide differently.

Uniform Consensus: Why does the previous solution does not work?

Uniform Consensus: Why does the previous solution does not work?



Uniform Consensus: Why does the previous solution does not work?



Uniform Consensus: Why does the previous solution does not work?

Algorithm 3: Regular Consensus Algorithm (Fail-Stop/Synchronous) - Part II/II

Upon crash(p **) do:**
  correct $\longleftarrow$ correct $\setminus p$
  if correct $\subset$ correct-this-round$[r]$ $\wedge$ decided $= \perp$ **then**
    Call `CheckRoundTermination( )`

Upon cPropose (v **) do:**
  proposal-set$[1]$ $\longleftarrow$ proposal-set$[1]$ $\cup \{v\}$
  trigger bebBroadcast(MYSET, 1, proposal-set$[1]$)

Upon bebDeliver(p**, MYSET,** r**,** set**) do:**
  correct-this-round$[r]$ $\longleftarrow$ correct-this-round$[r]$ $\cup \{p\}$
  proposal-set$[r]$ $\longleftarrow$ proposal-set$[r]$ $\cup set$
  if r = round $\wedge$ correct $\subset$ correct-this-round$[r]$ $\wedge$ decided $= \perp$ **then**
    Call `CheckRoundTermination( )`

Procedure `CheckRoundTermination( )`
  if correct-this-round$[r]$ = correct-this-round$[r-1]$ **then**
    decided $\longleftarrow$ min(proposal-set$[r]$)
    trigger cDecide(decided)
    trigger bebBroadcast(DECISION, decided)
  else
    round $\longleftarrow$ round $+1$
    trigger bebBroadcast(MYSET, round, proposal-set[round-1])

Upon bebDeliver(p**, DECISION,** $value$ **) do:**
  if $p \in$ correct $\wedge$ decided $= \perp$ **then**
    decided $\longleftarrow$ $value$
    trigger cDecide(decided)
    trigger bebBroadcast(DECISION, decided)

Uniform Consensus (bcast consensus): Intuition

- The problem with the previous solution is that some process might decide too soon.
- Soon here means that a process decides before being sure that all other processes knowns, or will know in the future, the value that he is deciding.
- The intuition to solve this problem is hence, to delay the decision. But for how long should we delay that decision?

Uniform Consensus (bcast consensus): Intuition

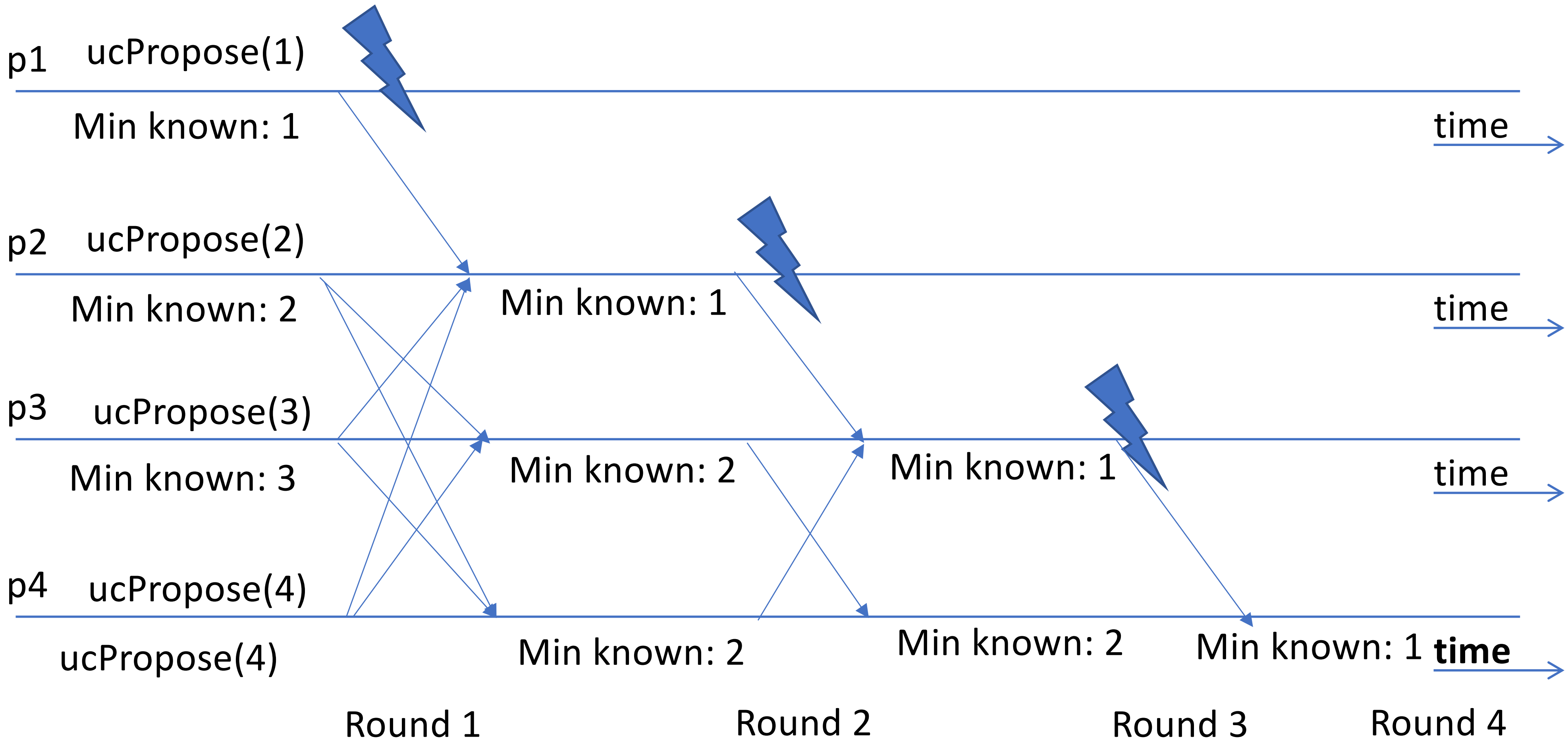
- The problem with the previous solution is that some process might decide too soon.
- Soon here means that a process decides before being sure that all other processes knowns, or will know in the future, the value that he is deciding.
- The intuition to solve this problem is hence, to delay the decision. But for how long should we delay that decision?
- Answer: Only decide in round n.

Uniform Consensus (bcast consensus): Intuition

- Why do we have to wait for round n?

Uniform Consensus (bcast consensus): Intuition

- Why do we have to wait for round n?



Uniform Consensus (bcast consensus)

- Interface:
- Request:
  - ucPropose(V): Used to propose a value v for consensus
- Indication:
  - ucDecide(V): Used to indicate the decided value v for consensus

Uniform Consensus (uniform bcast consensus)

Algorithm 4: Uniform Consensus Algorithm (Fail-Stop/Synchronous)

```
Interface:
    Requests:
        ucPropose( v )
    Indications:
        ucDecide ( v )

State:
    correct //set of correct processes
    round //current round number
    correct-this-round //map, per round, with a set containing the id of
                //processed that participated in that round
    decided //decided value (if any)
    proposal-set //set with all known proposed values

    Upon Init do:
        correct ⟵ Π
        round ⟵ 1
        decided ⟵ ⊥
        proposal-set ⟵ {}
        for all i ∈ 1 to #Π do
            correct-this-round[i] ⟵ {}

    Upon crash( p ) do:
        correct ⟵ correct \ p
        if correct ⊆ correct-this-round[round] ∧ decided = ⊥ then
            Call CheckRoundTermination( )

    Upon ucPropose ( v ) do:
        proposal-set ⟵ proposal-set ∪ {v}
        trigger bebBroadcast( MYSET, 1, proposal-set )

    Upon bebDeliver( p, MYSET, r, set) do:
        correct-this-round[r] ⟵ correct-this-round[r] ∪ {p}
        proposal-set ⟵ proposal-set ∪set
        if r = round ∧ correct ⊆ correct-this-round[r] ∧ decided = ⊥ then
            Call CheckRoundTermination( )

    Procedure CheckRoundTermination( )
        if r = #Π then
            decided ⟵ min(proposal-set)
            trigger ucDecide(decided)
        else
            round ⟵ round +1
            trigger bebBroadcast( MYSET, round, proposal-set)
```

Uniform Consensus (uniform bcast consensus) State and Init

Algorithm 5: Uniform Consensus Algorithm (Fail-Stop/Synchronous) - State and Init

Interface:
  Requests:
    ucPropose(v **)**
  Indications:
    ucDecide (v **)**

State:
```
correct //set of correct processes
round //current round number
correct-this-round //map, per round, with a set containing the id of
            //processed that participated in that round
decided //decided value (if any)
proposal-set //set with all known proposed values
```

Uniform Consensus (uniform bcast consensus) Main Algorithm

Algorithm 6: Uniform Consensus Algorithm (Fail-Stop/Synchronous) - Main Algorithm

Upon crash(p **) do:**
  correct $\longleftarrow$ correct $\setminus p$
  if correct $\subseteq$ correct-this-round$[round]$ $\wedge$ decided $= \perp$ **then**
    Call `CheckRoundTermination( )`

Upon ucPropose (v **) do:**
  proposal-set $\longleftarrow$ proposal-set $\cup \{v\}$
  trigger bebBroadcast(MYSET, 1, proposal-set)

Upon bebDeliver(p**, MYSET,** r**,** set**) do:**
  correct-this-round$[r]$ $\longleftarrow$ correct-this-round$[r]$ $\cup \{p\}$
  proposal-set $\longleftarrow$ proposal-set $\cup set$
  if $r =$ round $\wedge$ correct $\subseteq$ correct-this-round$[r]$ $\wedge$ decided $= \perp$ **then**
    Call `CheckRoundTermination( )`

Procedure `CheckRoundTermination( )`
  if $r = \#\Pi$ **then**
    decided $\longleftarrow$ min(proposal-set)
    trigger ucDecide(decided)
  else
    round $\longleftarrow$ round $+1$
    trigger bebBroadcast(MYSET, round, proposal-set)

Uniform Consensus (uniform bcast consensus): Correctness argument

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Uniform Agreement: No two processes decide differently.

- Termination: All correct processes reach round n:
  1. processes that do not fail will keep moving from round to round (failures will be detected for sure);

Uniform Consensus (uniform bcast consensus): Correctness argument

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Uniform Agreement: No two processes decide differently.

- Validity: Follows from the algorithm.
  1. To decide a value, that value has to be in the set of known proposals.
  2. For the value to be in that set, then some process broadcasted that value in its set.
  3. The only way to add a value to the set of proposals of a process, other than receiving the value from another process, is to the value to be proposed by that process.

Uniform Consensus (uniform bcast consensus): Correctness argument

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Uniform Agreement: No two processes decide differently.

- Integrity: Follows from the algorithm.
    1. A decision can only made if *decided* has a bottom value.
    2. When a process decides, its *decided* state becomes the decided value (hence different from bottom).

Uniform Consensus (uniform bcast consensus): Correctness argument

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Uniform Agreement: No two processes decide differently.

- Agreement: By construction of the algorithm
  - All processes that reach round n will have the same set of values in their local proposal-set.
  - Since the value to be decided is picked using a deterministic function, all process will decide the same value.

A bit more about consensus.

- Are these the only algorithms for solving regular consensus and uniform consensus?
- Are these the best algorithms to solve regular consensus and uniform consensus?

A bit more about consensus.

- Are these the only algorithms for solving regular consensus and uniform consensus?
  - No, there are multiple variants.
- Are these the best algorithms to solve regular consensus and uniform consensus?
  - No, in particular the complexity of messages is:
  - Bcast consensus: $N^2 + N^2 + N^2$ x #faults
  - Uniform bcast consensus: N^3
- **BUT**: The lower bound for solving consensus in synchronous fail-stop environment tolerating f faults, and where f faults happen is f+1 rounds (which is exactly what bcast consensus provides).

So now how do we use this to achieve State Machine Replication

- Servers receive operations from clients.
- Servers broadcast operations to all replicas.
- They run consensus among them, in which each process proposes an operation (or **ordered** set of operations) to be executed.

  (An independent instance of consensus is executed for each operation/set of operations to be executed, this can also be optimized, we will discuss that later)
- They all decide the same operation/set of operations to execute.
- They continue doing this forever, and by the properties of consensus each time they all decide the same operation/set of operations to be executed step by step.