Algorithms and Distributed Systems 2023/2024 (Revision for First Midterm)

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

Based on slides from João Leitão

Up to now in ASD

- Models for distributed computing
- Formal properties of Protocols
- Network Link Abstractions
- Reliable broadcast (Regular and Uniform Variants)
- Probabilistic Reliable Broadcast
- Unstructured Overlay Networks
- Structured Overlay Networks (DHTs)
- Replication Strategies
- 1W-NR Regular Register Replication
- Quorum-based Replication and Quorum Systems
- Consensus in Synchronous Systems (Regular and Uniform Variants)

Rules for the Midterm

- 2h hours.
- You can consult up to 2 sides of A4 that must be **handwritten.**
- You will answer directly on the pages of the Midterm.

- Midterm will be on the 27th October @ 18h30.
- Show up in the test room at least 5 minutes before
- Room is 127– Ed. II.

Typical Structure of the Midterm

- 1 Group with Agree/Disagree questions that must have a valid justification (incorrect answers do not discount).
- Several Groups of open questions focusing on the contents of the lecture:
  - Analyze pseudo-code.
  - Write pseudo-code.
  - Show general knowledge.
  - Reason about a particular scenario.
  - Propose a solution for a problem.

Revision material…

Timing assumptions

Two fundamental models

Synchronous System:

- Assumes that there is a known *upper bound* to the time required to deliver a message through the network and for a process to make all computations related with the processing of the message.

Asynchronous System:

- There are *no assumptions* about the time required to deliver a message or process a message.

Partially synchronous:

- Events complete, *eventually*.

Process Fault Model

- A process that never fails, is considered **correct.**
- **Correct processes** never deviate from their expected/prescribed behavior.
  - It executes the algorithm as expected and sends all messages defined by it.
- **Failed (or Faulty) processes** might deviate from their prescribed behavior in different ways.
  - The *unit of failure* is the process, meaning that when it fails, all its component fail at the same time.
- The (possible/considered) behaviors of a faulty process is defined by the process fault model.

Process Fault Models

- Crash Fault Model:
  - When a process fails it stops sending any messages (from that point onward).
  - This is the fault model that we will consider most of the times.
- Omission Fault Model:
  - A process that fails omits the transmission (or reception) of any number of messages (potentially not all of them).
- Fail-Stop Model:
  - Similar to the crash model, except that upon failure the process “notifies” all other processes of its own failure

    (only possible when considering a synchronous system).

Network Model

- The Network Model captures the assumptions made concerning the **links** that interconnect processes.
- Namely it captures what can go wrong in the network regarding:
  - Messages sent between processes being lost.
  - Possibility of duplication of messages.
  - Possibility for corruption of messages.

Network Model

- **Fair-loss**

- **Stubborn**

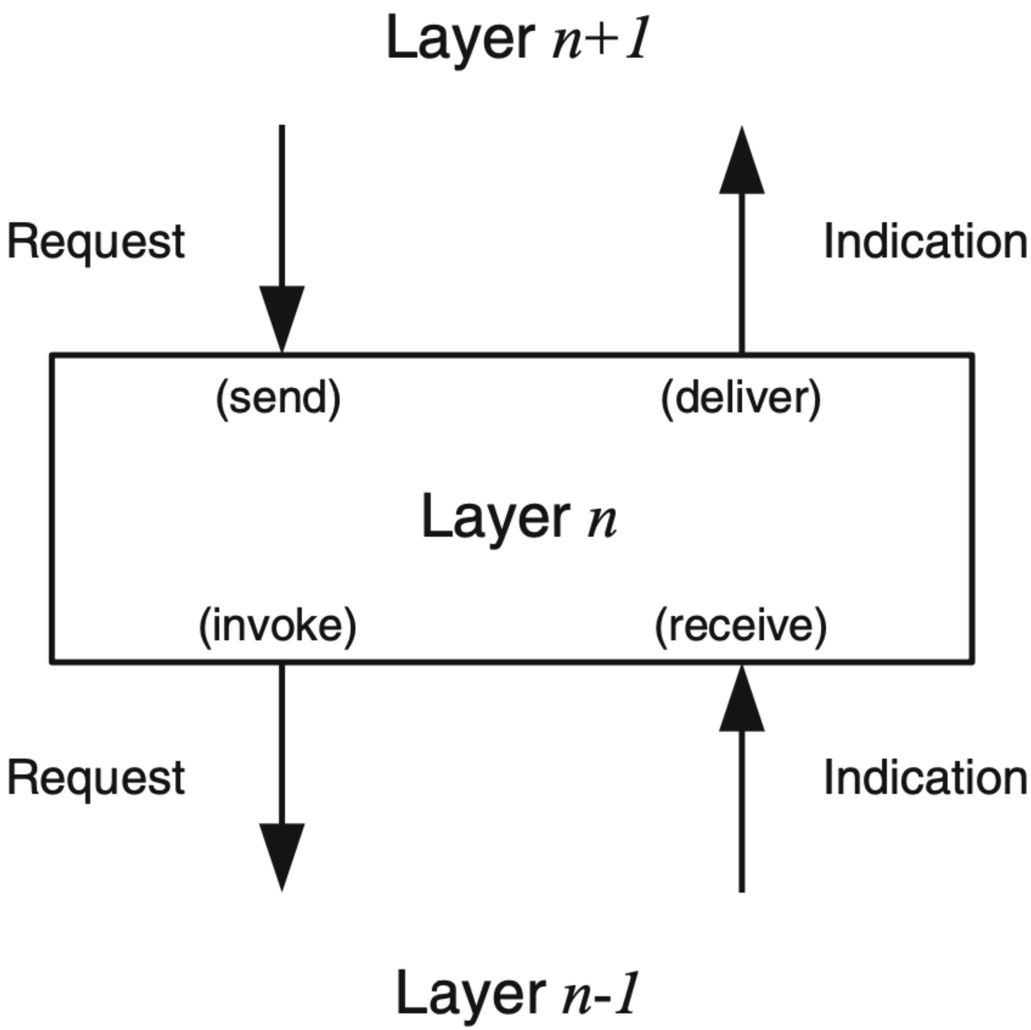
- **Perfect Link**
    - Example properties:
        - PL1 (Reliable Delivery): Considering two correct processes ***i*** and ***j***; if ***i*** sends a message to ***j***, then ***j eventually*** delivers *m*.
        - PL2 (No Duplication): No message is delivered by a process more than once.
        - PL3 (No Creation): If a correct process ***j*** delivers a message *m*, then *m* was sent to ***j*** by some process ***i.***

General points to know

- Under what conditions are certain models more favourable?

- Which models provide more accurate frameworks of the *real-world?*

Algorithmic Layering



Two important distinctions:

- Requests
- Indications

Broadcasts

- Should know the goals and the differences, but don't need to memorise every property in full
- **Best-effort**
- **Reliable**
- **Uniform**
- **Probabilistic**

Pseudocode

Algorithm 1: (Efficient) Reliable Broadcast (Crash Fault Model / Synchronous System (uses PP2PLink)

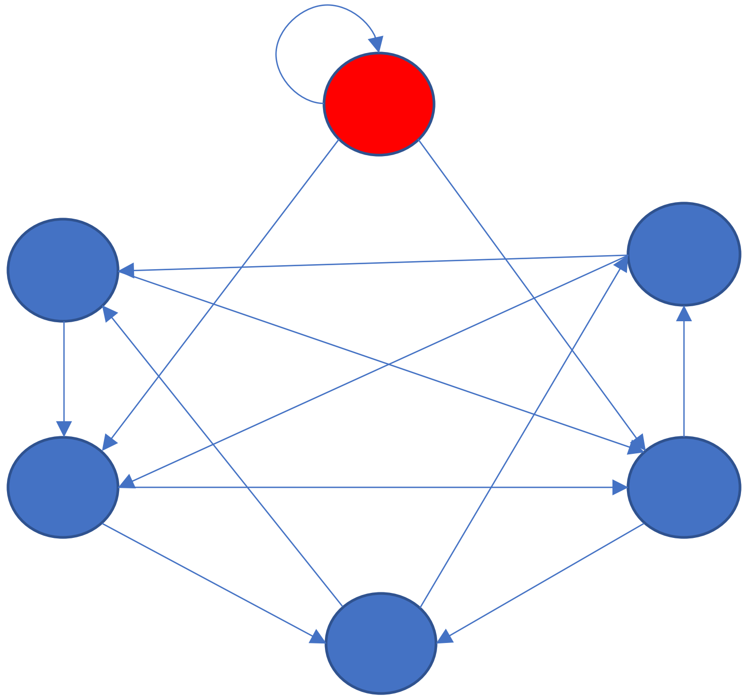
```
Interface:
    Requests:
        rBroadcast ( m )
    Indications:
        rBcastDeliver ( s, m ) //s is the sender, m the message

State:

    myself //my own identifier
    correct //correct processes identifiers
    delivered //messages already delivered
    messages //Map that associates to each process p the messages dependent on it

    Upon Init (Π, self) do:
        myself ⟵ self
        correct ⟵ Π
        delivered ⟵ {}
        Foreach p ∈ correct do:
            messages[p] ⟵ {}

    Upon rBroadcast ( m ) do:
        Trigger rBcastDeliver ( myself, m)
        delivered ⟵ delivered ∪ {m}
        p ⟵ p ∈ correct: p < p′, ∀p′ ∈ correct: p ≠ p′ ∧ p ≠ myself
        If p ≠ ⊥ Then
            Trigger Send( BCAST, p, self, m )
            messages[p] ⟵ messages[p] ∪ {(myself, m)}

    Upon Receive( BCAST, s, p, m ) do:
        If m ∉ delivered Then
            delivered ⟵ delivered ∪ {m}
            d ⟵ d ∈ correct: d > myself ∧ d ≠ p ∧ d < d′, ∀d′ ∈ correct: d ≠ d′ ∧ d′ ≠ p ∧ d′ > myself
            If d ≠ ⊥ Then
                Trigger Send( BCAST, d, p, m )
                messages[d] ⟵ messages[d] ∪ {(p, m)}

    Upon Crash ( p ) do:
        correct ⟵ correct \ {p}
        Foreach (s, m) ∈ messages[p] do:
            d ⟵ d ∈ correct: d > p ∧ d ≠ s ∧ d ≠ myself ∧ d < d′, ∀d′ ∈ correct: d ≠ d′ ∧ d′ ≠ s ∧ d′ > p
            If d ≠ ⊥ Then
                Trigger Send( BCAST, d, s, m )
                messages[d] ⟵ messages[d] ∪ {(s, m)}
```

- You should be familiar reading and writing it
- You will be asked to comment on certain pieces of code
- You will be asked to write (simple) pseudocode

Gossip protocols

Key concepts:

- Fanout
- Eager Push, Pull, Lazy push
- Overlays
- Optimisations (Anti-entropy etc.)



Resource location

Unstructured overlays

- Flooding vs Super-peers
- HyParView general idea (Active/Passive view(and improvements

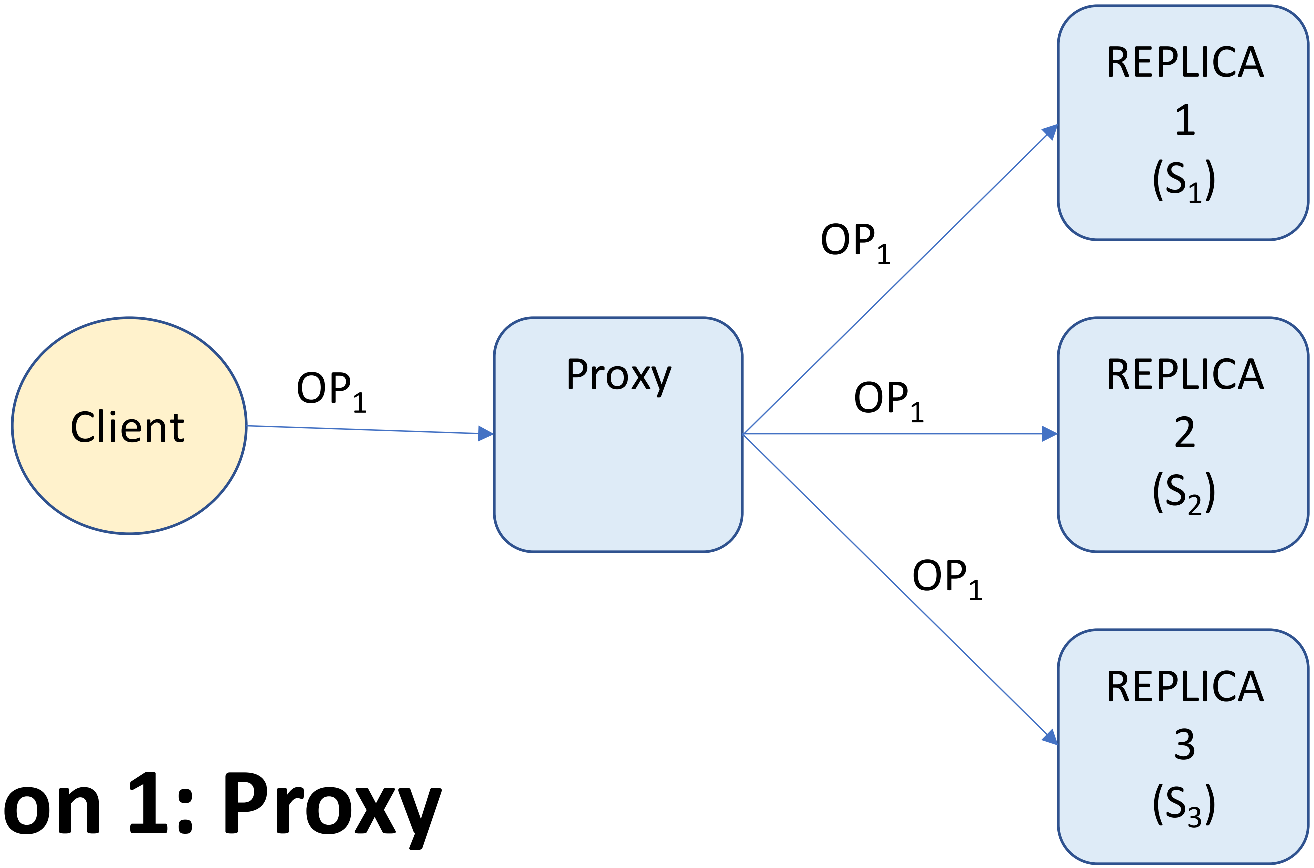
Structured overlays

- Distributed hash tables
- Examples (Chord, Kadmelia, etc)
- What circumstances are structured overlays preferable?

Replication

- All other aspects concern Broadcast, Membership, Resoure Location
- Replication concerns State
- **Fault-tolerance**
- **Performance**

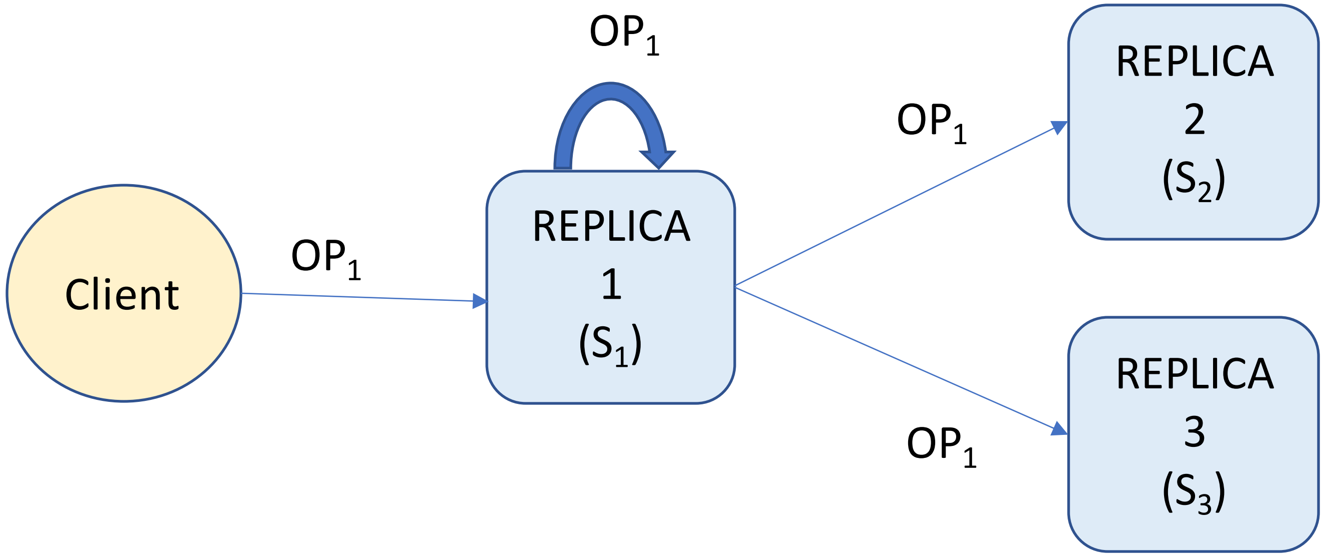
Transparency (architectural solutions)



Solution 1: Proxy

(Used in most Web Applications)

Transparency (architectural solutions)



Solution 2: Only one replica receives operations and interacts with the client

Replication: Strategies

- First Dimension:
  - Active Replication: Operations are executed by all replicas
  - Passive Replication: Operations are executed by a single replica, results are shipped to other replicas
- Second Dimension:
  - Synchronous Replication: Replication takes place before the client gets a response.
  - Asynchronous Replication: Replication takes place after the client gets a response.
- Third Dimension:
  - Primary-Backup: A single replica receives operations from clients that modify the state.
  - Multi-Primary: Any replica can receive and process any operation issued by a client.

A simplistic replication algorithm

Register Replication:

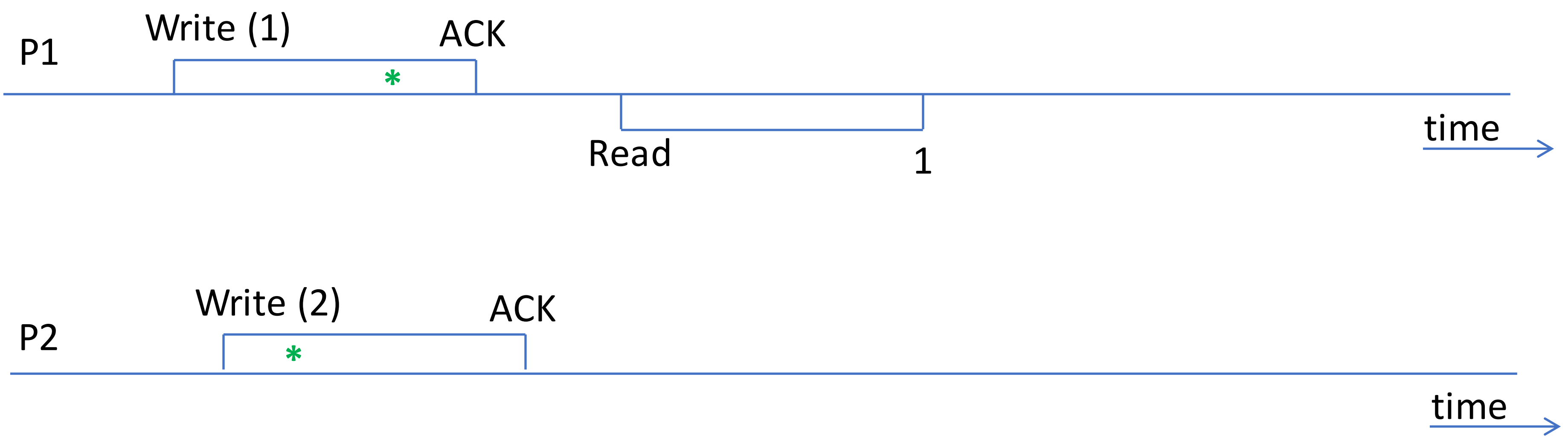
- A set of processes own a register, which store a single value (lets assume a positive integer value) initially set to zero.
- Processes have two operations: *read* and *write*.
- Each process has its own local copy of the register, but the register is shared among all processes.
- Processes invoke operations sequentially (each process executes one operation at a time).
- Values written to the register are uniquely identified (e.g., the id of the process performing the write and a timestamp or some monotonic [i.e., sequence] value).

Register Replication

Properties (high level):

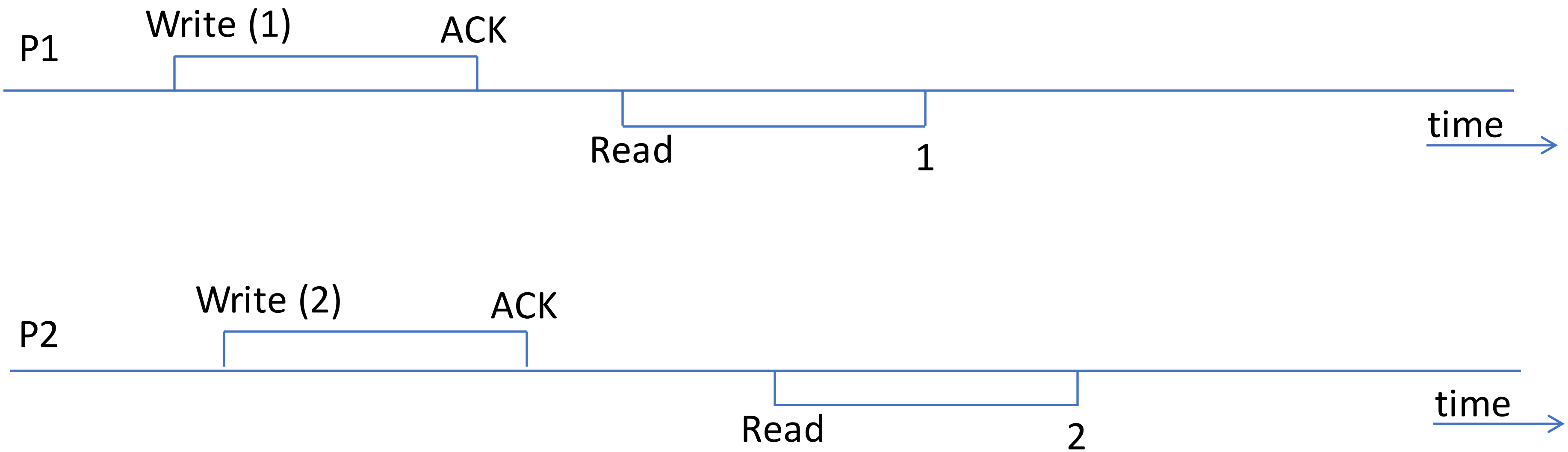
- **Liveness**: Every operation of a correct process eventually completes.
- **Safety**: Every read operation returns the *last value written*.

What does **last written value** means in this context.



This is a valid execution, since write operations are concurrent, we must define serialization points to arbiter their order.

What does **last written value** means in this context.



This is not a valid execution, there are no serialization points that explain the return of those two reads.

(1,N) Read-Write Register

Algorithm 2: Regular Register (1, N): Read and Write Operations

```
State:
   value //local copy of the register (it's value)
   writeSet //set containing identifiers of processes that have acknowledge
          //the write in progress
   correct //set containing the identifiers of processes that have not crashed

   Upon Init () do:
      value ⟵ 0;
      writeSet ⟵ {};
      correct ⟵ Π;

   Upon rregRead( ) do:
      Trigger rregReadReturn ( value );

   Upon rregWrite( v ) do:
      Trigger bebBroadcast ( {WRITE, v} );

   Upon bebDeliver ( s, {WRITE, v} ) do:
      value ⟵ v;
      Trigger pp2pSend( s, ACK );

   Upon pp2pDeliver ( s, ACK ) do:
      writeSet ⟵ writeSet ∪ {s};
      Call CHECKACKS();

   Procedure CheckAcks( ):
      if correct ⊆ writeSet do:
         writeSet ⟵ {};
         Trigger rregWriteReturn ( );

   Upon Crash ( p ) do:
      correct ⟵ correct \ {p};
      Call CHECKACKS();
```

Quorum Based Replication

Replication algorithms that are based on quorums execute operations over a *large-enough* replica set such that any two concurrent operations will have a **non-empty intersection**.

Quorum Types: Majority

- Replication strategy based on a quorum system where:
  - Every operation (either read or write) must be executed across a majority of replicas (>N/2).

- Properties:
  - Best fault tolerance possible from a theoretical point of view (can tolerate up to f faults with N >= 2f+1).
  - Read and Write operations have a similar cost.

Quorum Types: Weighted Voting

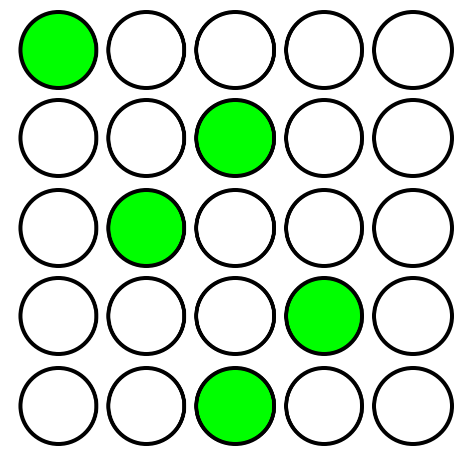
- A Replication strategy based on a read-write quorum system where:
  - To each replica i, it is assigned a weight w_i: $\Sigma w_i = w_{total}$ defining also the weight required for performing a read, w_R, and the weight required for performing a write operation, w_W, such that:

    $$w_R + w_w > w_{total} \quad \text{AND} \quad w_w + w_w > w_{total}$$
  - A read quorum can be composed by any subset of replicas such that $R=\{r_1, r_2, \ldots, r_m\}$: $\Sigma w(r_i) \geq w_R$
  - A write quorum can be composed by any subset of replicas $W=\{w_1, w_2, \ldots, w_m\}$: $\Sigma w(w_i) \geq w_w$

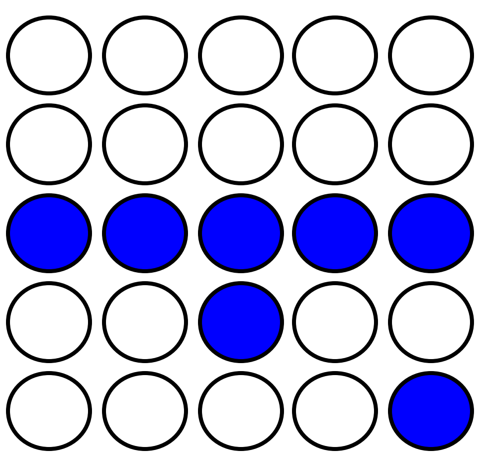
- Properties:
  - This allows to balance the size of different quorums for different read and write operations.
  - Replicas are no longer completely equivalent among them, meaning that the fault model is also not uniform (the failure of a given process might have a different impact in the availability of the system than the failure of a different process).

Quorum Types: Grid

- Processes are organized (logically) in a grid such that:



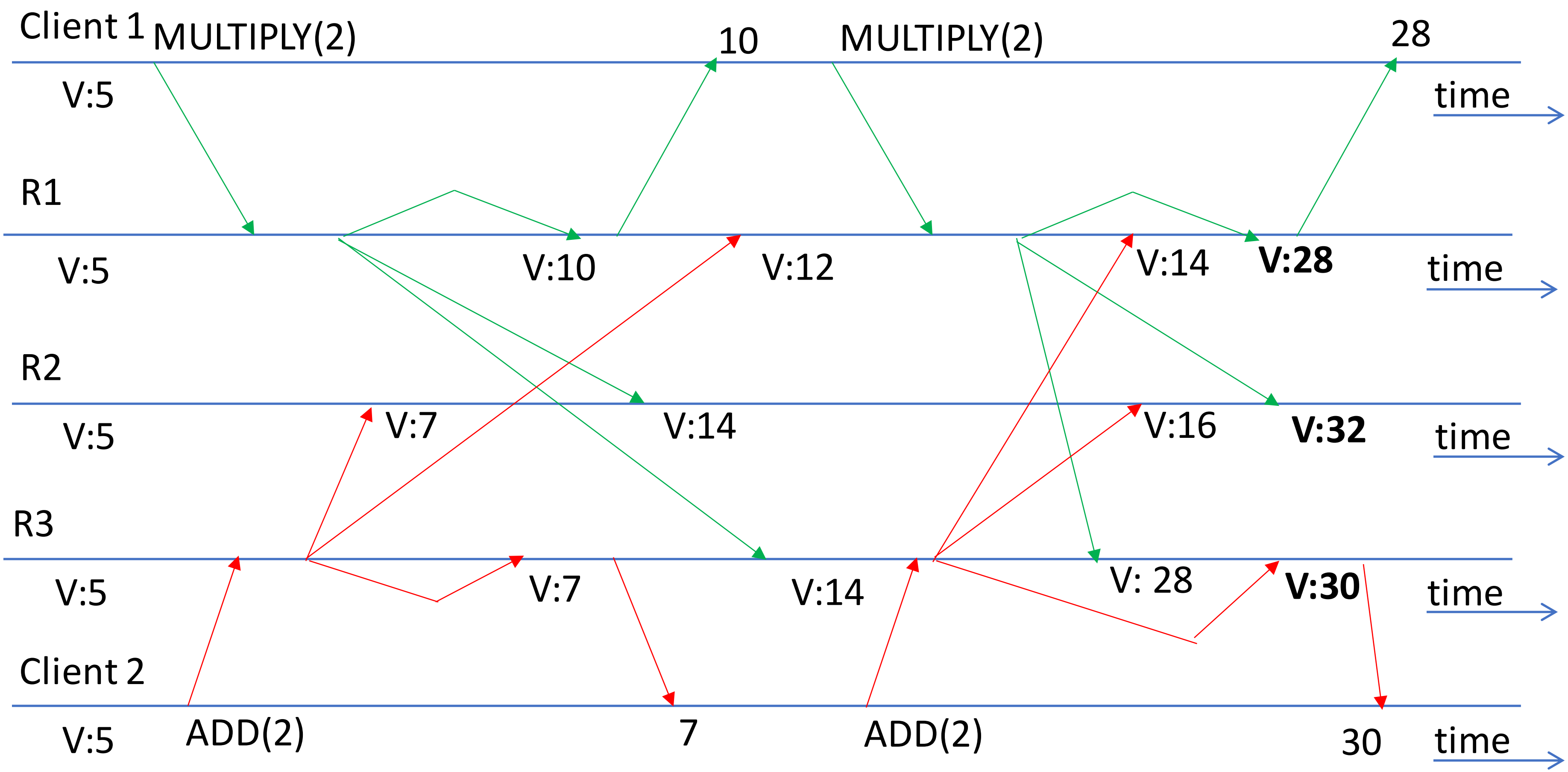
Read Quorum:
One element from each line



Write Quorum:
Full line + one element from each of the lines below that one.

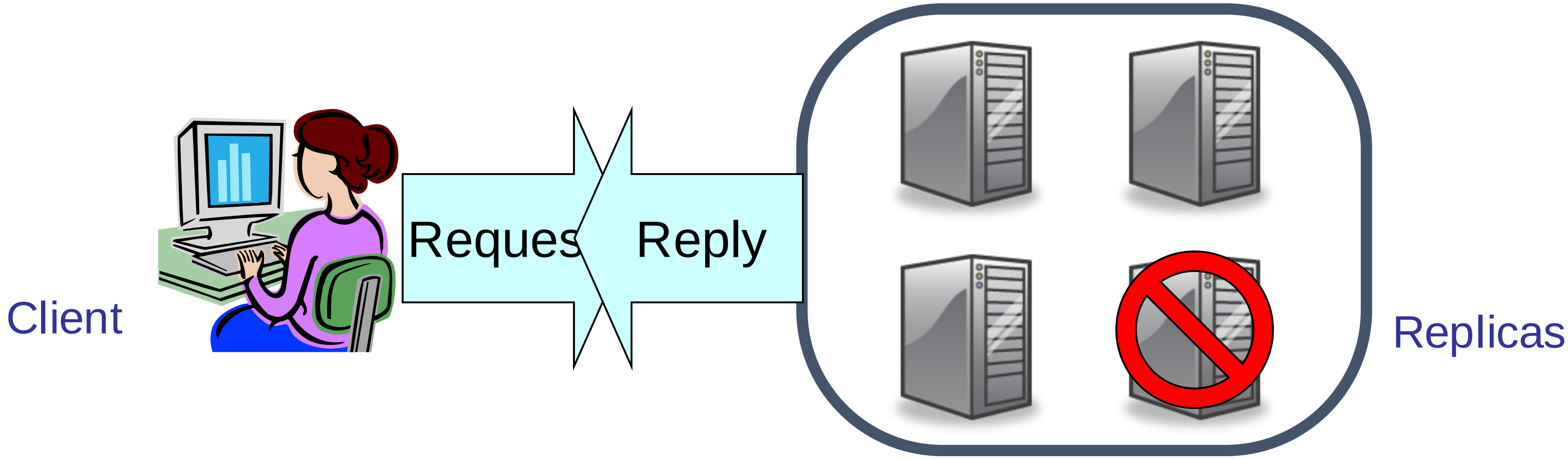
Replicating an Integer

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



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.



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.

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])

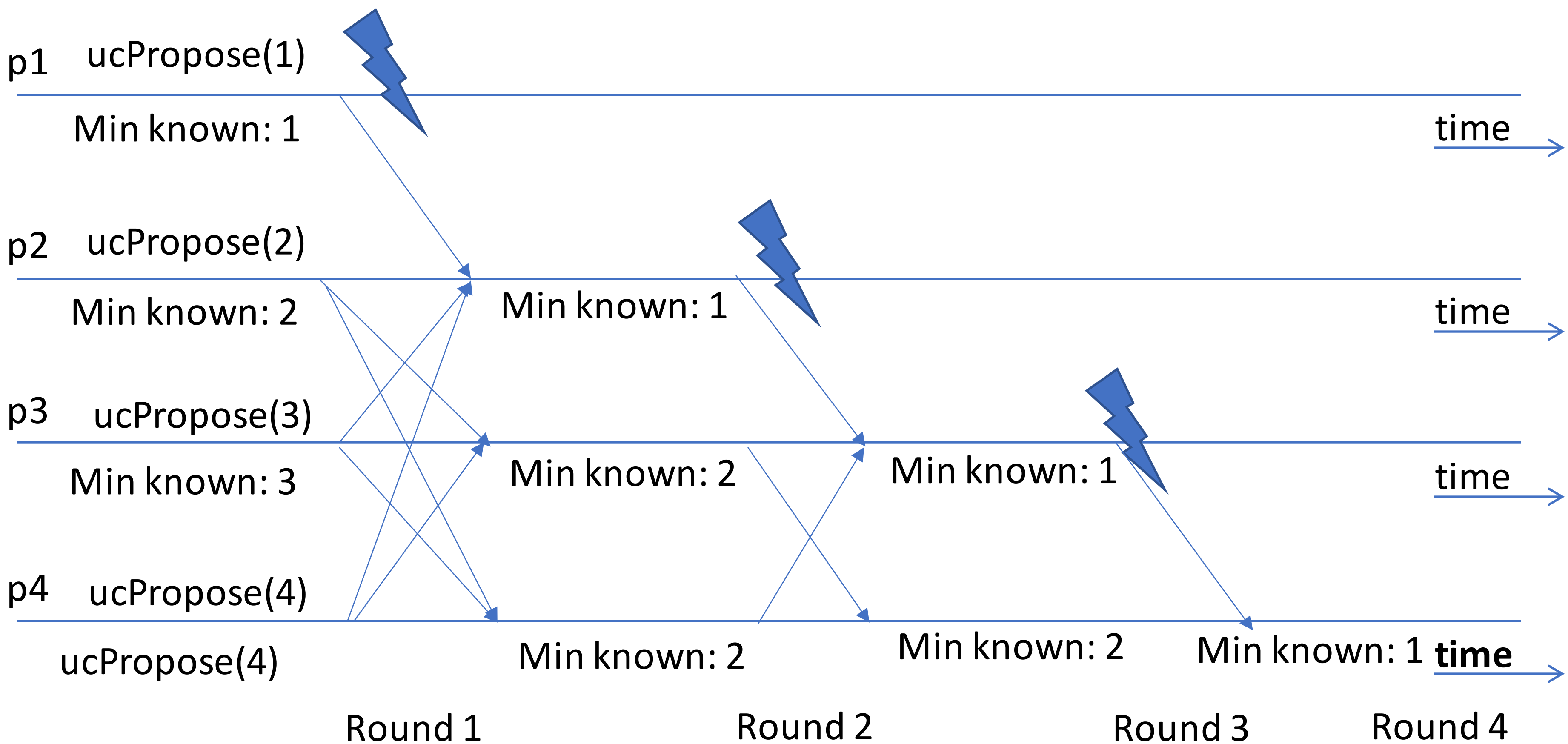
Upon bebDeliver(p**, MYSET,** r**,** set**) do:**
    correct-this-round$[r]$ $\longleftarrow$ correct-this-round$[r]$ $\cup \{p\}$
    proposal-set$[r]$ $\longleftarrow$ proposal-set$[r]$ $\cup set$
    if $r =$ round $\wedge$ correct $\subseteq$ correct-this-round$[r]$ $\wedge$ decided $= \perp$ **then**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

Uniform Consensus (bcast consensus): Intuition

- Why do we have to wait for round n?



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.

Remember

- The algorithms you see are **informative**
- They are **not** necessarily the **most efficient!**
- Try to understand why and how improvements can be made, and with respect to what parameters
  - Number of messages
  - Size of messages
  - Latency