SOFTWARE REPLICATION: SYNCHRONOUS SYSTEMS

DISTRIBUTED SYSTEMS

Master of Science in Cyber Security

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

Replica = istance of a distributed object

OUTLINE

- Motivating Example
- Definition and System Model:
  - Replicated Object
  - Processes and links
- Consistency Criteria:
  - Linearizability
  - Sufficient conditions for a linearizable object
- Active vs Passive replication:
  - Primary Backup (passive)
  - Active Replication

↳ total order broadcast

MOTIVATIONS

Client 1

Client 2

Append 3

get List[0]

3

List:

[3]

Server

MOTIVATIONS

Client 1

Client 2

If the server crashes the object is destroyed forever.

Failure probability p

depends on the structure of the server

$\neq 0$

Server

MOTIVATIONS

Client 1

Client 2

If the server crashes the object is destroyed forever.

Failure probability p

Fixed p, how do we increase the probability that our system works?

Server

MOTIVATIONS

Client 1

Client 2

If the server crashes the object is destroyed forever.

Server

Failure probability p

Fixed p, how do we increase the probability that our system works? Replication!

MOTIVATIONS

Client 1

Client 2

Server A

Server B

Server C

We replicate our distributed object on **k** servers. In **synchronous systems** we can build systems that tolerate **f=k-1 failures**.

k - 1

MOTIVATIONS

Client 1

Client 2

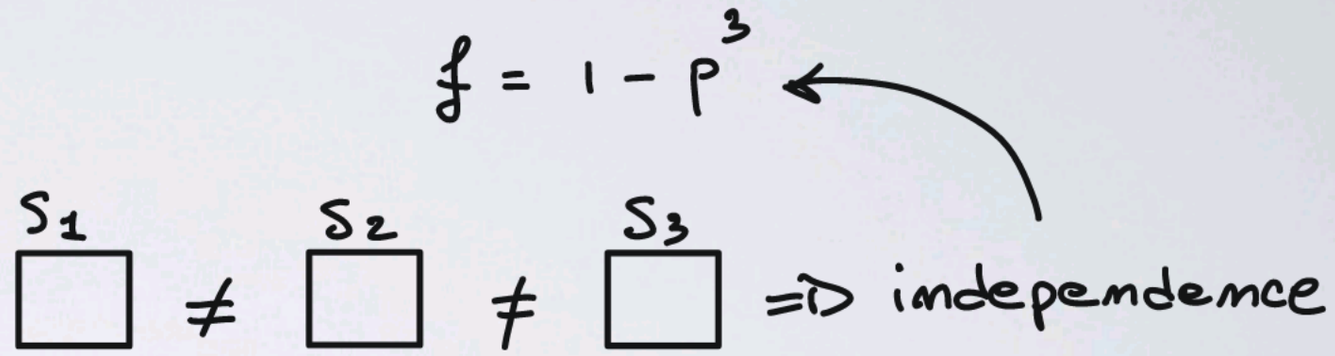
Single server failure probability

$$p$$

We replicate our distributed object on **k** servers. In **synchronous systems** we can build systems that tolerate **f=k-1 failures**.

If k=3 and the failure probability of a single server is p, what is the failure probability of our object?

MOTIVATIONS



- Fault Tolerance
  - Guarantees the availability of a service (also called object) despite failures
- Assuming *p* the failure probability of a Server. *O*'s **availability** in a centralized solution is *1-p*.
- On a **synchronous system** we can replicate an object *O* on ***N* nodes** and assuming *p* the (**independent- ensured with diversity)** failure probability of each replica, O's **availability** is $1 - p^N$

MOTIVATIONS

- Fault Tolerance
  - Guarantees the availability of a service (also called object) despite failures
- Assuming *p* the failure probability of a Server. *O*'s **availability** in a centralized solution is *1-p*.
- On a **synchronous system** we can replicate an object *O* on ***N*** **nodes** and assuming *p* the (**independent- ensured with diversity)** failure probability of each replica, O's **availability** is *1-* p^N

Diversity: A technique used to increase the availability of software of hardware. A set of replica is diverse if replica run on different hardware, different operative systems and different implementations of our object.

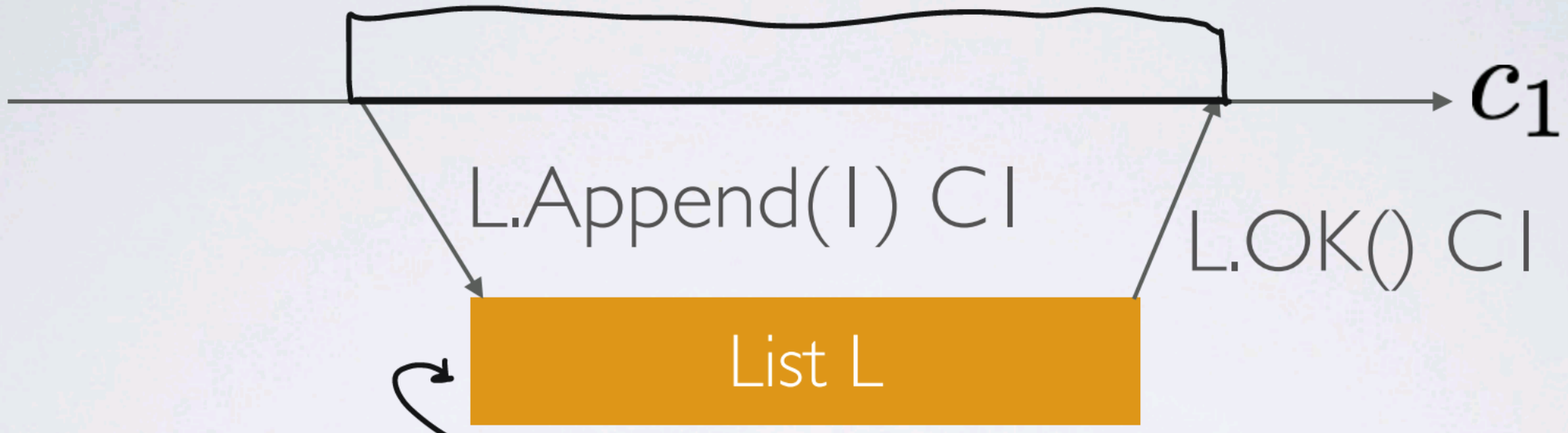
SYSTEM MODEL

- The system is composed of a set of processes
  - Processes are connected through perfect point-to-point links
  - Processes may fail by crashing
  - We have a perfect failure detector P
  - **C processes are clients** and other N are used to implement our distributed object.

SYSTEM MODEL

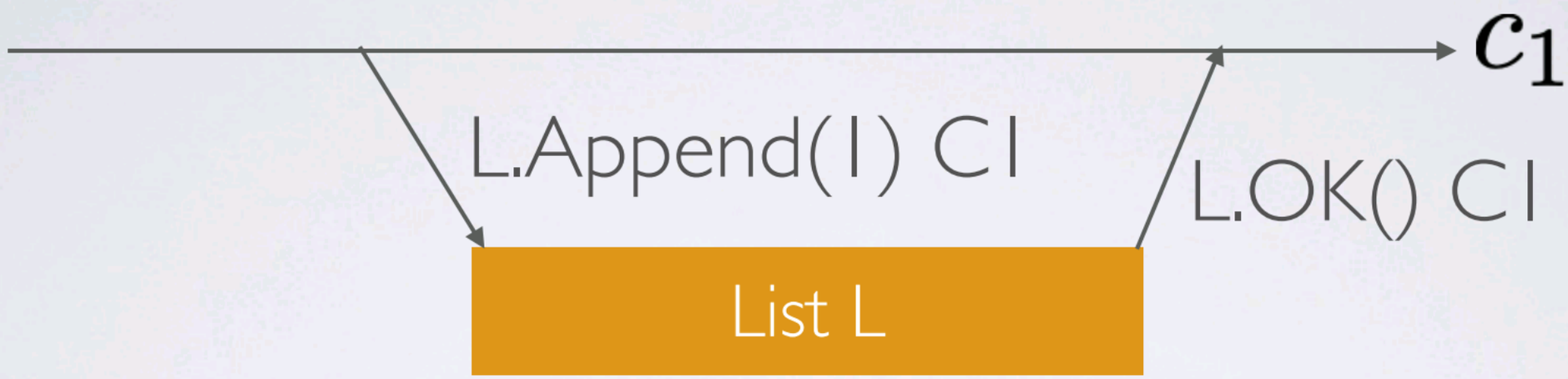
- The system is composed of a set of processes
  - Processes are connected through perfect point-to-point links
  - Processes may fail by crashing
  - We have a perfect failure detector P
  - C processes are clients and other **N are used to implement our distributed object**.

SYSTEM MODEL



- A client interacts with a **distributed object** implemented in our set of processes.
- Our object has a certain **state S** and a set of allowed **Operations**.
- Each operation is able to either read the state or to modify it.
- An operation is issued by a client with an invocation and it is pending until the client receives a response.
  - Invocation: OBJ.OP_NAME([Arguments]) id invoking client
  - Response: OBJ.OK([Return Values]) id client

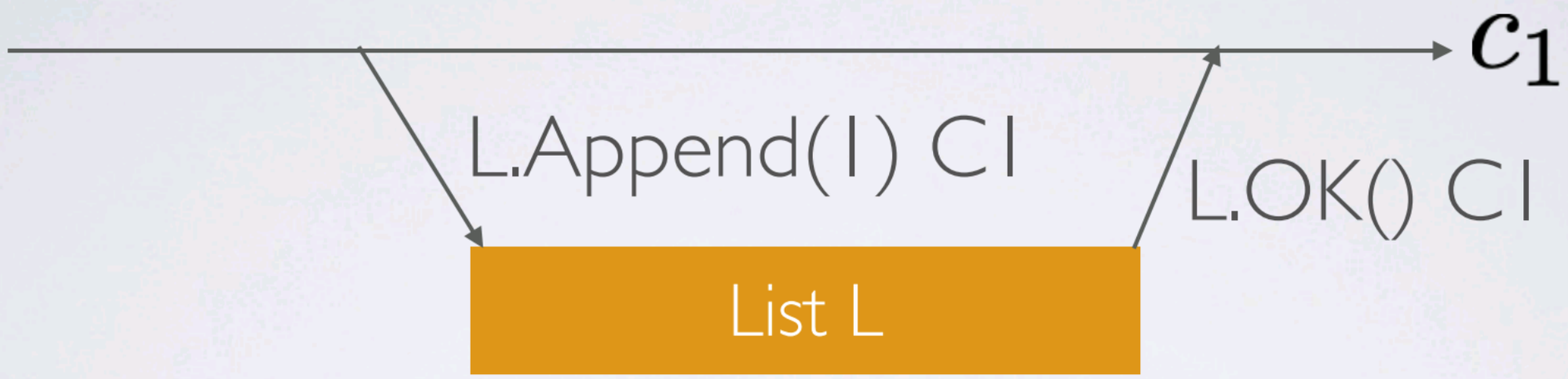
SYSTEM MODEL



- A client interacts with a **distributed object** implemented in our set of processes.
- Our object has a certain **state S** and a set of allowed **Operations**.
- Each operation is able to either read the state or to modify it.
- An operation is issued by a client with an invocation and it is pending until the client receives a respor
  - Invocation: OBJ.OP_NA
  - Response: OBJ.OK([Ret

We assume that a client does not invoke (or issue) an operation until the pending one completes.

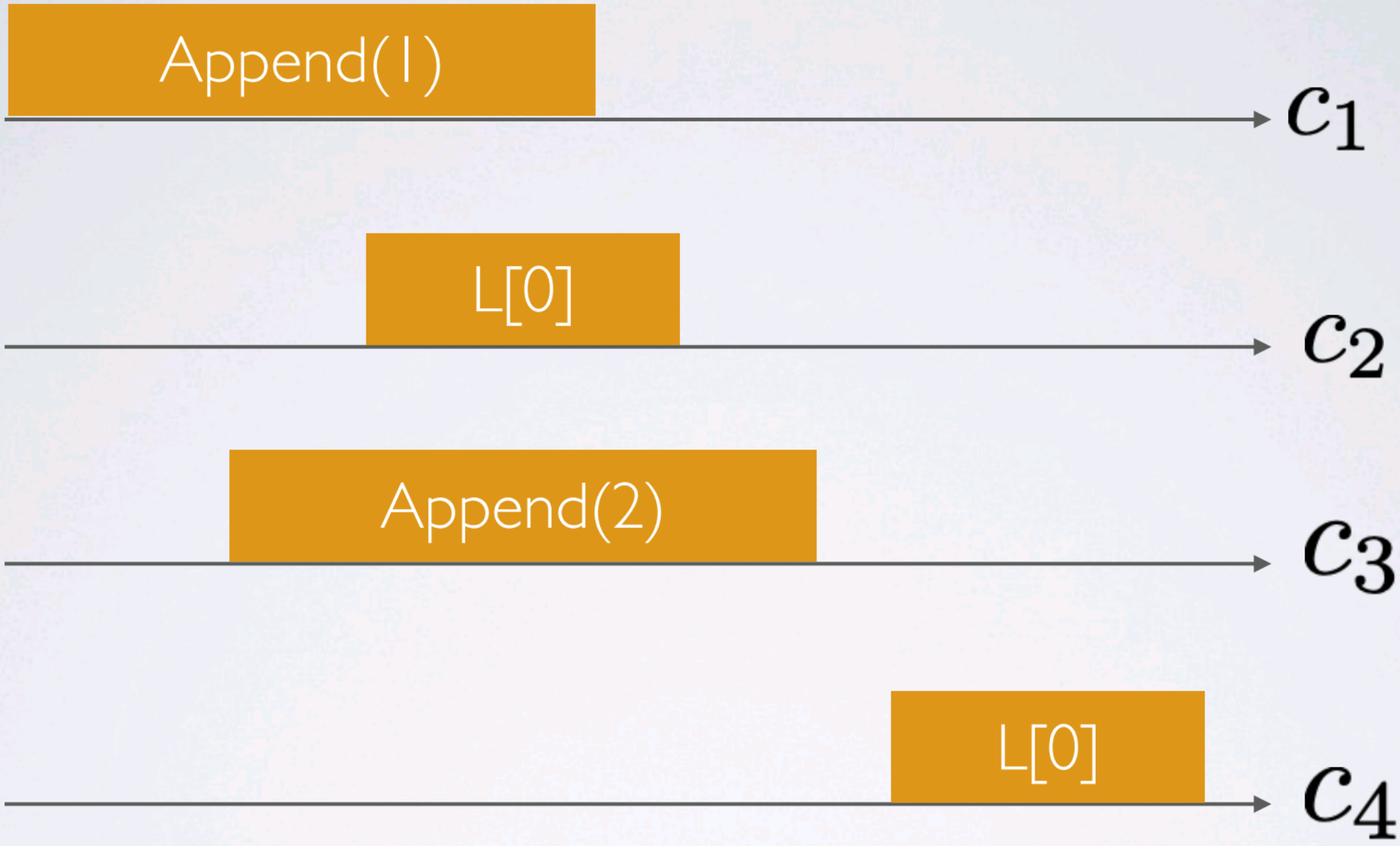
SYSTEM MODEL



P1 P2 ... Pn

Our object (list in the example) is implemented on a set of n processes. This detail is totally transparent to **clients**.

SYSTEM MODEL



SYSTEM MODEL

Many clients means concurrent operation. What we were defining to cope with concurrency in the register?

SYSTEM MODEL

Many clients means concurrent operation. What we were defining to cope with concurrency in the register? **CONSISTENCY SEMANTIC!**

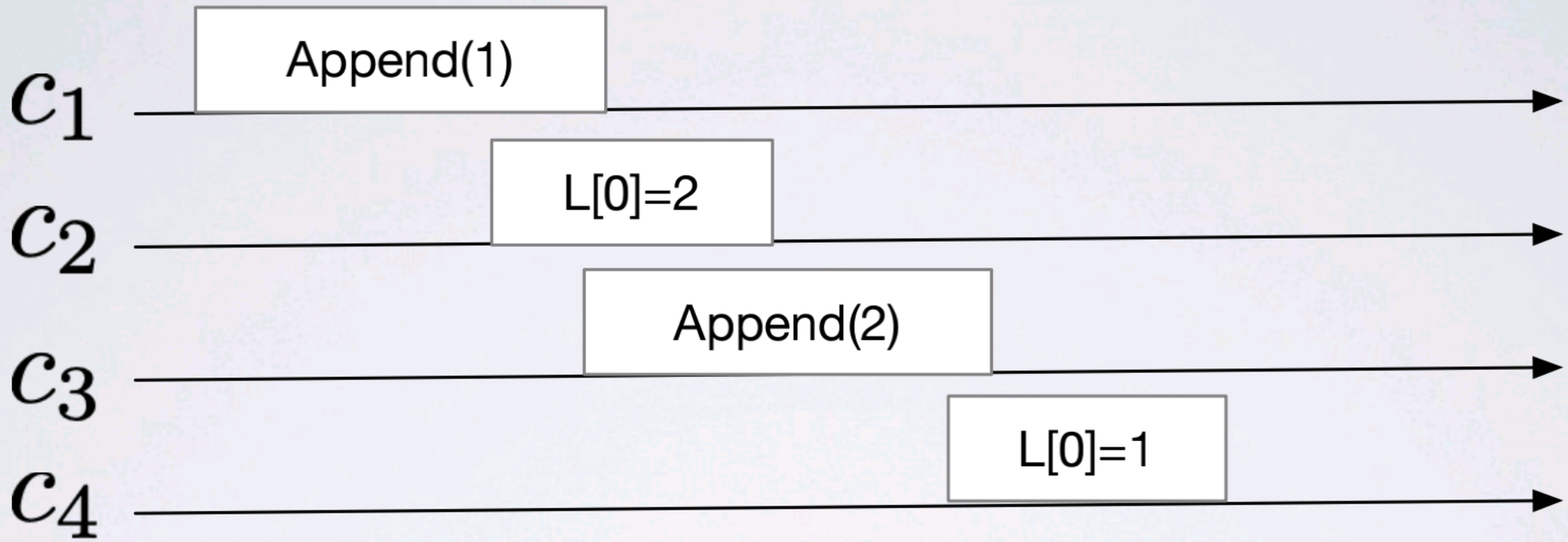
CONSISTENCY CRITERIA

- A consistency criterion defines the result returned by an operation
  - It defines all acceptable runs.
- Three main consistency criteria are defined in literature
  - Linearizability /Atomicity
  - Sequential consistency

  } Strong Consistency
  - **Causal consistency** } Weak Consistency

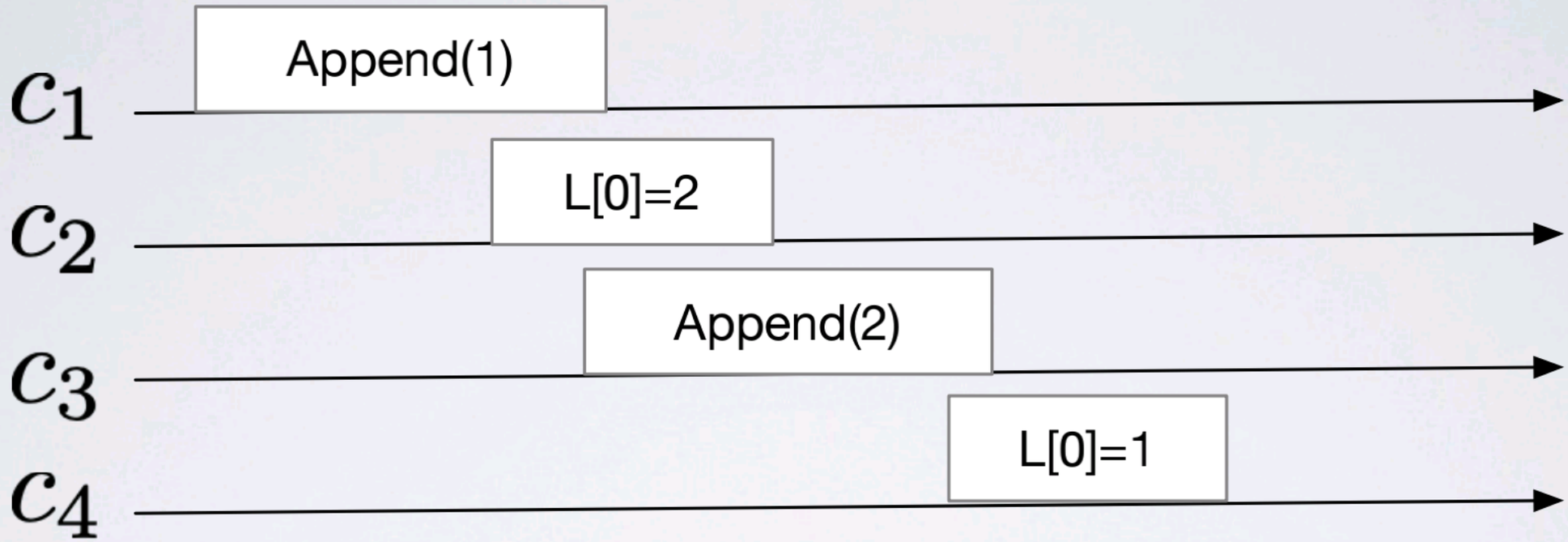
- We will only see the linearizability semantic.
- The Sequential consistency you can imagine how it works. Generalize what we have seen for a register.

LINEARIZABILITY



An **Execution on** our generic object **O** is **linearizable** if for any operation **Op** on O we can find a **fictional point (linearisation point)** between the start and the end of OP in which the operation happens instantaneously and is visible to our entire system.

LINEARIZABILITY

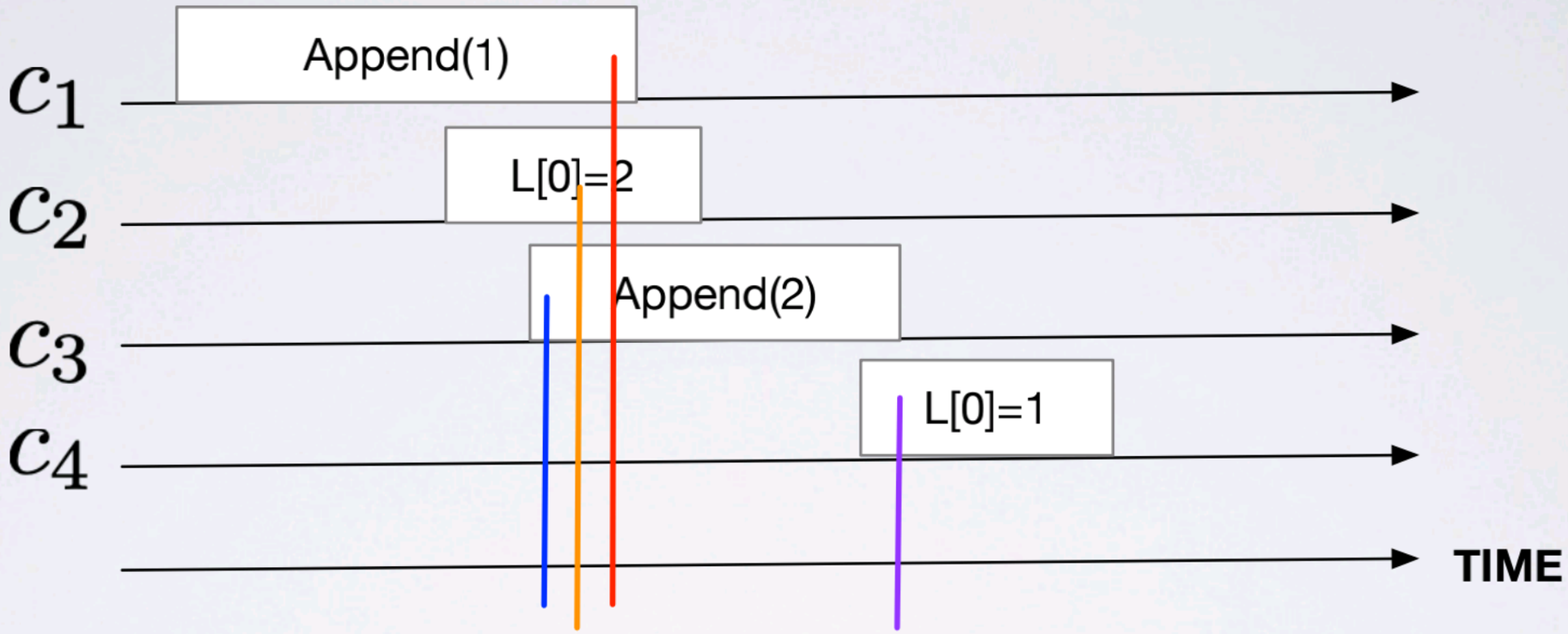


- An **Execution on** our generic object **O** is **linearizable** if for any operation **Op** on O we can find a **fictional point (linearisation point)** between the start and the end of OP in which the operation happens instantaneously and is visible to our entire system.

Is this execution linearizable?

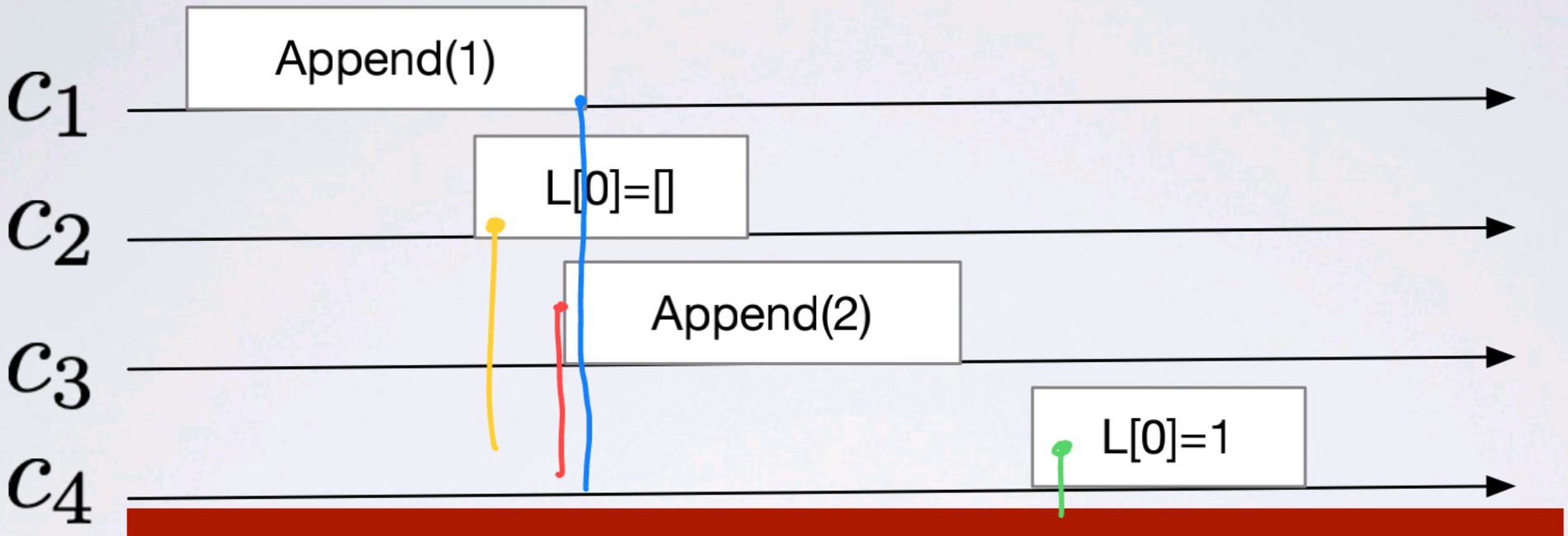
yes (c_3, c_2, c_1, c_4)

LINEARIZABILITY



- An **Execution on** our generic object **O** is **linearizable** if for any operation **Op** on O we can find a **fictional point (linearisation point)** between the start and the end of OP in which the operation happens instantaneously and is visible to our entire system.

LINEARIZABILITY

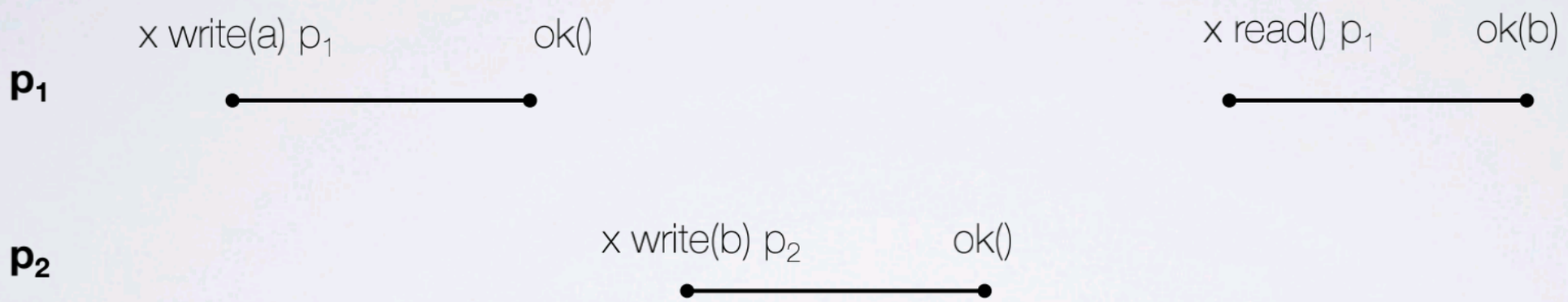


Is this execution linearizable?

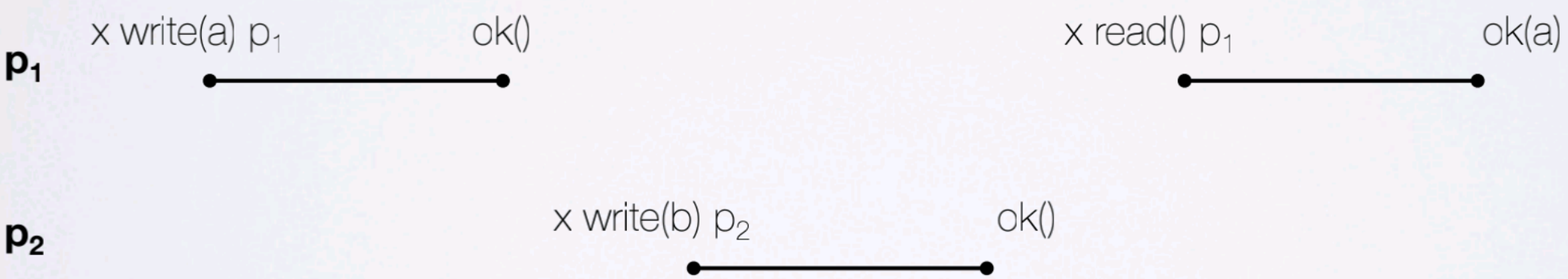
- An **Execution on** our generic object **O** is **linearizable** if for any operation **Op** on O we can find a **fictional point (linearisation point)** between the start and the end of OP in which the operation happens instantaneously and is visible to our entire system.

EXAMPLE: SHARED VARIABLE

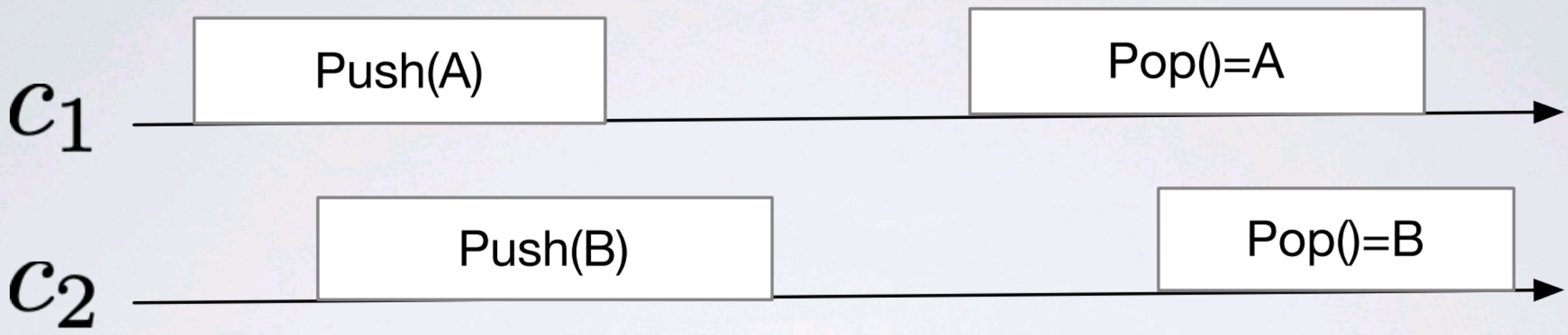
1. **LINEARIZABLE**



2. **NON LINEARIZABLE BUT SEQUENTIAL CONSISTENT**



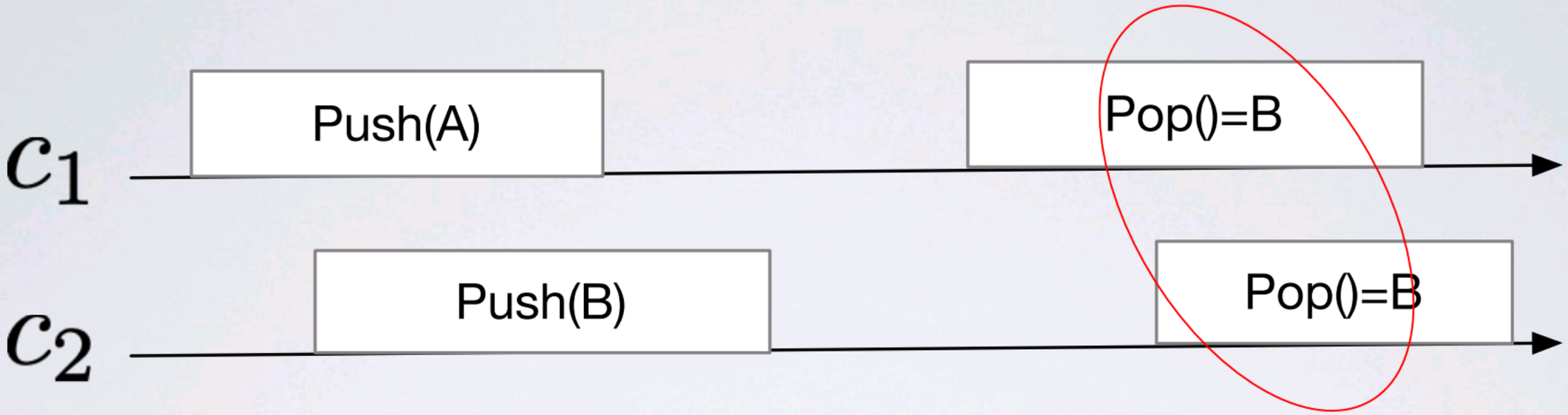
EXAMPLE: STACK



LINEARIZABLE?

E= {Push(B), Push(A), Pop()=A, Pop()=B}

EXAMPLE: FIFO QUEUE



- **LINEARIZABLE?**

NO!

Not Legal wrt the specification of a Stack!

violating the semantic of the stack

- **SEQUENTIAL CONSISTENT?**

NO!

HOW DO IMPLEMENT A GENERIC LINEARIZABLE (OR ATOMIC) **REPLICATED** OBJECT?

SUFFICIENT CONDITIONS FOR LINEARIZABILITY

- Replicas (the processes implementing O) **must agree** on the set of invocations they handle and **on the order** according to which they handle these invocations
- There are three conditions that if respected on the invocations automatically implies that O is linearizable.
- **Atomicity**
- **Ordering**
- **Blocking operations**

SUFFICIENT CONDITIONS FOR LINEARIZABILITY

operation

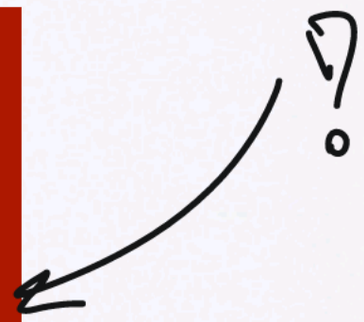
- **Atomicity**: Given an invocation [*x op(arg) p_i*], if one replica of the object *O* handles this invocation, then every correct replica of *O* also handles the invocation [*x op(arg) p_i*]. -> **All or nothing.**
- **Ordering**: Given two invocations [*x op1(arg1) p_i*] and [*x op2(arg2) p_j*] if two replicas handle both the invocations, they handle them in the same order. **-> Consistency of executed Operations.**
- **Blocking operations**: The client does not see the result of an operation until at least one (correct) replica handled it.

SUFFICIENT CONDITIONS FOR LINEARIZABILITY

AO → Total Order Broadcast ⇉

- **Atomicity (A)**: Given an invocation [*x op(arg) p_i*], if one replica of the object *O* handles this invocation, then every correct replica of *O* also handles the invocation [*x op(arg) p_i*]. -> **All or nothing.**
- **Ordering (O)**: Given two invocations [*x op1(arg1) p_i*] and [*x op2(arg2) p_j*] if two replicas handle both the invocations, they handle them in the same order. **-> Consistency of executed Operations.**
- **Blocking operations (B)**: The client does not see the result of an operation until at least one (correct) replica handled it.

The conditions are **sufficient**. If we have them we can implement any Object. However are not **necessary**. There are **some objects** that can be Atomic even if we do not have AOB.



SUFFICIENT CONDITIONS FOR LINEARIZABILITY

Exists an Atomic object we can implement without using consensus?

Asymc: F > 0 ; F < n/2 → Quorum & Atomic Registers

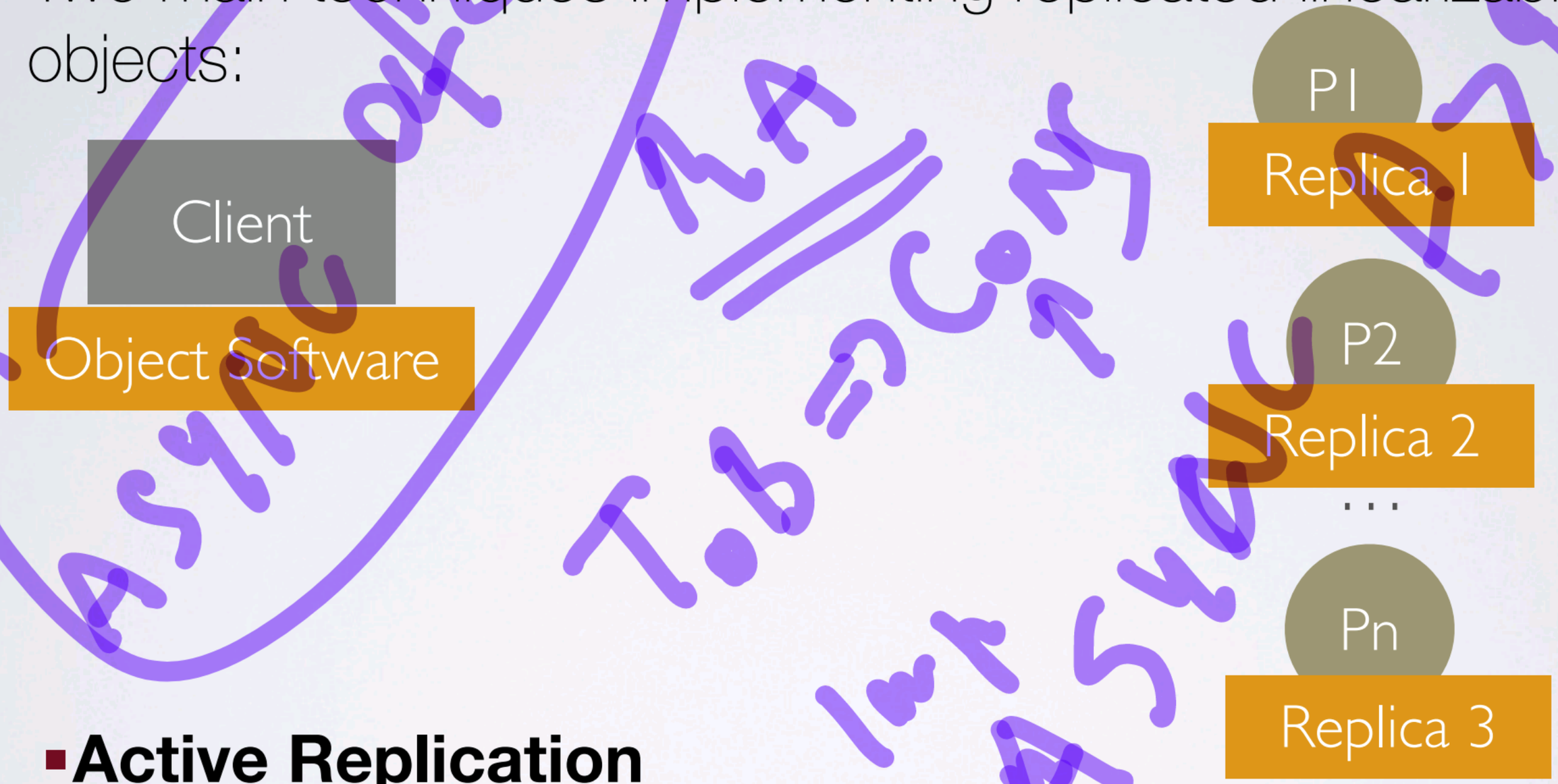
- **Atomicity (A)**: Given an invocation [*x op(arg) p_i*], if one replica of the object *O* handles this invocation, then every correct replica of *O* also handles the invocation [*x op(arg) p_i*]. -> **All or nothing.**
- **Ordering (O)**: Given two invocations [*x op1(arg1) p_i*] and [*x op2(arg2) p_j*] if two replicas handle both the invocations, they handle them in the same order. **-> Consistency of executed Operations.**
- **Blocking operations (B)**: The client does not see the result of an operation until at least one (correct) replica handled it.

The conditions are **sufficient**. If we have them we can implement any Object. However are not **necessary**. There are **some objects** that can be Atomic even if we do not have AOB.

(*) Can you name an atomic object that can be implemented without these conditions?

REPLICATION TECHNIQUES

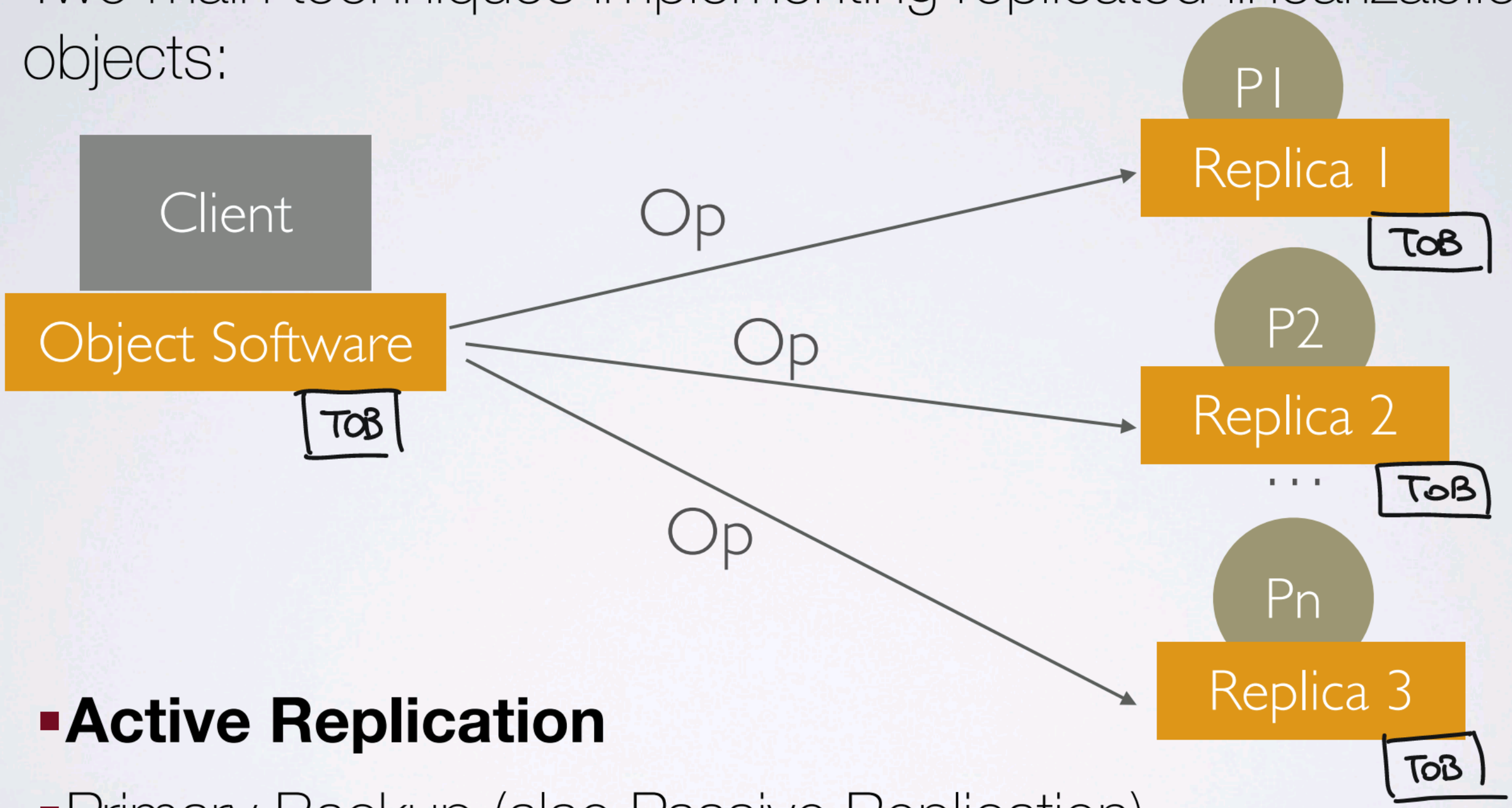
- Two main techniques implementing replicated linearizabile objects:



- **Active Replication**
- Primary Backup (also Passive Replication)

REPLICATION TECHNIQUES

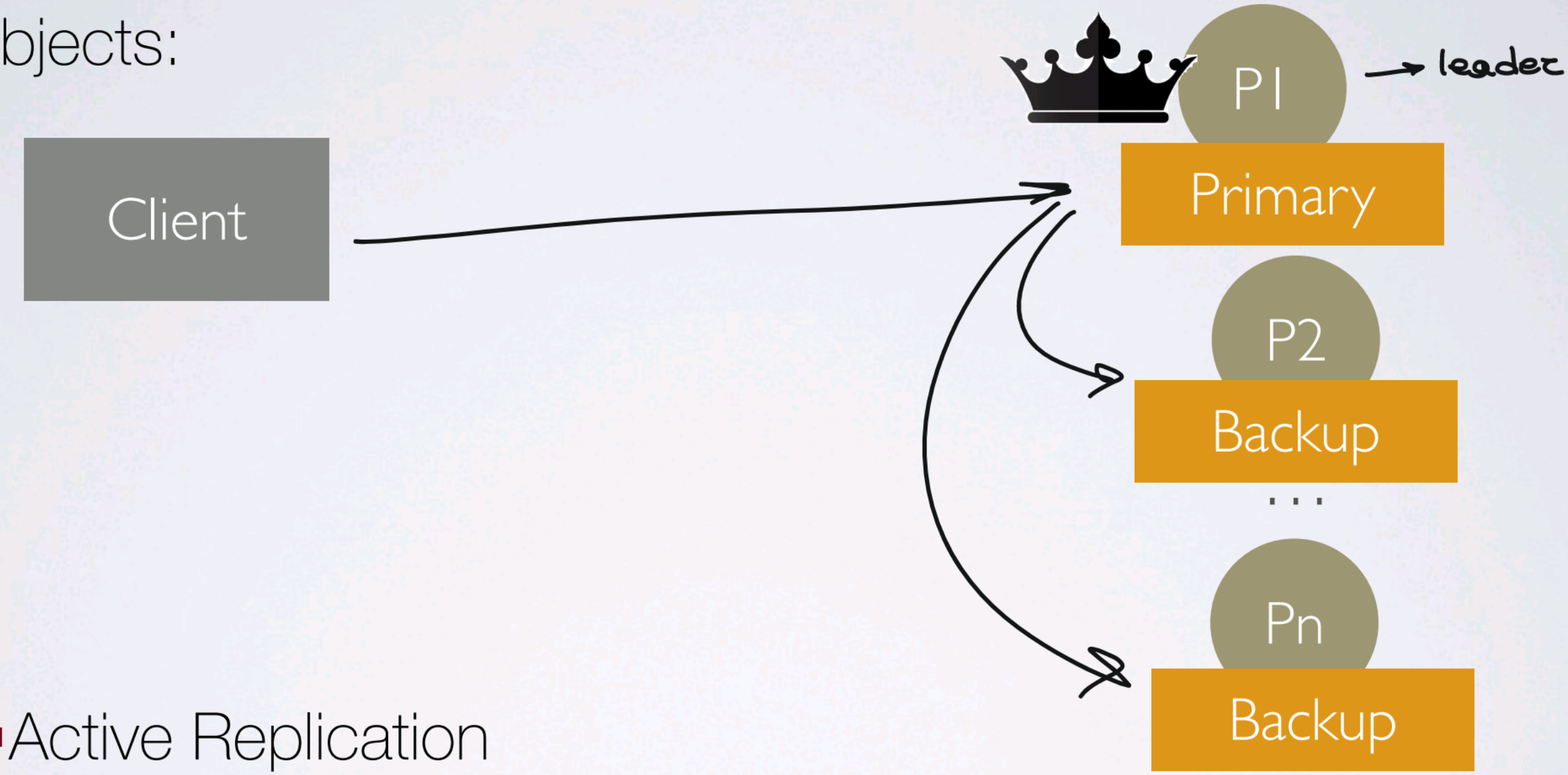
- Two main techniques implementing replicated linearizabile objects:



- **Active Replication**
- Primary Backup (also Passive Replication)

REPLICATION TECHNIQUES

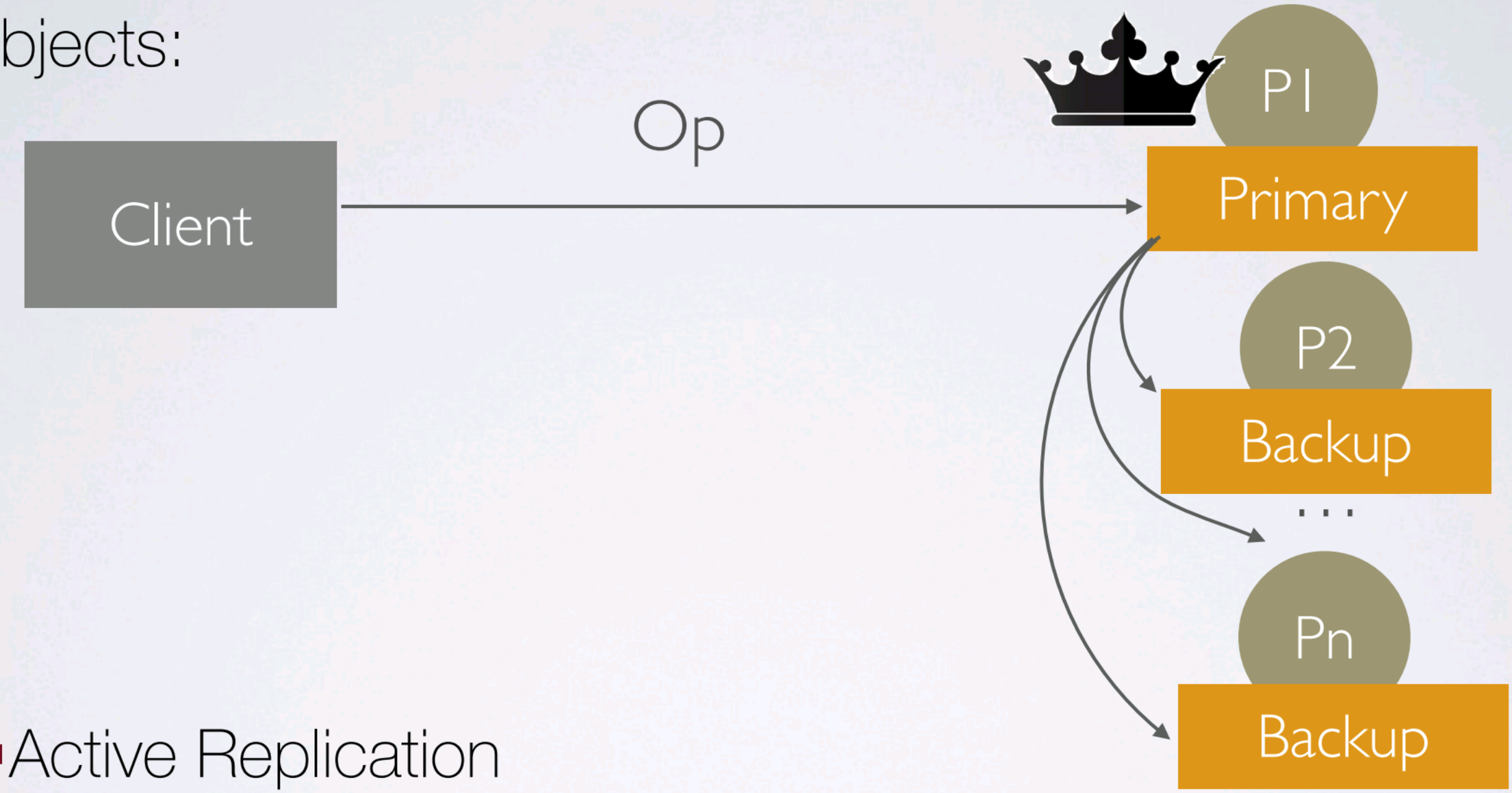
- Two main techniques implementing replicated linearizabile objects:



- Active Replication
- **Primary Backup** (also **Passive Replication**)

REPLICATION TECHNIQUES

- Two main techniques implementing replicated linearizabile objects:

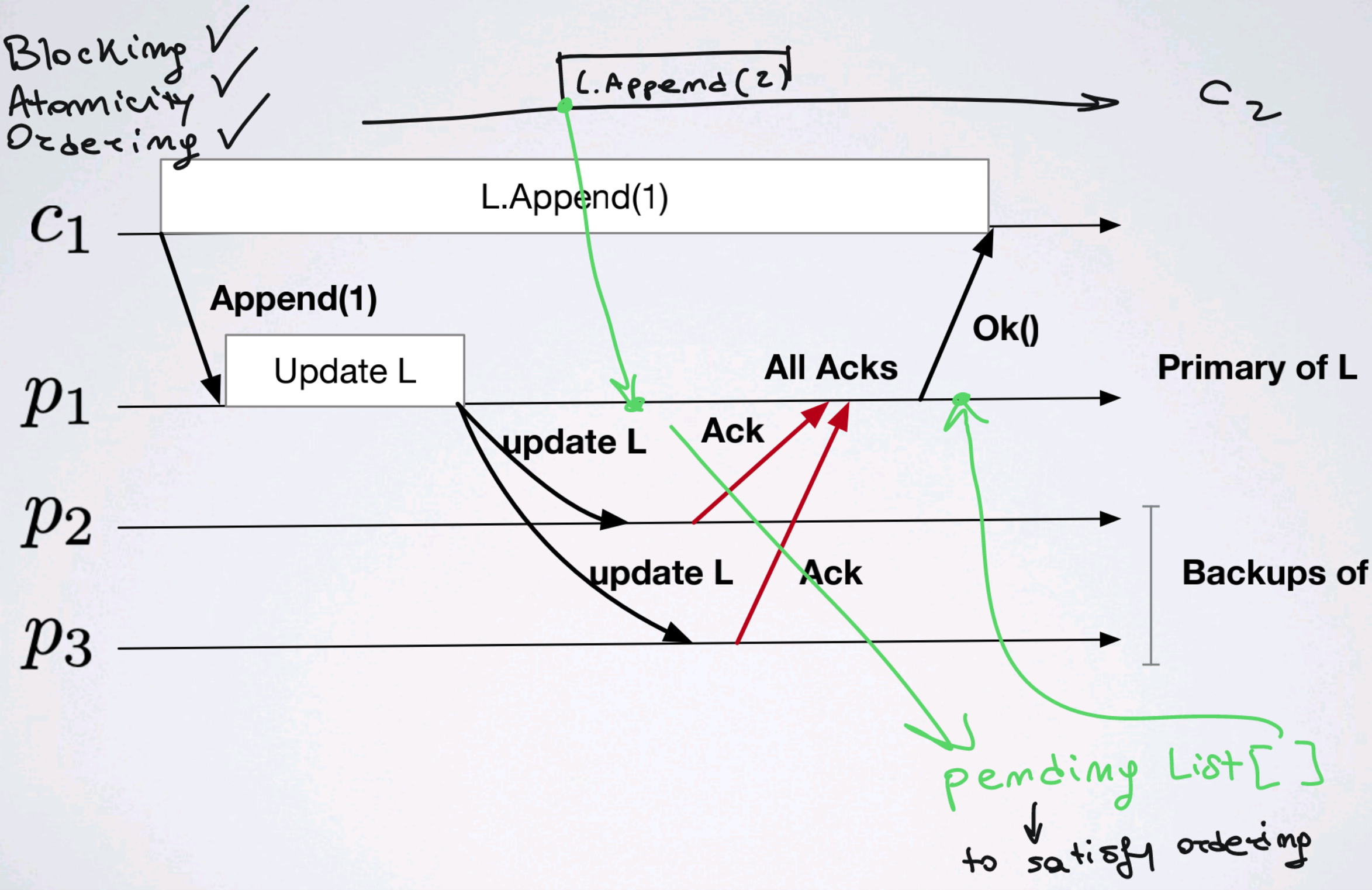


- Active Replication
- **Primary Backup** (also **Passive Replication**)

PRIMARY BACKUP

- Replicas are partitioned in two sets. Primary and backup. The Primary set is a singleton (a set with just one process - a leader).
- Primary (unique):
  - Receives invocations from clients and sends back the answers. Is the only ones that interacts with the clients.
  - Given an object *O*, *prim(O)* returns the primary of *O*.
- Backup:
  - Interacts with *prim(O)* only
  - is used to guarantee fault tolerance by replacing a primary when crashes
- NB: the following slides assume a synchronous system (you have **P**).

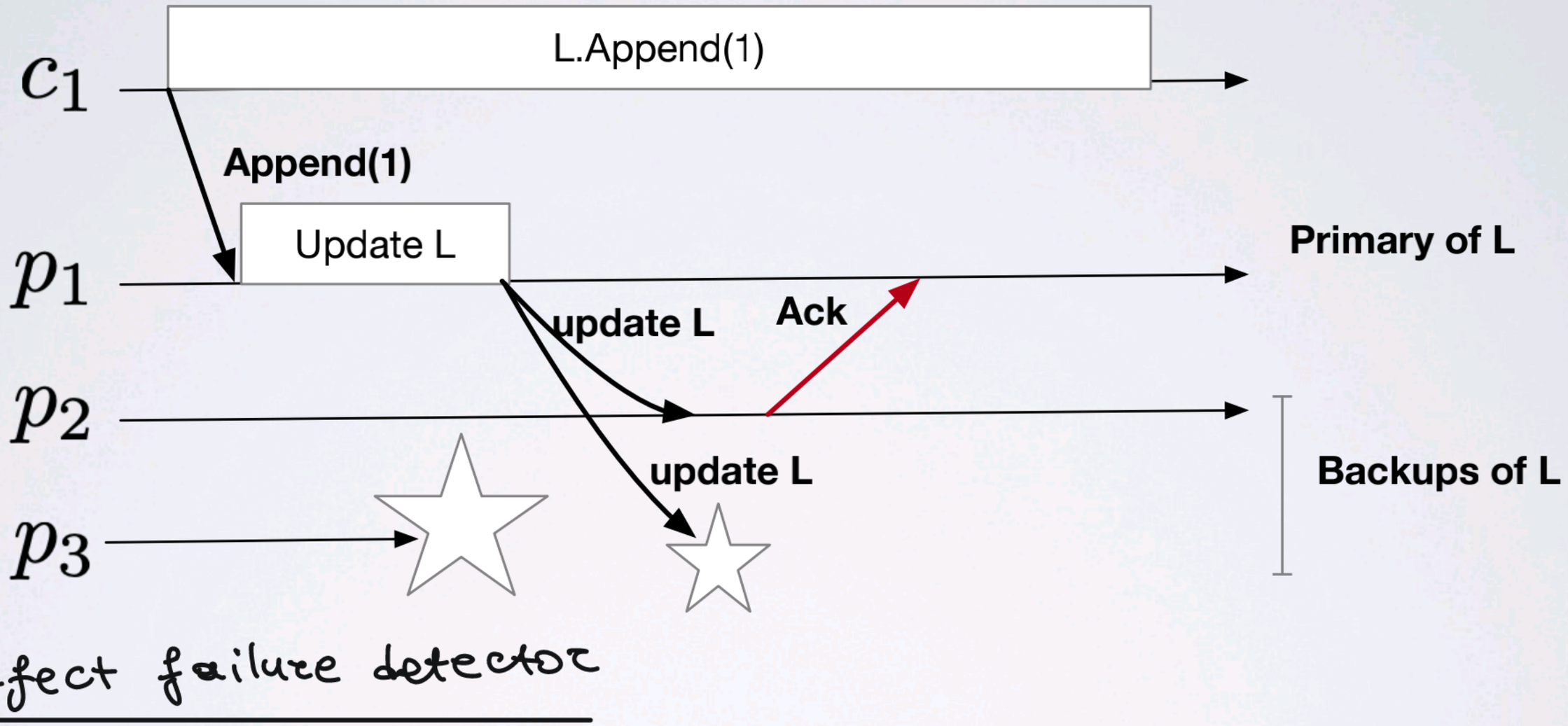
PRIMARY BACKUP SCENARIO



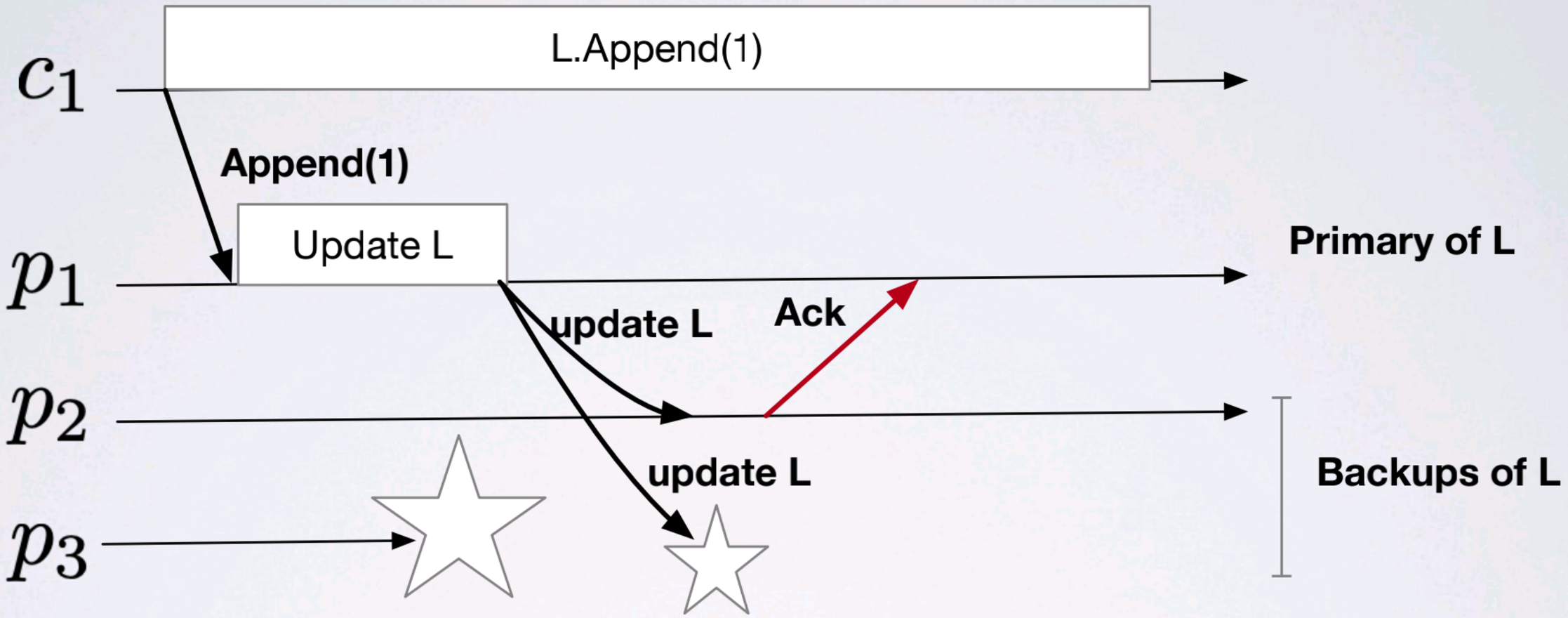
CASE 1: NO CRASH

- When update messages are received by backups, they update their state and send back the *ack* to *prim(O)*.
- *prim(O)* waits for an *ack* message from each correct backup and then sends back the answer, *res*, to the client.
- How to guarantee Linearizability:
  - **ATOMICITY:** an operation completes only when is executed by all backups
  - **ORDERING:** the order in which *prim(O)* receives clients' invocations defines the order of the operations on the object.
  - **Blocking operations**: each operation is blocked until the primary answers -> execution on all backups.

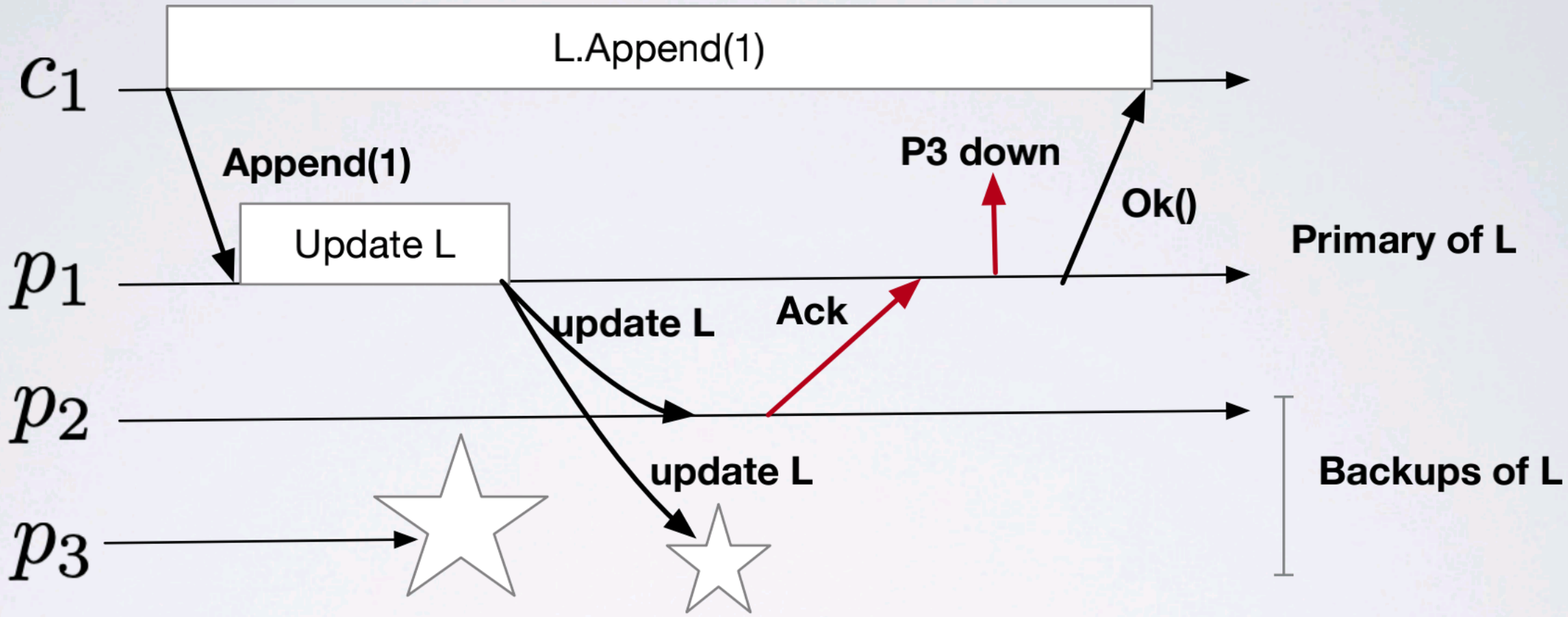
CASE 2: BACKUP CRASH



CASE 2: BACKUP CRASH



CASE 2: BACKUP CRASH

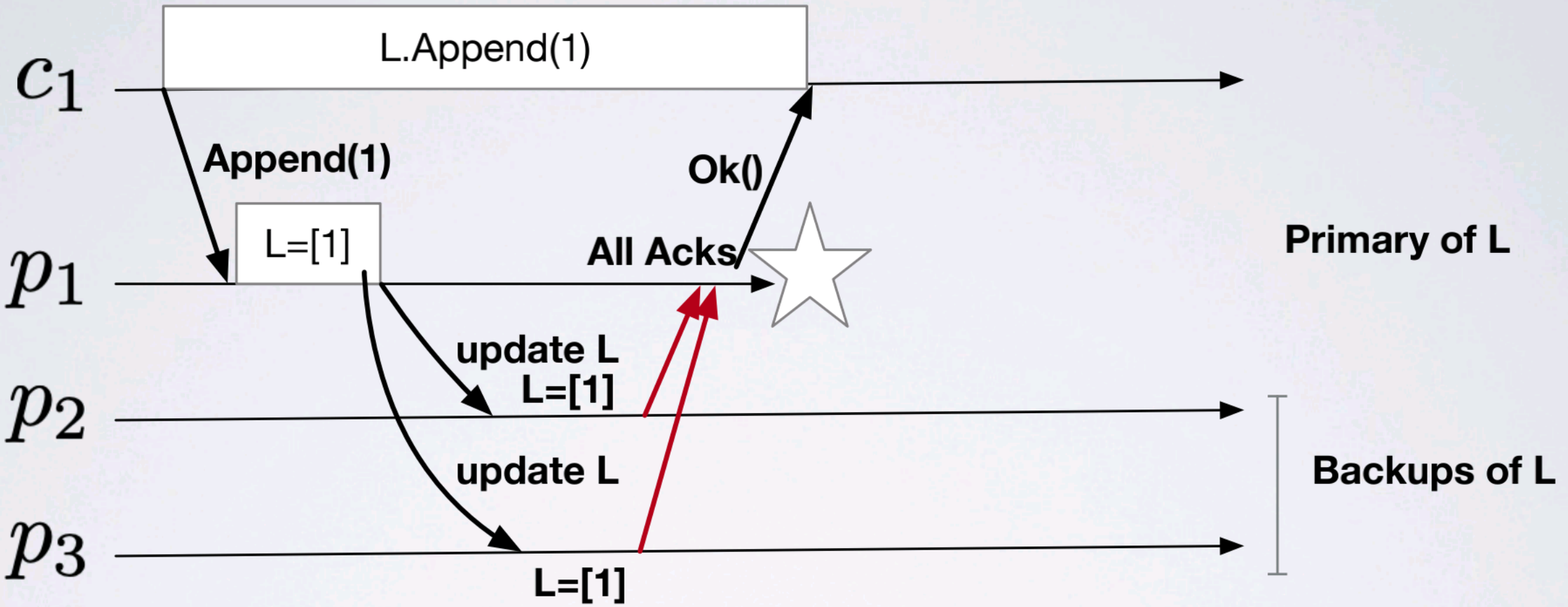


- Handling the crash of a Backup is easy.
- The primary waits until $Crashed \cup Acks = \Pi$

CASE 3: PRIMARY CRASH worst case

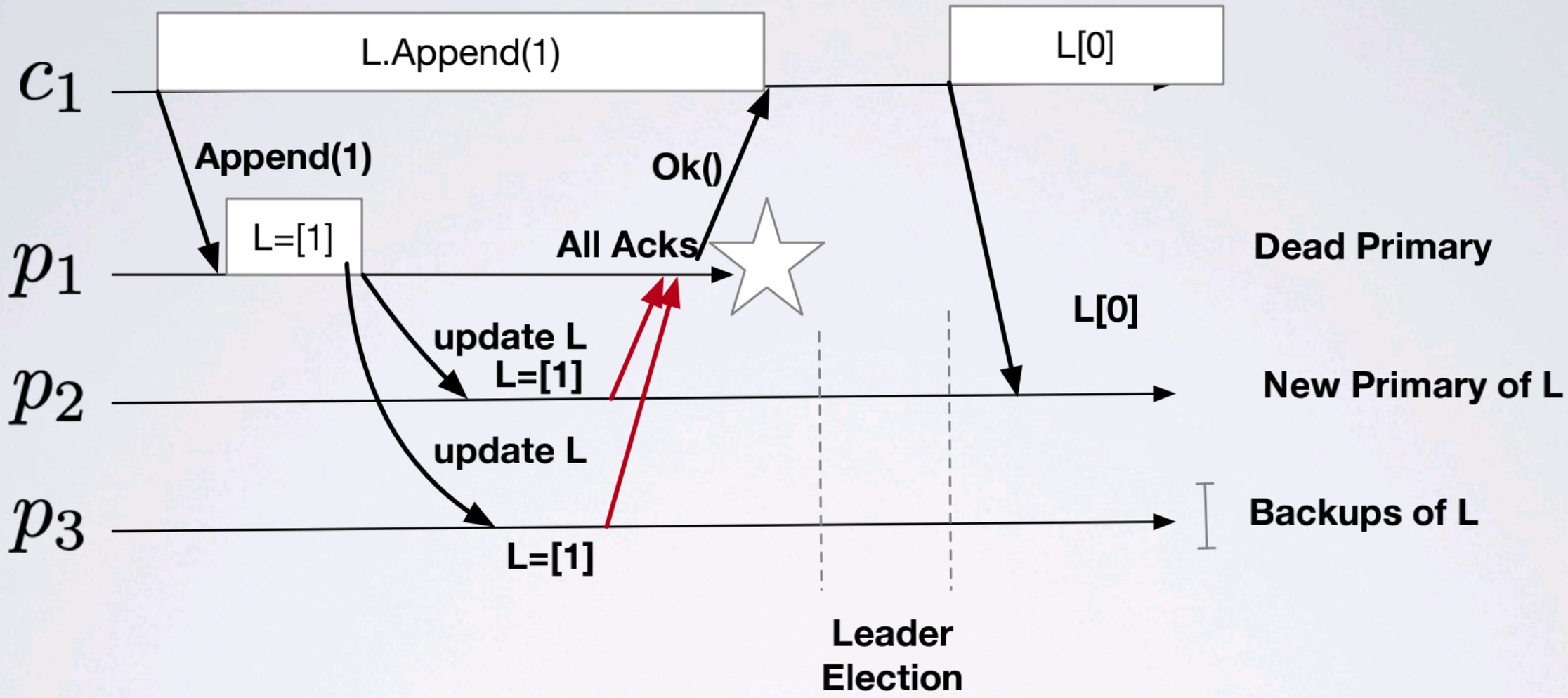
- The difficult case is when the **primary crashes**
- Three scenarios :
  - **Scenario 1**: Primary fails after the operation is completed and when is not handling any operation.
  - **Scenario 2**: Primary fails after its has received all the Acks and before the client receives the response. The client will not receive the response.
  - **Scenario 3**: Primary fails while sending update messages and before receiving all the *ack* messages.
- In all cases there is the need of **electing a new primary** (new leader).

SCENARIO 1



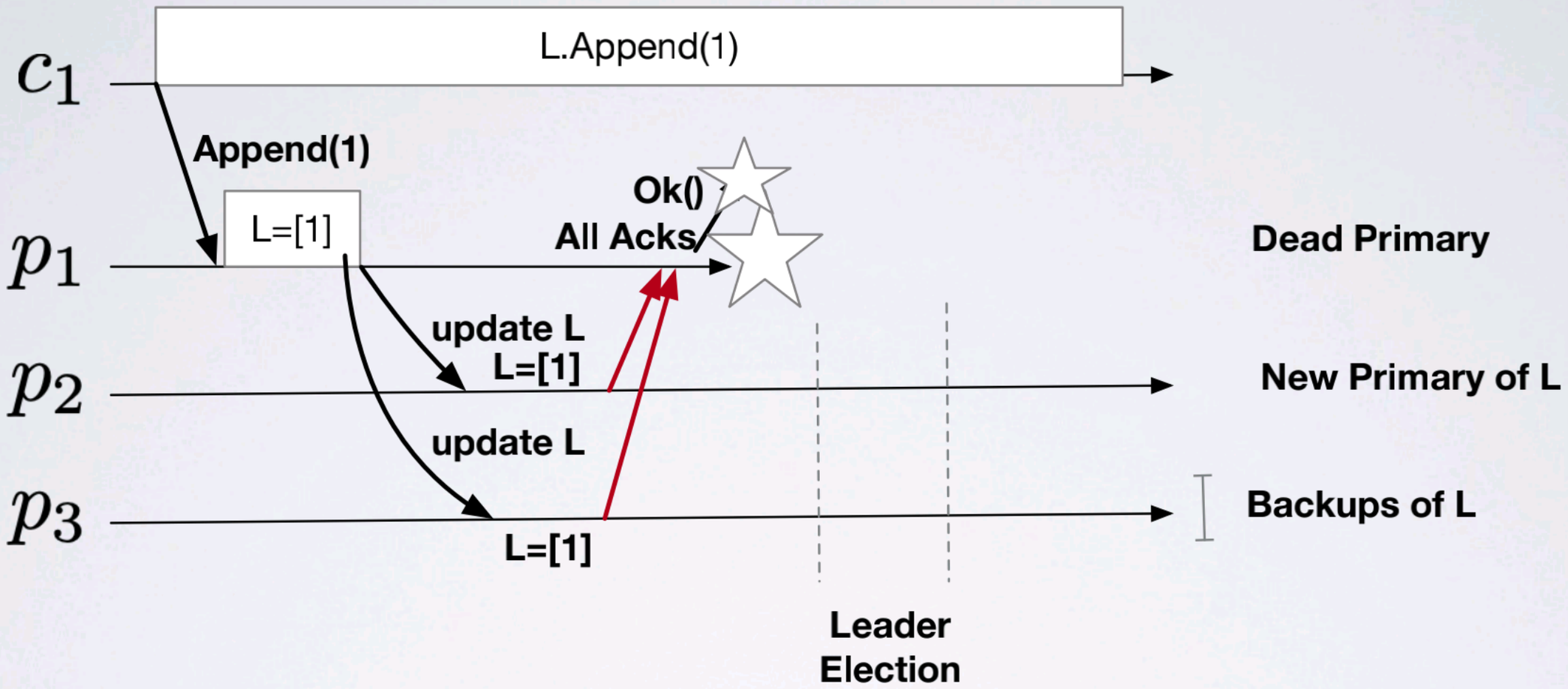
- **Scenario 1**: Primary fails after the operation is completed and when is not handling any operation (it fails after the response to client)

SCENARIO 1



- **Scenario 1**: Primary fails after the operation is completed and when is not handling any operation (it fails after the response to client).
- We elect a new primary using a LE algorithm. The new primary has the latest copy of the object and can answer to new requests

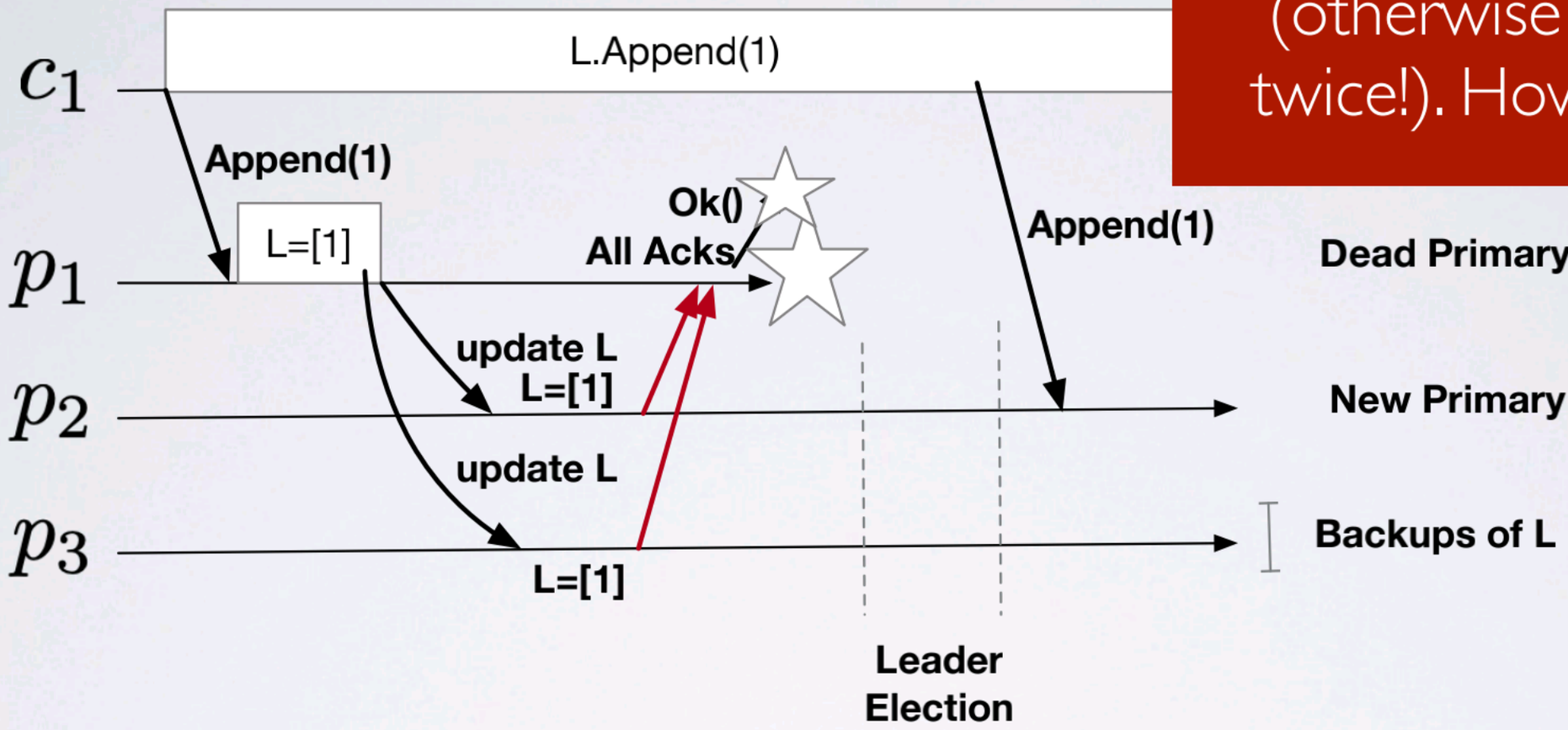
SCENARIO 2



- **Scenario 2**: Primary fails after its has received all the Acks and before the client receives the response. The client will not receive the response.

SCENARIO 2

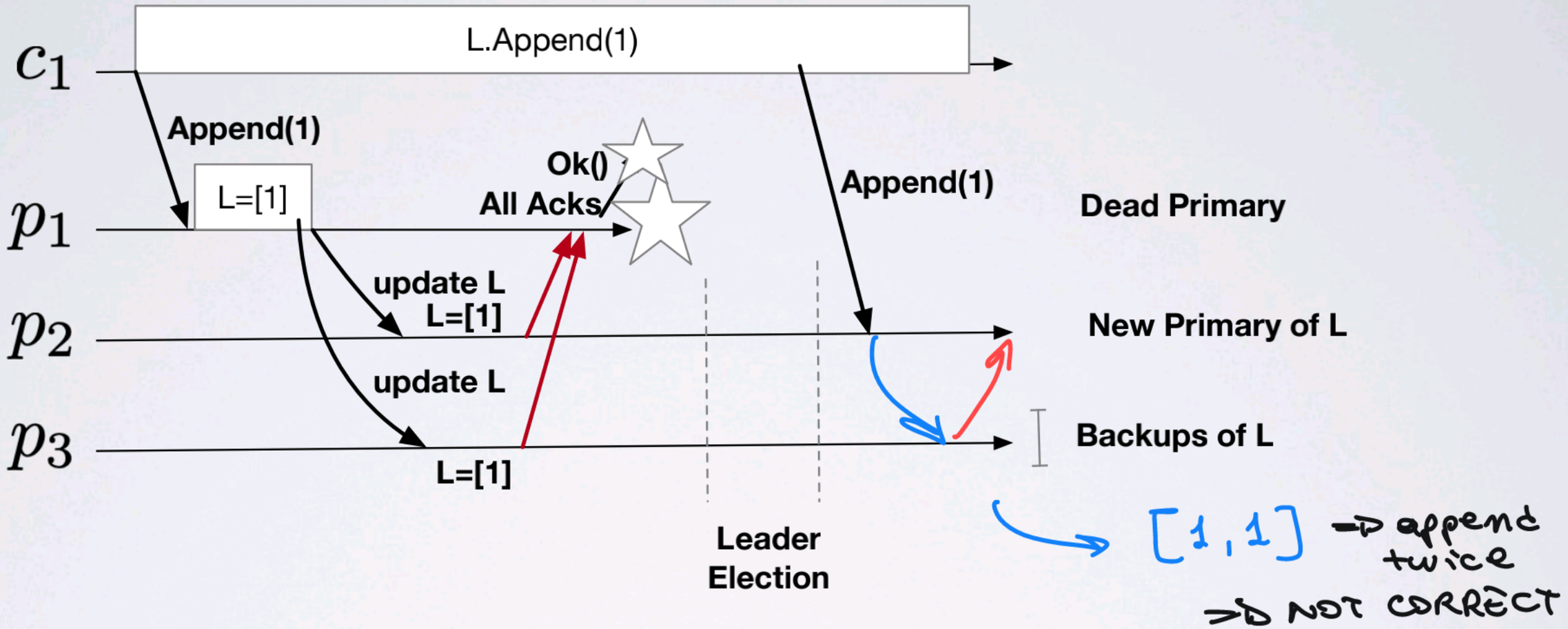
The new primary has to recognise that the request was already executed (otherwise we append twice!). How do we do?



- **Scenario 2**: Primary fails after its has received all the Acks and before the client receives the response. The client will not receive the response.
- The client timeouts and sends again the request to the new primary.

SCENARIO 2

using timestamps to avoid situation



- **Scenario 2**: Primary fails after its has received all the Acks and before the client receives the response. The client will not receive the response.
- The client timeouts and sends again the request to the new primary.
- The new primary will recognize the request re-issued by the client as already processed and sends back the result without updating the replicas. **REQUESTS HAVE TO BE UNIQUE (id_c,ts).**

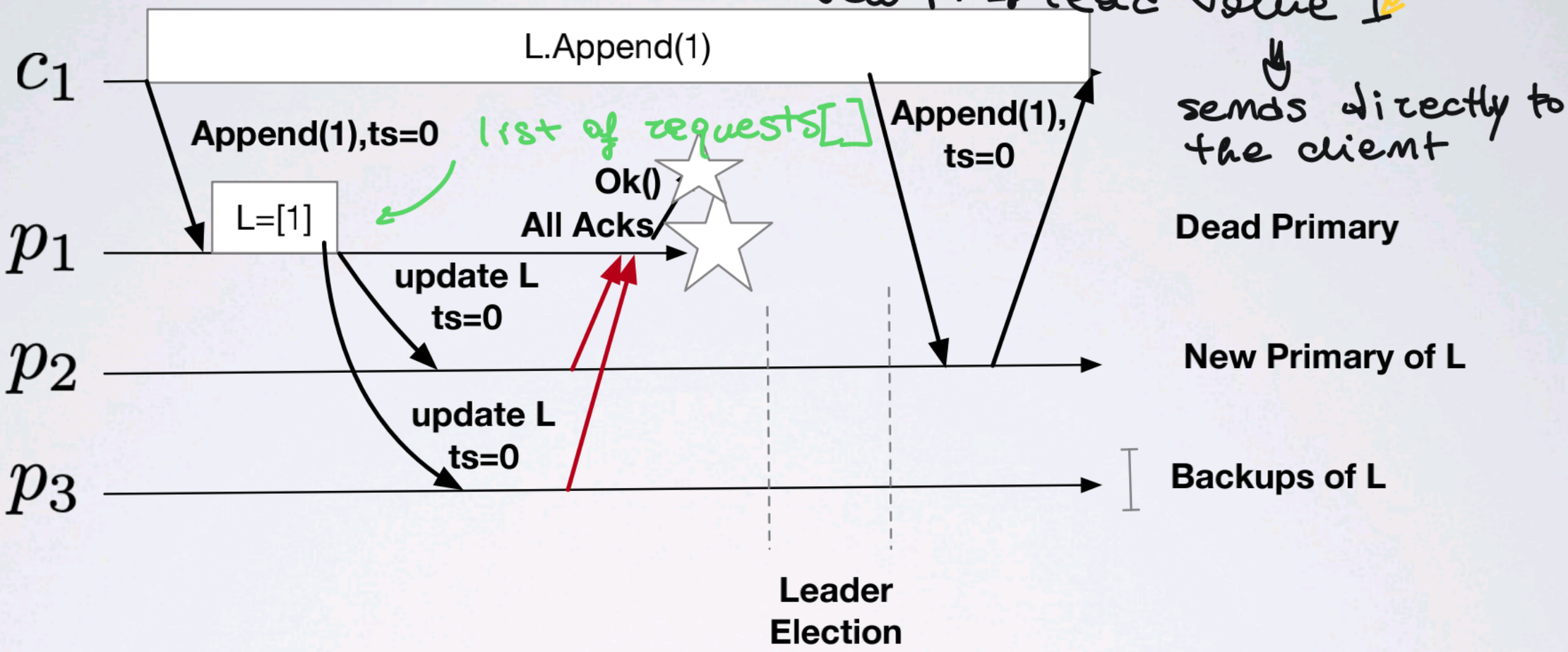
SCENARIO 2

if request ✗ ⇒ execute

Dead P. → (ts = 0, Append[1], ⊥)

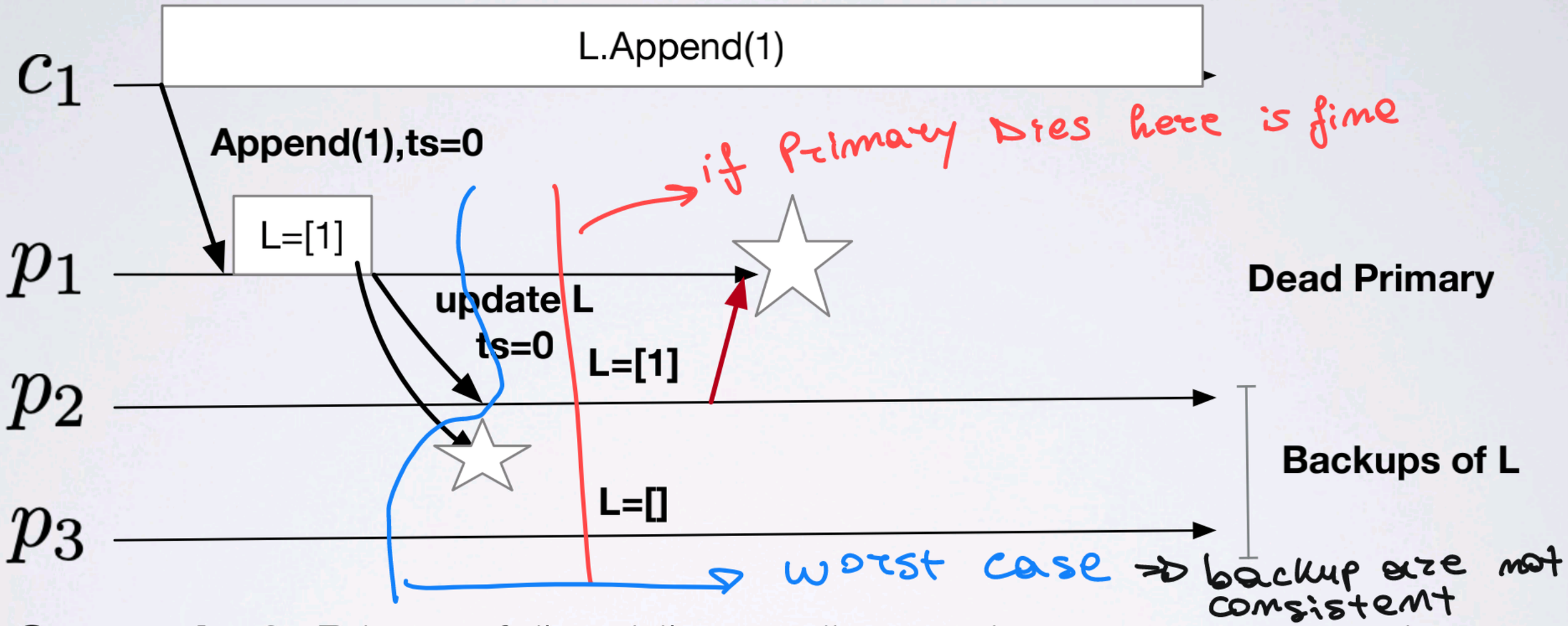
New P. → read value ⊥

sends directly to the client



- **Scenario 2**: Primary fails after its has received all the Acks and before the client receive the response. The client will not receive the response.
- The new primary will recognize the request re-issued by the client as already processed and sends back the result without updating the replicas. **REQUESTS HAVE TO BE UNIQUE (id_c,ts).**

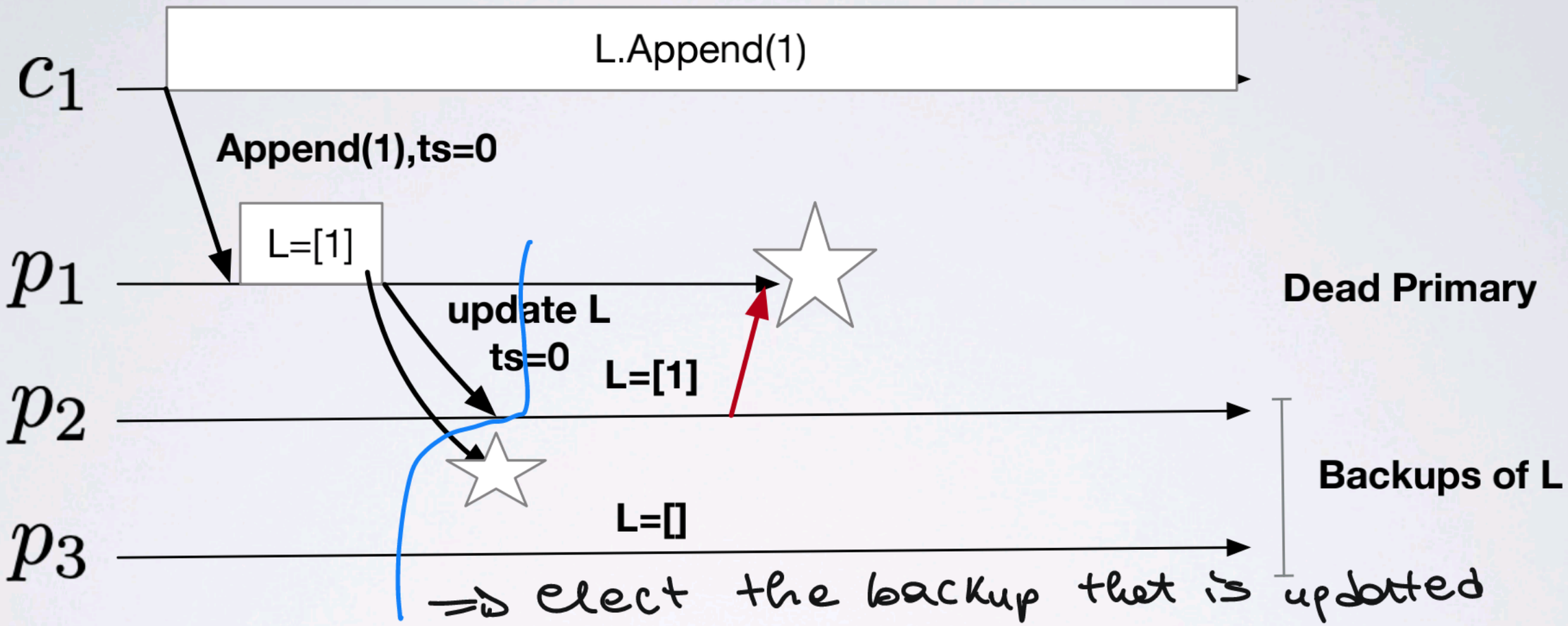
SCENARIO 3



- **Scenario 3**: Primary fails while sending update messages and before receiving all the *ack* messages. The problematic case is if the update is received by only some of the backups.

SCENARIO 3

new primary must get other backups at the same "level"

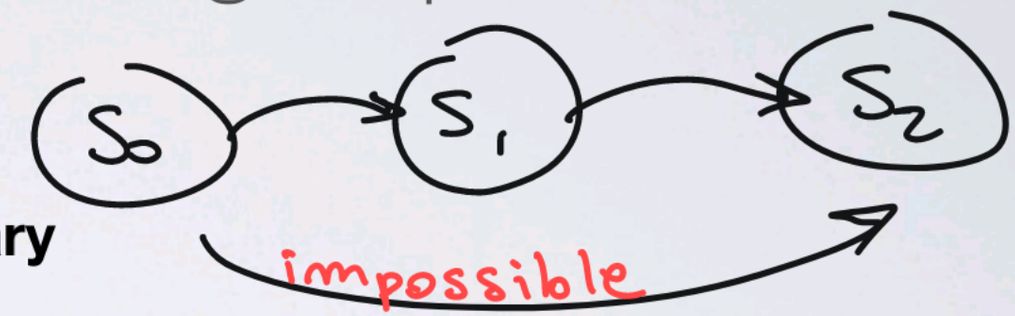


- **Scenario 3**: Primary fails while sending update messages and before receiving all the *ack* messages. The problematic case is if the update is received by only some of the backups.

We have to take the backup with the most updated state as primary.

SCENARIO 3

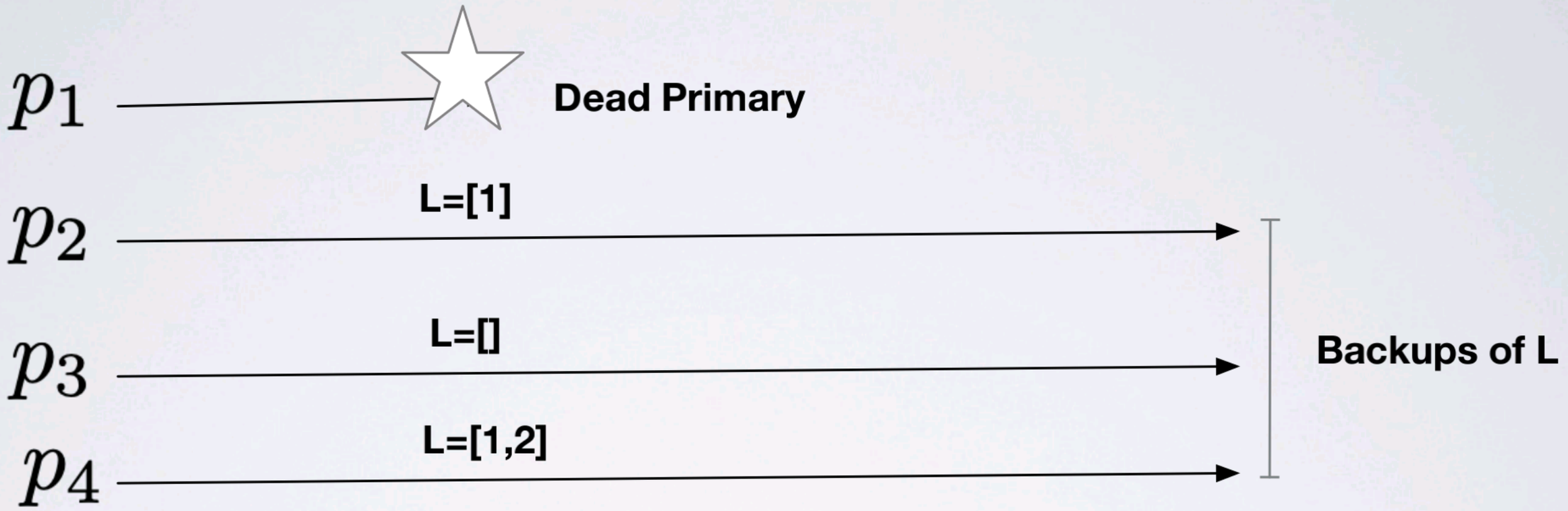
Assume only append and get operations



Possible?

SCENARIO 3

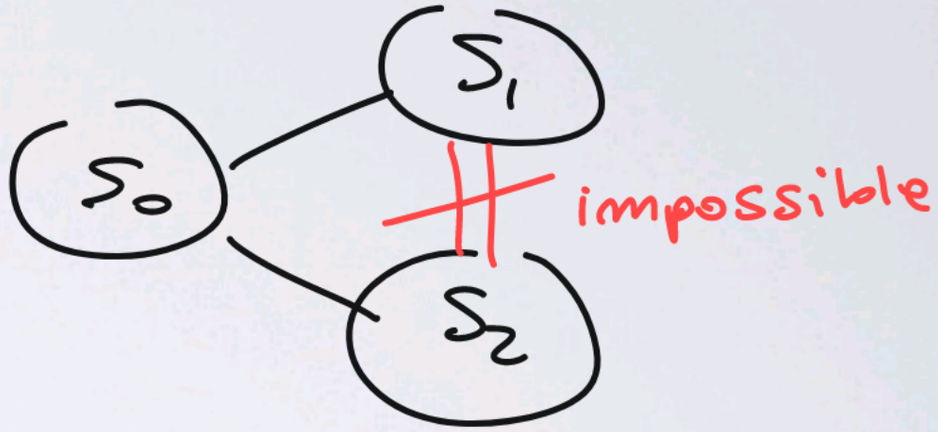
Assume only append and get operations



Possible? NO! This would implies that the primary executes the OP append(2) when P3 has not yet done append(1). This is impossible, a primary terminates an op only when all the backups updated.

SCENARIO 3

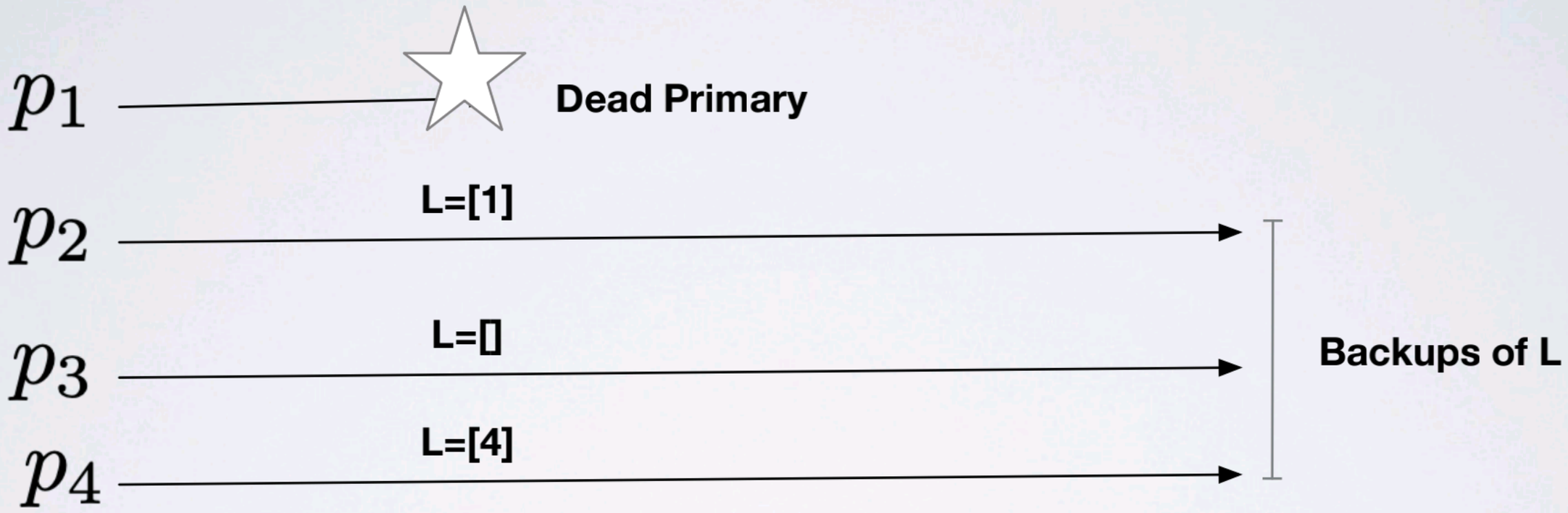
Assume only append and get operations



Possible?

SCENARIO 3

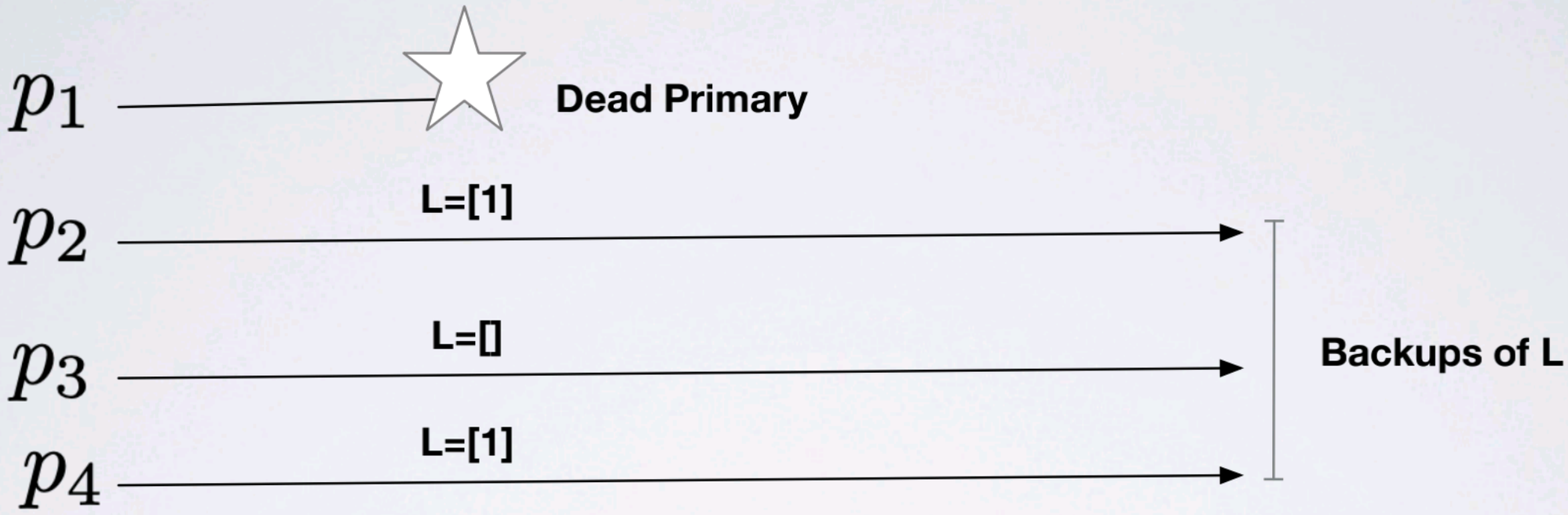
Assume only append and get operations



Possible? No! A primary terminates an op only when all replicas are updated. This inconsistency is not possible. It implies that the primary acts in a bizantine way.

SCENARIO 3

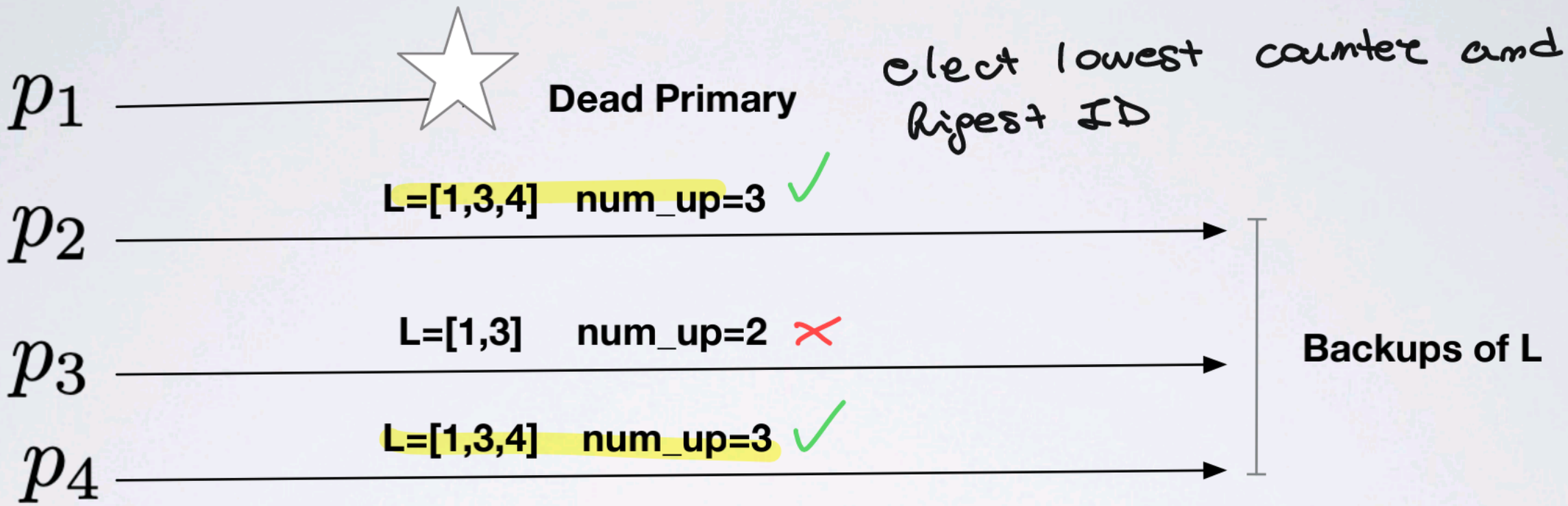
Assume only append and get operations



The only possibility is that we have a partition of the backups.
Last update seen vs Last update unseen.

SCENARIO 3

Assume only append and get operations



For general object each backup keeps a counter of the update seen this counter can only be different by 1.

ELECTING THE MOST UPDATED BACKUP

Algorithm 8 Primary- Backup Election Mechanism

upon event INIT
  $primary = \perp$
  $deadprimary = \perp$
  $Crashed = \emptyset$
  $update\_counter = 0$
  $ACKS = [\emptyset, \emptyset, \ldots, \emptyset]$ ▷ size equal to $|\Pi|$

upon event CRASH(p)
  $Crashed = Crashed \cup \{p\}$

upon event $primary \in Crashed$
  BebCast($< ELECTION, primary, myID, update\_counter >$)
  $primary = \perp$

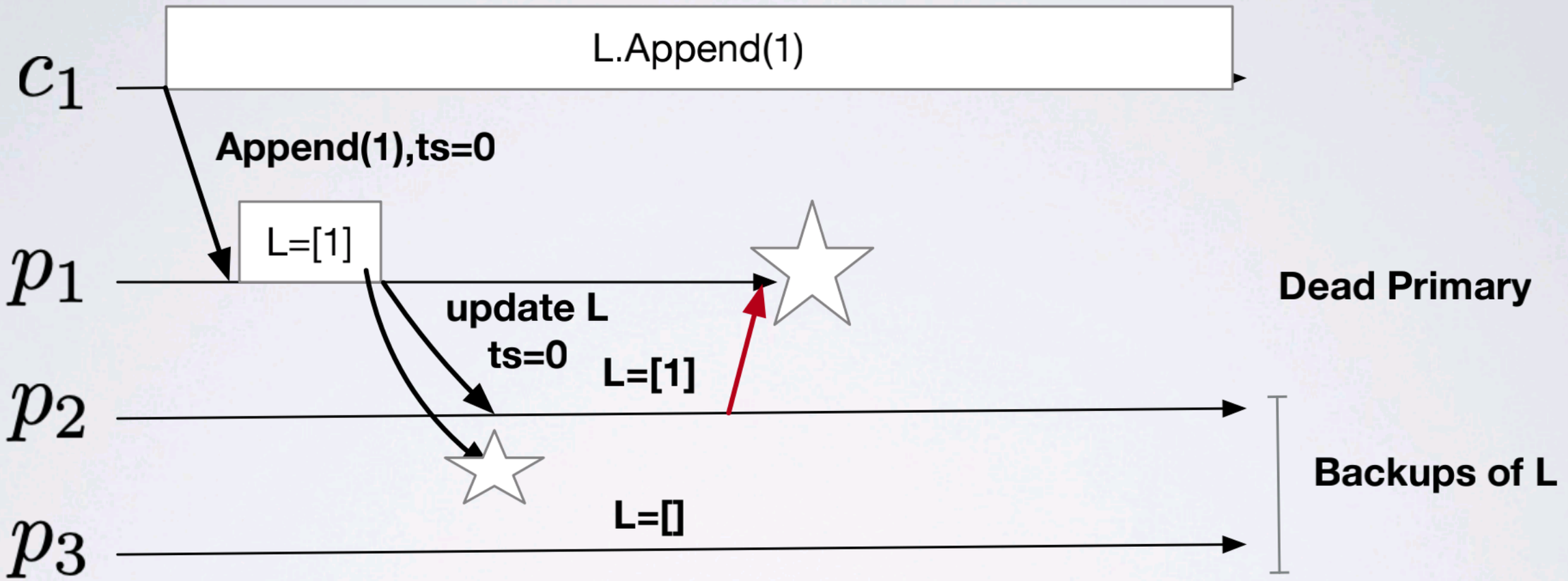
upon event BEB DELIVER($< ELECTON, IDDEADPRIMARY, ID, CNT >$ from p)
  if $primary == IDDEADPRIMARY$ **then**
    $Crashed = Crashed \cup \{primary\}$
    BebCast($< ELECTION, primary, myID, update\_counter >$)
    $primary = \perp$
  $ACKS[IDDEADPRIMARY] = ACKS \cup \{< ELECTON, IDDEADPRIMARY, ID, CNT\}$

upon event EXISTS j SUCH THAT $ACKS[j] \cup Crashed = \Pi$
  $primary$ = take the lowest ID with the highest counter in $ACKS[j]$
  $ACKS[j] = \emptyset$

[IDDEADPRIMARY]

every on the system knows that the primary is dead
⇩
we can terminate

SCENARIO 3



- **Scenario 3**: Primary fails while sending update messages and before receiving all the *ack* messages. The problematic case is if the update is received by only some of the backups.

We have to take the backup with the most updated state as primary.

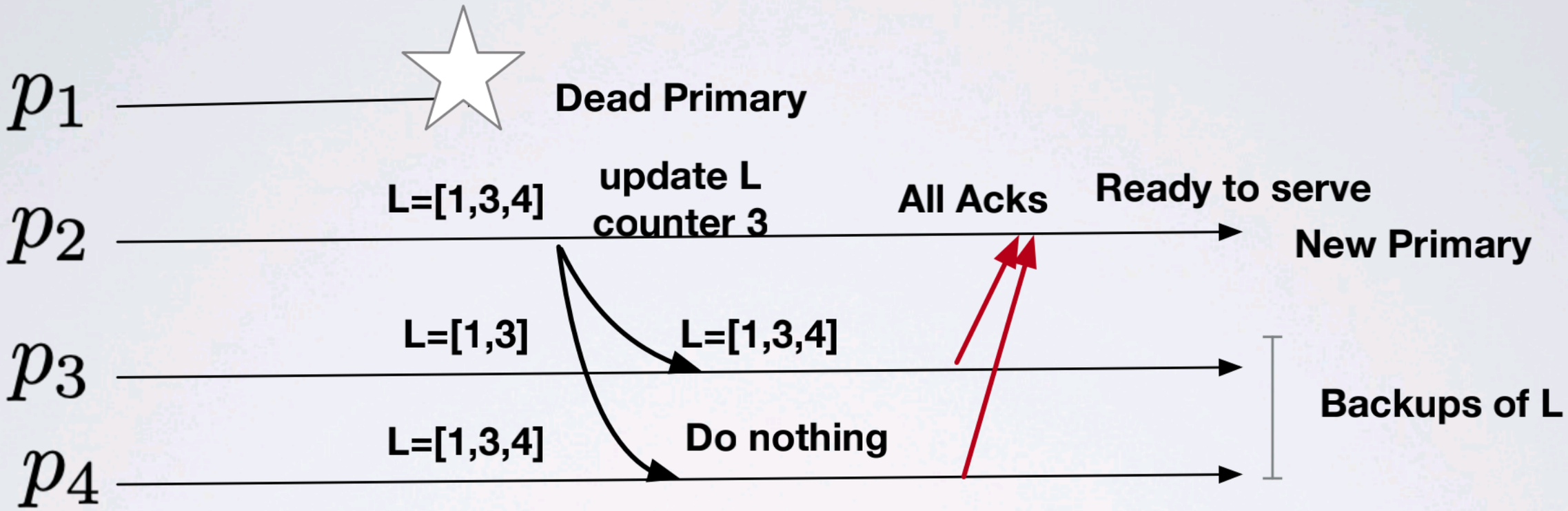
The most updated states is the one that has seen more updates.

SCENARIO 3

- **How to Guarantee atomicity?** the update must be received either by **all or by none** of the backups. (**You cannot escape from consensus**)
- When a primary fails there is the need to elect another primary among the correct replicas.
- This primary has to be the one with the most updated copy of the object
- The primary has the job of making the backups **on par** before answering new requests.

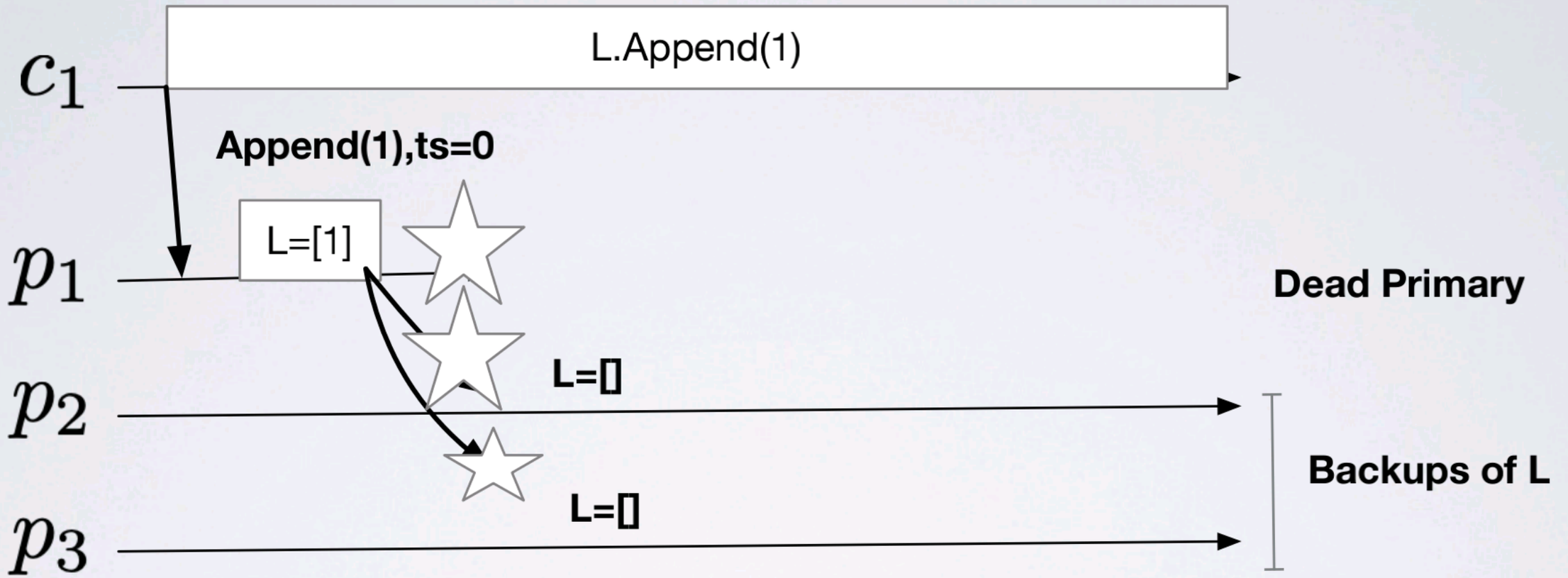
We have to update the state of the others otherwise we may lose the property on the operation counters.

SCENARIO 3 - MAKING BACKUPS ON PAR



- The new primary broadcast its state to everyone.
- If I receive a new state I update only if it has a greater counter
- Otherwise I do not update
- In any case I ack.

SCENARIO 3



- **Scenario 3**: Primary fails while sending update messages and before receiving all the *ack* messages. If no backups receive the OP is fine. We elect someone new, that will execute again the operation upon request.

PRIMARY BACKUP - PROS AND CONS

- Positive points:
  - Easy to understand: All operations passes through a primary that linearises operations.
  - Works even if execution is **non-deterministic** (Example, getRand(), currentTimeMillis(), any operation on an external oracle).
  - New backups can be added in a relatively easy way.
  - *Reads* can be satisfied locally (on the primary).
- Negative points:
  - Delivering state change can be costly, the client has to wait for a new primary.
  - Backups are just backups.
  - Scenario 3 (crash in the middle of the update) is complex to solve in eventually sync-systems. -> **RAFT solves it.**

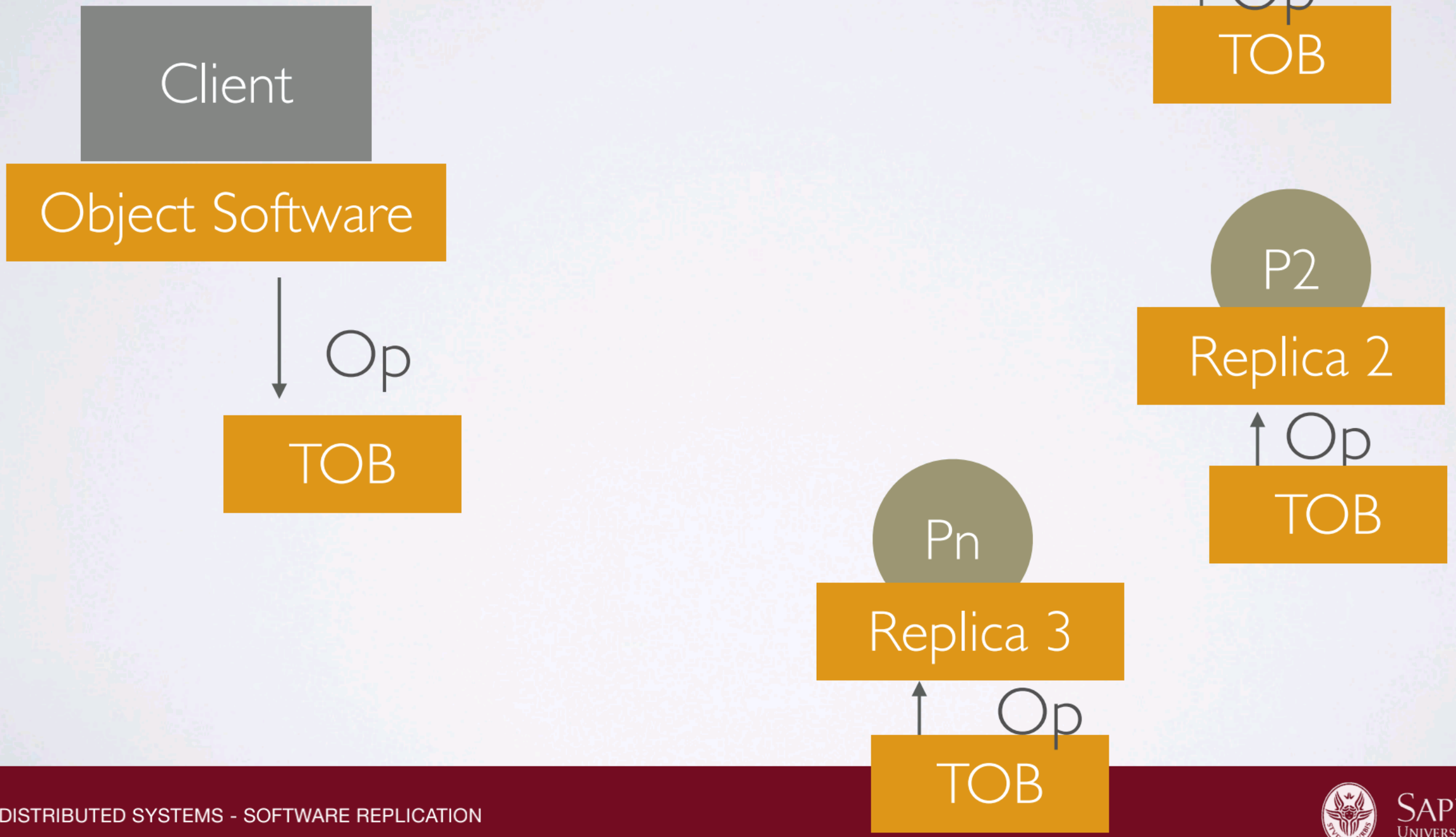
QUESTION

- Let us say that I give you the task of implementing a **generic distributed object O**, in an **asynchronous** systems with n=10 processes and f=1 failure. If you succeed in the task I will give you 100 dollars.
- Will you gain the money?

ACTIVE REPLICATION

- There is no coordinator
- All replicas have the same role
- Each replica is deterministic. If any replica starts from the same state and receives the same input, they will produce the same output
- The above implies that if each replica executes the same list of command, then the clients will receive the same response from all (non-faulty) replicas.

ACTIVE REPLICATION



ACTIVE REPLICATION

- To ensure linearizability we need to preserve:
  - **Atomicity**: if a replica executes an invocation, all correct replicas execute the same invocation.
  - **Ordering**: all correct replicas execute any two invocations in the same order.
  - **Blocking operation:** When I see the reply, my operation has been **committed.**
- We need: TOTAL ORDER Broadcast
  - **INCLUDING THE CLIENTS!**

ACTIVE REPLICATION: CRASH

- Active Replication does not need recovery actions upon the failure of a replica

ACTIVE REPLICATION - PROS AND CONS

- Pros:
  - Crashes are handled directly in the protocol.
  - Can be adapted to tolerate n/3-1 Byzantine replicas.
  - Clients do not have to wait if a crash happens.
- Cons:
  - It is hard to implement non-deterministic operations.
  - Total order broadcast is generally **expensive -** especially if we have 1000 clients.
  - **Reads cannot be local.** In primary backup a read can be satisfied directly by the primary. In active replication a read has to go through the Total Order Bcast.

REPLICATED GENERIC OBJECT

- A replicated generic object (deterministic) needs consensus to be implemented with linearisable semantic.
- Consensus is sufficient to implement any generic replicated object.
- Consensus is not necessary to implement all the replicated objects. As example we can implement

REFERENCES

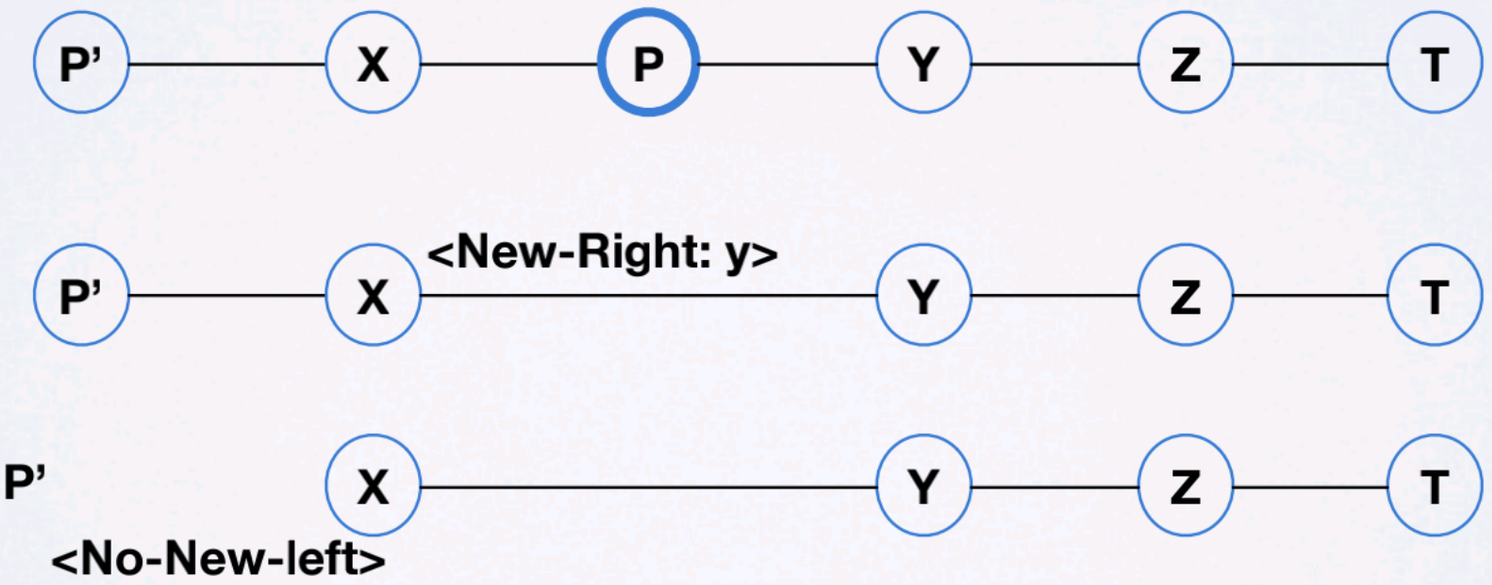
- Replication is not in the main book. The content of the slides is enough.
- Rachid Guerraoui and André Schiper: "***Fault-Tolerance by Replication in Distributed Systems***". In ***Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies*** (Ada-Europe '96).

EXERCISE

Consider processes arranged as a line, in a system where each one has a unique ID (see Figure). Processes are connected by perfect point2point links. Processes are not always correct, and when they fail, they fail by crashing. Processes can communicate only with neighbors by exchanging messages.

When a neighbor of a process fails a new neighbor is given: there is an oracle that notifies a process of a new link with a new-left or new-right neighbor. The only exception is if a process becomes an endpoint of the line (P' is an endpoint). In this case the event no-new-left (or no-new-right) is generated.

Suppose that the oracle does not shuffle the line, that is at each reconfiguration the relative order from left to right on the line is preserved (see figure) below. Create and write the pseudocode of an algorithm that implements a total order broadcast (TOBcast). A TOBcast is a regular reliable broadcast with the additional property that all correct processes deliver messages in the exact same order.



Delivers M and Then M'

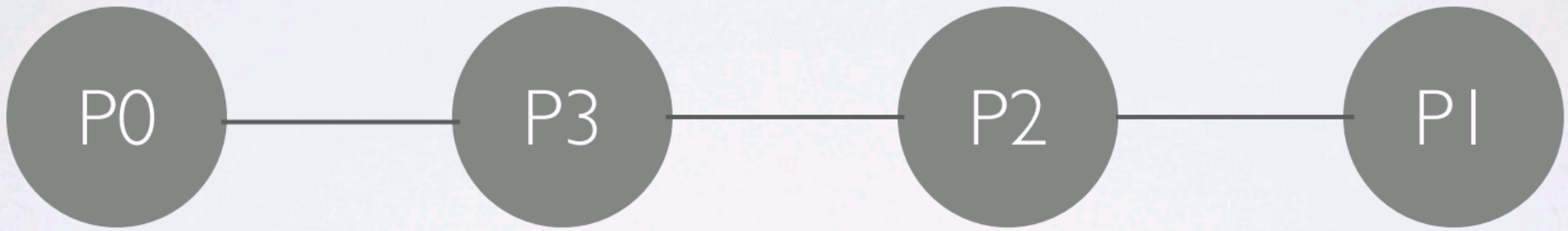
Delivers M and Then M'

Delivers M and Then M'

Delivers M and Then M'

M

M'



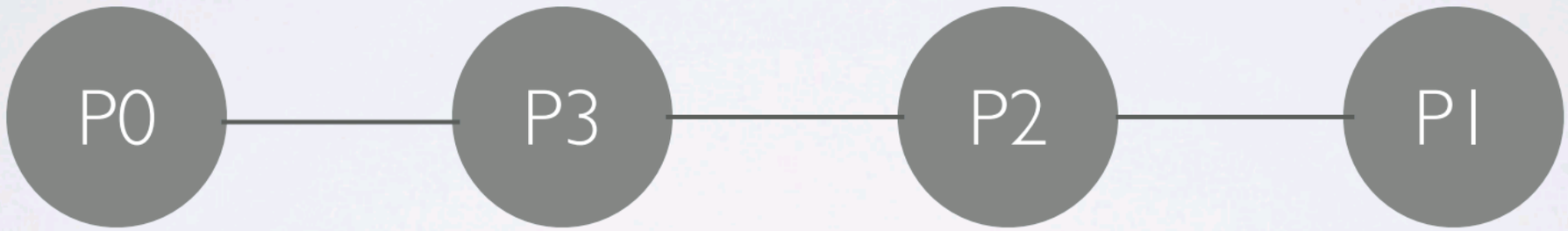
<Ping,[]>

<Ping,[M]>

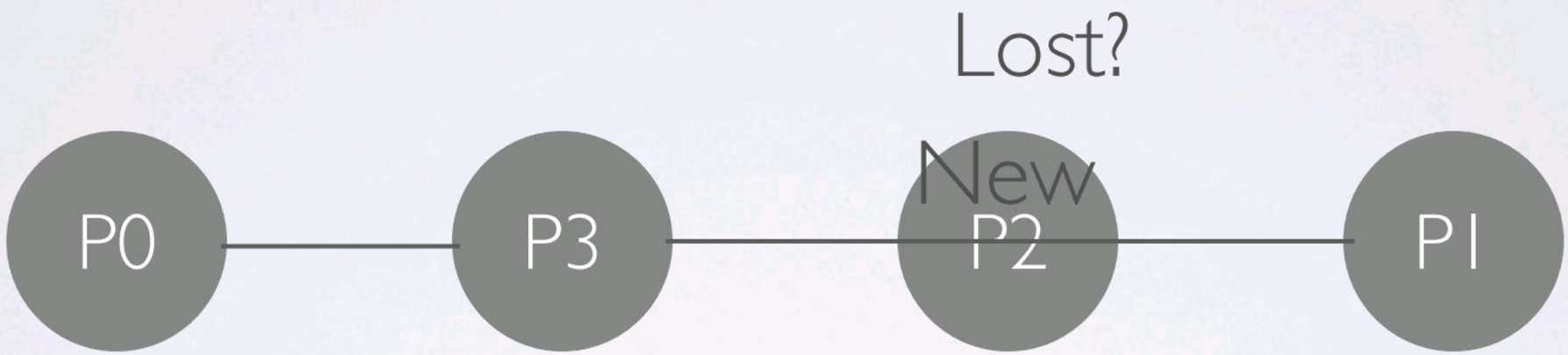
<Ping,[M,M']>

<Pong,[M,M']>

Ordered list



<Ping,[M]>



<Ping,[M]>
<Ping,[M]>

Delivers M and Then M'

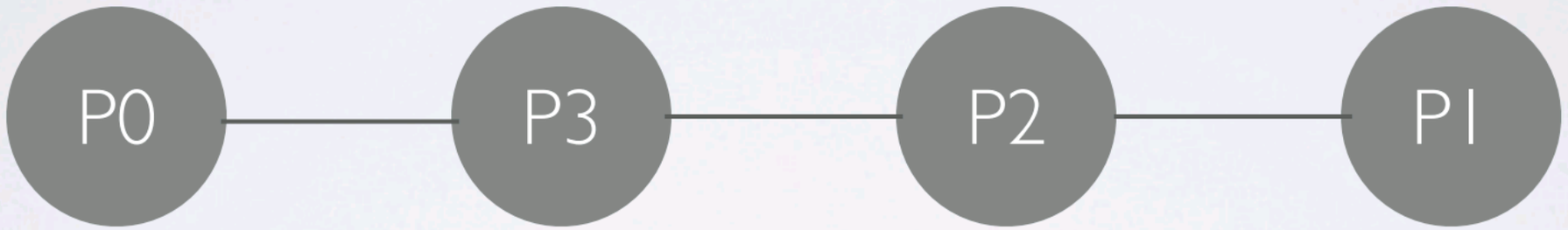
Delivers M and Then M'

Delivers M and Then M'

Delivers M and Then M'

M

M'



<Ping-1,[]> <Ping-1,[M]>

<Ping-1,[M,M']>

<Pong-1,[M,M']>

Ordered list

Algorithm 20 Ping-Pong Algorithm, process p_i

upon event INIT
- $left = P2pLink()$
- $right = P2pLink()$
- $head = False$ ▷ True on the leftmost process, trivial check $left = \perp$
- $tail = False$ ▷ True on the rightmost process, trivial check $right = \perp$
- $Pending = \emptyset$
- $last\_ping = \perp$
- $last\_pong = \perp$

upon event TO BROADCAST(m)
- $m =< p_i, m >$
- $Pending = Pending \cup \{m\}$

upon event DELIVERY FROM PERFECT LINK(MSG)
- **if** $MSG =< PING, sn, list > \wedge head = False \wedge sn > last\_ping.sn$ **then** ▷ assumes $\perp.sn == -1$
  - rcv_ping($PING, sn, list$)
- **else if** $MSG =< PONG, sn, list > \wedge tail = False \wedge sn > last\_pong.sn$ **then** ▷ assumes $\perp.sn == -1$
  - rcv_pong($PONG, sn, list$)

upon event $head \wedge last\_ping.sn = last\_pong.sn$ ▷ assumes $\perp.sn == -1$
- $last\_ping =< PING, last\_ping.sn + 1, [] >$
- rcv_ping($last\_ping$)

upon event $tail \wedge last\_ping.sn = last\_pong.sn + 1$
- $last\_pong =< PONG, last\_ping.sn + 1, last\_ping.list >$
- rcv_pong($last\_pong$)

upon event NEW LEFT NEIGHBOUR(new left p)
- $left = p$
- **if** $p = \perp$ **then**
  - $head = True$
- **else**
  - **if** $last\_pong \neq \perp$ **then**
    - sendLeft($last\_pong$)

upon event NEW RIGH NEIGHBOUR(new right p)
- $left = p$
- **if** $p = \perp$ **then**
  - $tail = True$
- **if** $last\_ping \neq \perp$ **then**
  - sendRight($last\_ping$)

procedure RCV_PING($PING, sn, list$)
- $list = list$+LIST($Pending$)
- $Pending = \emptyset$
- $last\_ping =< PING, sn, list >$
- sendRight($last\_ping$)

procedure RCV_PONG($PONG, sn, list$)
- **for all** $(source, message) \in list$ **do**
  - TRIGGER TO DELIVER($source, message$)
- $last\_pong = MSG$
- sendLeft($last\_pong$)