Algorithms and Distributed Systems 2019/2020 (Lecture Four)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Leitão (jc.leitao@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Lecture structure:

- Epidemic Broadcast + Recovery (Project related)
- Replication (Why and How?)
- Register Replication
- Quorums and Quorums Systems

Epidemic Broadcast + Recovery

- Very similar to the Probabilistic Reliable Broadcast:
    - A Crash fault model and asynchronous system.
    - You have to use gossip (in the project flood).
    - Assume a partial view-based membership service that has the request getNeighbors(), whose indication is neighbors(N), where N is the set of process identifiers of the local process.
    - The membership is most of time correct, but for some time connectivity might not be ensured.
    - *[Performance Aspects:]*
        - *Assume that messages being broadcasted are large.*
        - *Try to minimize the use of the network and reduce the time required to disseminate the message to everyone.*

Algorithm 1: Reliable Broadcast I – Push Gossip + Anti-Entropy

```
Interface:
    Requests:
        rBroadcast ( $m$ )
    Indications:
        rBcastDeliver ( $m$ )

State:
    t //fanout of the protocol
    delivered //set of Ids of messages and respective payload already delivered
    pending //Messages to be forwarded
    currentNeighbors //Set of neighbors received on the last Neighbors up call

    Upon Init () do:
        t $\longleftarrow \ln(\Pi)$;
        delivered $\longleftarrow \{\}$;
        pending $\longleftarrow \{\}$;
        Setup Periodic Timer AntiEntropy ( T ); //T is a fixed configuration parameter
        currentNeighbors $\longleftarrow \perp$;

    Upon rBroadcast( $m$ ) do:
        mid $\longleftarrow$ generateUniqueID(m);
        Trigger rBcastDeliver ($m$);
        delivered $\longleftarrow$ delivered $\cup \{(mid, m)\}$;
        pending $\longleftarrow$ pending $\cup \{(mid, m, myself)\}$;
        Trigger GetNeighbors( );

    Upon Neighbors( $N$ ) do:
        currentNeighbors $\longleftarrow N$;
        Foreach (mid, m, sender) $\in$ pending do:
            gossipTargets $\longleftarrow$ randomSelection(t, ($N \backslash sender$) );
            Foreach $p \in$ gossipTargets do:
                Trigger Send( GossipMessage, $p$, $mid$, $m$ );
        pending $\longleftarrow \{\}$;

    Upon Receive (GossipMessage, $s$, $mid$, $m$ ) do:
        If (mid, m) $\notin$ delivered do:
            delivered $\longleftarrow$ delivered $\cup \{(mid, m)\}$;
            Trigger rBcastDeliver ($m$);
            pending $\longleftarrow$ pending $\cup \{(mid, m, s)\}$;
            Trigger GetNeighbors( );

    Upon AntiEntropy ( ) do:
        If currentNeighbors $\neq \perp$ then
            p $\longleftarrow$ randomPick(currentNeighbors);
            knownMessages $\longleftarrow \{\}$;
            ForEach (mid, m) $\in$ delivered do:
                knownMessages $\longleftarrow$ knownMessages $\cup \{mid\}$;
            Trigger Send(AntiEntropyMsg, p, knownMessages);

    Upon Receive (AntiEntropyMsg, $s$, $kmsgs$ ) do:
        ForEach (mid, m) $\in$ delivered do:
            If mid $\notin$ kmsgs Then
                Trigger Send ( GossipMessage, $s$, $mid$, $m$ );
```

Algorithm 2: Reliable Broadcast `II` – Lazy Push Gossip + Anti-Entropy (Part 1/2)

```
Interface:
    Requests:
        rBroadcast ( $m$ )
    Indications:
        rBcastDeliver ( $m$ )

State:
    t //fanout of the protocol
    delivered //set of Ids of messages and respective payload already delivered
    pending //Messages to be forwarded
    requested //Messages ids that were already requested
    currentNeighbors //Set of neighbors received on the last Neighbors up call

    Upon Init () do:
        t $\longleftarrow$ ln($\Pi$);
        delivered $\longleftarrow$ $\{\}$;
        pending $\longleftarrow$ $\{\}$;
        Setup Periodic Timer AntiEntropy ( T ); //T is a fixed configuration parameter
        requested $\longleftarrow$ $\{\}$;
        currentNeighbors $\longleftarrow$ $\perp$;

    Upon rBroadcast( $m$ ) do:
        mid $\longleftarrow$ generateUniqueID(m);
        Trigger rBcastDeliver ($m$);
        delivered $\longleftarrow$ delivered $\cup \{(mid, m)\}$;
        pending $\longleftarrow$ pending $\cup \{(mid, m, myself)\}$;
        Trigger GetNeighbors( );

    Upon Neighbors( $N$ ) do:
        currentNeighbors $\longleftarrow$ $N$;
        Foreach (mid, m, sender) $\in$ pending do:
            gossipTargets $\longleftarrow$ randomSelection(t, $(N \backslash sender)$ );
            Foreach $p$ $\in$ gossipTargets do:
                Trigger Send( GossipAnnouncement, $p$, $mid$);
        pending $\longleftarrow$ $\{\}$;

    Upon Receive (GossipAnnouncement, $s$, $mid$) do:
        If (mid, -) $\notin delivered \wedge mid \notin requested$ Then
            requested $\longleftarrow$ requested $\cup \{mid\}$;
            Trigger Send( GossipRequest, $s$, $mid$);

    Upon Receive (GossipRequest, $s$, $mid$) do:
        m $\longleftarrow$ m' : (mid', m') $\in$ delivered $\wedge$ mid $=$ mid';
        Trigger Send( GossipMessage, $s$, $mid$, $m$ );
```

Algorithm 3: Reliable Broadcast `II` – Lazy Push Gossip + Anti-Entropy (Part 2/2)

```
Upon Receive (GossipMessage, $s$, $mid$, $m$ ) do:
    If (mid, m) $\notin$ delivered do:
        delivered $\longleftarrow$ delivered $\cup \{(mid, m)\}$;
        requested $\longleftarrow$ requested $\setminus$ mid;
        Trigger rBcastDeliver ($m$);
        pending $\longleftarrow$ pending $\cup \{(mid, m, s)\}$;
        Trigger GetNeighbors( );

Upon AntiEntropy ( ) do:
    If currentNeighbors $\neq \perp$ then
        p $\longleftarrow$ randomPick(currentNeighbors);
        knownMessages $\longleftarrow$ $\{\}$;
        ForEach (mid, m) $\in$ delivered do:
            knownMessages $\longleftarrow$ knownMessages $\cup \{mid\}$;
        Trigger Send(AntiEntropyMsg, p, knownMessages);

Upon Receive (AntiEntropyMsg, $s$, $kmsgs$ ) do:
    ForEach (mid, m) $\in$ delivered do:
        If mid $\notin$ kmsgs Then
            Trigger Send ( GossipMessage, $s$, $mid$, $m$ );
```

Algorithm 4: Reliable Broadcast III – Eager Push/Lazy Push Gossip + Anti-Entropy (Part 1/2)

```
Interface:
    Requests:
        rBroadcast ( $m$ )
    Indications:
        rBcastDeliver ( $m$ )

State:
    t //fanout of the protocol
    delivered //set of Ids of messages and respective payload already delivered
    pending //Messages to be forwarded
    requested //Messages ids that were already requested
    currentNeighbors //Set of neighbors received on the last Neighbors up call
    r //Number of re-transmissions made in eager push before commuting to lazy p

    Upon Init () do:
        t $\longleftarrow \ln(\Pi)$;
        delivered $\longleftarrow \{\}$;
        pending $\longleftarrow \{\}$;
        Setup Periodic Timer AntiEntropy ( T ); //T is a fixed configuration parameter
        requested $\longleftarrow \{\}$;
        currentNeighbors $\longleftarrow \perp$;
        r $\longleftarrow$ 3; Assuming lots of nodes, could be different value

    Upon rBroadcast( $m$ ) do:
        mid $\longleftarrow$ generateUniqueID(m);
        Trigger rBcastDeliver ($m$);
        delivered $\longleftarrow$ delivered $\cup \{(mid, m, 0)\}$;
        pending $\longleftarrow$ pending $\cup \{(mid, m, 0, myself)\}$;
        Trigger GetNeighbors( );

    Upon Neighbors( $N$ ) do:
        currentNeighbors $\longleftarrow N$;
        Foreach (mid, m, sender, hop) $\in$ pending do:
            gossipTargets $\longleftarrow$ randomSelection(t, ($N \setminus sender$) );
            Foreach $p \in$ gossipTargets do:
                If hop $\leq$ r Then
                    Trigger Send( GossipMessage, $p$, $mid$, $hop + 1$ $m$);
                Else
                    Trigger Send( GossipAnnouncement, $p$, $mid$);
        pending $\longleftarrow \{\}$;

    Upon Receive (GossipAnnouncement, $s$, $mid$) do:
        If (mid, -, -) $\notin delivered \wedge mid \notin requested$ Then
            requested $\longleftarrow$ requested $\cup \{mid\}$;
            Trigger Send( GossipRequest, $s$, $mid$);

    Upon Receive (GossipRequest, $s$, $mid$) do:
        (mid, m, hop) $\longleftarrow$ (mid', m', hop') $\in$ delivered $\wedge$ mid $=$ mid';
        Trigger Send( GossipMessage, $s$, $mid$, $m$, $hop + 1$ );
```

Algorithm 5: Reliable Broadcast `III` – Eager Push/Lazy Push Gossip + Anti-Entropy (Part 2/2)

```
Upon Receive (GossipMessage, $s$, $mid$, $m$, $hop$ ) do:
    If (mid, m) $\notin$ delivered do:
        delivered $\longleftarrow$ delivered $\cup \{(mid, m, hop)\}$;
        requested $\longleftarrow$ requested $\setminus$ mid;
        Trigger rBcastDeliver ($m$);
        pending $\longleftarrow$ pending $\cup \{(mid, m, s, hop)\}$;
        Trigger GetNeighbors( );

Upon AntiEntropy ( ) do:
    If currentNeighbors $\neq \perp$ then
        p $\longleftarrow$ randomPick(currentNeighbors);
        knownMessages $\longleftarrow$ $\{\}$;
        ForEach (mid, m, hop) $\in$ delivered do:
            knownMessages $\longleftarrow$ knownMessages $\cup \{mid\}$;
        Trigger Send(AntiEntropyMsg, p, knownMessages);

Upon Receive (AntiEntropyMsg, $s$, $kmsgs$ ) do:
    ForEach (mid, m, hop) $\in$ delivered do:
        If mid $\notin$ kmsgs Then
            Trigger Send ( GossipMessage, $s$, $mid$, $m$, $hop + 1$ );
```

Up to now…

- We have seen how to address some issues in distributed systems under different system models (fault models and synchrony models):
  - Broadcast problem (information dissemination)
  - Membership problem (who is part of the system)
  - Resource Location problem (what is where)

- But in some sense, we have not yet focused on a important aspect of distributed systems

Up to now…

- We have seen how to address some issues in distributed systems under different system models (fault models and synchrony models):
  - Broadcast problem (information dissemination)
  - Membership problem (who is part of the system)
  - Resource Location problem (what is where)

- But in some sense, we have not yet focused on a important aspect of distributed systems

State

Think about a distributed database

- It has multiple processes (most likely in different machines, or nodes).
- Each process is responsible for managing a fraction of the data in the database (using consistent hashing for instance – we already covered this).
- What happens if one of these processes fails?

Think about a distributed database

- It has multiple processes (most likely in different machines, or nodes).
- Each process is responsible for managing a fraction of the data in the database (using consistent hashing for instance – we already covered this).
- What happens if one of these processes fails?
- What can we do to ensure that the system operates as expected despite failures?

Think about a distributed database

- It has multiple processes (most likely in different machines, or nodes).
- Each process is responsible for managing a fraction of the data in the database (using consistent hashing for instance – we already covered this).
- What happens if one of these processes fails?
- What can we do to ensure that the system operates as expected despite failures?
- Answer is: **Replication**

Think about a distributed database

- It has multiple processes (most likely in different machines, or nodes).
- Each p raction of the hashi is).
- What ?
- What can we do to ensure that the system operates as expected despite failures?
- Answer is: **Replication**

This is generalizable for many distributed systems other than databases… since state is a key aspect of many systems.

Replication

- In its essence **replication** entails having multiple copies of “something” across multiple processes.
- Can we only replicate data?

Replication

- In its essence **replication** entails having multiple copies of something across multiple processes.
- Can we only replicate data?
- No
  - Data replication
  - Computation replication (e.g., Seti@Home)

Replication

- In its essence **replication** entails having multiple copies of something across multiple processes.
- Can we only replicate data?
- No
  - Data replication (**in this course we focus on this**)
  - Computation replication (e.g., Seti@Home)

Replication

- Why do we want to replicate?

Replication

- Why do we want to replicate?

- **Fault tolerance**: If some replicas fail, the system does not lose information and clients can still interact with the system (and modify its state)

Replication

- Why do we want to replicate?

- **Fault tolerance**: If some replicas fail, the system does not lose information and clients can still interact with the system (and modify its state)

- **Performance**: If there are many clients issuing many operations, a single process might not be enough to handle the whole load with adequate latency.

Replication (Model)

- A process has a given state S, and a set of operations *Ops* = {Op_1, Op_2… Op_n} that return or modify that state (*Read operations* and *Write operations*, respectively).
- The process (logic) is replicated, meaning that there are multiple copies. Lets assume that the set of all replicas is known and static: **π** and that **#π = n**.
- Clients (processes outside the **set π**) invoke operations from the set *Ops* over the system.

Replication Algorithm (or Protocol)

- A replication algorithm is responsible for managing the multiple replicas of the process.
  - Under a given fault model.
  - Under a given synchrony model.

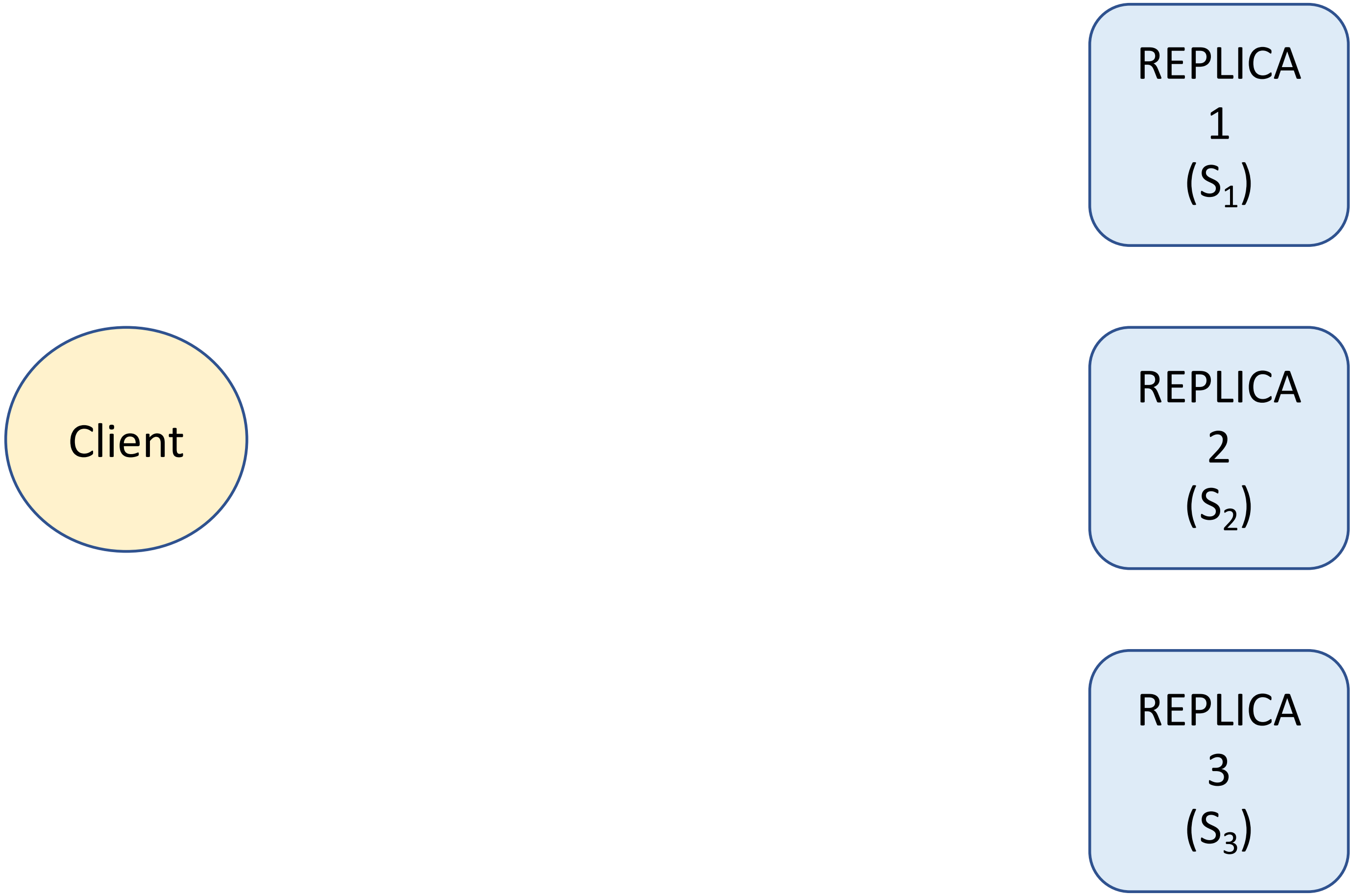
- In its essence the replication algorithm will enforce properties over what are the effects of operations observed by clients given the story of the system (and potentially the story of the cliente issuing a particular operation).

Replication Algorithm (or Protocol) – Cont.

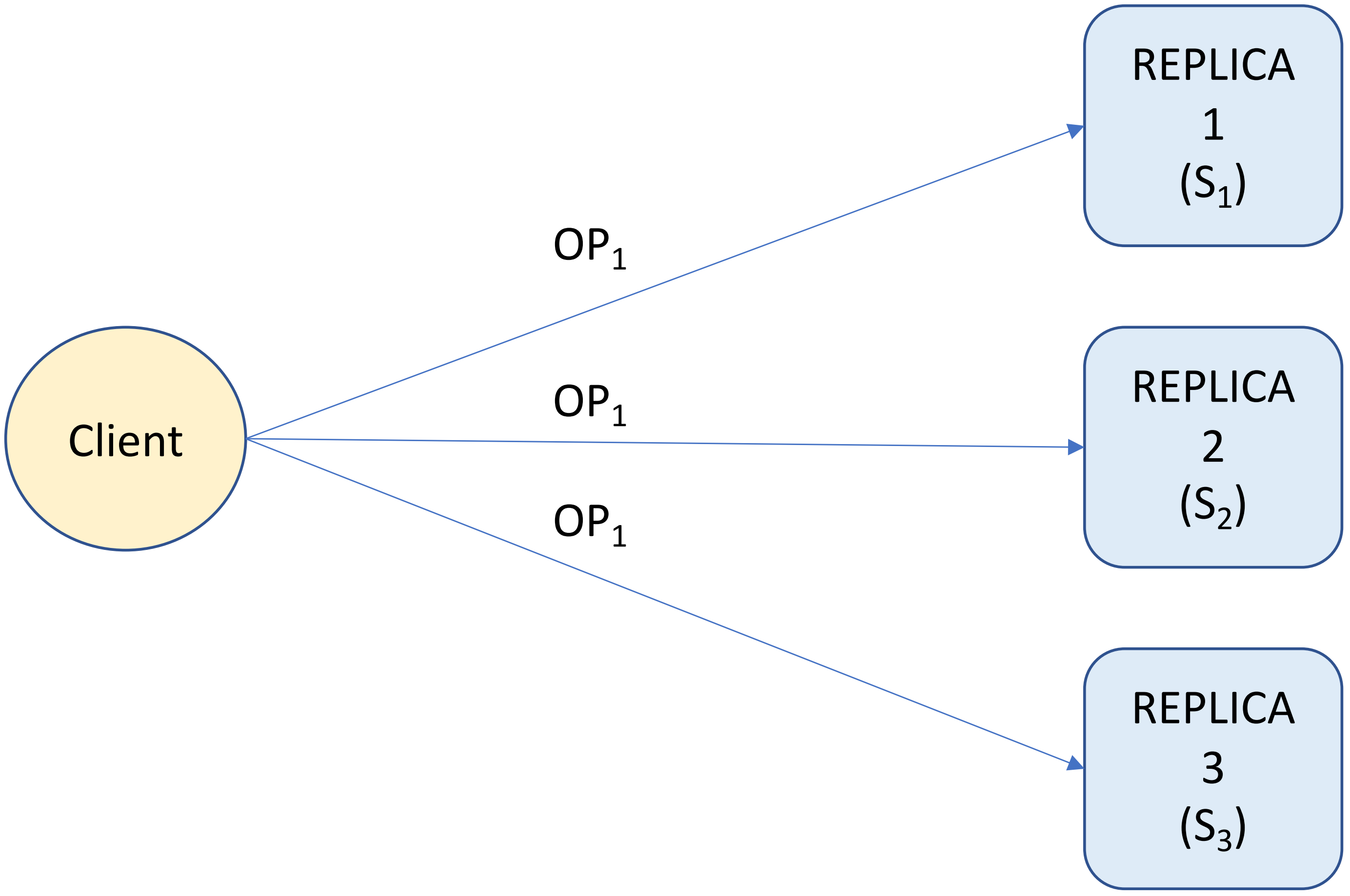
High Level Aspects:

- **Transparency**: The client is not aware that multiple replicas exist. In fact clients only observe a single logical state (or collection of data objects) and are fully unaware of the existence of multiple copies.
- **Consistency**: Despite the individual state of each replica, enforcing consistency implies reestricting the state that can be observed by a client given its past (operations executed by that client) and the system history (operations executed previously by any client).

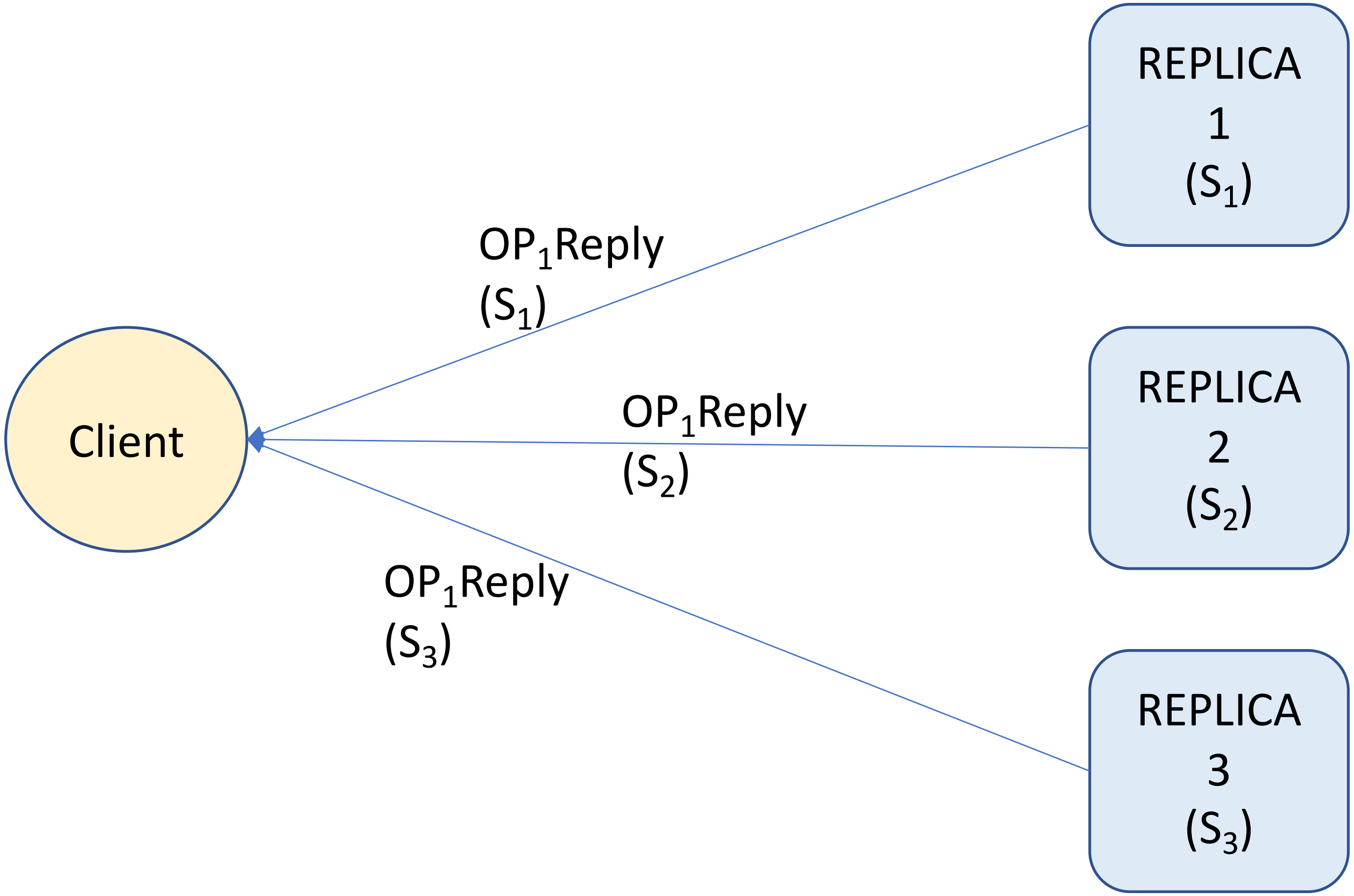
Transparency (architectural solutions)



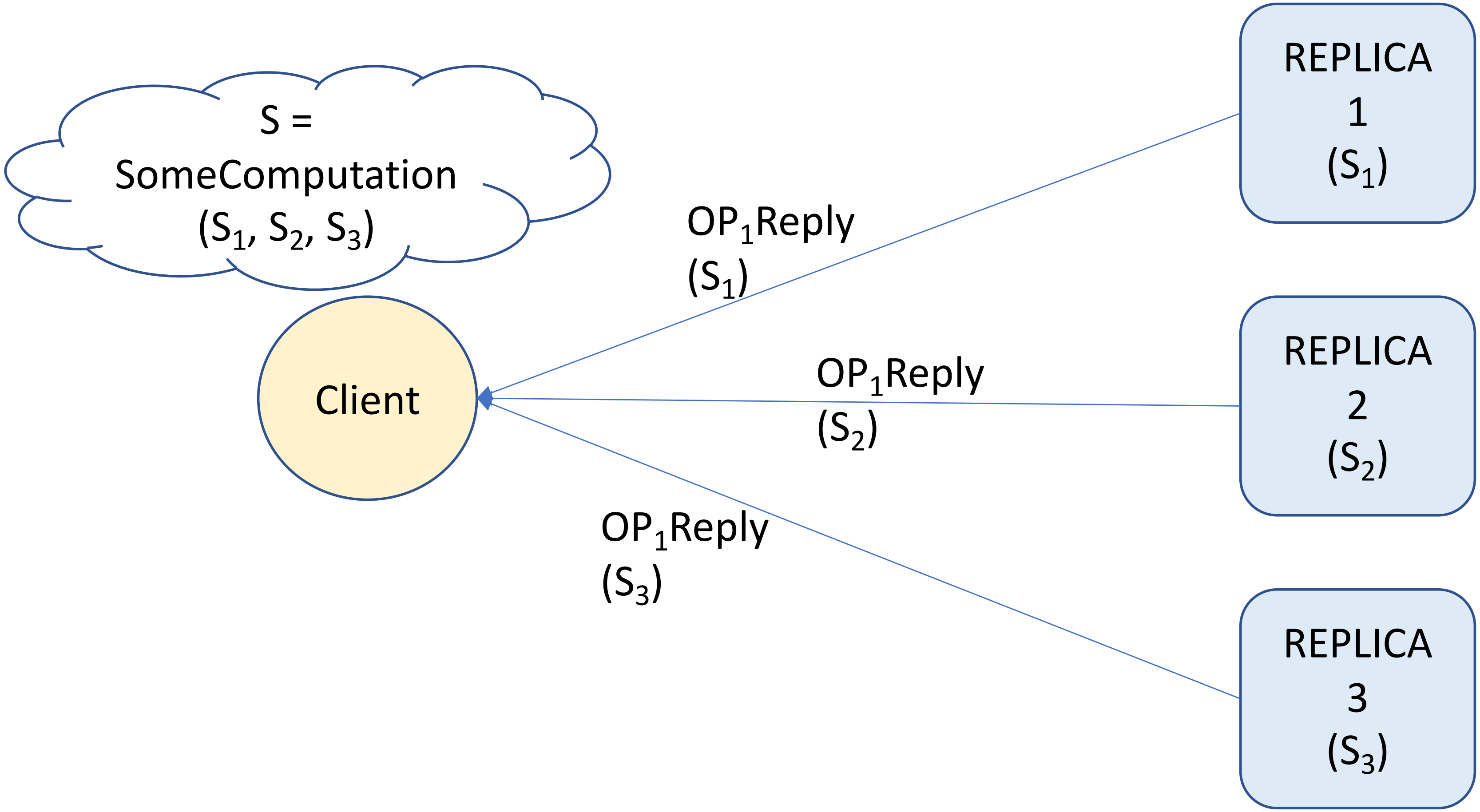
Transparency (architectural solutions)



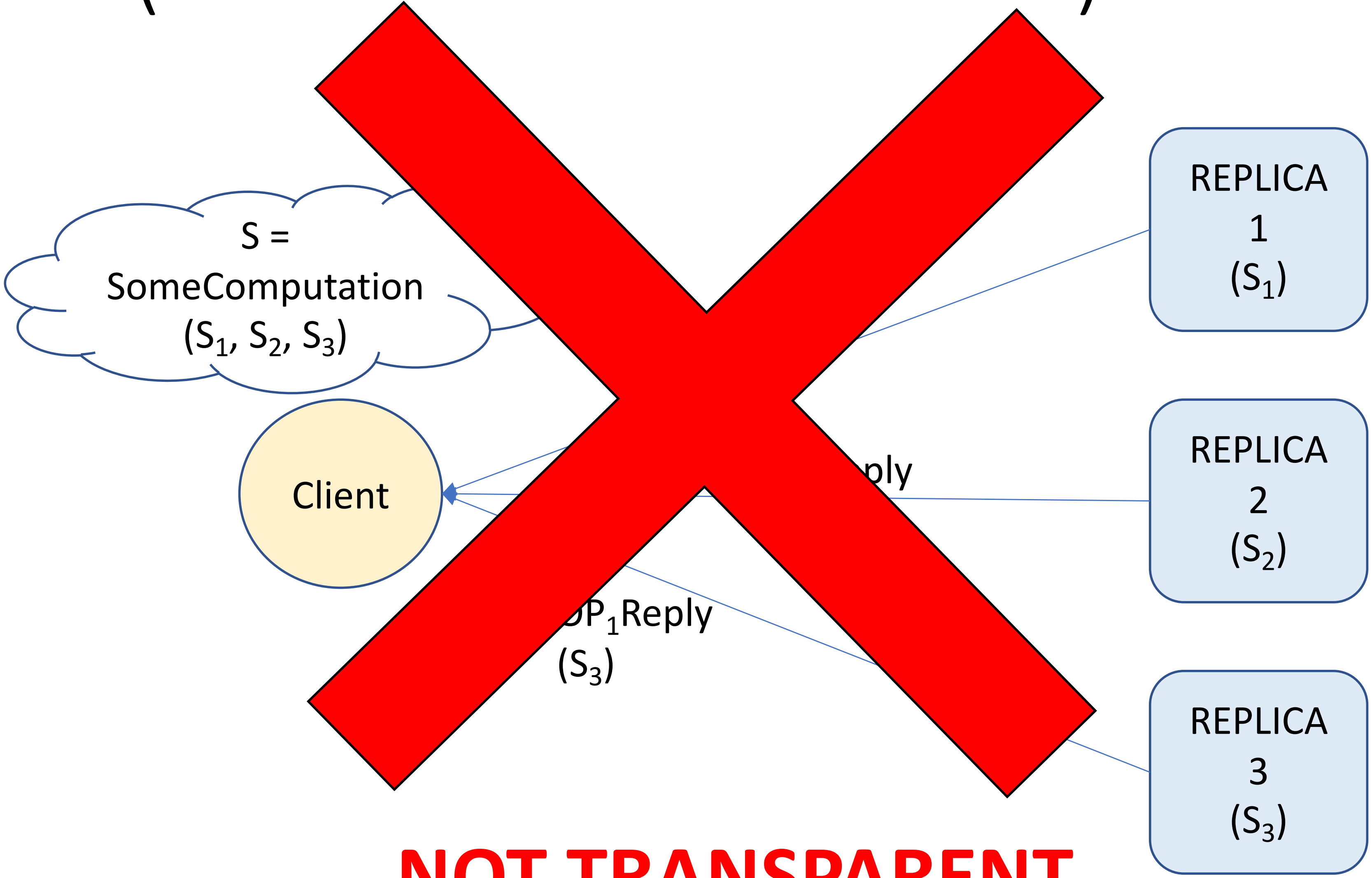
Transparency (architectural solutions)



Transparency
(architectural solutions)

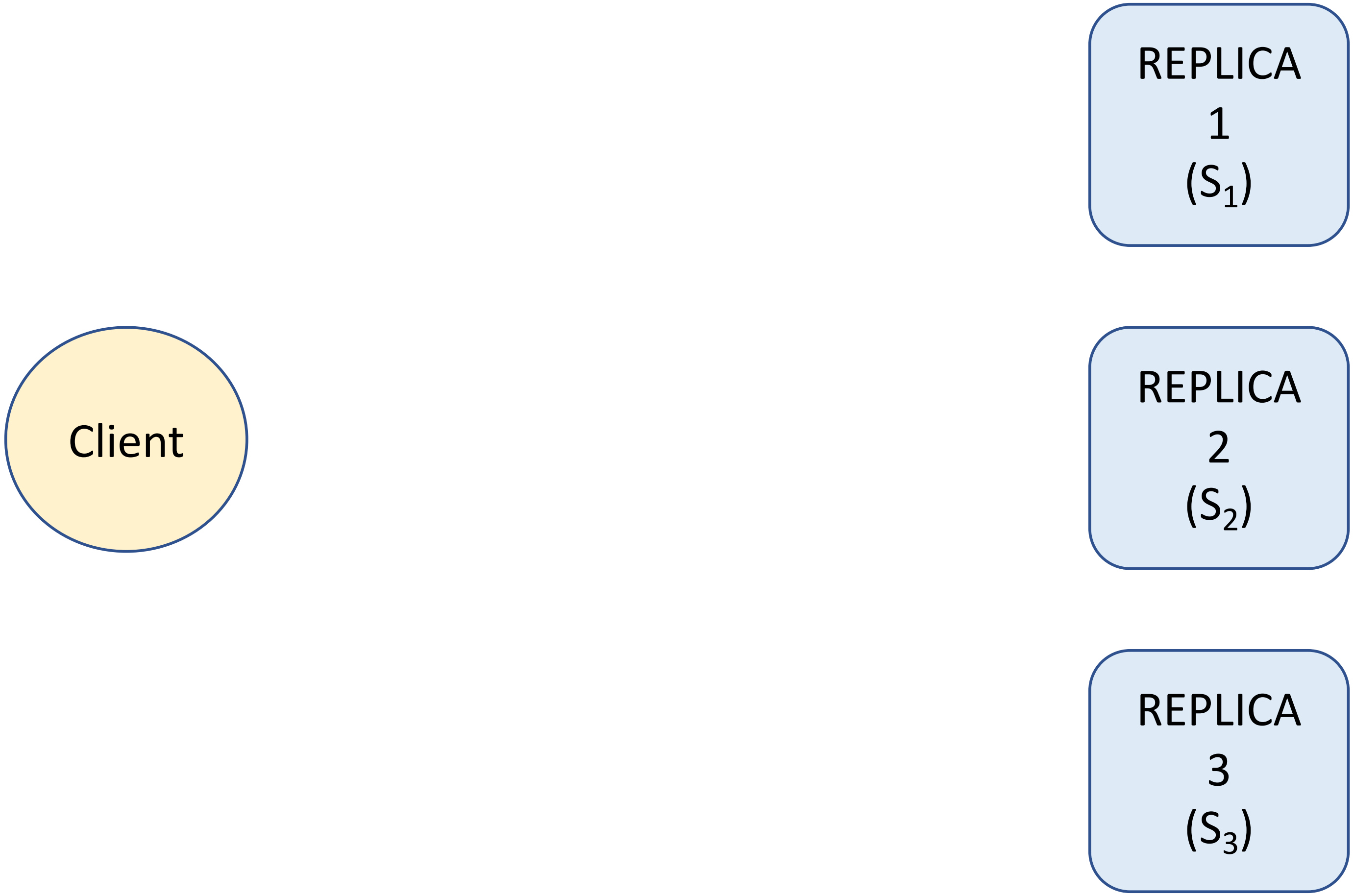


Transparency (architectural solutions)



NOT TRANSPARENT

Transparency (architectural solutions)



Client

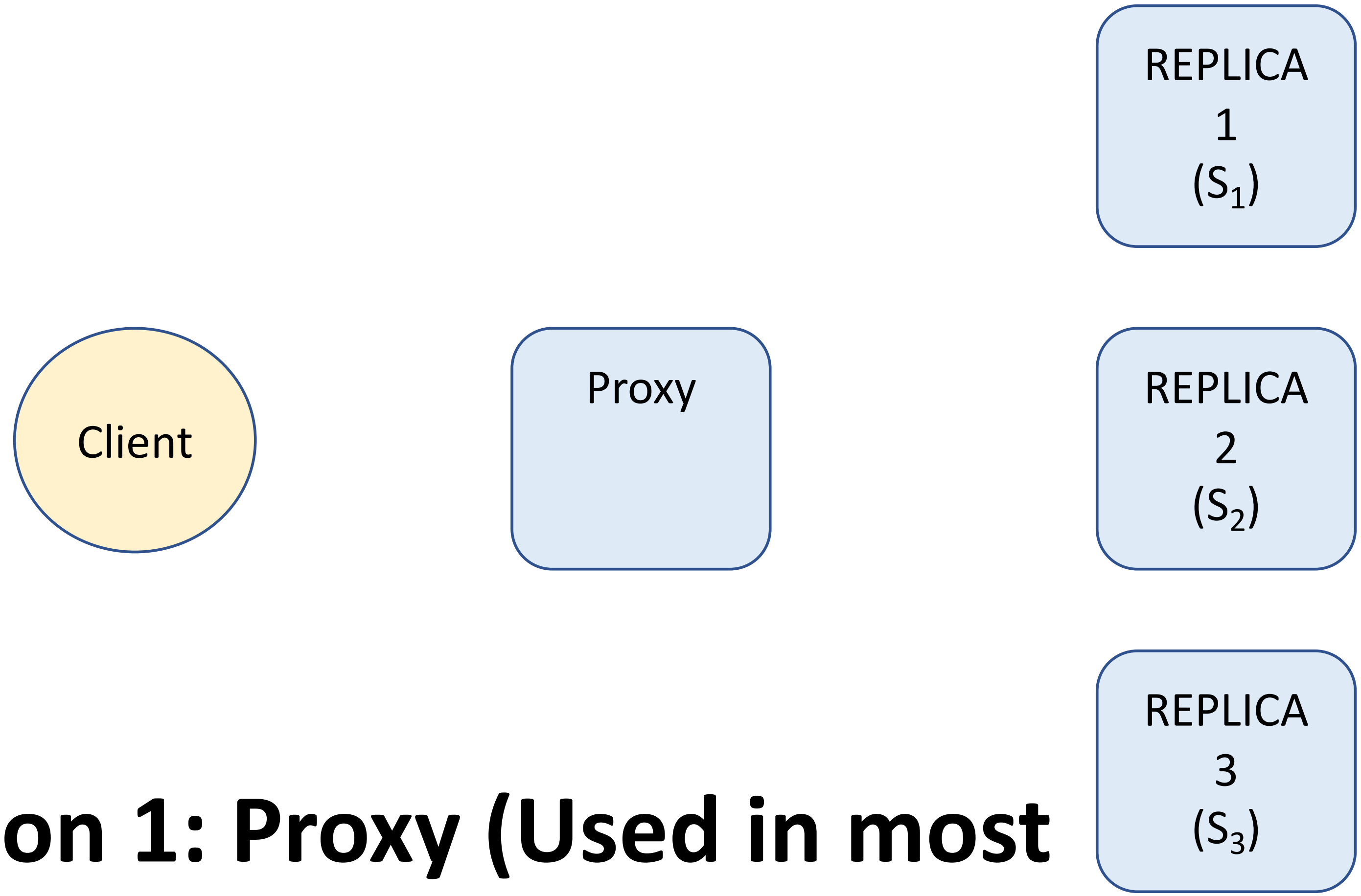
REPLICA 1 (S_1)

REPLICA 2 (S_2)

REPLICA 3 (S_3)

Solutions??

Transparency (architectural solutions)



Client

Proxy

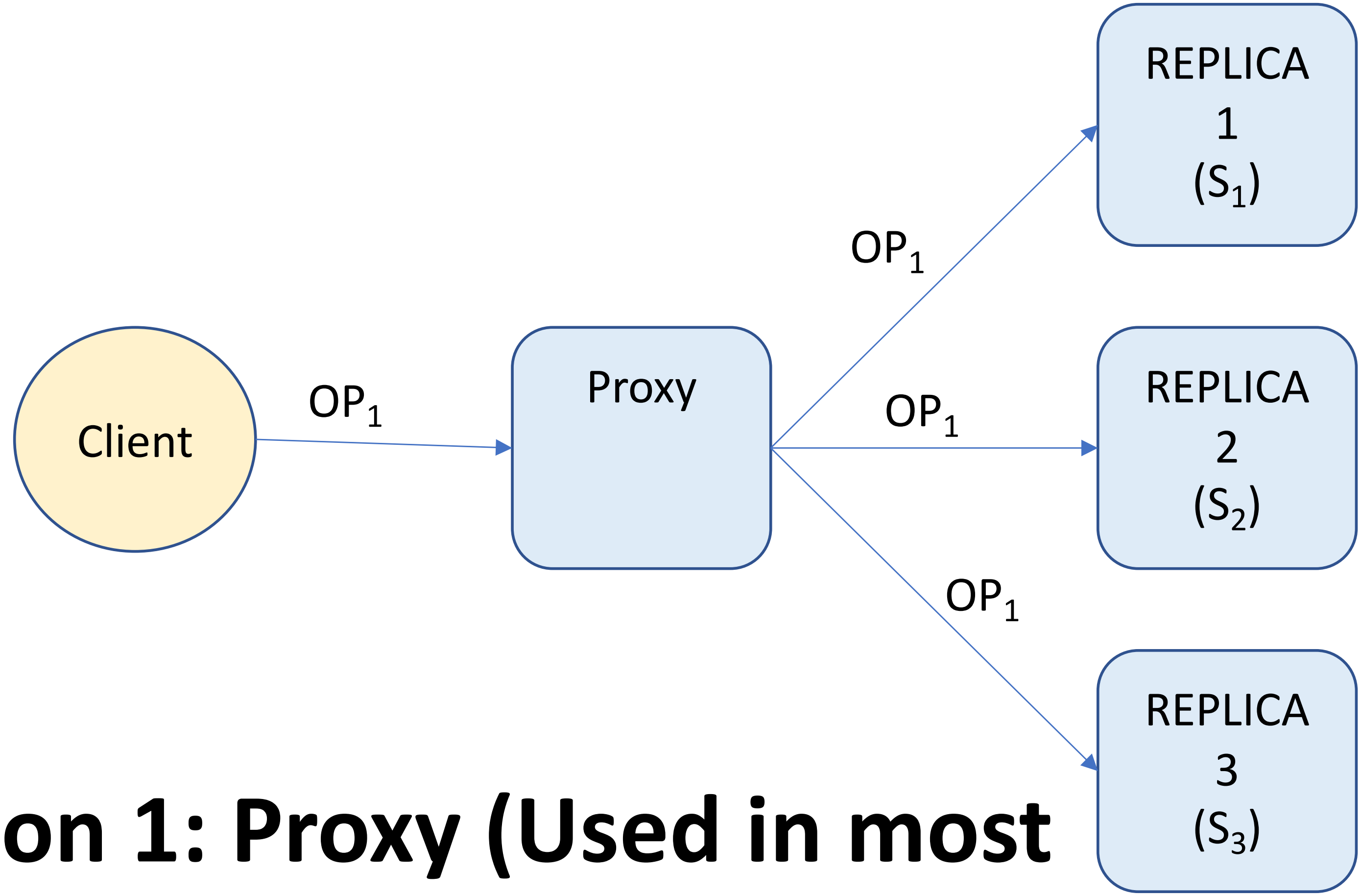
REPLICA 1 (S_1)

REPLICA 2 (S_2)

REPLICA 3 (S_3)

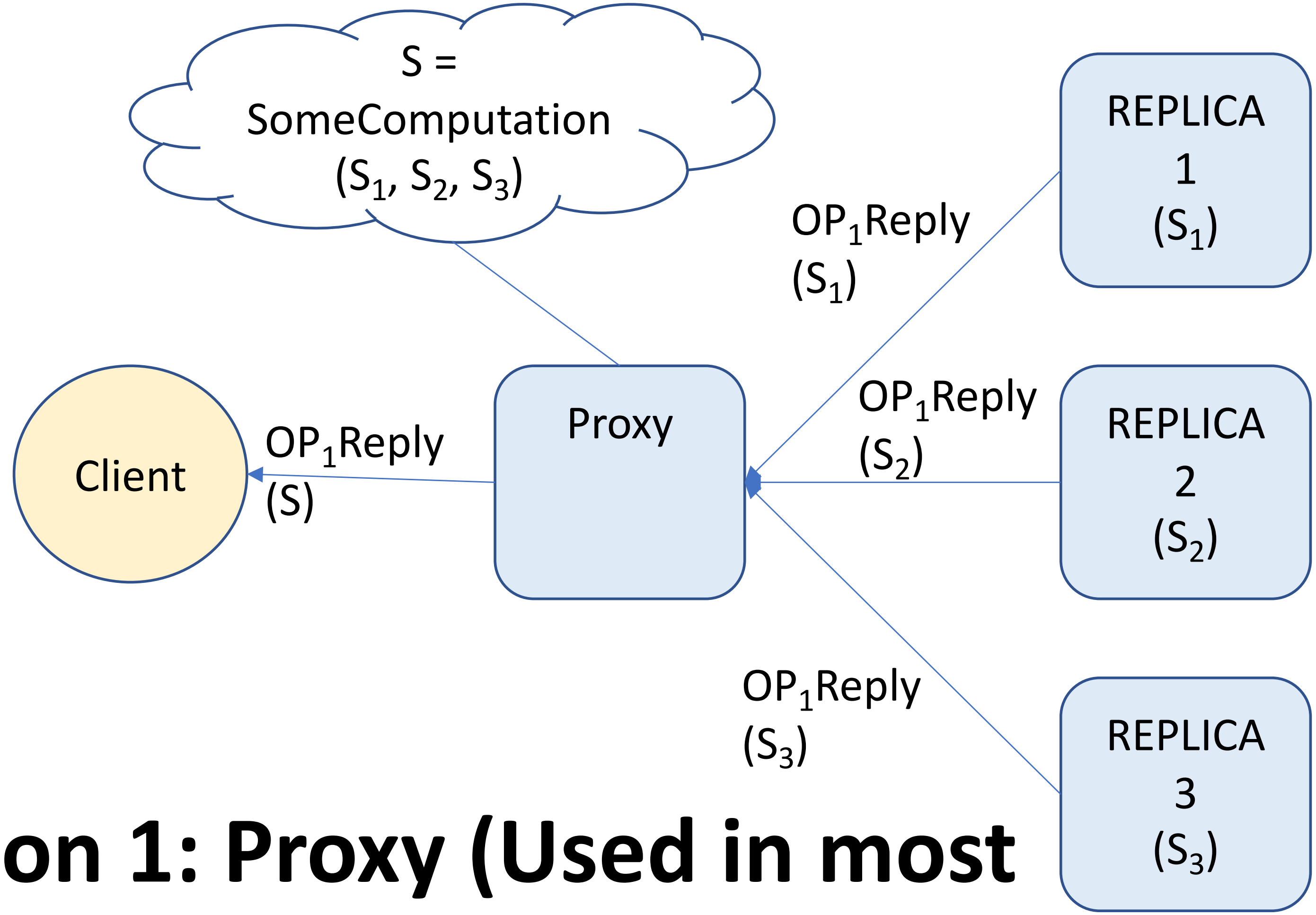
Solution 1: Proxy (Used in most Web Applications)

Transparency (architectural solutions)



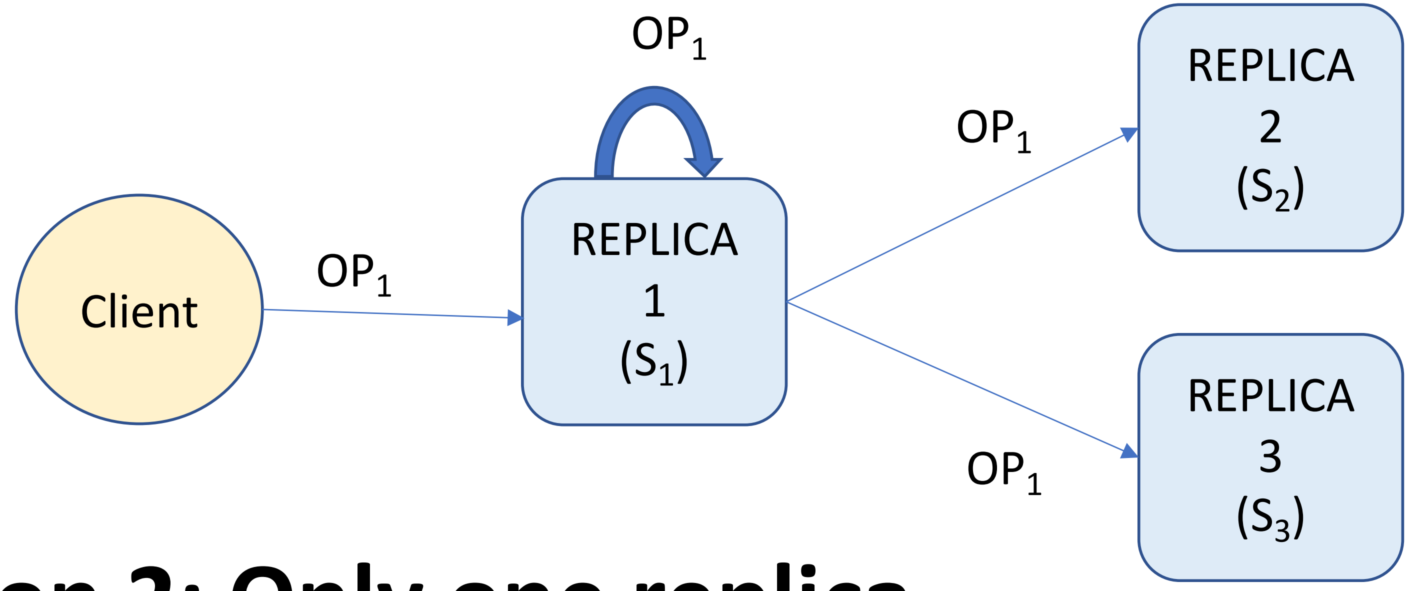
Solution 1: Proxy (Used in most Web Applications)

Transparency (architectural solutions)



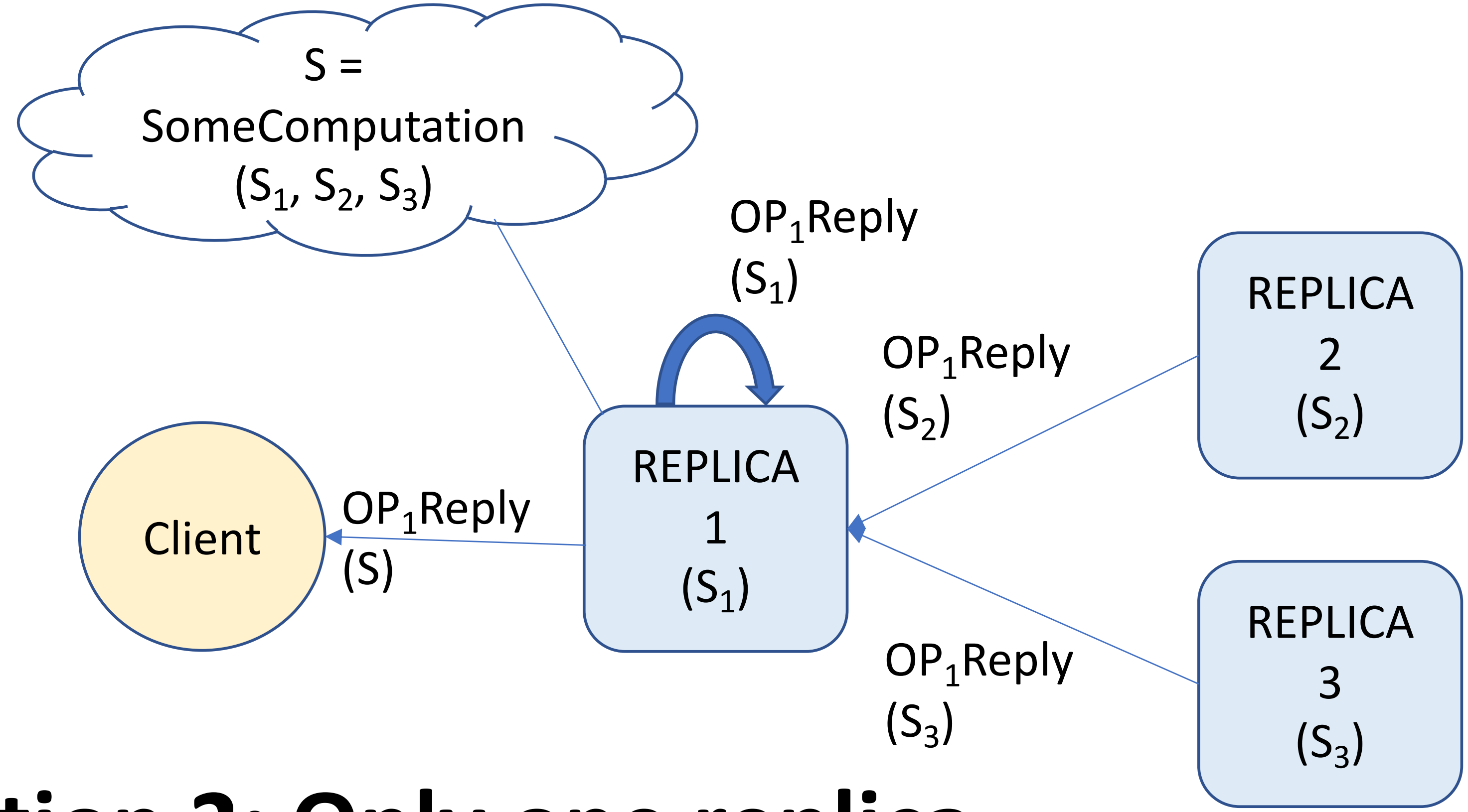
Solution 1: Proxy (Used in most Web Applications)

Transparency
(architectural solutions)



Solution 2: Only one replica received operations interacts with client

Transparency (architectural solutions)



Solution 2: Only one replica received operations interacts with 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:
  - Single Master (also known as Master-Slave): A single replica receives operations that modify the state from clients.
  - Multi-Master: Any replica can process any operation.

Active Replication

- All replicas execute operations.
- State is continuously updated at every replica, which might lower the impact of a replica failure.
- Can only be used when operations are deterministic (i.e., they do not depend from non-deterministic variables, such as local time, or generating a random value).
- If operations are not commutative (execution of the same set of operations in different orders lead to different results) then, all replicas must agree on the order operations are executed.

Passive Replication

- Only one replica executes operations.
- Other replicas are only informed of results (to update their local state).
- Good when operations depend on non-deterministic data or inputs (random number, local replica time, etc...)
- Load across replicas is not balanced (only one replica effectively executes the operation and computes the result, other replicas only observe results – for write operations).

Synchronous Replication

- Replicas are updated before replying to the client.
- *Notice that a client operation is only considered as complete after the client obtains a reply from “the system”.*
- This can delay (significantly) the response times of the system (client experiences higher latency) leading to lower overall performance.
- When a client obtains the answer, it is easier to guarantee that the effects of her operation are not going to be lost.

Asynchronous Replication

- Replicas are updated sometime after the client obtaining a reply (or concurrently with the reply being sent to the client).
- Clients will obtain replies more quickly (no need to wait for more than one replica to update their state), which promotes better performance overall.
- Effects of client operations may be lost, even if she got a reply, for instance due to the failure of replicas.

Single Master (Master-Slave or Primary-Backup)

- Only a single replica, named the *master*, processes operations that modify the state (write operations).
- Other replicas might process client operations that only observe the state (read operations), leading clients to potentially observe stale values (depends on consistency guarantees enforces by the system).
- When the master fails, one of the secondary replicas must take over the role of master.
- If two processes believe themselves to be the master *safety properties* might be compromised.

Multi-Master

- Any replica can process any operation issued by a client (i.e., both read and write operations)
- All replicas behave in the same way, which implies better load balancing.
- Adding more replicas can increase the overall capacity of the system to process client operations.
- Multiple replicas might attempt to do conflicting operations at the same time, which requires some form of coordination (distributed locks or other coordination protocols that typically are expensive).

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 of them.
- 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 [sequence] value).

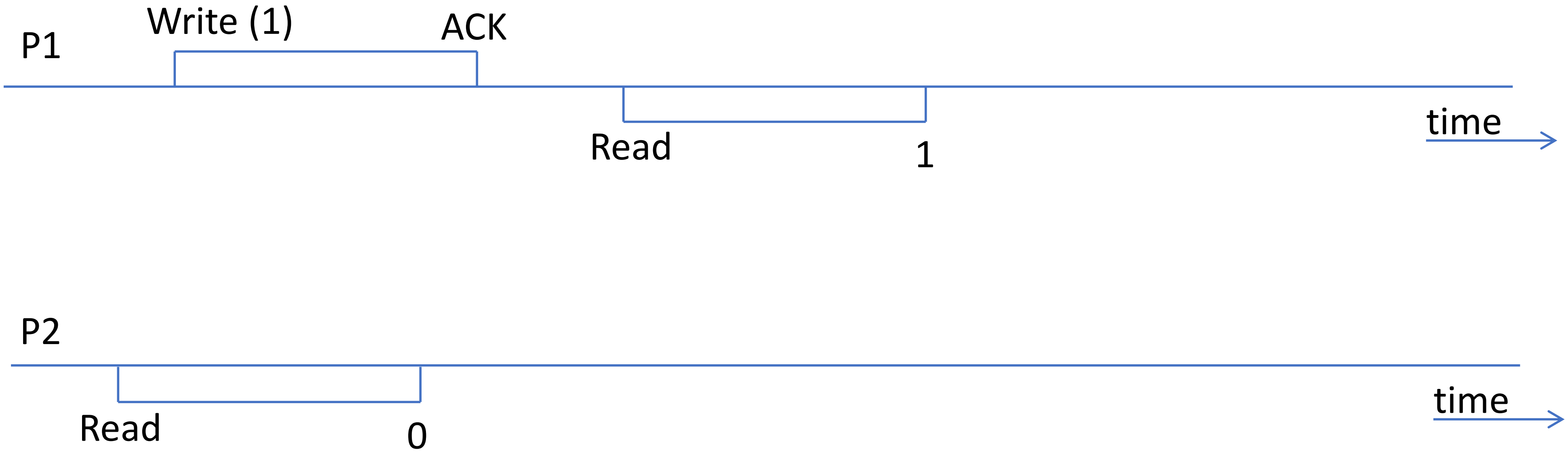
Register Repication

- Properties (high level):

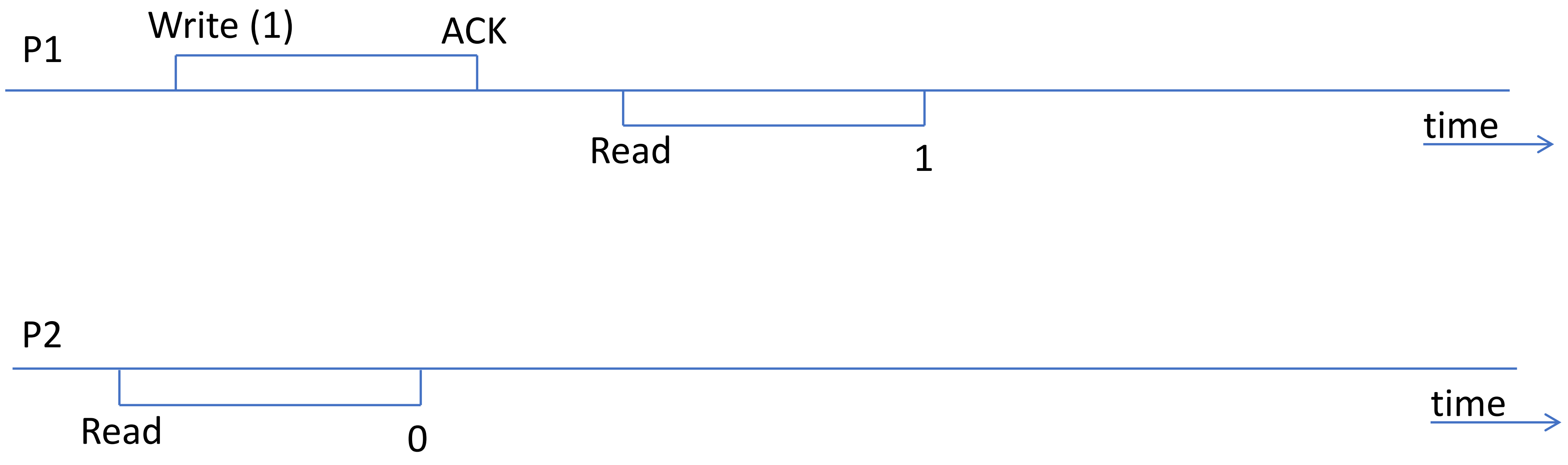
- Liveness: Every operation of a correct process eventually completes.

- Safety: Every read operation returns the *last* value written.

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

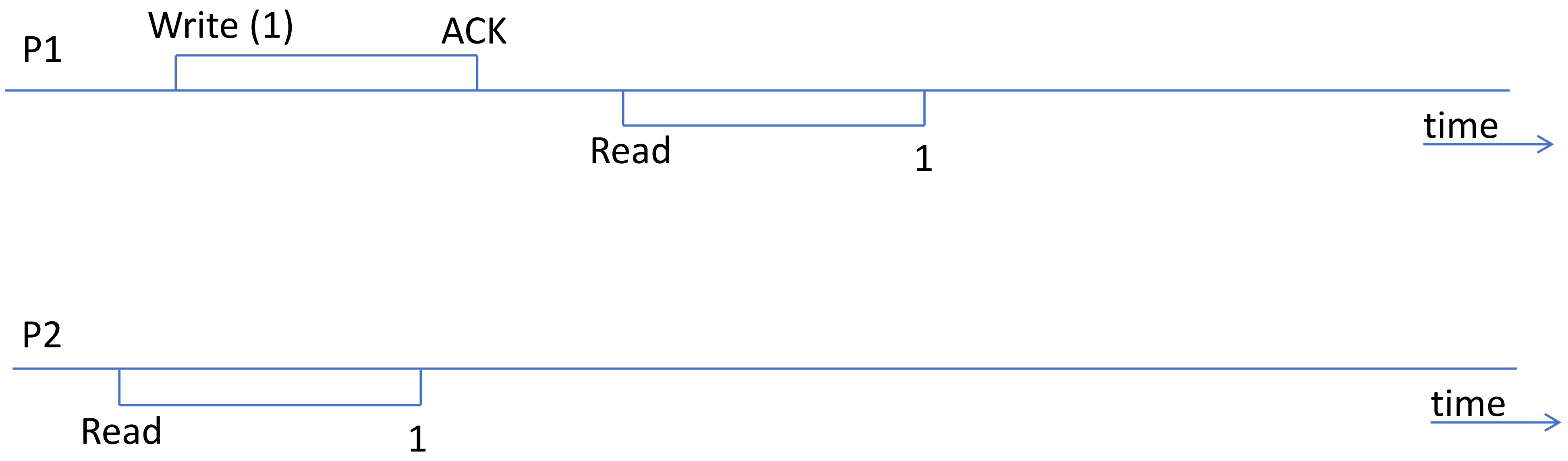


What does **last written value** means in this context.

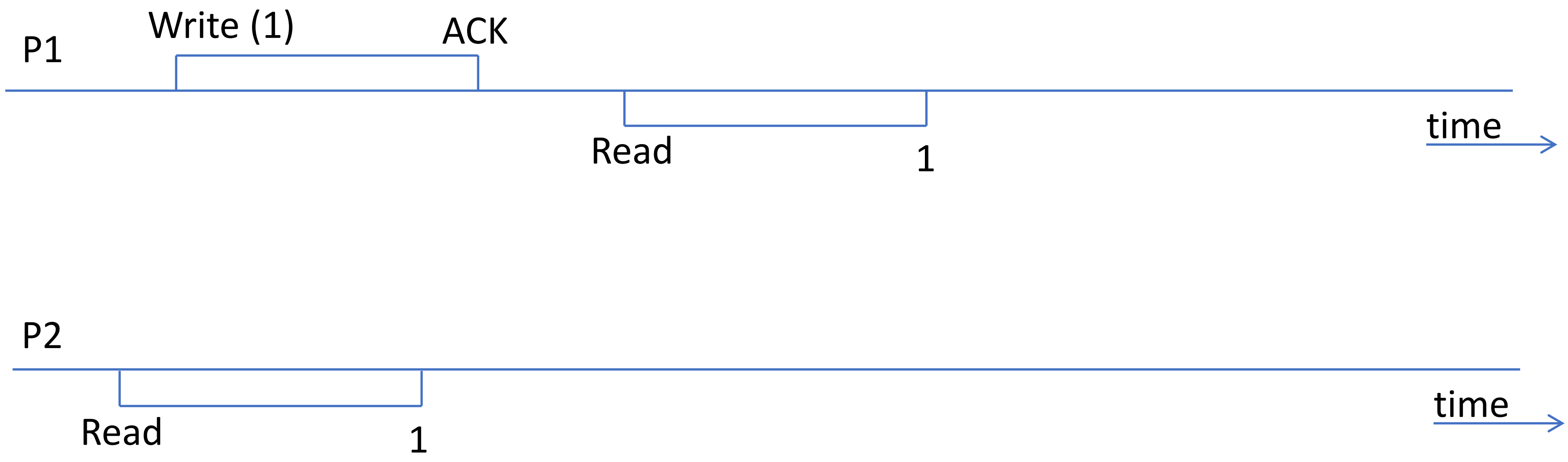


This is a valid execution

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

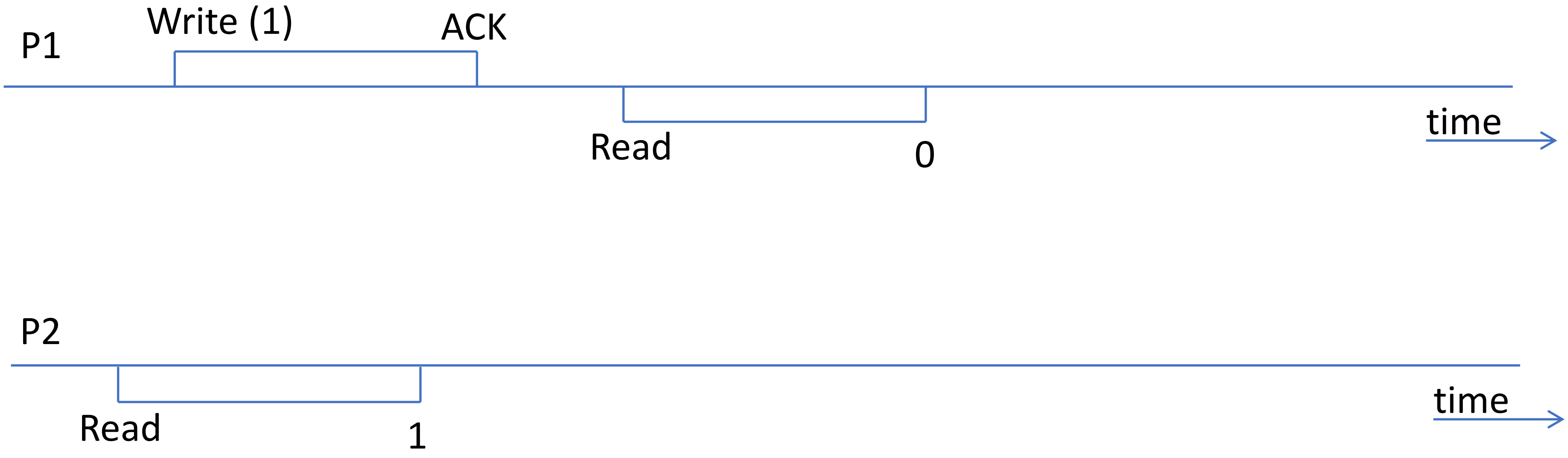


What does **last written value** means in this context.



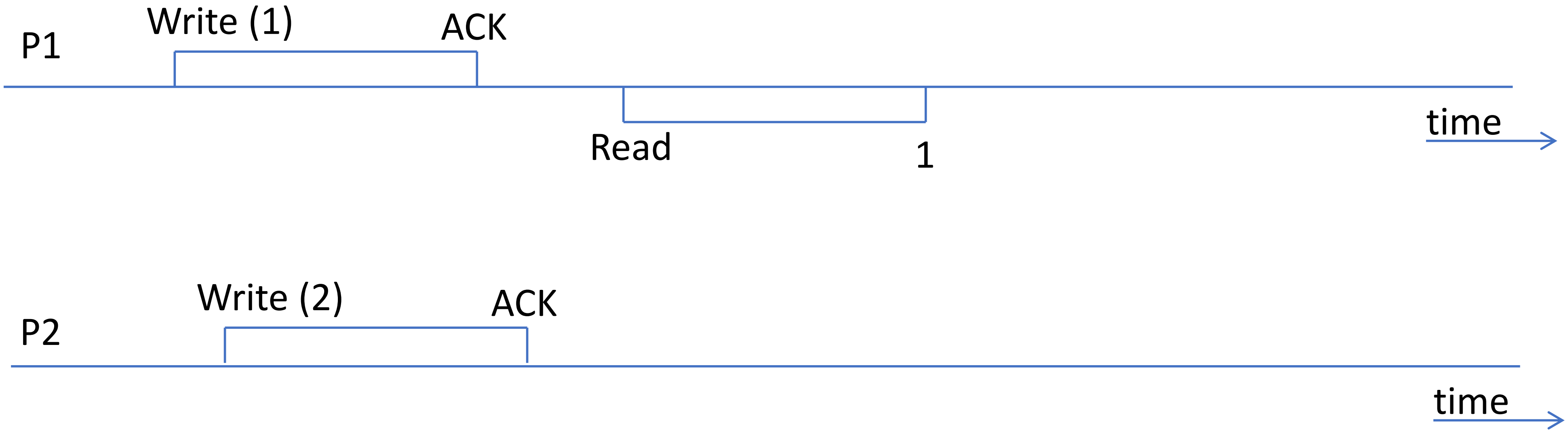
This is a valid execution

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



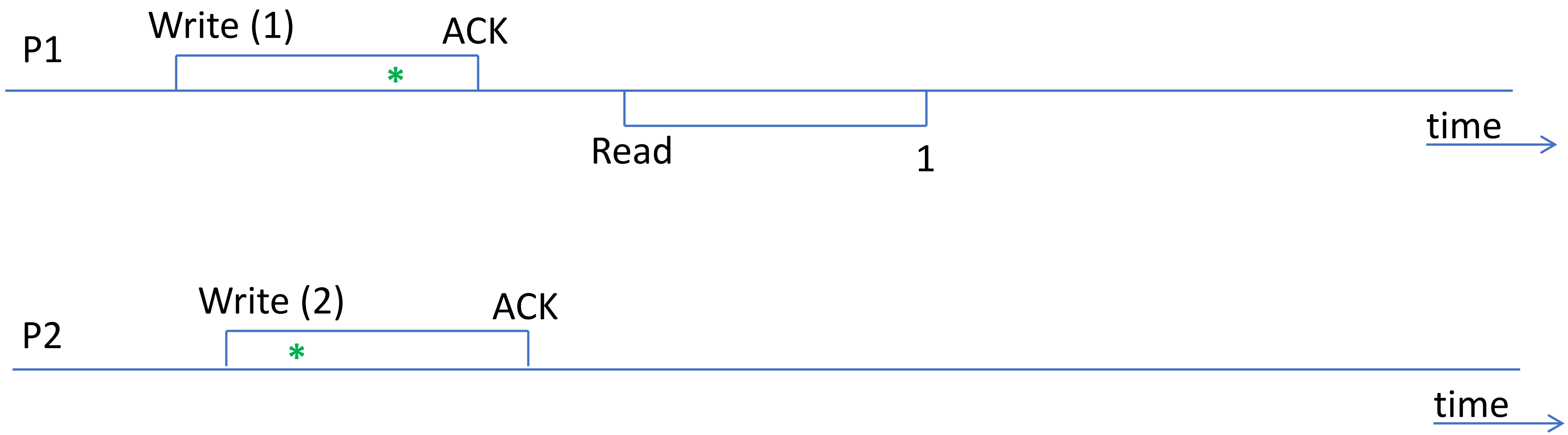
This is NOT a valid execution

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



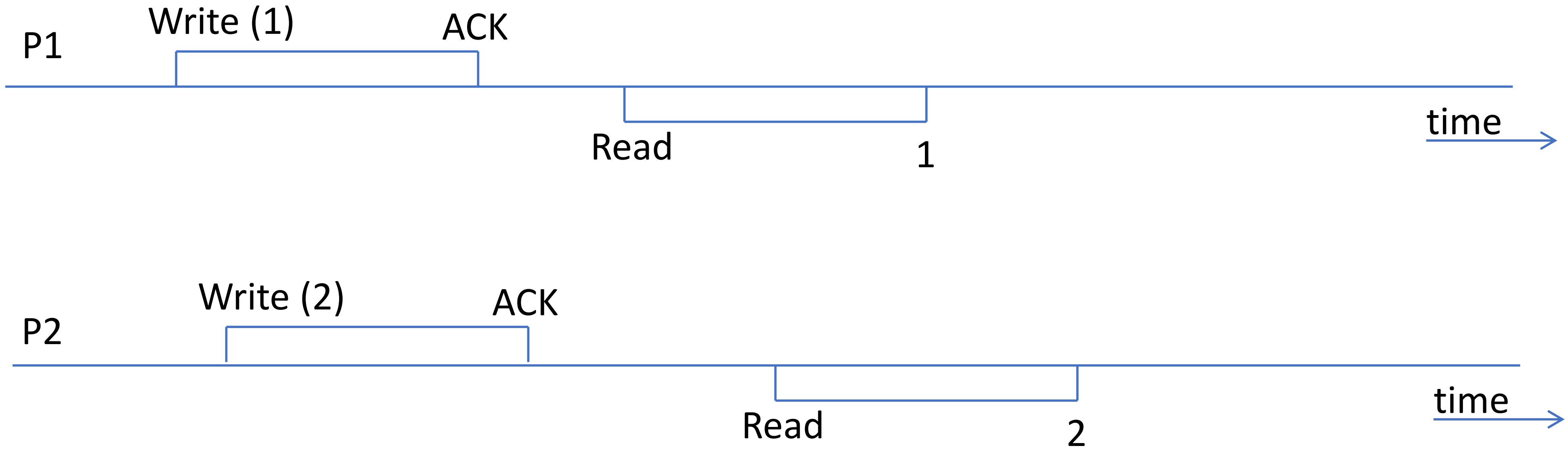
What about this case?

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



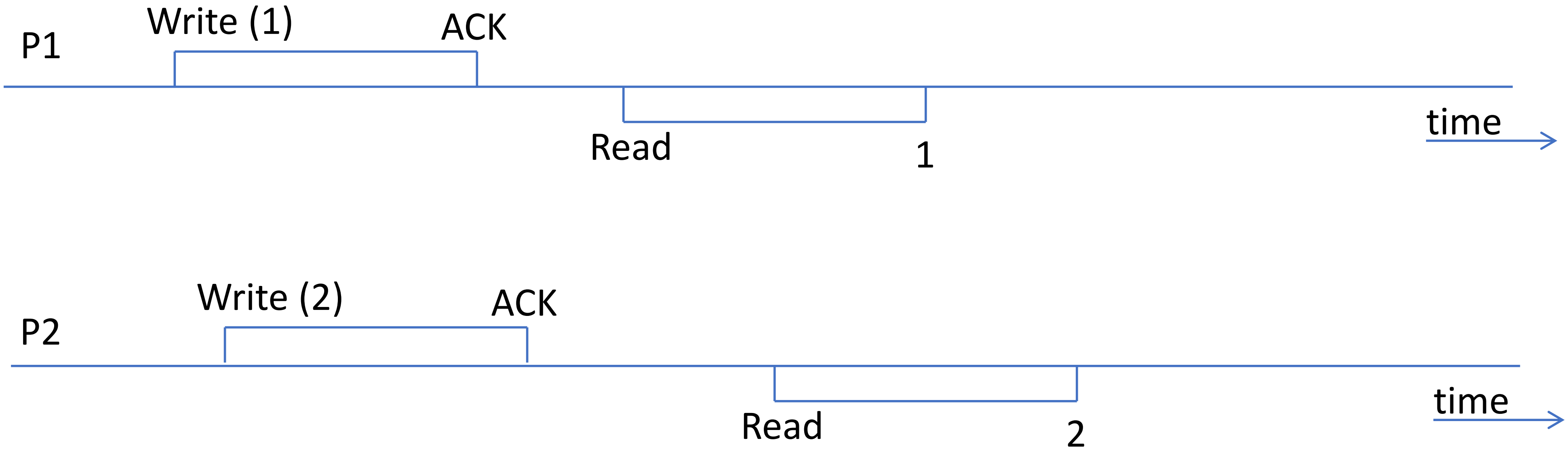
This is a valid execution, since write operations are concurrent, we must define serialization points to arbiter their order.

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



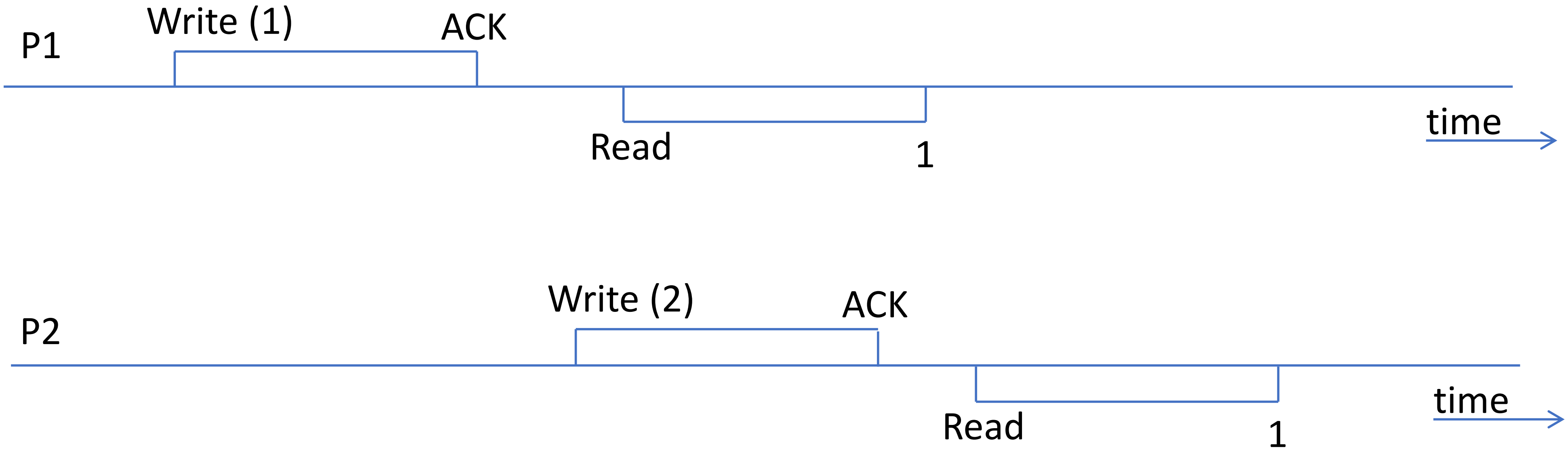
What about this case?

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



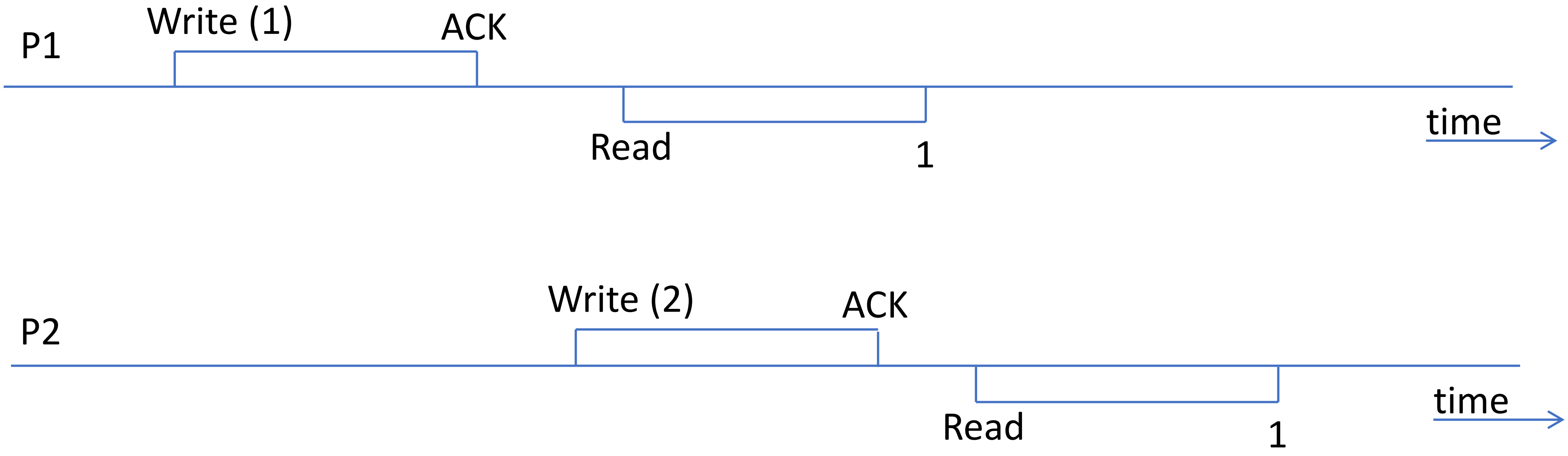
This is not a valid execution, there are no serialization points that explain the return of those two reads.

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



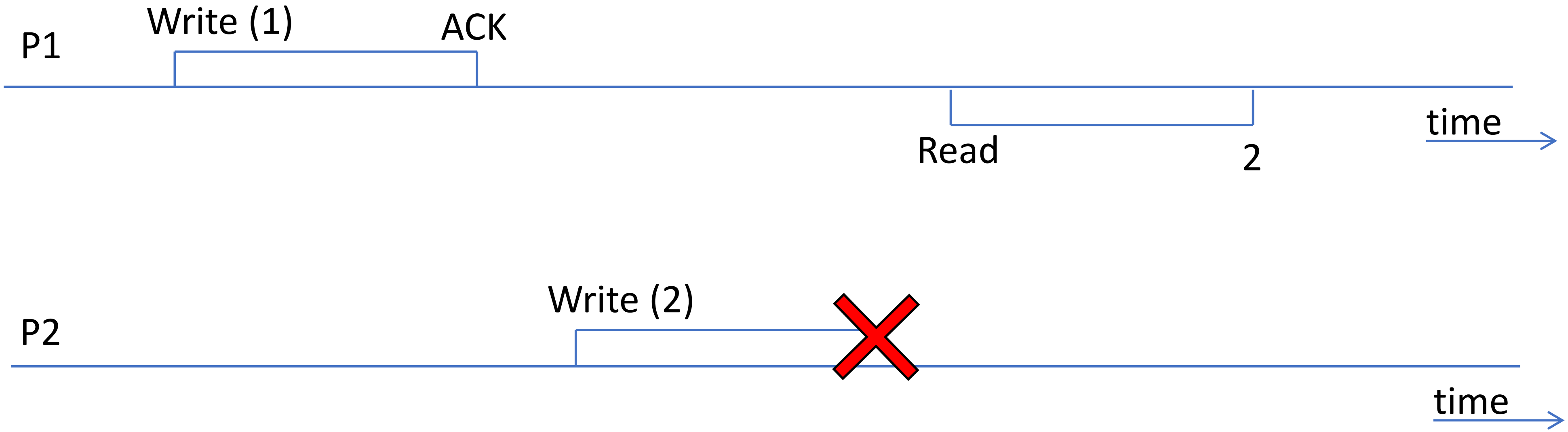
What about this case?

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



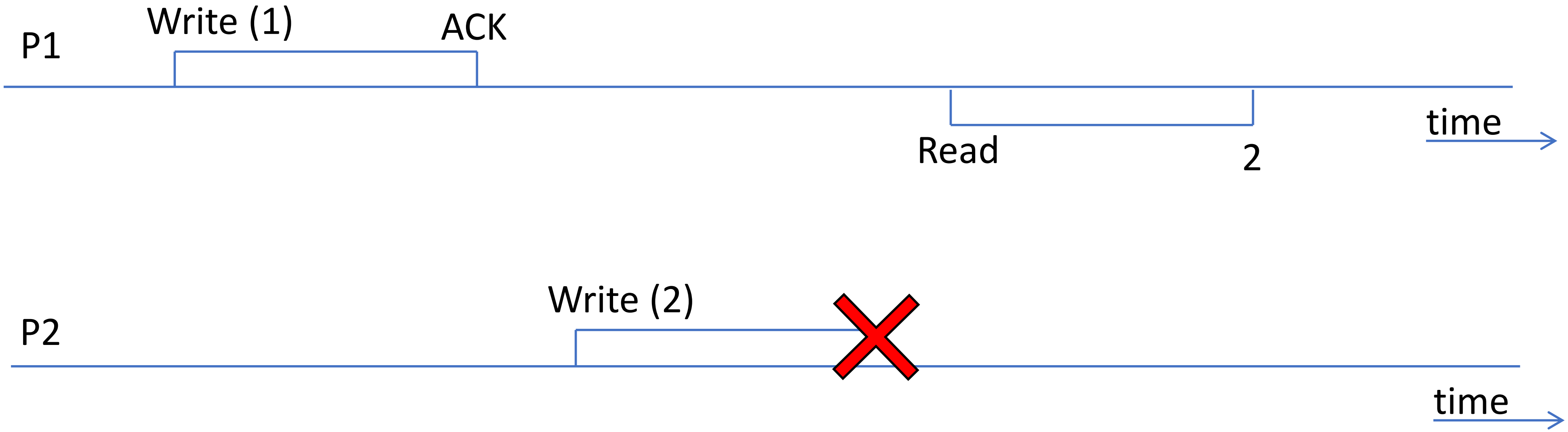
This is not a valid execution evidently, but only because of the read performed by P2.

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



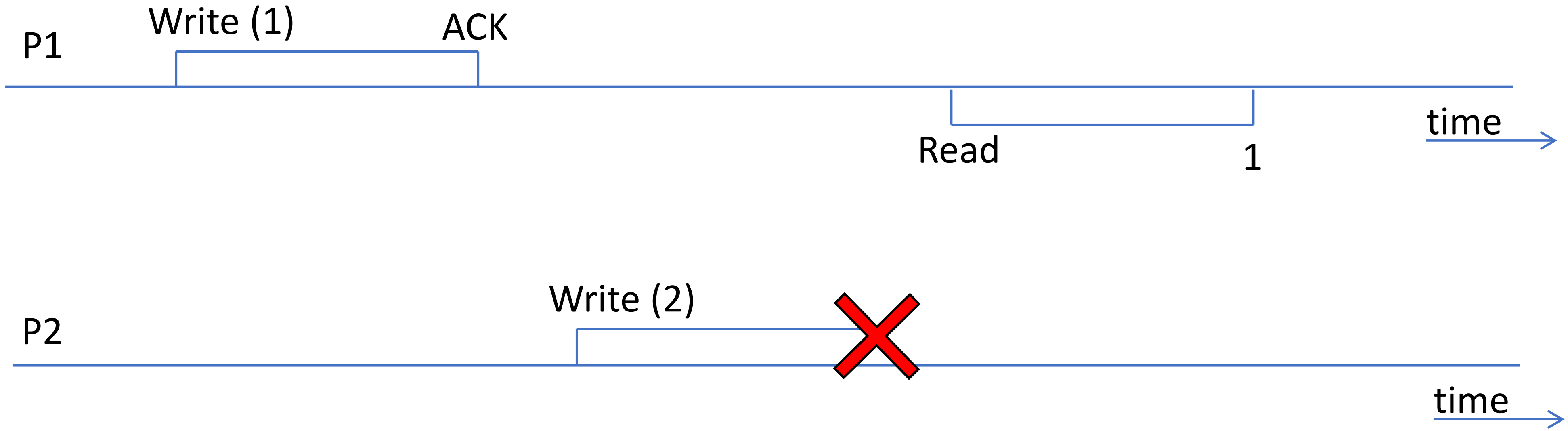
What about this last case?

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



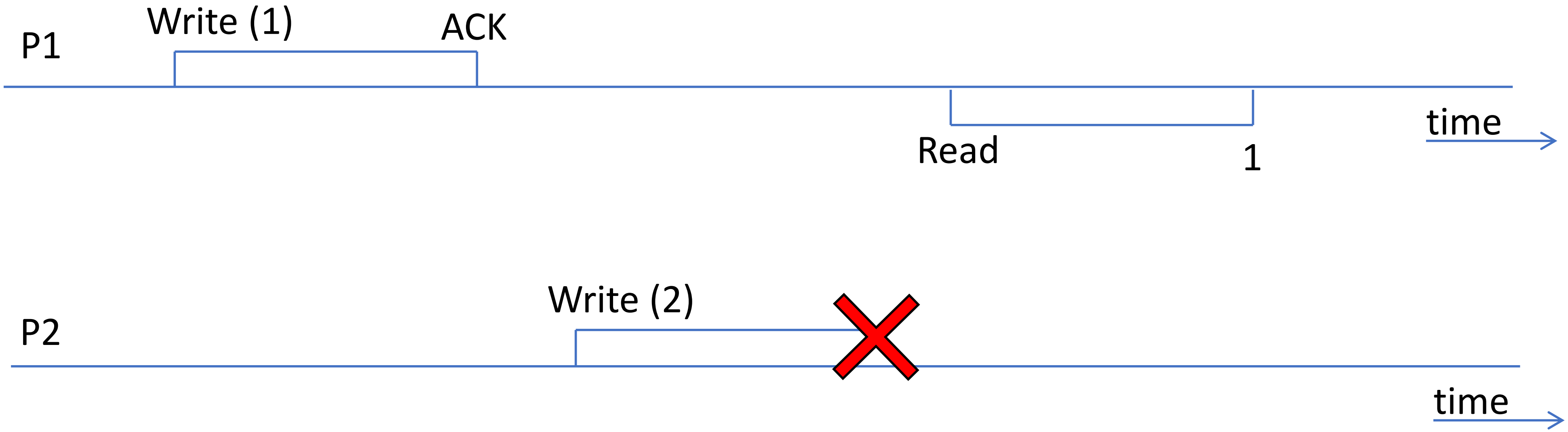
This is a valid execution.

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



What about this one?

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



This is also a valid execution, since P2 failed, we cannot assure that its operation completed (it’s not in the target properties).

Let’s build a simple register

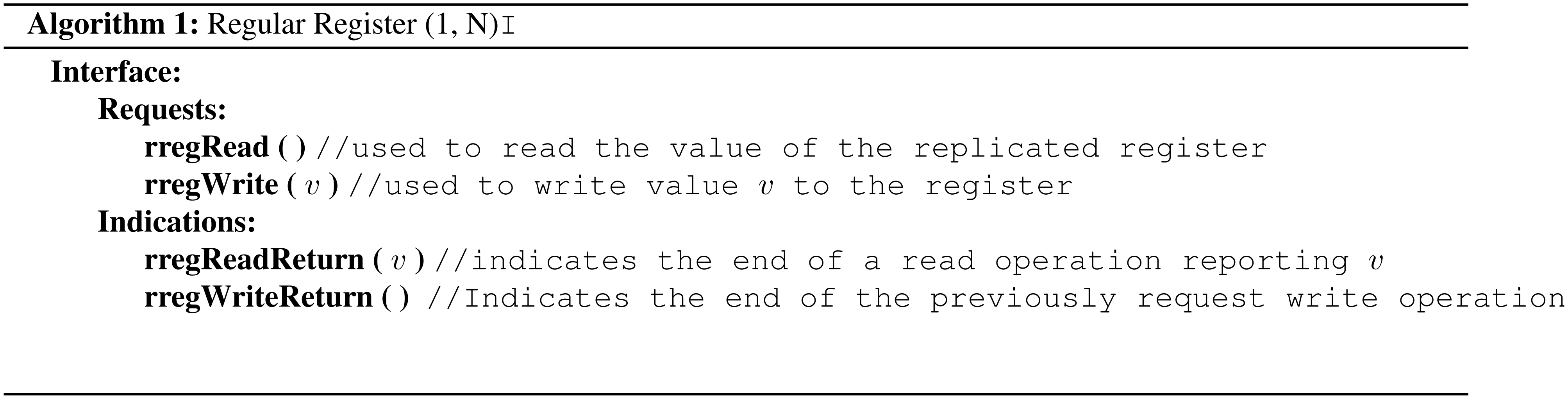
(1, N) register -> 1 Writer (say p1) N readers

Assume the following system model:

- Fault Model: Fail-Stop
- Synchrony Model: Synchronous System
- Known and fixed membership.

*(Remember that in this case you have access to Crash detection through the **Upon Crash (p)** event.*

Let’s build a simple register



```
Algorithm 1: Regular Register (1, N)I
  Interface:
    Requests:
      rregRead ( ) //used to read the value of the replicated register
      rregWrite ( $v$ ) //used to write value $v$ to the register
    Indications:
      rregReadReturn ( $v$ ) //indicates the end of a read operation reporting $v$
      rregWriteReturn ( ) //Indicates the end of the previously request write operation
```

Properties:

Termination: If a correct process invokes an operation then the process eventually receives the corresponding indication of its termination.

Validity: A read returns the last value written, or the value being concurrently written.

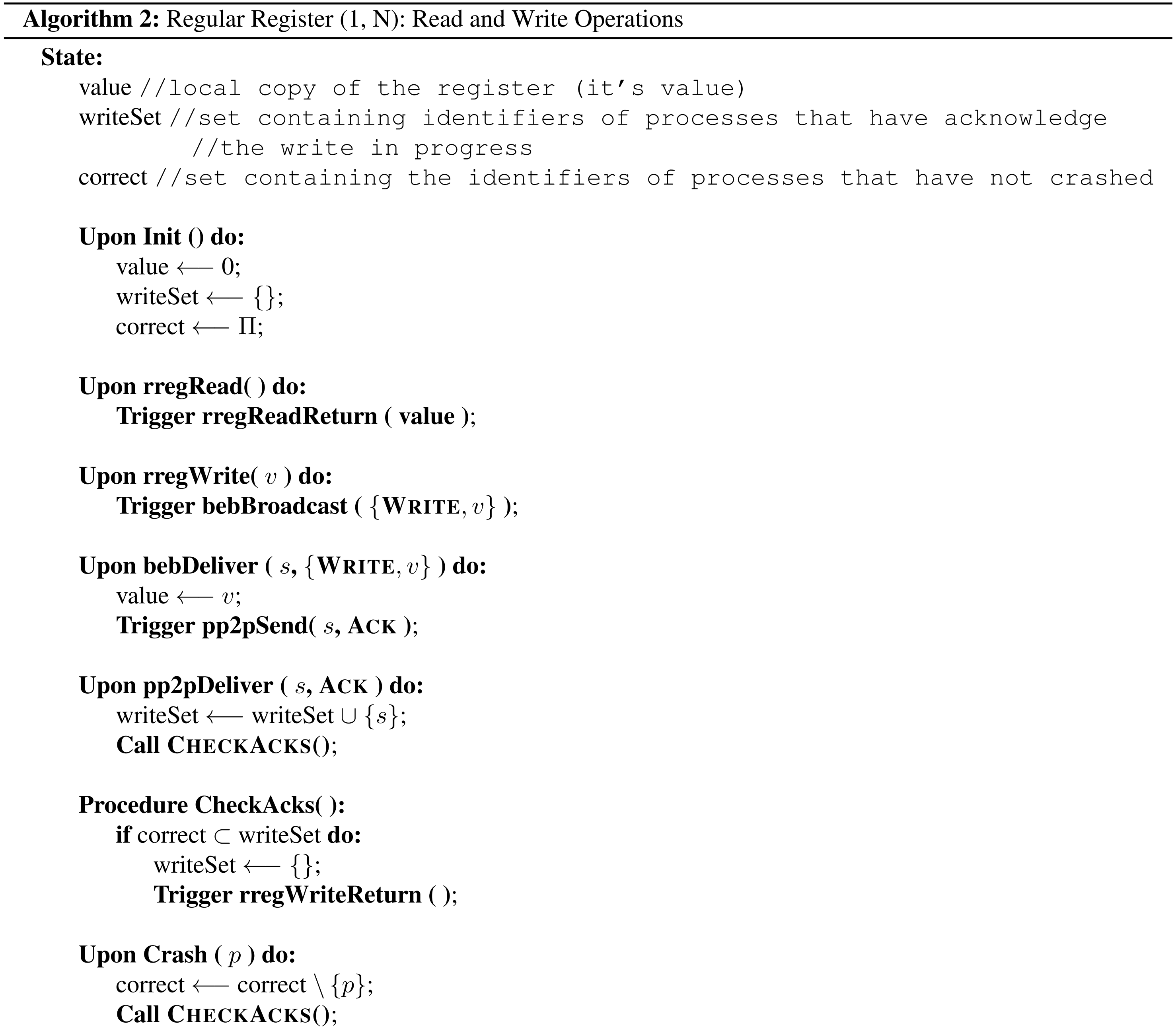
How do you propose yourself to handle this?

How do you propose yourself to handle this?

Intuition:

- Whenever I read, I always read from my local copy.
- Whenever I write, I send the write to everyone (best effort broadcast) and wait for confirmations from every correct process.
- When I detect a process has crashed, I no longer wait for his confirmations.
- Whenever I get confirmations from all correct processes then I complete the write.

The Algorithm



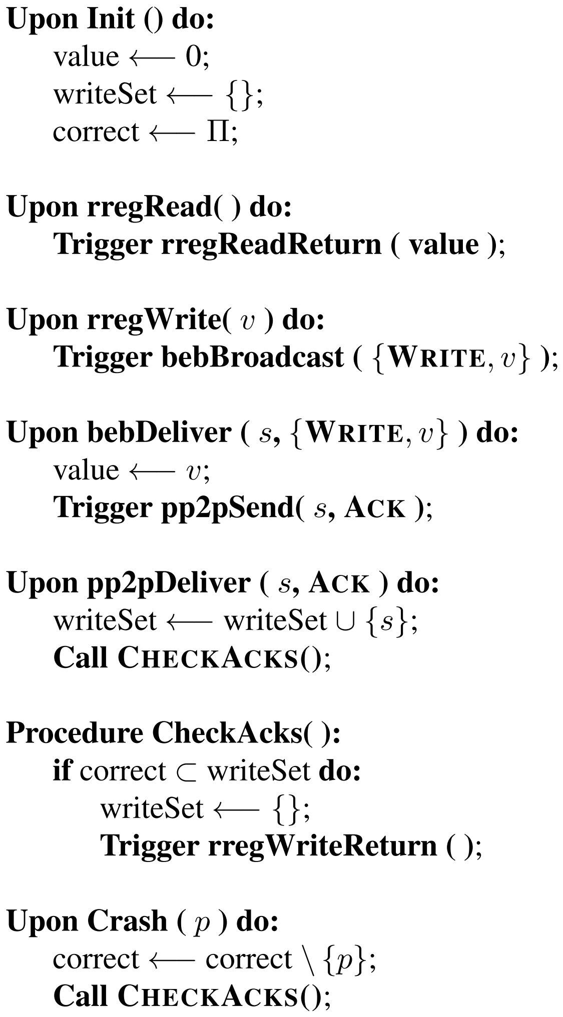
Algorithm 2: Regular Register (1, N): Read and Write Operations

```
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 $\longleftarrow$ 0;
        writeSet $\longleftarrow \{\}$;
        correct $\longleftarrow \Pi$;

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

    Upon rregWrite( $v$ ) do:
        Trigger bebBroadcast ( $\{\textsc{Write}, v\}$ );

    Upon bebDeliver ( $s$, $\{\textsc{Write}, v\}$ ) do:
        value $\longleftarrow v$;
        Trigger pp2pSend( $s$, Ack );

    Upon pp2pDeliver ( $s$, Ack ) do:
        writeSet $\longleftarrow$ writeSet $\cup \{s\}$;
        Call CheckAcks();

    Procedure CheckAcks( ):
        if correct $\subset$ writeSet do:
            writeSet $\longleftarrow \{\}$;
            Trigger rregWriteReturn ( );

    Upon Crash ( $p$ ) do:
        correct $\longleftarrow$ correct $\setminus \{p\}$;
        Call CheckAcks();
```

The Algorithm

Termination: If a correct process invokes an operation then the process eventually receives the corresponding indication of its termination.

Validity: A read returns the last value written, or the value being concurrently written.



```
Upon Init () do:
    value ⟵ 0;
    writeSet ⟵ $\{\}$;
    correct ⟵ $\Pi$;

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 $\cup \{s\}$;
    Call CHECKACKS();

Procedure CheckAcks( ):
    if correct $\subset$ writeSet do:
        writeSet ⟵ $\{\}$;
        Trigger rregWriteReturn ( );

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

Let’s build the correctness argument.

Making it more interesting…

- Same problem (1,N) Register.

- Different System Model:

- Fault Model: Crash
- Synchrony Model: Asynchronous System
- Known and fixed membership.

(You no longer know when a process has crashed…)

How do you propose yourself to handle this?

How do you propose yourself to handle this?

Intuition:

- We can no longer wait for all processes to acknowledge a write (they might have failed and that can make a process wait forever, which is not a nice strategy to ensure liveness…)
- What can we do?

How do you propose yourself to handle this?

Intuition:

- We can no longer wait for all processes to acknowledge a write (they might have failed and that can make a process wait forever, which is not a nice strategy to ensure liveness…)
- What can we do?
- The truth is... We cannot do much... We have to make our fault model stronger (through additional assumptions).

Making it more interesting…

- Same problem (1,N) Register.

- Different System Model:

- Fault Model: Crash (**with at most f faults**)
- Synchrony Model: Asynchronous System
- Known and fixed membership with size n >= 2f+1.

- Translation: A majority of processes will not fail.

How do you propose yourself to handle this?

How do you propose yourself to handle this?

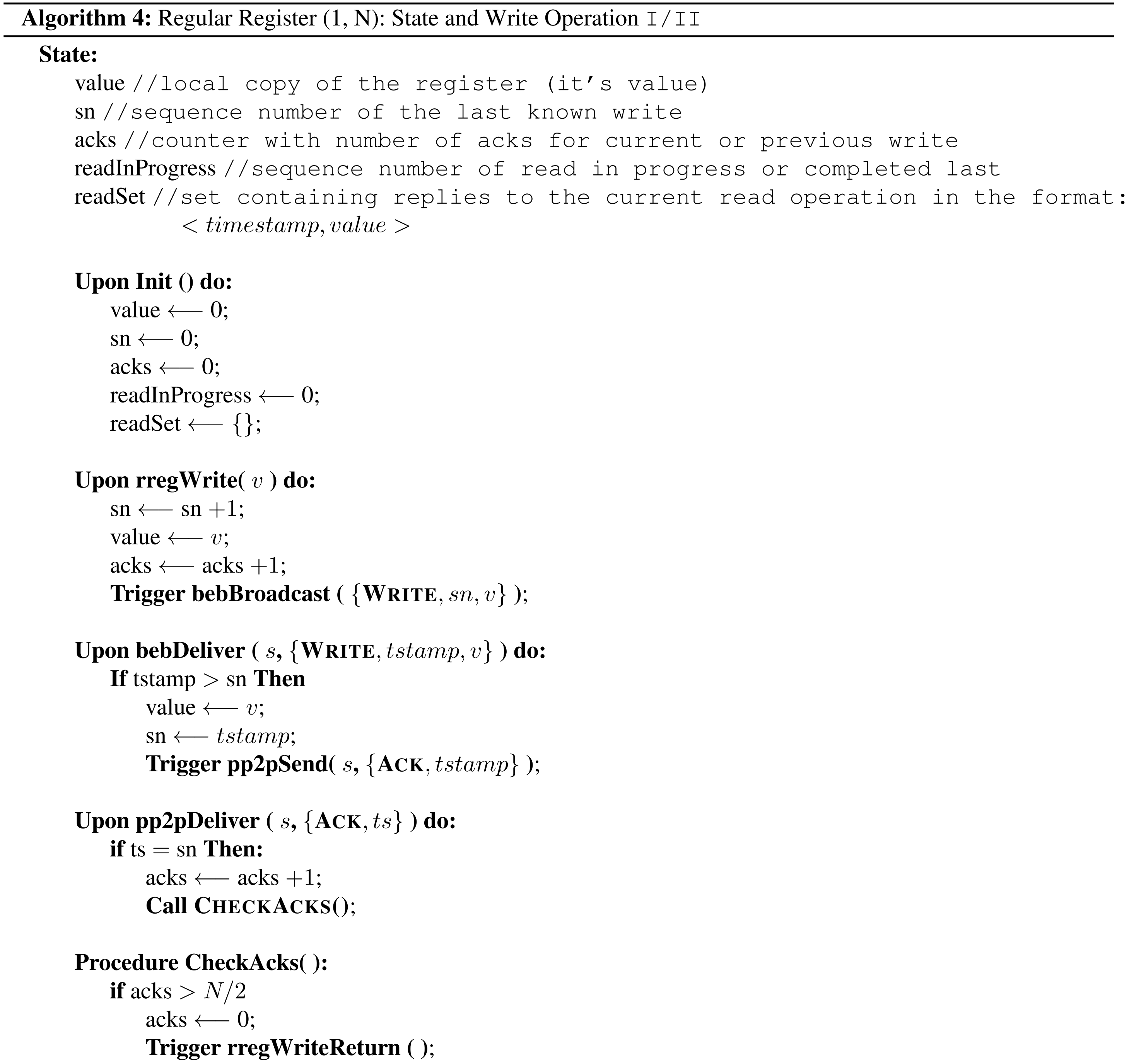
Intuition:

- We wait for at least N/2 + 1 processes to reply to us when we write, that ensures our writes will finish *(even if a total of f processes crash N/2+1 are sure to remain correct by the additional assumption).*
- But now… when I read, how can I be sure that I am reading the last value, If I read locally, I might have missed the last write(s).

How do you propose yourself to handle this?

- Intuition
- We wait for at least N/2 + 1 processes to reply to us when we write, that ensures our writes will finish.
- But now… when I read, how can I be sure that I am reading the last value, If I read locally, I might have missed the last write(s).
- So now I read from at least N/2+1 processes.
- This ensures that I will read at least from one process that knows the last write.
- I just need to figure out which is the last write (remember the timestamps?).

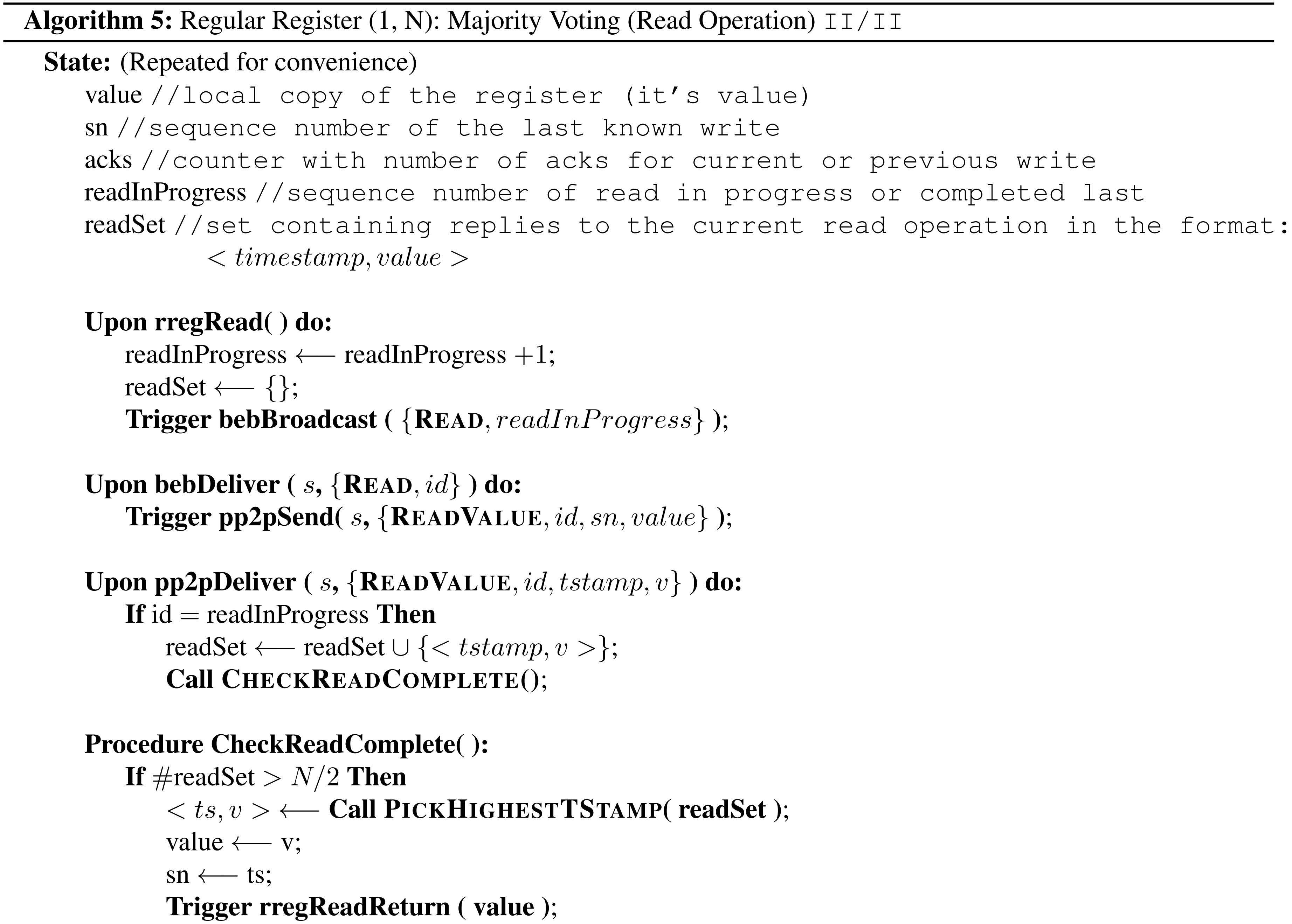
The Algorithm 1/2



Algorithm 4: Regular Register (1, N): State and Write Operation I/II

```
State:
    value //local copy of the register (it's value)
    sn //sequence number of the last known write
    acks //counter with number of acks for current or previous write
    readInProgress //sequence number of read in progress or completed last
    readSet //set containing replies to the current read operation in the format:
            $< timestamp, value >$

    Upon Init () do:
        value $\longleftarrow$ 0;
        sn $\longleftarrow$ 0;
        acks $\longleftarrow$ 0;
        readInProgress $\longleftarrow$ 0;
        readSet $\longleftarrow$ $\{\}$;

    Upon rregWrite( $v$ ) do:
        sn $\longleftarrow$ sn $+1$;
        value $\longleftarrow$ $v$;
        acks $\longleftarrow$ acks $+1$;
        Trigger bebBroadcast ( $\{\textsc{Write}, sn, v\}$ );

    Upon bebDeliver ( $s$, $\{\textsc{Write}, tstamp, v\}$ ) do:
        If tstamp $>$ sn Then
            value $\longleftarrow$ $v$;
            sn $\longleftarrow$ $tstamp$;
            Trigger pp2pSend( $s$, $\{\textsc{Ack}, tstamp\}$ );

    Upon pp2pDeliver ( $s$, $\{\textsc{Ack}, ts\}$ ) do:
        if ts $=$ sn Then:
            acks $\longleftarrow$ acks $+1$;
            Call CheckAcks();

    Procedure CheckAcks( ):
        if acks $>$ $N/2$
            acks $\longleftarrow$ 0;
            Trigger rregWriteReturn ( );
```

The Algorithm 2/2



Algorithm 5: Regular Register (1, N): Majority Voting (Read Operation) II/II

```
State: (Repeated for convenience)
    value //local copy of the register (it's value)
    sn //sequence number of the last known write
    acks //counter with number of acks for current or previous write
    readInProgress //sequence number of read in progress or completed last
    readSet //set containing replies to the current read operation in the format:
            $< timestamp, value >$

    Upon rregRead( ) do:
        readInProgress ⟵ readInProgress +1;
        readSet ⟵ {};
        Trigger bebBroadcast ( {READ, $readInProgress$} );

    Upon bebDeliver ( $s$, {READ, $id$} ) do:
        Trigger pp2pSend( $s$, {READVALUE, $id, sn, value$} );

    Upon pp2pDeliver ( $s$, {READVALUE, $id, tstamp, v$} ) do:
        If id = readInProgress Then
            readSet ⟵ readSet ∪ {$< tstamp, v >$};
            Call CHECKREADCOMPLETE();

    Procedure CheckReadComplete( ):
        If #readSet $> N/2$ Then
            $< ts, v >$ ⟵ Call PICKHIGHESTTSTAMP( readSet );
            value ⟵ v;
            sn ⟵ ts;
            Trigger rregReadReturn ( value );
```

Let’s try to understand why is this correct.

- And the key aspects in showing the correctness of the solution is twofold:

1. Operations always terminate because you only wait for a number of processes that will never fail (because at most there are f failures).
2. Any write and read operation (more generally any pair of operations) will intersect in one correct process.

Let’s try to understand why is this correct.

- And the key aspects in showing the correctness of the solution is twofold:

1. Operations always terminate because you only wait for a number of processes that will never fail (because at most there are f failures).
2. Any write and read operation (more generally any pair of operations) will intersect in one correct process.

This intersection is the basis for quorum-based replication algorithms.

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.
- If any pair of operations executed in the lifetime of the system has a non-empty intersection in the set of replicas executing it, we call this a quorum system.

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.
- If any pair of operations executed in the lifetime of the system has a non-empty intersection in the set of replicas executing it, we call this a quorum system.

(Notice that, you still send (write) operations to all replicas in the system, however you make progress as soon as a valid quorum of replies is received)

Quorum System

- Given a set of replicas $P=\{p_1,p_2,...,p_n\}$, we call a quorum system as a set $Q=\{q_1,q_2,...,q_m\}$ of subsets of P such that $\forall i,j,\ q_i \cap q_j \neq \emptyset$

Read-Write Quorum System

- Given a set of replicas $P=\{p_1,p_2,\ldots,p_n\}$, we define a Read-Write Quorum System as a pair of sets $\mathbf{R=\{r_1,r_2,\ldots,r_m\}}$, $\mathbf{W=\{w_1,w_2,\ldots,w_r\}}$, of subsets of P, such that:
  - $\forall i,j,\ r_i \cap w_j \neq \emptyset$ (read intersects writes)
  - $\forall i,j,\ w_i \cap w_j \neq \emptyset$ (writes intersects writes)

- This has the benefit of allowing potentially smaller read quorums, which is important for making read operations faster (in systems where read operations are much more frequent).

Quorum Types: Read one/write all

- Replication strategy based on a read-write quorum system :
  - Read operations can be executed in any (and a single) replica.
  - Write operations must be executed in all replicas.

- Properties:
  - Very fast read operations.
  - Heavy write operations. If a single replica fails, then write operations can no longer be executed with sucess.

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.

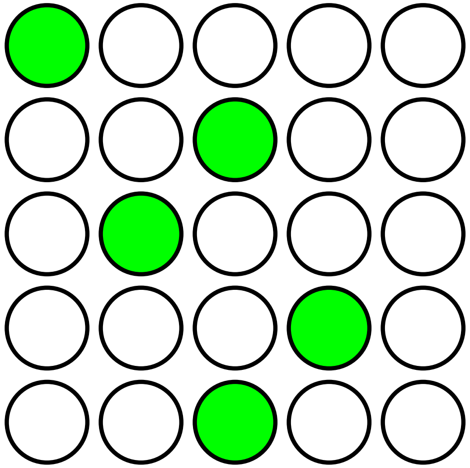
Quorum Types: Weighted Voting

- 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 $w_R + w_w > w_{total}$
  - 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 $W=\{w_1,w_2,\ldots,w_m\}$: $\Sigma w(w_i) \geq w_w$

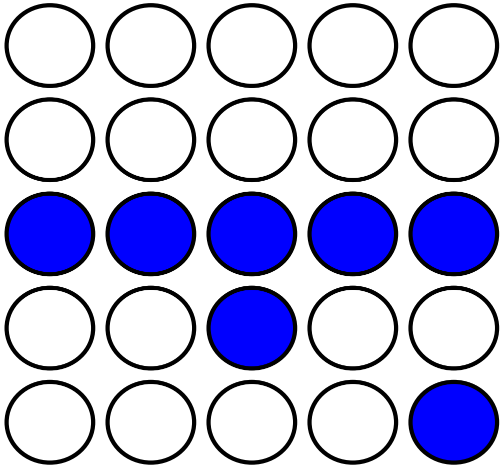
- Properties:
  - This allows to balance the size of different quorums for different read and write operations.
  - Replicas are no longer completely equivalent among them.

Quorum Types: Grid

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



Read Quorum:
Na element from each line



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

Quorum Types: Grid

- Properties:
  - Size of quorums grows sub-linearly with the total number of replicas in the system: O(sqrt(2))
    - Load on each replica also increases sub-linearly with the total number of operations.
  - It allows to balance the dimension of read and write quorums (for instance to deal with different rates of each type of request) by manipulating the size of the grid (i.e., making it a rectangle)
  - Somewhat more complex to use.

What does it matter?

Quorum Sizes

- 9 replicas:
  - Majority: 5/9 ≈ 0.5
  - Grid: 5/9 ≈ 0.5
- 100 replicas:
  - Majority: 51/100 ≈ 0.5
  - Grid: 19/100 ≈ 0.2
- 10000 replicas:
  - Majority: 5001/10000 ≈ 0.5
  - Grid: 199/10000 ≈ 0.02

Fault Tolerance

- +-15 replicas:

Prob failure	0.1	0.3	0.5
Majority	0.00003	0.05	0.5
Grid	0.0058	0.24	0.7

- +-28 replicas:

Prob failure	0.1	0.3	0.5
Majority	0.00000	0.01	0.5
Grid	0.0019	0.17	0.7

Homework 2 (A challenge):

- Give me (in pseudo-code) your best proposal to an algorithm with the following interface and properties.
  - Interface:
    - **request: cPropose(*v*)** (*v* is some value, there is some mechanism where all processes in the system will receive simultaneously a request of this type, but potentially with different values of v from process to process).
    - **notification: cDecide(*v*)** (this notification should only be triggered once by each process in the system, when a process triggers this notification we say it has decided value v).
  - Properties:
    - **Termination**: Every correct process eventually decides a value.
    - **Validity**: If a process decides v, then v was proposed by some process.
    - **Integrity**: No process decides twice.
    - **Agreement**: No two correct processes decide differently.

- Your solutions should be correct in a **synchronous** system in the **fail-stop model** (you have access to the ***crash(p)*** event). The membership is static and known by everyone (all processes know **π**). You can assume **f < # π**. You can use **Best Effort Broadcast**.