# Software Replication

# Motivation

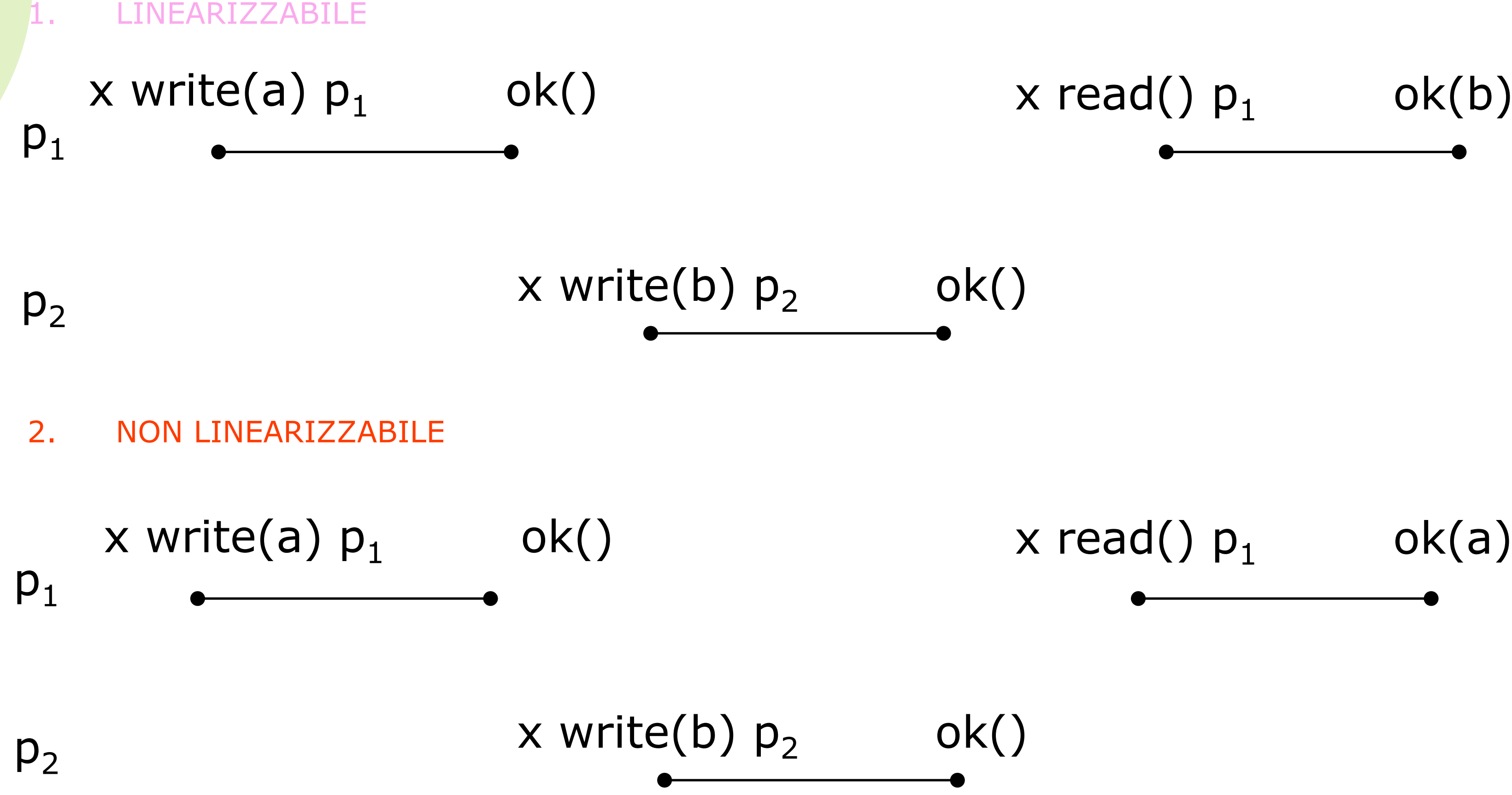
- Assure the availability of an object (service) despite failures
- Assuming p the failure probability of an object O. O's availability is 1-p.
- Replicating an object O on n nodes and assuming p the failure probability of each replica, O's availability is $1- p^{n}$

  (considering independent failures probability)

# Linearizability In a Shared Memory System

1. LINEARIZZABILE

2. NON LINEARIZZABILE



# Linearizability

- Liveness: every invocation has a matching response
- Safety: there exists a permutation P of all operations on O such that
  - P is legal wrt O (respects the semantic of the object)
  - If the response of O occurs before O’ in the run, then O precedes O’ in P (respects the real-time ordering of operations)

# Replication Techniques

- Due main techniques implementing linearizability:
  - Primary Backup
  - Active Replication

# Sistem Model

- Processes are either clients or replicas
- Clients and replicas exchanges messages through Perfect point-point links
- Processes can crash

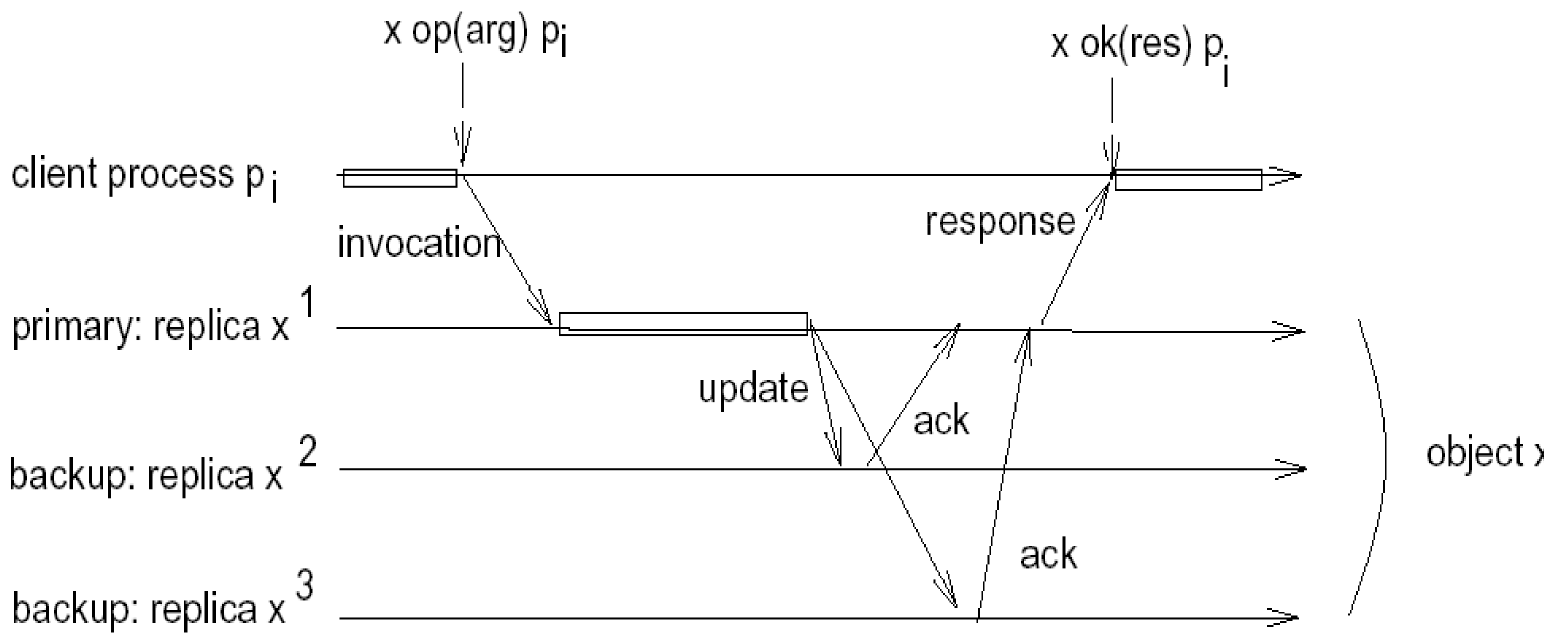
# Passive replication: Primary-Backup

# Primary Backup

- *Primary*:
  - Receives invocations from clients and sends back the answers.
  - Given an object x, *prim(*x*)* returns the primary of x.

- *Backup:*
  - Interacts with *prim(*x*)*
  - is used to guarantee fault tolerance by replacing a primary when crashes

# Primary Backup Scenario



# Primary Backup: the case of no crash

- When update messages are received by backups, they update their state and send back the ack to *prim(*x*)*.
- *prim(*x*) waits* for an ack message from each correct *backup and then sends back the answer*, *res*, to the client.
- **Guarantee Linearizability**: the order in which *prim(*x*)* receive clients' invocations define the order of the operation on the object.

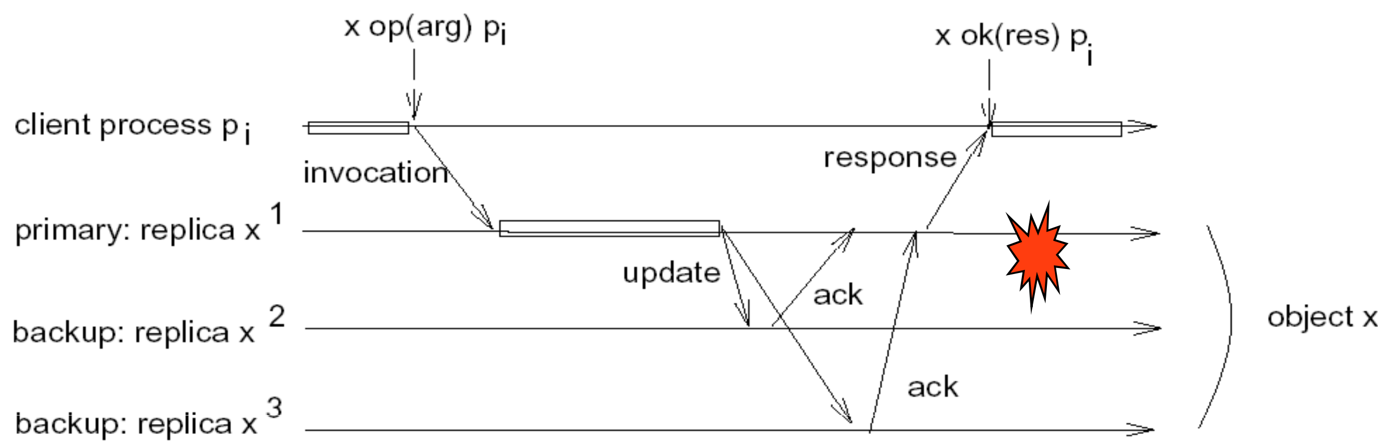
# Primary Backup: Presence of Crash

- Three scenarios *:*

1. **Scenario 1**: Primary fails after the client receives the answer.
2. **Scenario 2**: Primary fails before sending update messages
3. **Scenario 3**: Primary fails after sending update messages and before receiving all the ack messages.

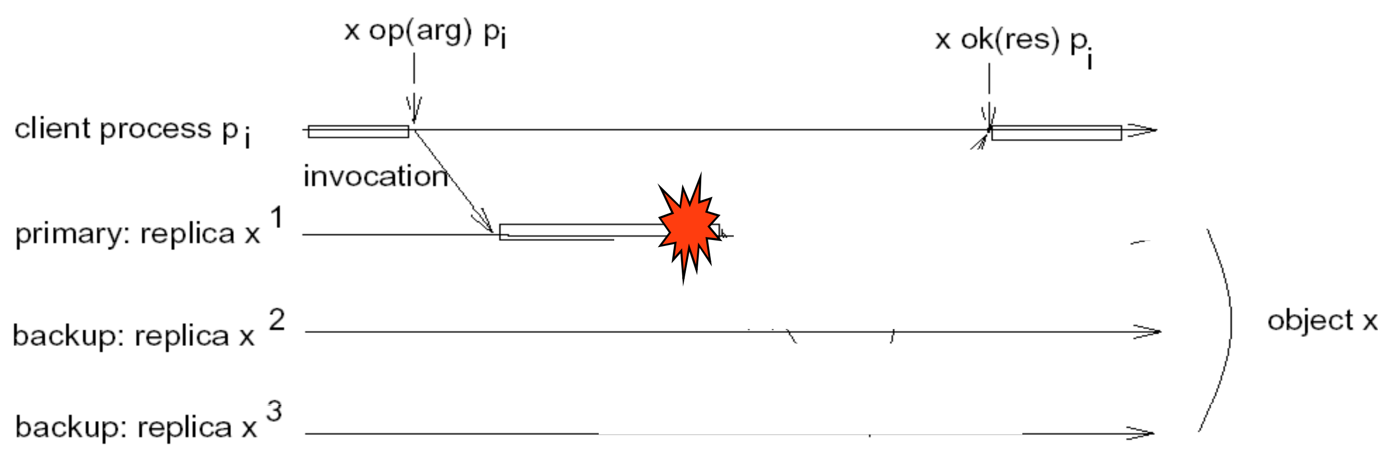
- In all cases there is the need of electing a new leader.

# Scenario 1



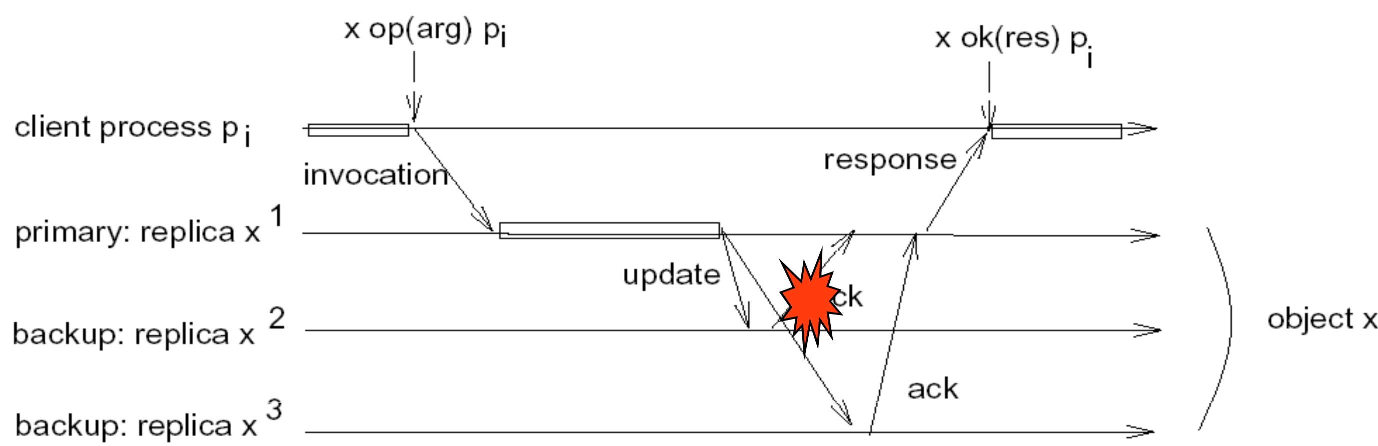

- Primary fails after the client receives the answer:
- Client could not receive the response due to perfect point-to-point link
- If the response is lost, client retransmits the request after a timeout
- The new primary will recognize the request re-issued by the client as already processed and sends back the result without updating the replicas

# Scenario 2




- Primary fails before sending update messages
- Client does not get an answer and resends the requests after a timeout
- The new primary will handle the request as new

# Scenario 3




- Primary fails after sending update messages and before receiving all the ack messages:
  - **Guarantee atomicity**: update it is received either by all or by no one.
- When a primary fails there is the need to elect another primary among the correct replicas

# Example of Primary Backup on Synchronous Systems

**USE**
Perfect Point-to-point link;
rb broadcast, leader election;
perfect failure detector;

**upon event** <Init> **do**
recovery:= false; primary:=p; state:=initial;
storage:= Ø; lastreq:=(-,-,-,-,initial);
ack_recovery:=Ø; correct:=Π;

**upon event** < pp2pdeliver | client, [request(arg,sn]>
**and not** recovery **do**
**If** (sn,c,arg) **in** storage
**then** result:=retrieve(sn,c,arg)from storage
**trigger** <pp2psend | c, [response(result)]>
**else** (result,new state):= **execute** request(arg)
**trigger** <rbBcast|[up(sn,c,arg,result,newstate)]>;
state:=newstate;
**create**($ack_set_{c,sn}$); $ack_set_{c,sn}$:= Ø;

**Upon event** <pp2pdeliver | replica, ack(c,sn)>
$ack_set_{c,sn}$:= $ack_set_{c,sn}$ U replica;

**Upon event** $ackset_{c,sn} \supseteq$ correct **do**
**store** (sn,c,arg,result,newstate) **into** storage;
**trigger** <pp2psend | c, [response(result)]>;

**Upon event** ack_recovery $\supseteq$ correct **do**
recovery:=false;

**Upon** <rbdeliver|[up(sn,c,arg,result,newstate)]> **do**
lastreq := (sn,c,arg,result,newstate);
state:=newstate;
**store** (sn,c,arg,result,newstate) **into** storage;
**trigger** <pp2psend | primary, [ack(sn,c)]>;

**upon event** <crash| p > **do**
correct:=correct-p;
**if** p=primary
**then trigger** <leaderelection> ;

**upon event** <leader| p> **do**
**if** p=myself
**then** recovery:= true;
**trigger** <rbbcast | [recovery(lastreq)]>;
*register the primary to a name server*;

**Upon** <rbdeliver | [recovery(lastreq)]> **do**
**if** state ≠ lastreq.state
**then** remove lastreq from storage;
state:=lastreq.state;
**trigger** <pp2psend | primary, [ack_rec]>;

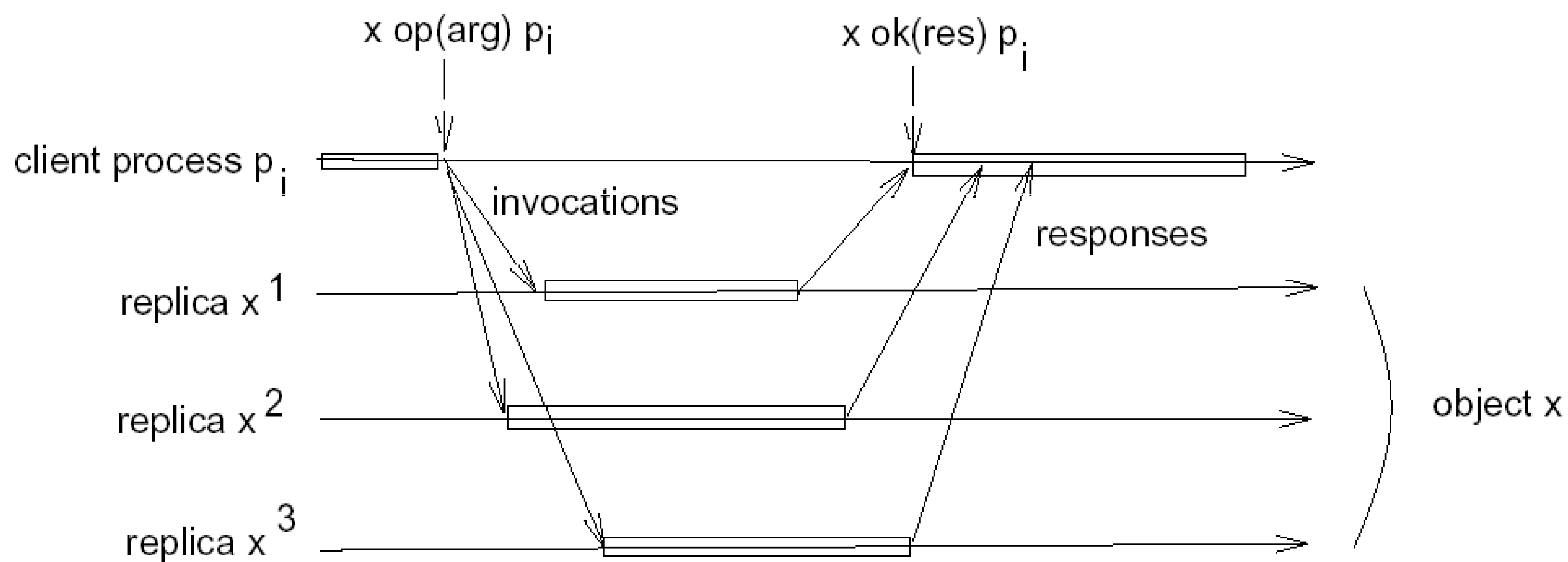
**Upon event** <pp2pdeliver | replica, ack_recovery>
ack_recovery:= ack_recovery U replica;

# Active Replication

# 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
- As a matter of fact clients will receive the same response one from each replica

# Active Replication



# Active Replication: Garantire Linearizability

- To ensure linearlizability we need to preserve:
  - Atomicity: if a replica executes an invocation, all correct replicas execute the same invocation.
  - Ordering: (at least) no two correct replicas have to execute two invocations in different order.

- We need: TOTAL ORDER Broadcast
  - INCLUDING THE CLIENTS!

# Active Replication:Crash

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