



Software Replication

Milani, Baldoni, Querzoni



Replication: why ?

- Guarantee the availability of a software object/service despite possible failures
- Crash: a process executes correctly its algorithm until it stops prematurely its execution.
- The object/service must be available with reasonable response times for a fraction of time t as close as possible to 100%
- Ex ample: Consider an object **O** replicated on n nodes whose failure probability is p (independent from the failure probability of other nodes).

Availability of O:

$$1 - p^n$$



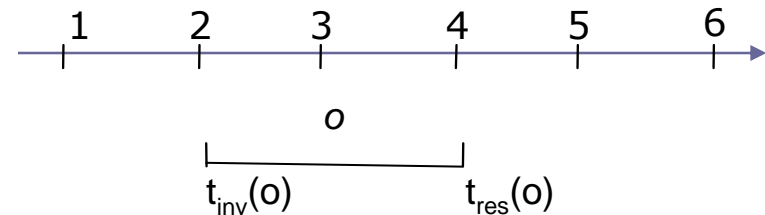
Logical Objects

- N sequential processes p_1, p_2, \dots, p_n interact through a set X of shared objects (files, variables, queues, ecc...)
- Each object $x \in X$ is characterized by a **state**
- Processes can access the object state through operations \bullet
- An operation \bullet executed by a process p_i on an object x is a couple invocation-response, $[x \bullet(\text{arg}) p_i] / [\text{ok}(\text{res}) p_i]$
- The set of operations that can be invoked on the object state define its semantic

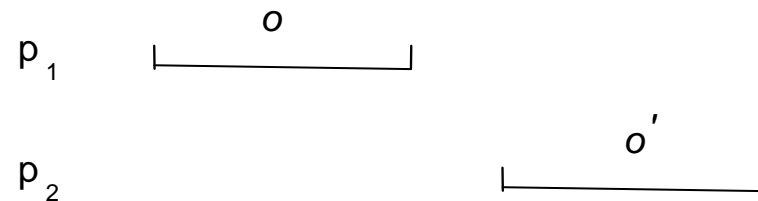
Temporal ordering of events

Notazione:

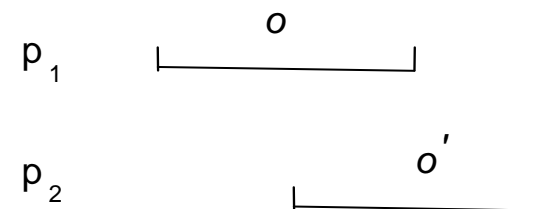
1. $t_{inv}(o)$ time at which p_i invokes o .
2. $t_{res}(o)$ time at which p_i receives the response



Two operations o and o' are **sequential**, defined as $o < o'$, if the response of o precedes the invocation of o' , i.e. $t_{res}(o) < t_{inv}(o')$



Two operations o and o' are **concurrent**, defined as $o || o'$, if $\neg(o < o')$ and $\neg(o' < o)$





A model for replication

- Every “logical object” x is implemented by a finite set $\{x^1, x^2 \dots x^m\}$ of physical copies, called “replicas”
- Each replica is placed on a distinct node inside the system
- Let s be the node hosting the i -th x^i replica of an object x . Then, s executes a sequential process, the *replica manager*, that manages every invocation for x^i
- x^i denotes both the i -th replica of object x and the corresponding replica manager.



Requirements

- **Transparency:**
 - The client process interacts directly with the logical object
 - The replication does not change:
 - the way a process invokes an operation on the object;
 - the way responses are delivered.
- **Consistency:**
 - Every operation must produce a result that respects the correctness specification of the logical object even if invoked on its replicas
 - The correctness specification depends on the semantics of the logical object.

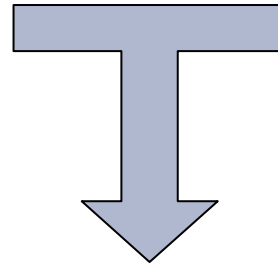
Linearizability

Sequential system:

- At each time instant there is a single process in the system that invokes an operation on the object
- The semantic of the operation is defined by pre/post conditions

Concurrent system:

- Various client processes concur to access an object
- A semantic must be defined for the various possible ordering of concurrent operations



LINEARIZABILITY

Defines the correctness of a concurrent object through a sequential specification



Linearizability

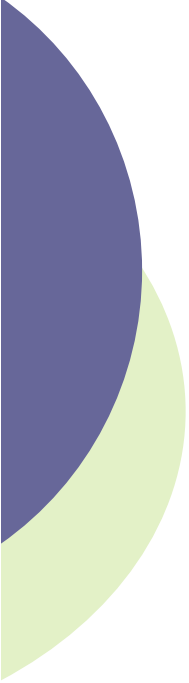
- Give to clients the illusion of interacting with the object in a sequential way even under consistency.
- The effect of an operation is the same that the a process would experience if the operation was executed in zero time
- The order among sequential operations must be preserved



Implementations of Linearizability: Software Replication Techniques



Linearizability implementations

- 
- A sufficient condition to guarantee linearizability: all replicas must agree on the set of invocations and on their order.
 - Formally:
 - **Order**: given two invocations on two replicas, both must execute them in the same order.
 - **Atomicity**: if a replica executes an operation, then all non faulty replica execute that operation.



Replication techniques

- Two fundamental techniques for linearizability implementation:
 - Primary Backup
 - Active Replication



System model

- Finite set of processes that communicate exchanging messages through a network infrastructure
- Processes are either clients or replicas
- Perfect point-point links:
 1. **Reliable delivery** – If a correct process p_i sends a message m to a correct process p_j , then p_j eventually delivers m .
 2. **No duplication** – No message is delivered by a process more than once.
 3. **No creation** – If a message m is delivered by a process p_j , then m has been previously sent to p_j by some process p_i
- Asynchronous system
- Processes may fail by crashing
- A process that crashes can recover (crash-recovery model)



Passive Replication: Primary-Backup

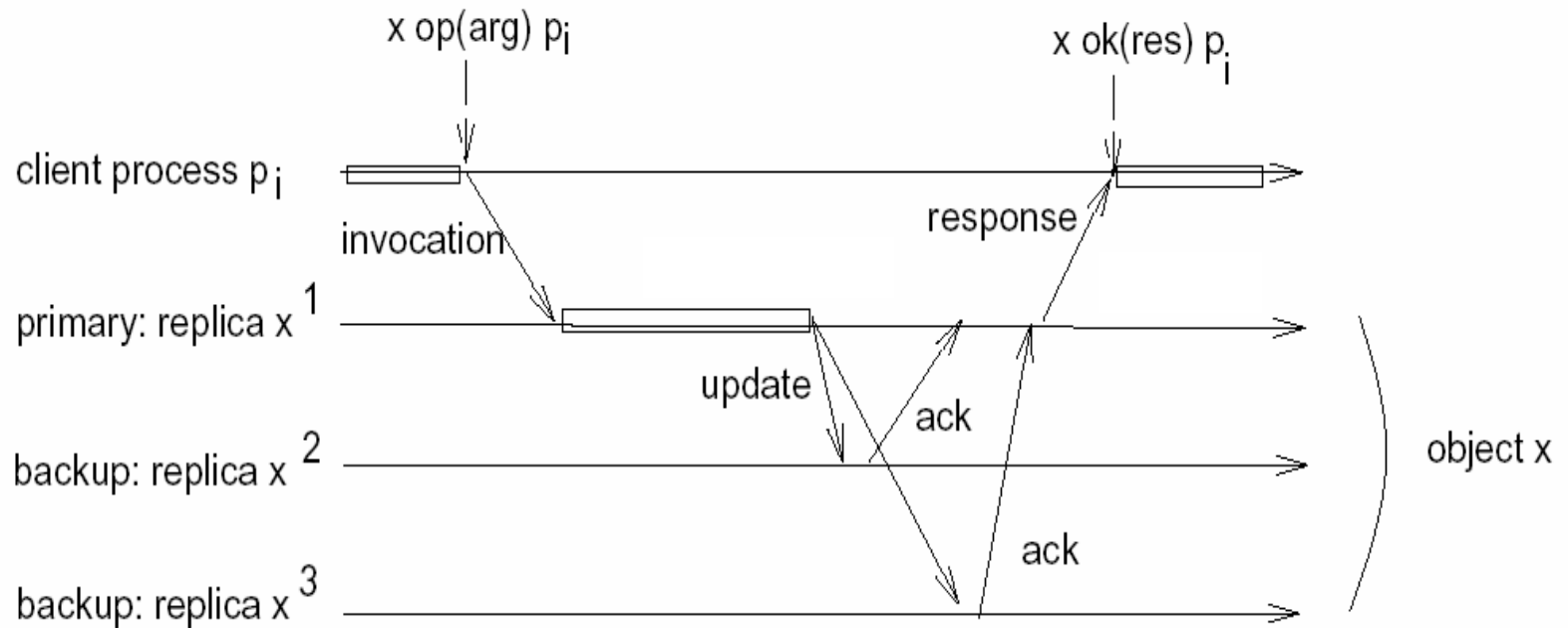




Primary Backup

- Replicas have two different roles:
 - *Primary*:
 - Receives invocations from client processes and sends back responses.
 - Given an object x , the primary of x is denoted $prim(x)$.
 - *Backup*:
 - Interacts only with $prim(x)$
 - Are used to guarantee fault tolerance

A possible scenario





Primary Backup: no crashes

- Client p_i invokes $o(\text{arg})$ on x .
- $\text{prim}(x)$ receives the invocation and executes the operation.
- After the execution $\text{prim}(x)$ updates its local state and prepare the response res .
- $\text{prim}(x)$ sends update messages (invId , res , state-update) to all *backups*:
 - **invId**: invocation identifiers
 - **res**: response for the client
 - **state-update**: state of the primary after the execution of the operation.



Primary Backup: no crashes

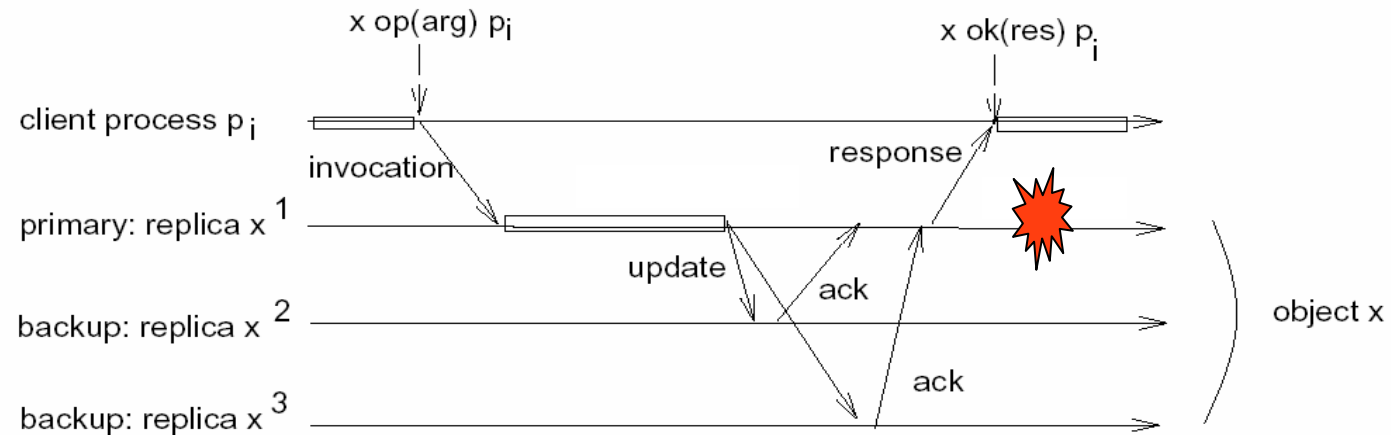
- When *backups* receive the update message, they update their local state and send an *ack* message to *prim(x)*.
- *prim(x)* waits for *acks* from all backups and then sends the response *res* to the client process.
- **Linearizability**: the order used to execute operations is decided by *prim(x)*.



Primary Backup:crash

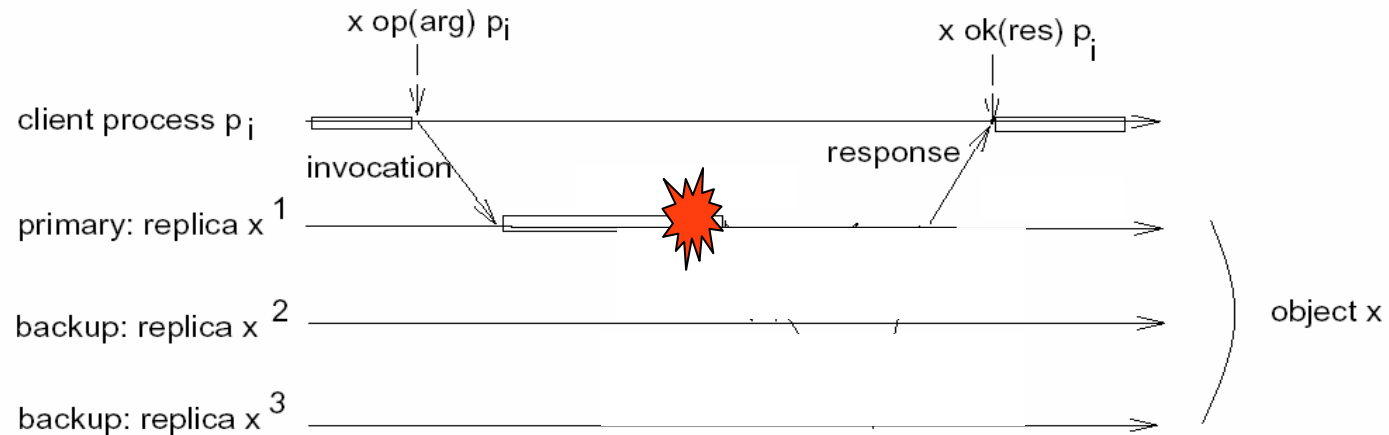
- Three possible failures of the *primary*:
 1. **Scenario 1**: it fails after the client has received the response
 2. **Scenario 2**: it fails before sending the state update messages
 3. **Scenario 3**: it fails after sending the state update messages but before sending the response to the client.
- In all these cases a new primary must be elected: *leader election*

Scenario 1



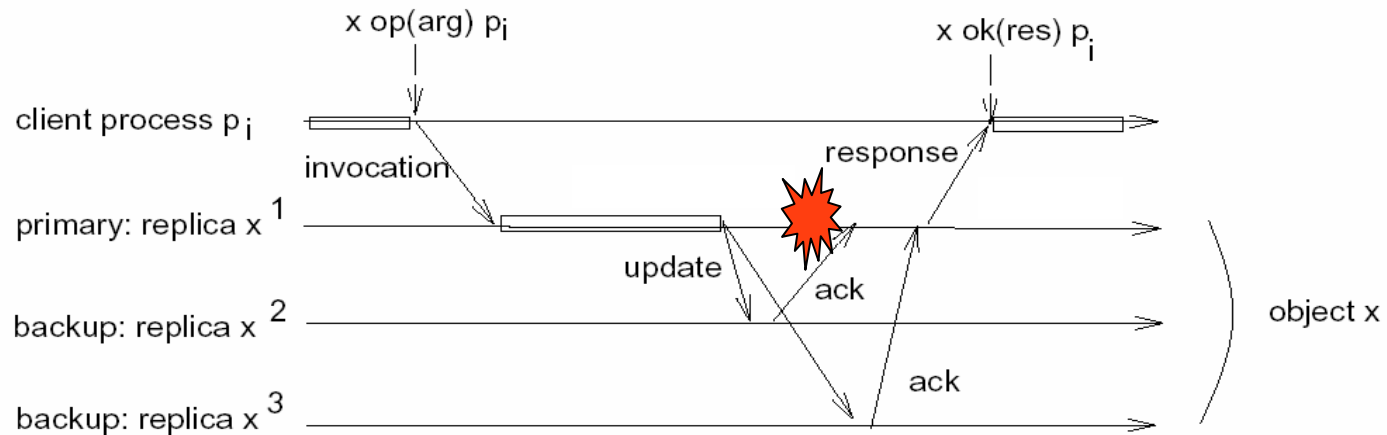
- Failure after the client has received the response:
 - The client is unaware of the failure

Scenario 2



- It fails before sending the state update messages
 - The client does not receive a response, suspects the current primary, and send again the invocation
 - The new primary manages the invocation as a new one

Scenario 3



- it fails after sending the state update messages but before sending the response to the client (before the acks):
 - **Atomicity**: the update is received by all or none of the backup processes.

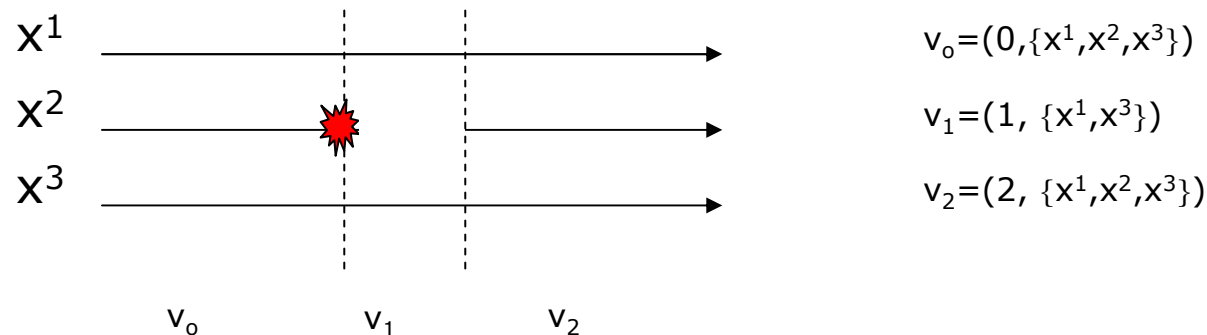


Scenario 3: possible variants

- **No** backups received the update:
 - Same as **Scenario 2**
- **All** backups received the update, i.e. the operation has been executed but the client received no response:
 - The state of backup processes is updated
 - The client invokes again the same op
 - The new primary uses the information (invId,res) to understand if the invocation has been executed and then send the response to the client.

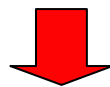
Primary Backup: dynamic groups

- The set of replicas can change during time due to crashes and recoveries.
- **Grup M**: set of replicas that manage the object x .
- **Dynamic groups**: the group composition varies during the system lifetime:
 - When a replica crashes it is removed from the group;
 - When a replica recovers it is added to the group.
- **View v_i** : it is a tuple $v_i = (i, M_i)$ that models the evolution of the group composition, where i is the view identifier and M_i is the corresponding group
- $V_0 = (0, M_0)$ is the initial view



Primary-Backup: Leader Election

- When the primary crashes a new one must be elected among the correct ones: **Leader election**
- Assume that a rule R exists to order the replicas within a view.
 - E.g. R=increasing identifier order:
 - x^1, x^2, x^4 **OK**
 - x^2, x^1, x^4 **ERROR**
- The new primary is the first replica in the **current view** according to R.
 - Primary selection:
 - $v_0 = (0, \{x^1, x^2, x^4\})$, primary x^1
 - $v_1 = (1, \{x^2, x^4\})$, primary x^2
- **Correctness: all the processes must elect the same primary starting from the same set of replicas**



ORDERED DELIVERY OF VIEWS

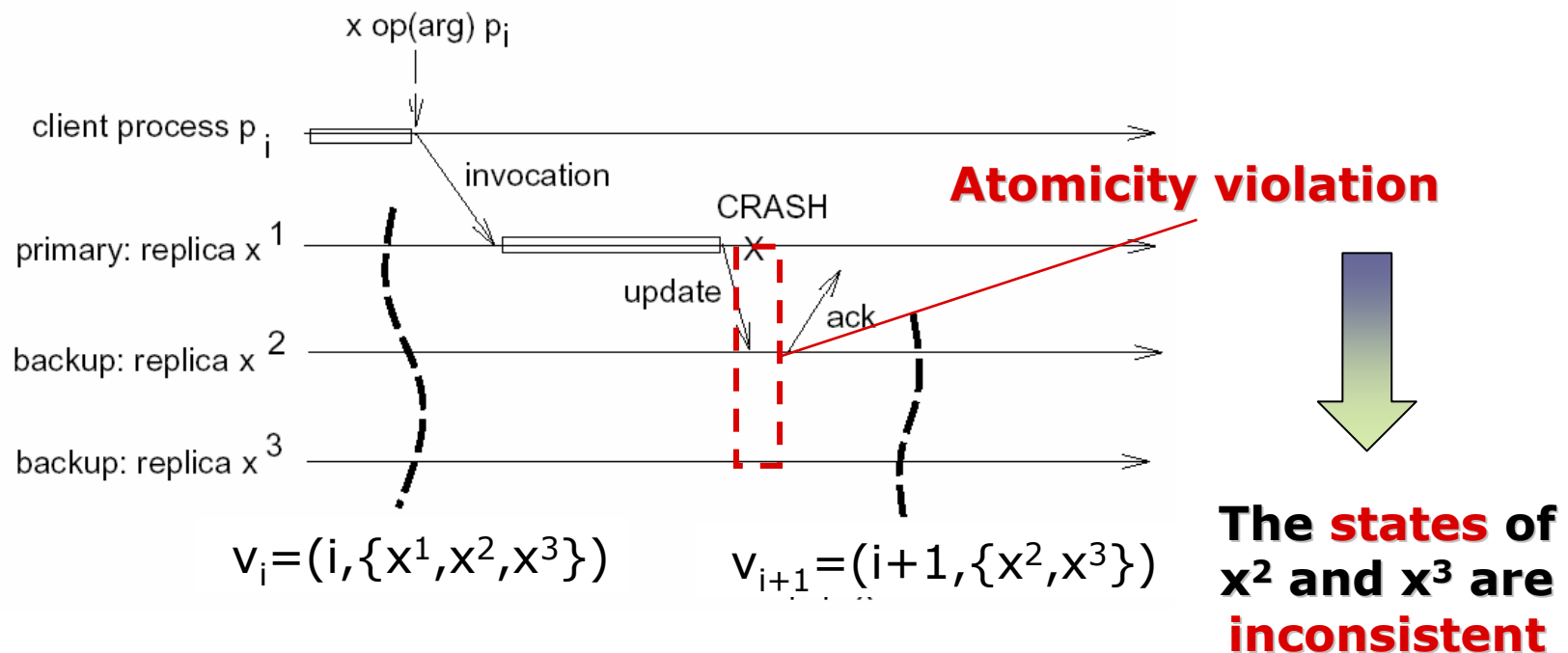


Ordered View Delivery: Specification

- The view ordering must respect the the following rules:
 - If a process $p \in g_x$ fails a new view that does not contain p is defined
 - If a process p recovers and become again part of g_x a new view that contains p is defined
 - Given a view $v_i(g_x)$ such that a process $p \in v_i(g_x)$, then either p delivers $v_i(g_x)$ or $\exists k > 0$ t.c. $p \notin v_{i+k}(g_x)$.
 - $\forall p, q \in g_x$, if p and q deliver $v_i(g_x)$ and $v_j(g_x)$ where $i \neq j$, they deliver the views in the same order
- **NOTE:** the fact that a replica is removed from a view because it crashed or because it is erroneously suspected is **IRRELEVANT**.

Primary-Backup: Correctness

- Is it sufficient to guarantee the ordered delivery of views to guarantee correctness? No !



- The correctness of the primary-backup implementation is based on the usage of a new primitive: VIEW SYNCHRONOUS MULTICAST



View-Synchronous Multicast: idea

- It extends the semantic of reliable multicast taking into account possible changes to the view
 - Let t be the instant of time when a process p delivers view $v_i = (i, M_i)$.
 - Every message m sent by p before the delivery of v_{i+1} :
 - Includes the identifier i
 - Will be sent to all processes in M_i
- Guarantees:
 - Atomicity: every replica in $v_i \cap v_{i+1}$ delivers m before delivering v_{i+1}
 - the delivery of a new view when a process crashes/recovers
 - Ordered view delivery



View-Synchronous Multicast: Specification

Assumption: $v_0 = (0, M_0)$ is the initial view for every process in M_0

Def. View-synchronous multicast

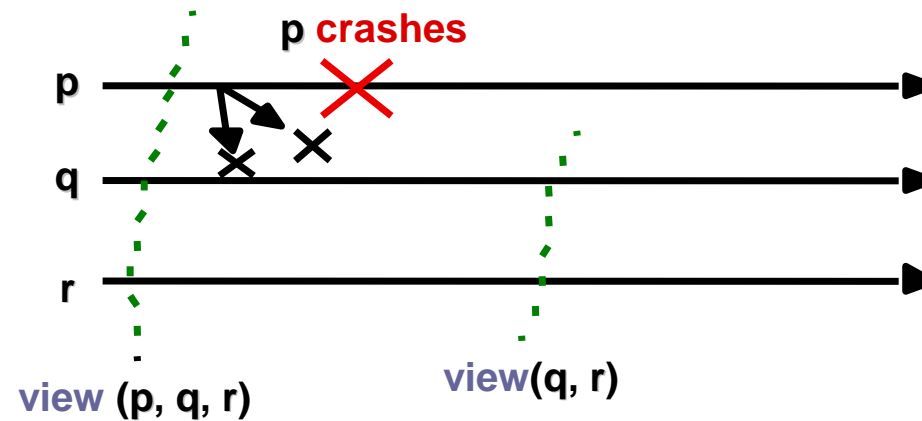
Given a group M_i , a view v_i and a message m sent via multicast to every process in M_i , the the multicast of m is view-synchronous in v_i iff the following condition is verified:

“if $\exists p \in v_i$ that delivered m in v_i and delivered v_{i+1} , then every process $q \in v_i$ that delivered v_{i+1} delivered m before delivering v_{i+1} ”

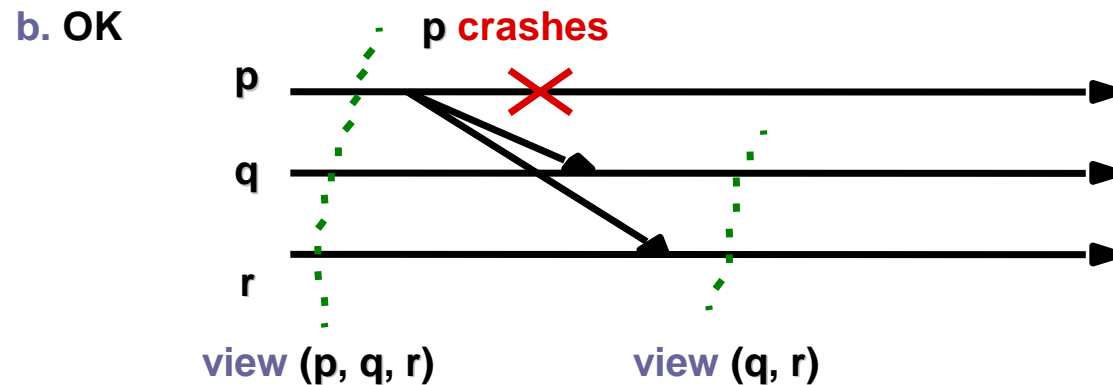
View-Synchronous Multicast: Scenario a

- Initial group: p,q,r
- p fails, q and r are correct

a. OK



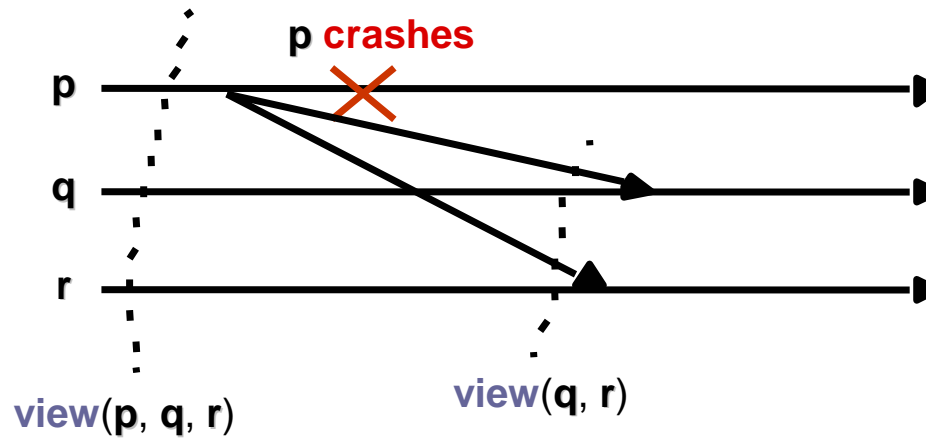
View-Synchronous Multicast: Scenario b



- At least one process among q and r delivered m before p crashes \Rightarrow q and r first deliver m and then deliver the new view {q,r}.

View-Synchronous Multicast: Scenario c

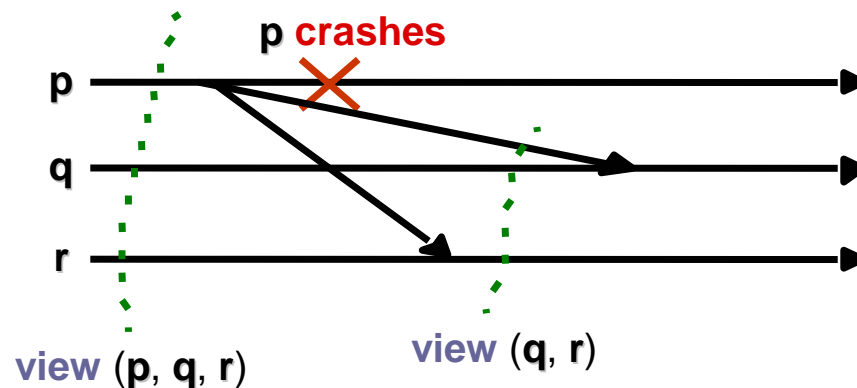
c. NO



- q and r cannot deliver a message from a process that failed.

View-Synchronous Multicast: Scenario d

d. NO.



- Processes must respect a same order in the delivery of the view and of the message.



Active Replication



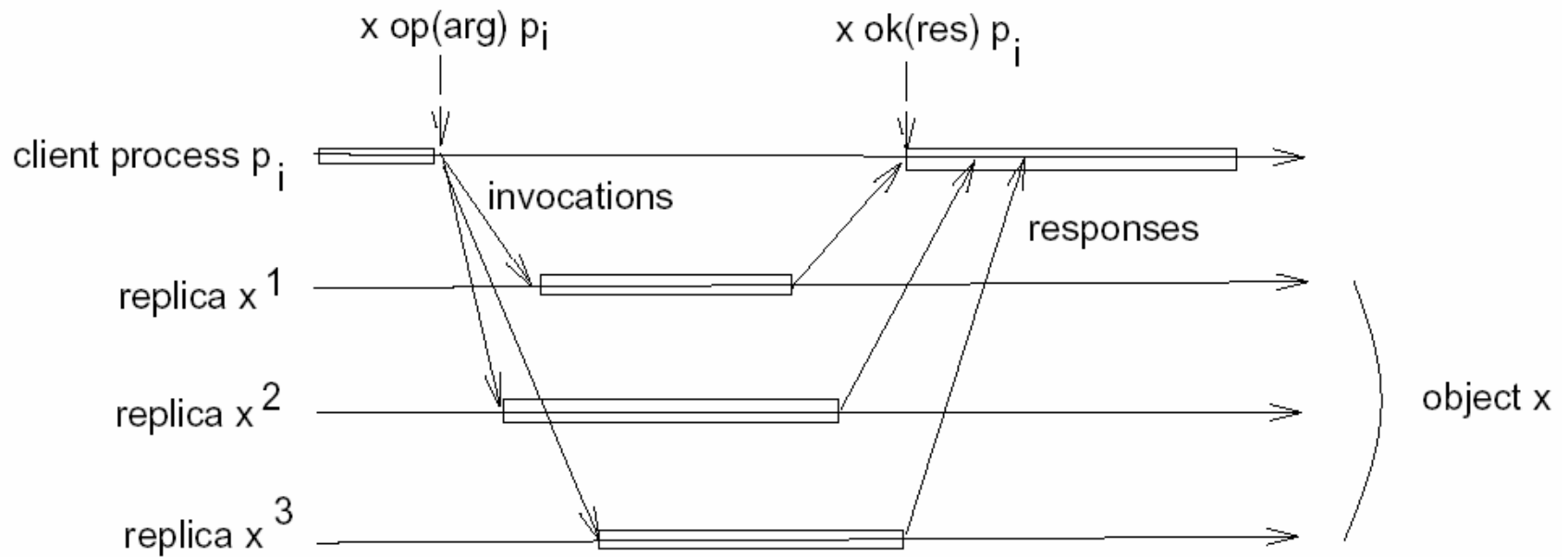
Active Replication

- There is no coordinator: all the replicas play the same role
- Every replica is **deterministic**: the outcome of an operation depends only on the state of the replica before the operation execution and on the operation itself

When a process p_i invokes an operation on an object x :

- The invocation is sent to every replica
- Every replica processes the invocation, updates its state and sends its response to the client.
- The client waits for the first response

Active Replication: a possible scenario



Active Replication: Linearizability

- This technique requires:
 - **Atomicity**: if a replica executes an operation, all the replicas execute that operation.
 - **Ordering**: given two operations executed by two replicas, the operations are executed in the same order.
- Communication primitive: **TOTAL ORDER
MULTICAST**



Active Replication: Crash

- Active replication does not require any action in case of a crash
- It is based on **static groups**:
 - Their composition does not change
 - Their composition does not reflect failures: a crashed replica remains in the group.



Active Replication: Ripristino di una replica

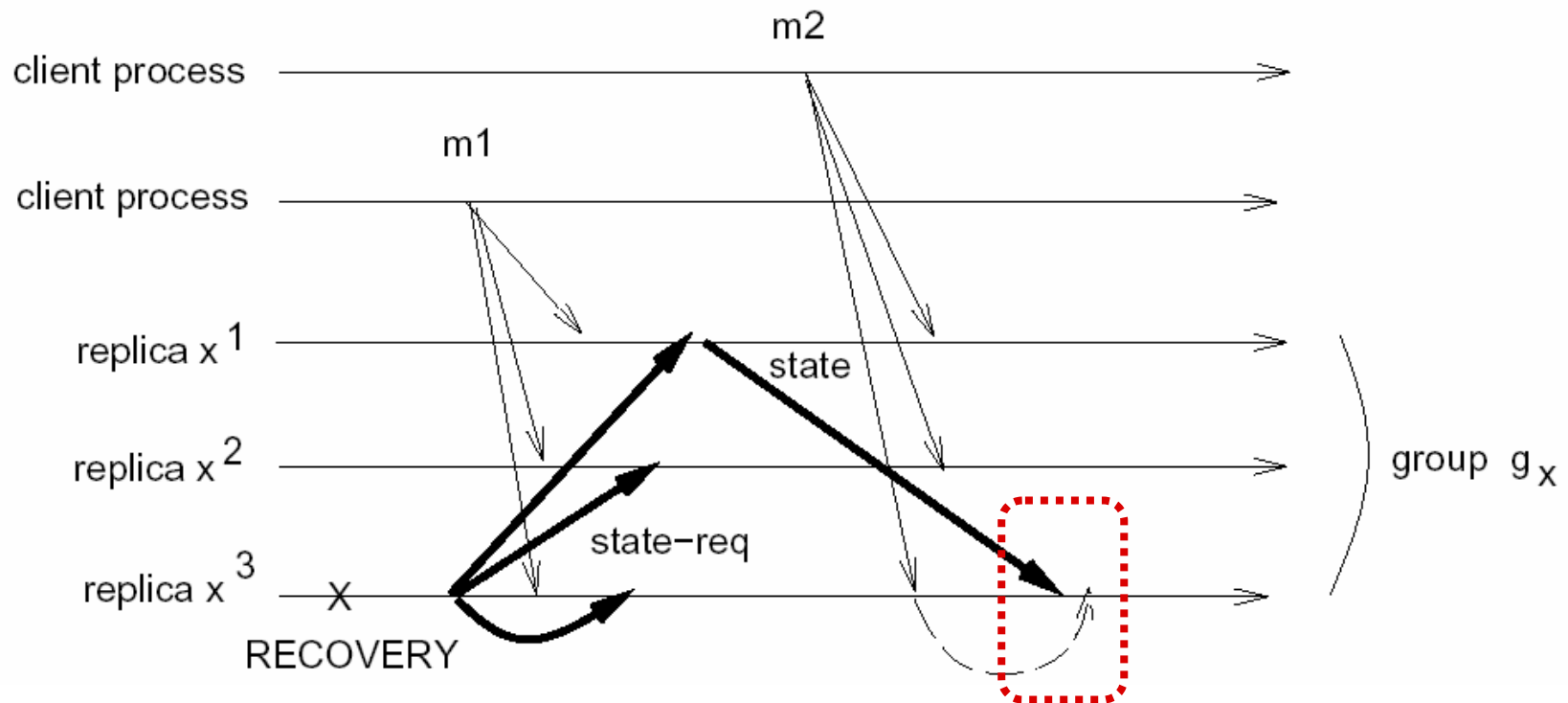
- Suppose that a replica x^k crashes at time t and recovers at time t' ($t < t'$)
- Due to *atomicity*, x^k must deliver every process sent between t and t'
- Then, when x^k recovers it must update its state from another replica x^j through the *State Transfer* procedure



Active Replication: State Transfer

- x^k sends a state request message (state-req, x^k) using the total order multicast primitive.
- Every replica, after delivering the message (state-req, x^k) sends its state to x^k .
- x^k waits to deliver its state request message.
- x^k waits the current state from one of the members of g_x .
- Meanwhile x^k buffers every message received after the send of (state-req, x^k).
- It updates its local state.
- Manages all the buffered messages.

State Transfer: example



Comparison

| | PRIMARY BACKUP | ACTIVE REPLICATION |
|----------------------------|--|---|
| Approach | Centralized: The primary is the coordinator | Totally distributed: All the replicas play the same role |
| Replica determinism | Yes/no | Yes |
| Replica failure | <ol style="list-style-type: none">1. Transparent if the replica is a backup2. If the primary fails then clients can experience large latencies: this is unacceptable for REAL-TIME applications | A replica failure is always transparent for a client |
| Resources usage | Less than Active Replication | More than Primary Backup |



Exercise

- The system is constituted by a finite number of processes
 - Every process has
 - a Perfect Failure Detector (P),
 - a BestEffortBroadcast primitive (beb)
 - a Uniform Consensus primitive (uc)
 - Failure model: crash-stop
- Define an algorithm that implements uniform view synchronous multicast:
 - No two processes install different views
 - Every process delivered by a process is delivered by all correct processes



Solution: Perfect Failure Detector

Module:

Name: PerfectFailureDetector (\mathcal{P}).

Events:

Indication: $\langle crash \mid p_i \rangle$: Used to notify that process p_i has crashed.

Properties:

PFD1: *Strong completeness:* Eventually every process that crashes is permanently detected by every correct process.

PFD2: *Strong accuracy:* If a process p is detected by any process, then p has crashed.

Module 2.5 Interface and properties of the perfect failure detector.



Solution: BestEffortBroadcast

Module:

Name: BestEffortBroadcast (beb).

Events:

Request: $\langle \text{bebBroadcast} \mid m \rangle$: Used to broadcast message m to all processes.

Indication: $\langle \text{bebDeliver} \mid \text{src}, m \rangle$: Used to deliver message m broadcast by process src .

Properties:

BEB1: *Best-effort validity:* For any two processes p_i and p_j . If p_i and p_j are correct, then every message broadcast by p_i is eventually delivered by p_j .

BEB2: *No duplication:* No message is delivered more than once.

BEB3: *No creation:* If a message m is delivered by some process p_j , then m was previously broadcast by some process p_i .

Module 3.1 Interface and properties of best-effort broadcast.



Solution: Uniform Consensus

Module:

Name: UniformConsensus (uc).

Events:

$\langle ucPropose \mid v \rangle$, $\langle ucDecide \mid v \rangle$: with the same meaning and interface as the consensus interface.

Properties:

C1: *Termination:* Every correct process eventually decides some value.

C2: *Validity:* If a process decides v , then v was proposed by some process.

C3: *Integrity:* No process decides twice.

C4': *Uniform Agreement:* no two processes decide differently.

Module 5.2 Interface and properties of uniform consensus.



Solution

```
upon event  $\langle \text{Init} \rangle$  do
  current-view := (0,  $\Pi$ );
  correct :=  $\Pi$ ; flushing := false; blocked := false;
  undelivered := delivered := dset :=  $\emptyset$ ; forall m do ackm :=  $\emptyset$ ;
```

```
upon event  $\langle \text{vsBroadcast} \mid m \rangle \wedge (\neg \text{blocked})$  do
  delivered := delivered  $\cup$  {m};
  trigger  $\langle \text{bebBroadcast} \mid [\text{DATA}, \text{current-view.id}, m] \rangle$ ;
```

When a process invokes *vsBroadcast* to send a message *m*:

- *m* is put in the *delivered* buffer
- *m* is sent to other processes using *bebBroadcast*



Delivery of messages in a view

```
upon event  $\langle \text{bebDeliver} \mid p_i, [\text{DATA}, \text{vid}, m] \rangle$  do
  if  $(\text{current-view.id} = \text{vid}) \wedge (\neg \text{blocked})$  then  $\text{ack}_m := \text{ack}_m \cup \{p_i\}$ 
  if  $(m \notin \text{delivered})$  then
    delivered := delivered  $\cup \{m\}$ ;
    trigger  $\langle \text{bebBroadcast} \mid [\text{DATA}, \text{current-view.id}, m] \rangle$ ;
```

- When a process receives m sent in the current view, adds the sender process p_i in ack_m (because p_i put m in *delivered*)
- Puts m in *delivered* if necessary
- Sends m using bebBroadcast

```
upon exists  $m: (\text{current-view} \subset \text{ack}_m) \wedge (m \notin \text{udelivered})$  do
  udelivered := udelivered  $\cup \{m\}$ ; trigger  $\langle \text{vsDeliver} \mid \text{src}_m, m \rangle$ ;
```

- When all the processes in the current view put m in the *delivered* buffer, m can be delivered guaranteeing uniformity



Creation of a new view

```
upon event  $\langle crash \mid p_i \rangle \wedge (\text{flushing} = \text{false})$  do  
  correct := correct  $\setminus \{p_i\}$ ;  
  flushing := true;  
  trigger  $\langle vsBlock \rangle$ ;
```

- When a process p understand that q crashed it invokes the creation of a new view that does not include q
- The a flushing phase starts where processi that are in the current view exchange the content of *delivered*
- The delivery of messages in the current view id blocked

```
upon event  $\langle vsBlockOk \rangle$  do  
  blocked := true; dset :=  $\emptyset$ ;  
  trigger  $\langle bebBroadcast \mid [DSET, \text{current-view.id}, \text{delivered}] \rangle$ ;
```



Creation of a new view

```
upon event  $\langle bebDeliver \mid src, [DSET, vid, mset] \rangle$  do
  dset := dset  $\cup \{(src, mset)\}$ ;
  if  $\forall p \in correct \exists \{(p, mset)\} \in dset$  then
    trigger  $\langle ucPropose \mid current-view.id+1, correct, dset \rangle$ ;
```

- Every process collects all the messages buffered by correct processes in the current view and then starts a run of uniform consensus
- The process propose a view:
 - Whose id is the id of the current view plus 1
 - Where M is the set of correct processes (in its opinion)
 - Propose also a set of messages that should be delivered before delivering the new view



Creation of a new view

```
upon event  $\langle ucDecided \mid id, memb, vs-dset \rangle$  do
  forall  $\{(p, mset)\} \in vs-dset: p \in memb$  do
    forall  $(src_m, m) \in mset: (src_m, m) \notin udelivered$  do
       $udelivered := udelivered \cup \{(src_m, m)\}$ ;
      trigger  $\langle vsDeliver \mid src_m, m \rangle$ ;
  flushing := blocked := false; delivered := udelivered  $\emptyset$ ;
  current-view := (id, memb); trigger  $\langle vsView \mid current-view \rangle$ ;
```

- Before delivering the new view every process must deliver every message contained in the delivered set decided by the consensus run