



Sistemi Distribuiti

Teoria della Replicazione

Dott. Ing. Silvia Bonomi

bonomi@dis.uniroma1.it

Replicazione: Motivazione

- Assicurare la disponibilità di un oggetto (servizio) a fronte di un certo numero e tipo di guasti
- Crash: un processo esegue correttamente il suo algoritmo fino ad un istante t dopo il quale interrompe prematuramente l'esecuzione del suo algoritmo
- L'oggetto deve essere accessibile con tempo di risposta ragionevole per una frazione di tempo prossima al 100%
 - Es. Sia O un oggetto replicato su n nodi la cui probabilità di guastarsi sia p (indipendente dalla probabilità di guasto degli altri nodi).

Disponibilità di O :

$$1 - p^n$$

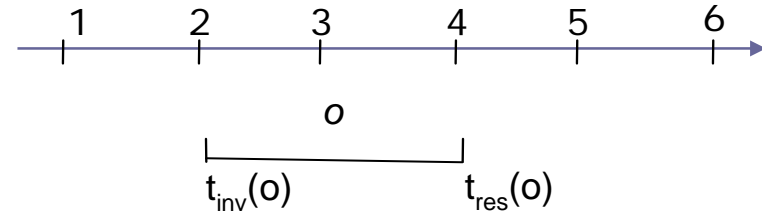
Oggetto Logico

- N processi sequenziali p_1, p_2, \dots, p_n interagiscono attraverso un insieme X di oggetti condivisi (file, variabili, code, ecc...)
- Un oggetto $x \in X$ ha uno stato (**oggetto logico**)
- I processi accedono allo stato dell'oggetto attraverso operazioni o
- Un'operazione o eseguita da un processo p_i su un oggetto x è una coppia invocazione-risposta, $[x \ o(\arg) \ p_i] / [ok \ (res) \ p_i]$
- Il set di operazioni che gestiscono l'oggetto ne determinano la semantica

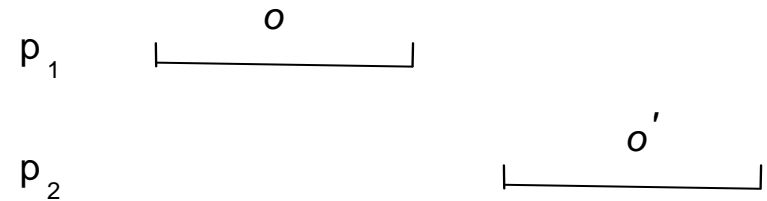
Relazioni temporali(real time) tra operazioni

Notazione:

1. $t_{inv}(o)$ istante (real time) in cui p_i invoca o .
2. $t_{res}(o)$ istante (real time) in cui p_i riceve la corrispondente risposta



Due operazioni o e o' , sono dette **sequenziali**, denotato $o < o'$, se la risposta di o precede l'invocazione di o' . ($t_{res}(o)$ minore $t_{inv}(o')$)



Due operazioni o e o' , sono dette **concorrenti**, denotato $o || o'$, se $\neg(o < o')$ e $\neg(o' < o)$



Modello di Replicazione

- Ogni “oggetto logico” x è implementato da un set finito $\{x^1, x^2 \dots x^m\}$ di copie fisiche, “repliche”
- Ogni replica è collocata su un nodo differente del sistema
- Sia s il nodo su cui è collocata la replica i -esima dell’oggetto x , x^i , allora su s è in esecuzione un processo sequenziale, *gestore della replica*, che gestisce le invocazioni su x^i
- x^i denota sia la i -esima replica dell’oggetto x che il corrispondente processo gestore

Requisiti

- **Trasparenza:**

- Il processo deve credere di interagire con l'oggetto logico
- La replicazione non cambia:
 - Il modo in cui un processo invoca un'operazione.
 - Il modo in cui vengono restituite le risposte.

- **Consistenza:**

- Le operazioni devono produrre risultati che rispettino le specifiche di correttezza dell'oggetto logico anche se eseguite su una collezione di repliche
- Le specifiche di correttezza dipendono dalla semantica dell'oggetto logico

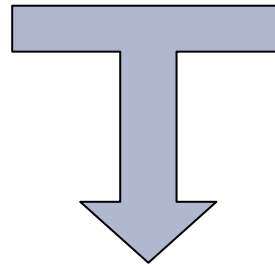
Consistenza: Linearizability

Sistema sequenziale:

- In ogni istante di tempo un solo processo client nel sistema esegue un'operazione sull'oggetto
- Il significato di ogni operazione è definito dalle pre e post condizioni

Sistema concorrente:

- Più processi client concorrono per accedere all'oggetto
- Si deve dare un significato al possibile ordinamento delle operazioni



LINEARIZABILITY

Definisce la correttezza di un oggetto concorrente in termini della sua specifica sequenziale

Linearizability: Idea

- Dare ai client l'illusione di accedere all'oggetto in maniera sequenziale anche a fronte di concorrenza
- L'effetto di ogni operazione deve essere lo stesso che si avrebbe qualora tale operazione fosse stata eseguita in maniera istantanea
- L'ordine tra operazioni sequenziali deve essere preservato

Linearizability: Specifica

- Un'esecuzione E è **linearizzabile** se esiste una sequenza S contenente tutte le operazioni in E tali che:
 1. \forall due operazioni o e o' in E , se $o < o'$ allora o appare prima di o' in S
 2. La sequenza S sia **legale**: S rispetta la semantica dell'oggetto come definita dalla sua specifica sequenziale

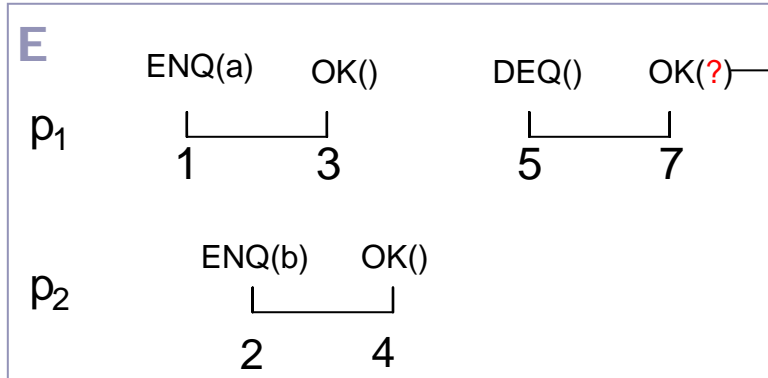


Esempio Coda FIFO

Coda FIFO (First In First Out): Oggetto Logico

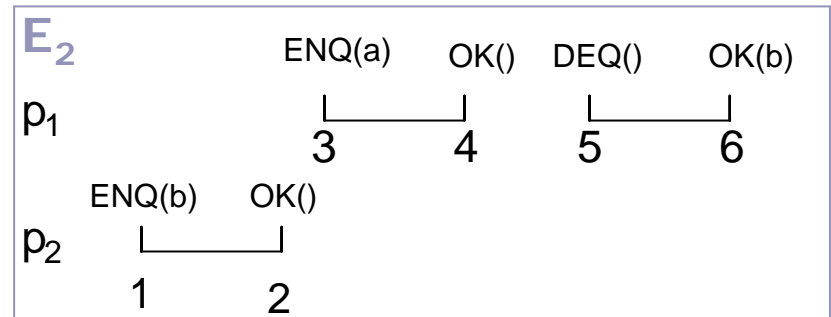
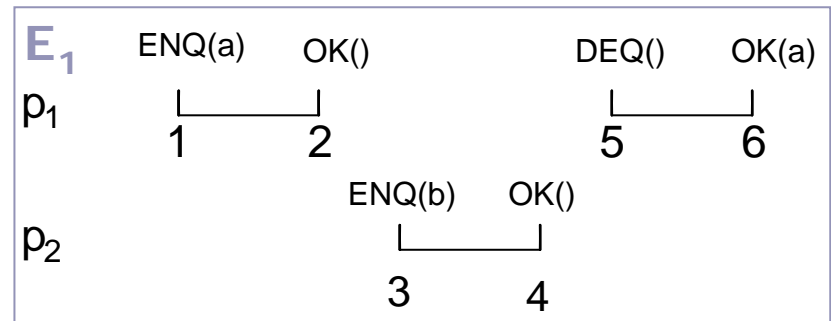
- Specifica sequenziale definita in termini delle operazioni che permettono l'accesso alla coda:
 - Quando un processo vuole inserire un elemento nella coda, es. *a*, invoca ENQ(*a*). Quando riceve Ok() e' sicuro che l'elemento e' stato inserito
 - Quando un processo vuole estrarre un elemento dalla coda invoca DEQ(). La coda deve restituire il *primo* elemento inserito in coda o un messaggio di coda vuota se nessun elemento è stato *precedentemente* inserito in coda
 - NOTA: in un sistema sequenziale non esiste ambiguità sul concetto di *primo* e *precedente*. Tali concetti diventano ambigui nel caso di sistema concorrente.

Coda FIFO: concorrenza



Quale valore deve restituire l'operazione DEQ() invocata da p₁?
Sia a che b sono valori ammissibili

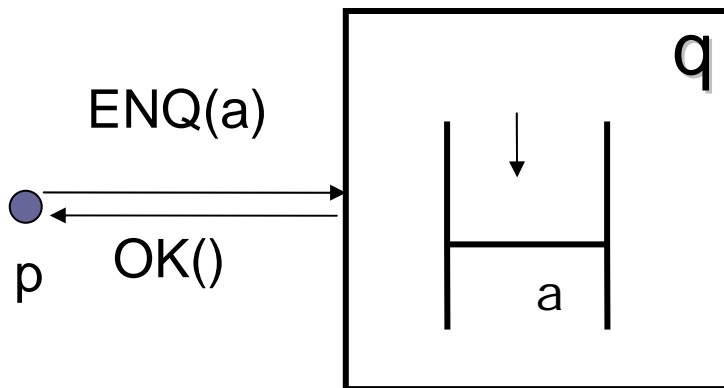
Se l'operazione DEQ() restituisce a, l'esecuzione concorrente E da ai client l'illusione di aver eseguito E₁. Se restituisce b, da l'illusione dell'esecuzione sequenziale E₂.



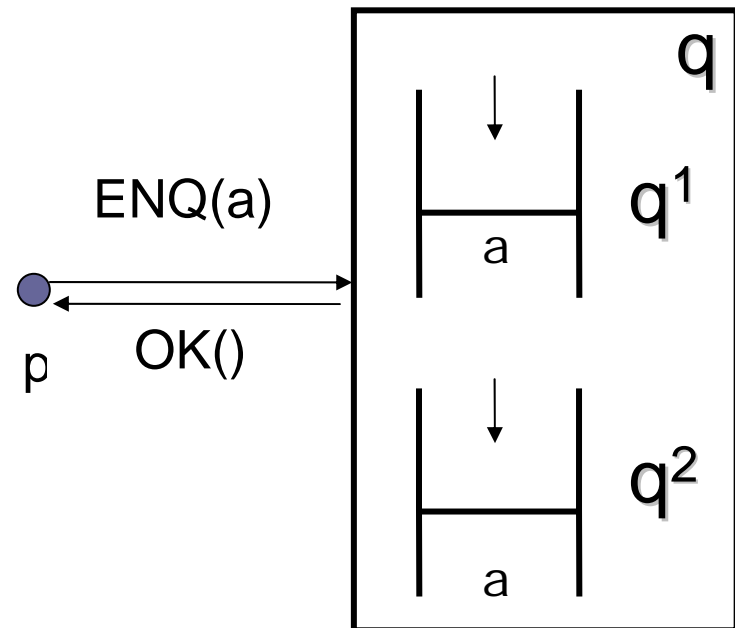
Coda FIFO: Trasparenza

- Assumiamo q inizialmente vuota
- Il modo in cui il client p invoca le operazioni e riceve risposta è indipendente da come la coda è implementata

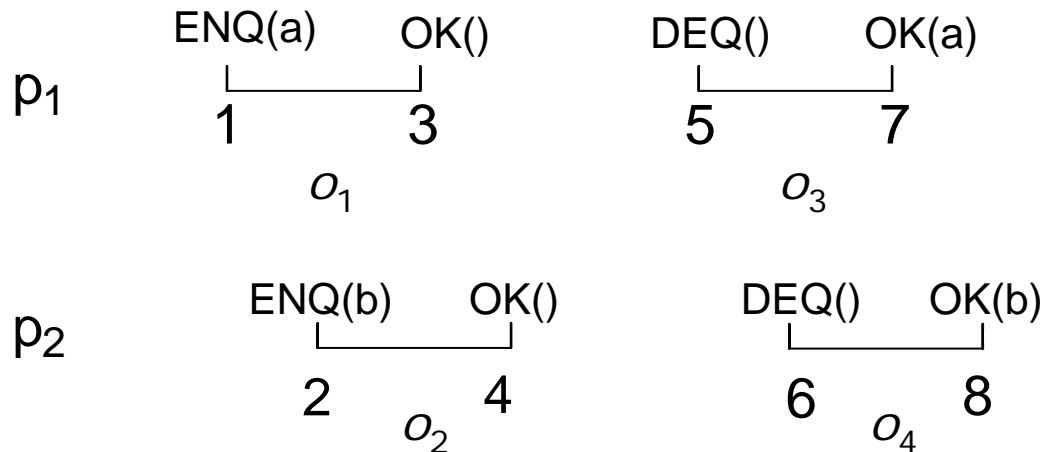
1. Unico oggetto fisico



2. Coda FiFo implementata con 2 repliche

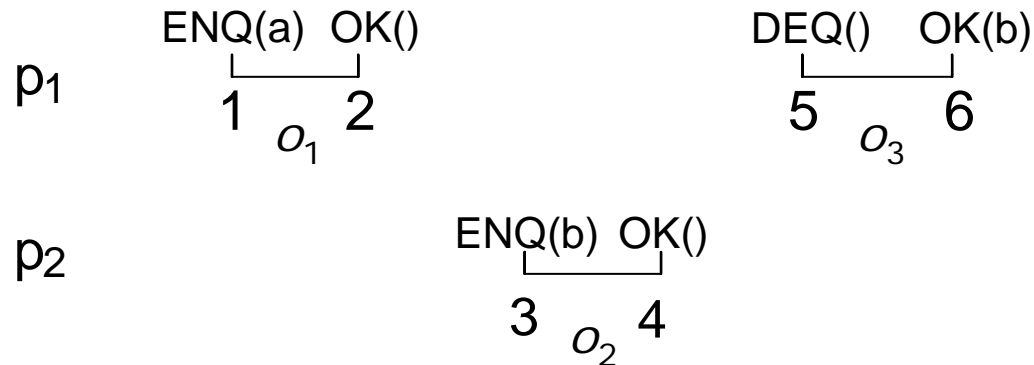


Coda FIFO consistenza: Esecuzione “Linearizzabile”



- Assumiamo la coda inizialmente vuota.
- E e' l'esecuzione costituita dal seguente insieme di operazioni $\{o_1, o_2, o_3, o_4\}$ e t.c. $o_1 < o_3, o_1 < o_4, o_2 < o_3$ e $o_2 < o_4$.
- E e' linearizzabile perche' esiste $S = o_2 o_1 o_4 o_3$ tale che:
 - Rispetta l'ordinamento real time,
 - S è *legale* perché rispetta le specifiche di una coda FIFO: b è messo in coda prima di a, quindi la prima operazione di estrazione restituisce b e la seconda a

Coda FIFO consistenza: Esecuzione non Linearizzabile



- Assumiamo la coda inizialmente vuota
- E e' l'esecuzione costituita dal seguente insieme di operazioni $\{o_1, o_2, o_3\}$ e t.c. $o_1 < o_2, o_1 < o_3, o_2 < o_3$
- E NON e' LINEARIZZABILE perche' NON ESISTE S tale che:
 - Rispetta l'ordinamento real time tra le operazioni,
 - Rispetta le specifiche di una coda FIFO: a è messo in coda prima di b, quindi la prima operazione di estrazione dovrebbe restituire a



Implementare Linearizability: Tecniche di Replicazione Software

Implementare Linearizability

- Condizione sufficiente a garantire la linearizability e' che le repliche siano concordi sul set di invocazioni ricevute e sull'ordine delle stesse.
- Formalmente:
 - **Ordine**: date due invocazioni, se due repliche eseguono entrambe le invocazioni, le eseguono in uno stesso ordine.
 - **Atomicità**: se una replica esegue un'invocazione allora tutte le repliche non guaste devono eseguire tale invocazione.

Tecniche di Replicazione

- Due fondamentali tecniche di replicazione che implementano linearizability sono:
 - Primary Backup
 - Active Replication

Modello di Sistema

- Insieme finito di processi che comunicano scambiandosi messaggi su una rete di comunicazione
- I processi si dividono in client e gestori delle repliche
- Perfect point-point links:
 1. **Reliable delivery** - Sia p_i un processo che invia un messaggio m ad un altro processo p_j . Se né p_i né p_j si guasta, allora p_j prima o poi consegna m .
 2. **No duplication** – Nessun messaggio viene consegnato da un processo più di una volta.
 3. **No creation** – Se un messaggio m è consegnato da un processo p_j , allora m è stato precedentemente inviato a p_j da qualche processo p_i
- Sistema asincrono
- Un processo può guastarsi a causa di un crash
- Un processo che ha subito crash potrebbe ripristinarsi (recovery)



Replicazione Passiva: Primary-Backup

Primary Backup

- Le repliche non hanno tutte lo stesso ruolo:
 - *Primary*:
 - riceve le invocazioni dal processo client e restituisce la risposta.
 - Dato un oggetto x , il primary di x è denotato $prim(x)$.
 - *Backup*:
 - interagiscono solo con la $prim(x)$
 - servono per garantire la tolleranza ai guasti

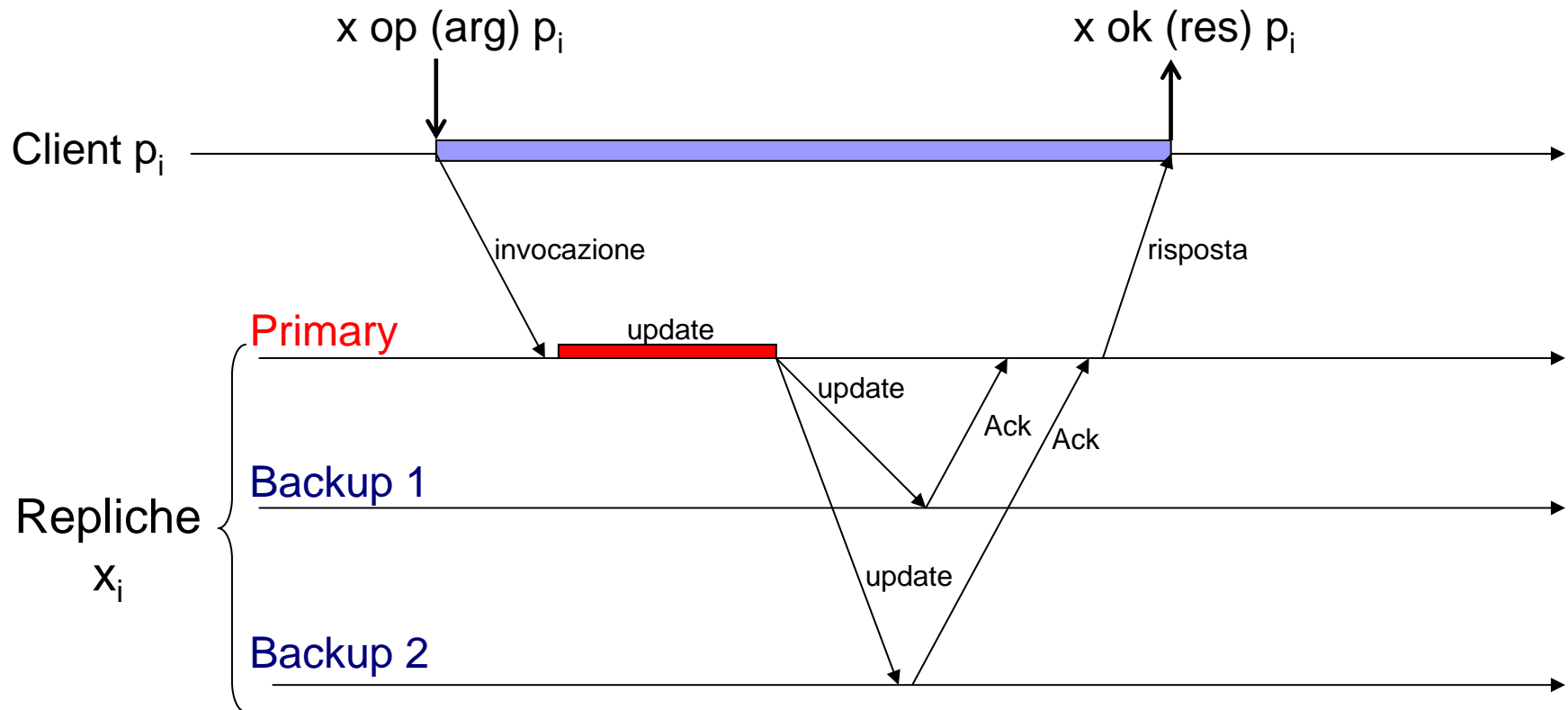
Primary Backup: Assenza di Crash

- Il client p_i invoca l'operazione $op(arg)$ su x .
- Il primary riceve l'invocazione ed esegue l'operazione.
- Dopo l'esecuzione il primary ha aggiornato il suo stato e la risposta, res , è disponibile.
- Il primary invia dei messaggi di update ($invld$, res , $state-update$) ai *backup*:
 - **invld**: identificativo univoco dell'invocazione
 - **res**: risposta per il client
 - **state-update**: stato del primary dopo aver eseguito la corrente invocazione.

Primary Backup: Assenza di Crash (2)

- Quando ricevono gli update, i *backup* aggiornano il loro stato e inviano l'ack al primary
- Il primary aspetta gli ack di tutti i *backup* corretti e poi invia la risposta, *res*, al client.
- **Garantire Linearizability**: l'ordine in cui *il primary* riceve le invocazioni determina l'ordinamento totale delle operazioni sull'oggetto.

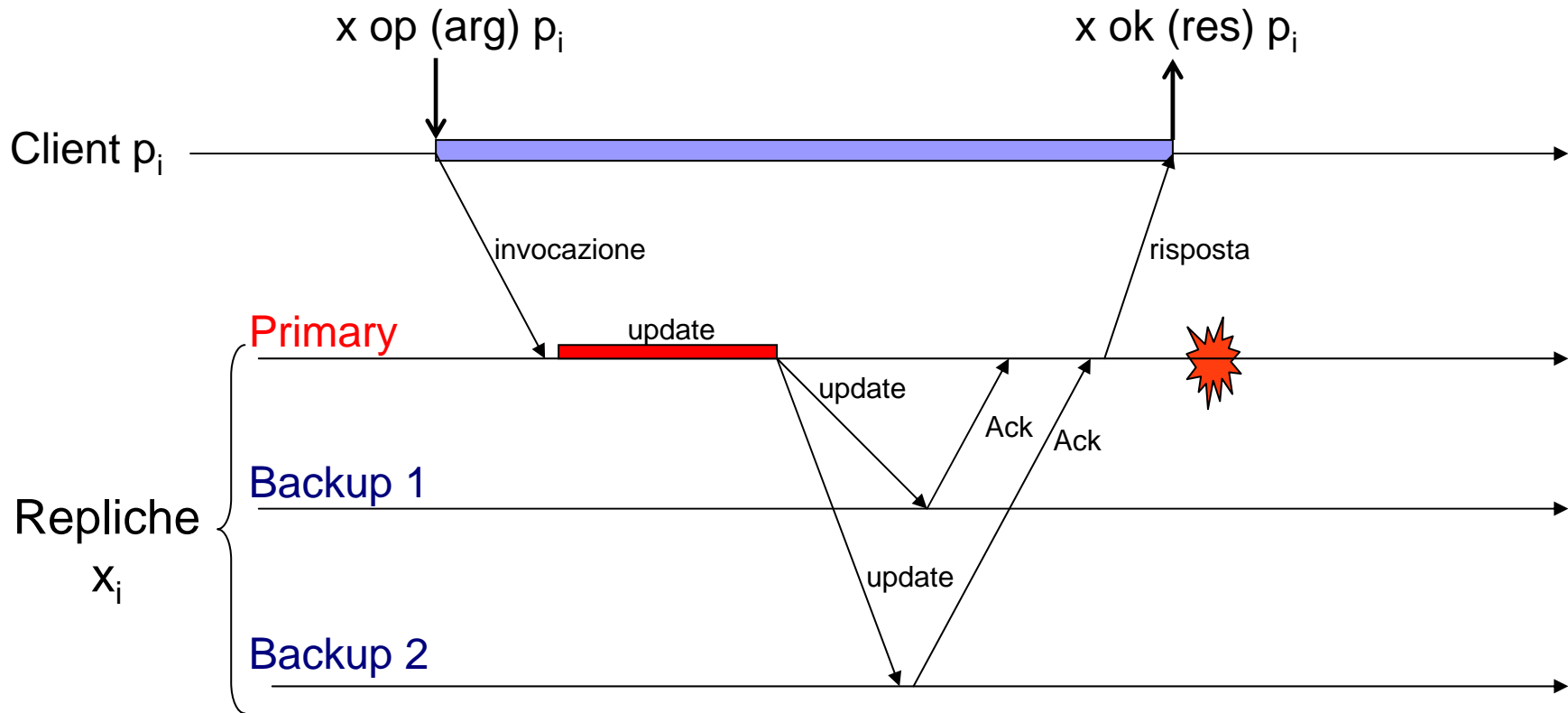
Scenario di Funzionamento in assenza di crash



Primary Backup: Presenza di Crash

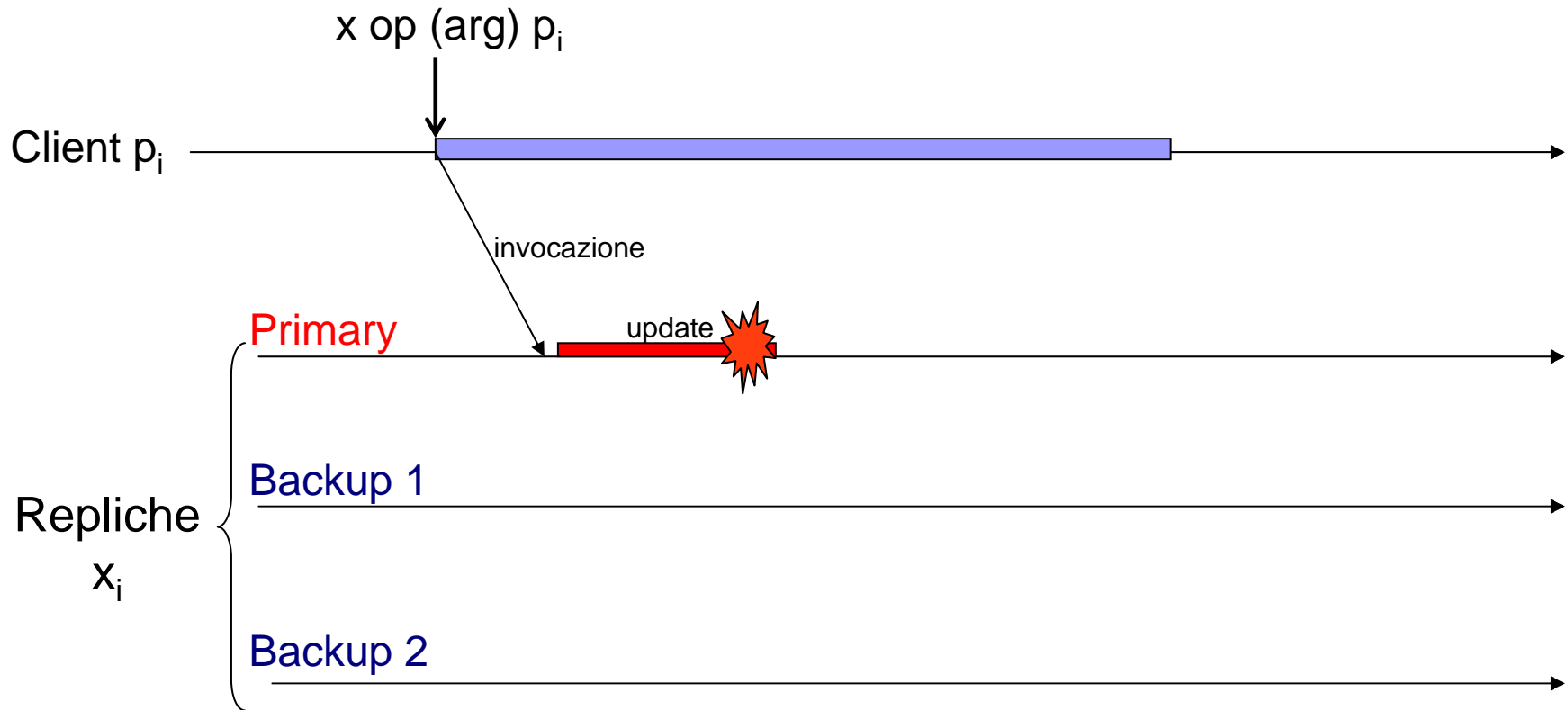
- Distinguiamo tre scenari di guasto del *primary*:
 1. **Scenario 1**: Guasto dopo che il client ha ricevuto la risposta
 2. **Scenario 2**: Guasto prima di inviare i messaggi di update
 3. **Scenario 3**: Guasto dopo aver inviato i messaggi di update ma prima di aver ricevuto gli ack.
- In tutti e tre i casi si deve eleggere un nuovo *primary*:
elezione del leader

Scenario 1



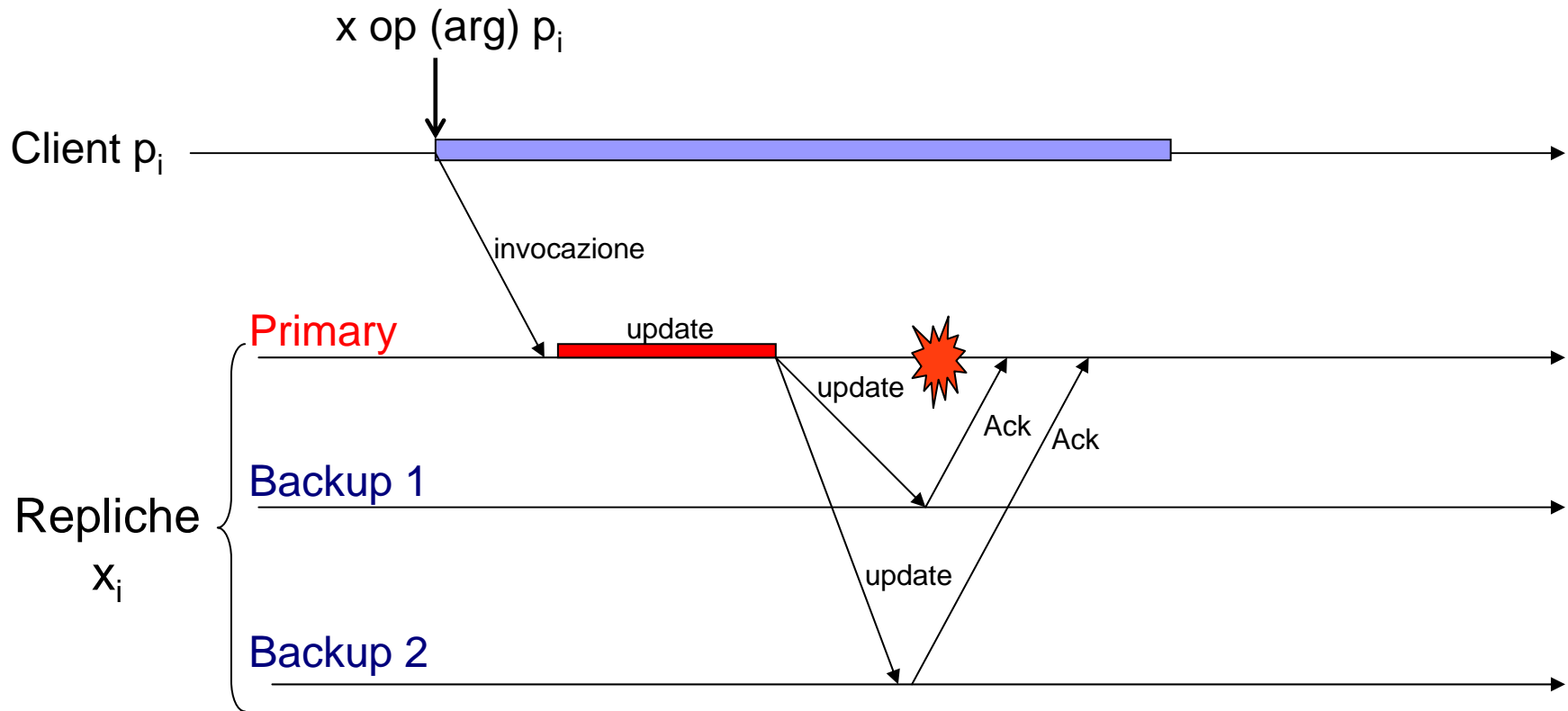
- Guasto dopo che il client ha ricevuto la risposta:
 - Il client non percepisce il guasto

Scenario 2



- Guasto prima di inviare i messaggi di update
 - Il client non ottiene una risposta e quindi sospetta il guasto.
 - Il nuovo primary tratta l'invocazione come se fosse nuova

Scenario 3



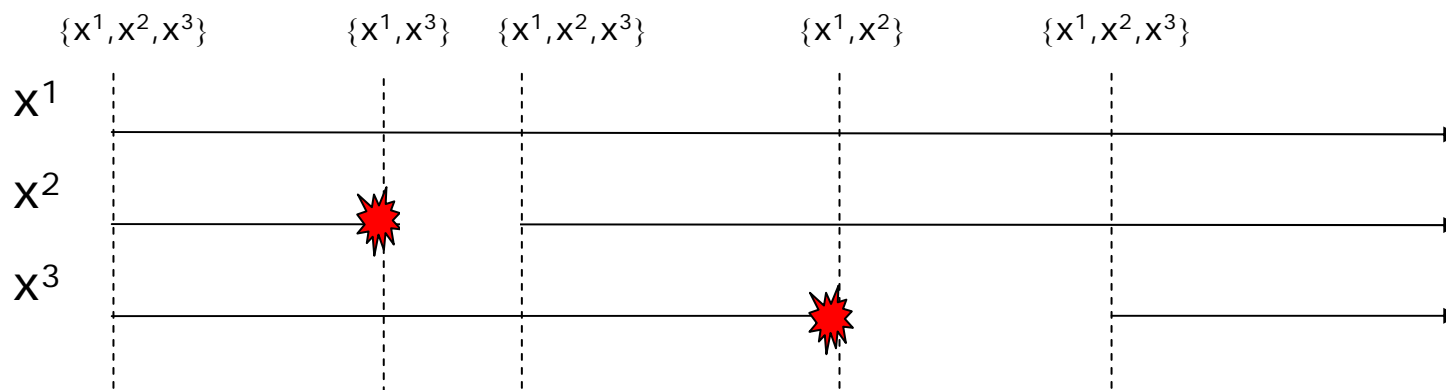
- Guasto dopo aver inviato i messaggi di update ma prima di aver ricevuto gli ack:
 - **Garantire atomicità:** l'update è ricevuto da **tutti** o da **nessuno**.

Scenario 3: possibili sottoscenari

- **Nessun** backup ha ricevuto l'update:
 - si torna allo **Scenario 2**
- **Tutti** i backup hanno ricevuto l'update implica che l'operazione è stata eseguita ma il client non ha avuto risposta:
 - Lo stato dei backup è aggiornato
 - Il client invoca nuovamente la stessa operazione
 - Il nuovo primary usa le informazioni (invld,res) per capire che l'invocazione è già stata gestita e mandare al client la res precedentemente calcolata.

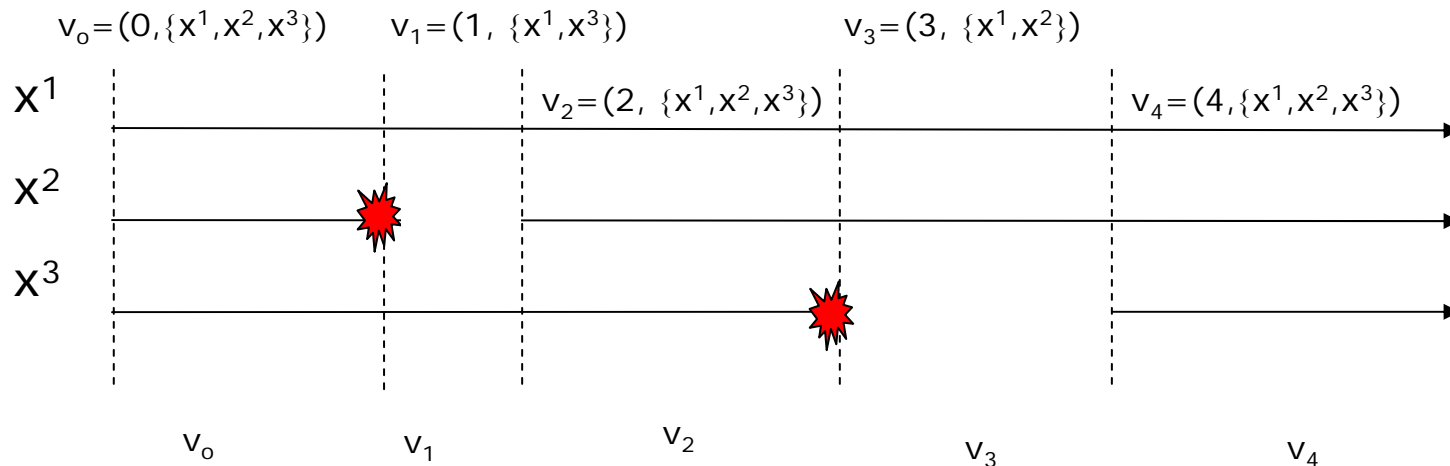
Primary Backup: Gruppi dinamici

- L'insieme di repliche corrette nel sistema può variare nel tempo a causa dei crash e dei ripristini
- **Gruppo M**: insieme di repliche che gestiscono l'oggetto x .
- **Gruppi dinamici**: la composizione del gruppo varia durante il ciclo di vita del sistema:
 - quando una replica x^i si guasta, viene rimossa dal gruppo,
 - quando una replica x^i si ripristina viene reinserita nel gruppo.



Primary Backup: Gruppi dinamici

- **View** v_i : è una tupla $v_i=(i,M_i)$ che modella l'evolvere della composizione del gruppo, dove i è l'identificativo della i -esima view e M_i è la corrispondente composizione del gruppo
- $V_0=(0,M_0)$ è la view iniziale.



Primary-Backup: Leader Election

- Quando il **primary** si guasta è necessario eleggerne un altro tra le repliche corrette: **Leader election**
- Supponiamo che ci sia una regola R che ordina le repliche all'interno di una view.
 - Es. R=ordine crescente dell'identificativo della replica:
 - x^1, x^2, x^4 **OK**
 - x^2, x^1, x^4 **ERRORE**
- Il nuovo primary può essere la prima replica nella **view corrente** in accordo ad R.
 - Definizione primary:
 - $v_0 = (0, \{x^1, x^2, x^4\})$, primary x^1
 - $v_1 = (1, \{x^2, x^4\})$, primary x^2

Primary-Backup: Leader Election

- **Correttezza: tutti i processi devono eleggere uno stesso primary a partire da uno stesso set di repliche corrette**



CONSEGNA ORDINATA DELLE VIEW

Consegna Ordinata delle view: Specifica

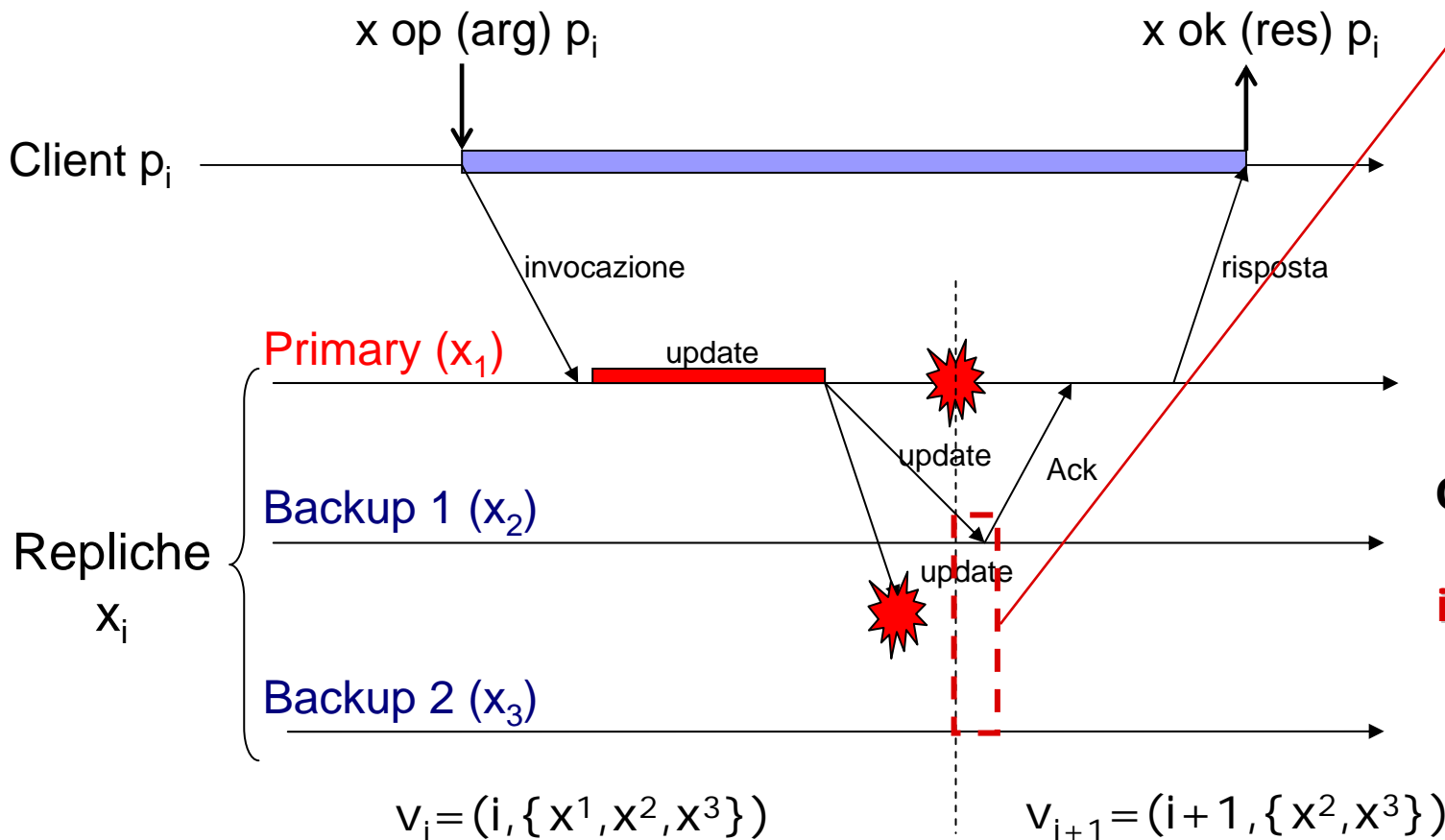
- L'ordinamento delle view deve soddisfare le seguenti specifiche:
 - Se un processo $p \in g_x$ si guasta, viene definita una nuova vista di cui p non fa parte
 - Se un processo p viene ripristinato e fa nuovamente parte del gruppo g_x , viene definita una nuova vista che comprende p
 - Data una vista $v_i(g_x)$ t.c. un processo $p \in v_i(g_x)$, allora o p consegna $v_i(g_x)$ oppure \exists un $k > 0$ t.c. $p \notin v_{i+k}(g_x)$.
 - $\forall p, q \in g_x$, se p e q consegnano $v_i(g_x)$ e $v_j(g_x)$ con $i \neq j$, allora le consegnano nello stesso ordine
- **NOTA:** è **IRRILEVANTE** se una replica è stata eliminata dal gruppo perchè realmente guasta oppure perchè erroneamente sospettata.

Primary-Backup: Correttezza

E' sufficiente garantire ordinamento delle invocazioni e delle viste per garantire **CORRETTEZZA**?

NO

VIOLAZIONE ATOMICITA'



Gli stati di x^2 e x^3 sono inconsistenti

Primary-Backup: Correttezza

- La correttezza del primary-backup si basa sull'utilizzo di una primitiva di comunicazione molto forte: VIEW SYNCHRONOUS MULTICAST
- Estende la semantica del reliable multicast prendendo in considerazione i cambiamenti delle view del gruppo
 - Sia t l'istante in cui un processo p consegna la view $v_i = (i, M_i)$.
 - Ogni msg m spedito da p prima della consegna della vista V_{i+1} :
 - conterrà l'identificativo i
 - sarà spedito a tutti i processi in M_i

View-Synchronous Multicast

■ Garantisce:

- Atomicità: tutte le repliche in $v_i \cap v_{i+1}$ consegnano m prima di consegnare v_{i+1}
- La consegna di una nuova vista quando qualche processo si guasta o si ripristina
- Ordinamento nella consegna delle viste

View-Synchronous Multicast: Specifica

Assunzione: $v_0=(0, M_0)$ è la view iniziale di ogni processo appartenente a M_0

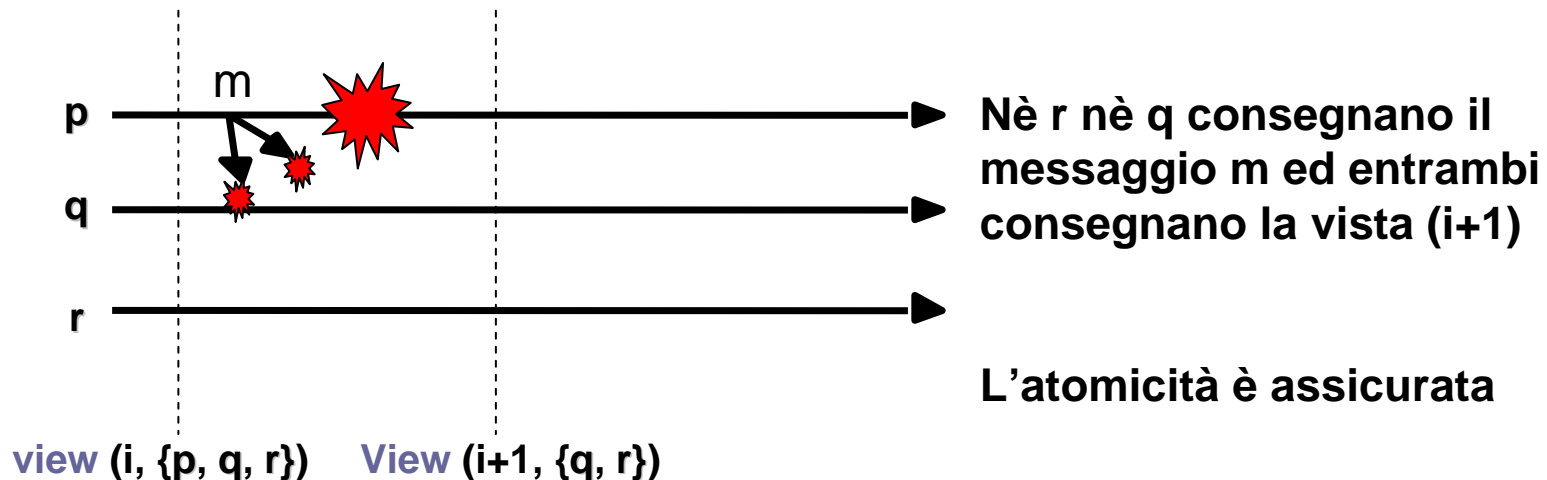
Def. View-synchronous multicast

Dato un gruppo M_i , una view v_i e un messaggio m mandato in multicast a tutti i processi appartenenti a M_i , allora

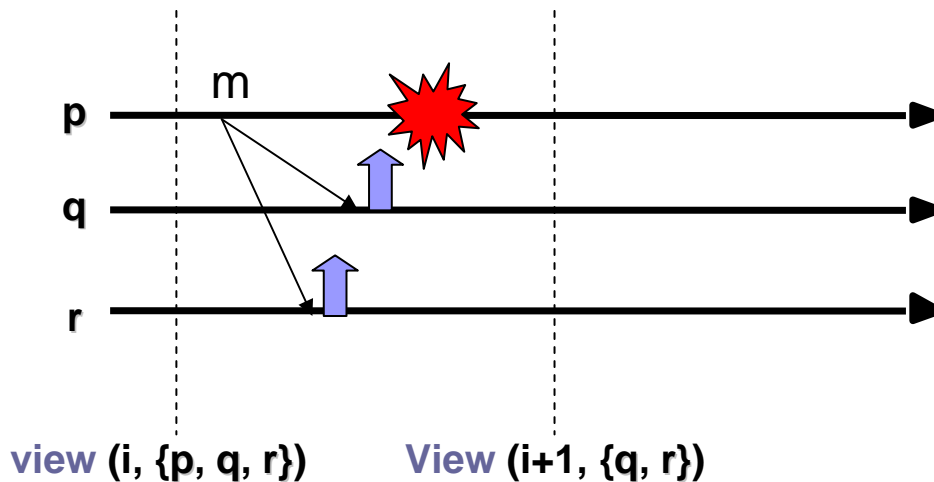
SE $\exists p \in v_i$ che ha consegnato m in v_i e ha consegnato v_{i+1} , allora tutti i processi $q \in v_i$ che hanno consegnato v_{i+1} hanno consegnato m prima di consegnare v_{i+1}

View-Synchronous Multicast: Scenario a

- Gruppo: p,q,r
- p si guasta, q ed r sono corretti



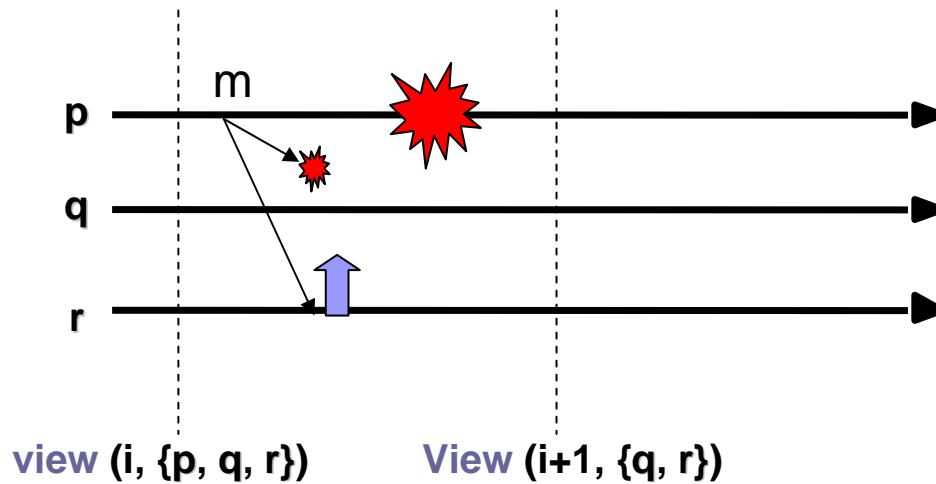
View-Synchronous Multicast: Scenario b



Sia r che q consegnano il messaggio m prima del crash di p e poi entrambi consegnano la vista (i+1)

L'atomicità è assicurata

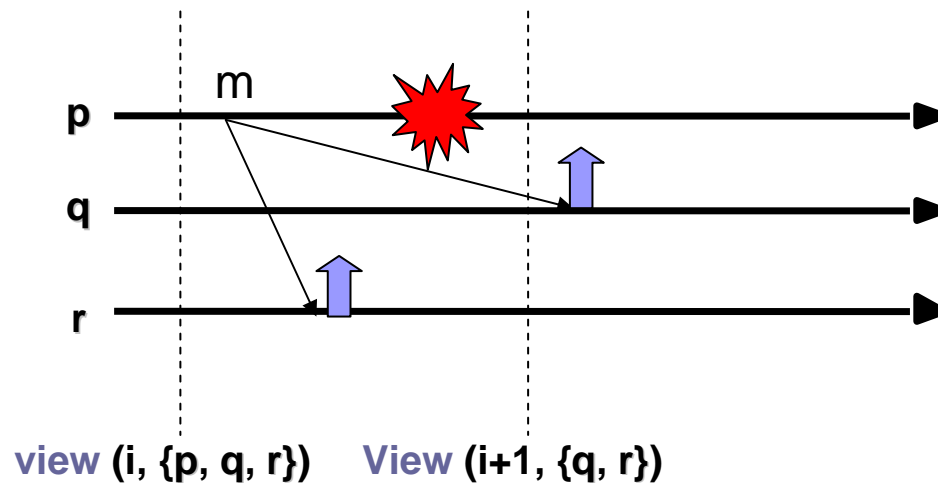
View-Synchronous Multicast: Scenario c



r consegna il messaggio m e poi la vista (i+1) mentre q solo la nuova vista

L'atomicità è violata

View-Synchronous Multicast: Scenario d



r consegna il messaggio m e poi la vista (i+1) mentre q consegna la nuova vista e poi m

L'atomicità è violata



Active Replication

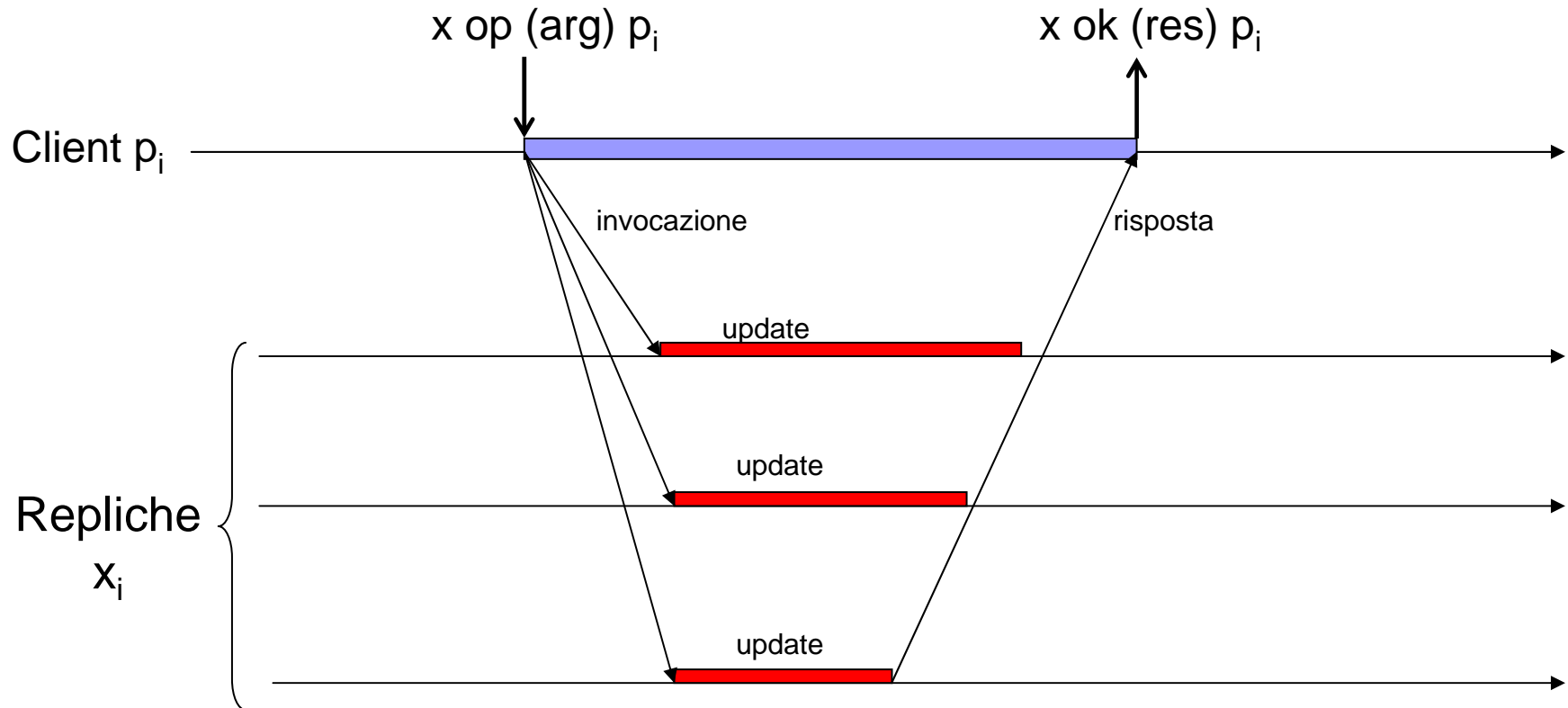
Active Replication

- Non c'è un coordinatore: tutte le repliche hanno uno stesso ruolo
- Ogni **replica** è **deterministica**: l'outcome di una operazione dipende soltanto dallo stato iniziale della replica e dalla sequenza di operazioni precedentemente eseguite dalla replica.

Quando un processo p_i invoca un'operazione sull'oggetto x :

- L'invocazione è spedita a tutte le repliche
- Ogni replica processa l'invocazione, aggiorna il suo stato e restituisce la risposta al client.
- Il client aspetta una sola risposta

Active Replication: Scenario di Funzionamento



Active Replication: Garantire Linearizability

- Questa tecnica richiede:
 - **Atomicità**: se una replica gestisce un'invocazione allora ogni replica corretta la deve gestire a sua volta.
 - **Ordinamento**: date due invocazioni, se due repliche gestiscono entrambe, lo fanno secondo uno stesso ordinamento.
- Primitiva di comunicazione adeguata: **TOTAL ORDER
MULTICAST**

Active Replication: Crash

- L'active replication non richiede alcun tipo di azione nel caso di guasto di una replica
- Si basa su **gruppi statici**:
 - la composizione del gruppo non cambia nel tempo,
 - la composizione non riflette i guasti: una replica guasta resta nel gruppo.

Active Replication:

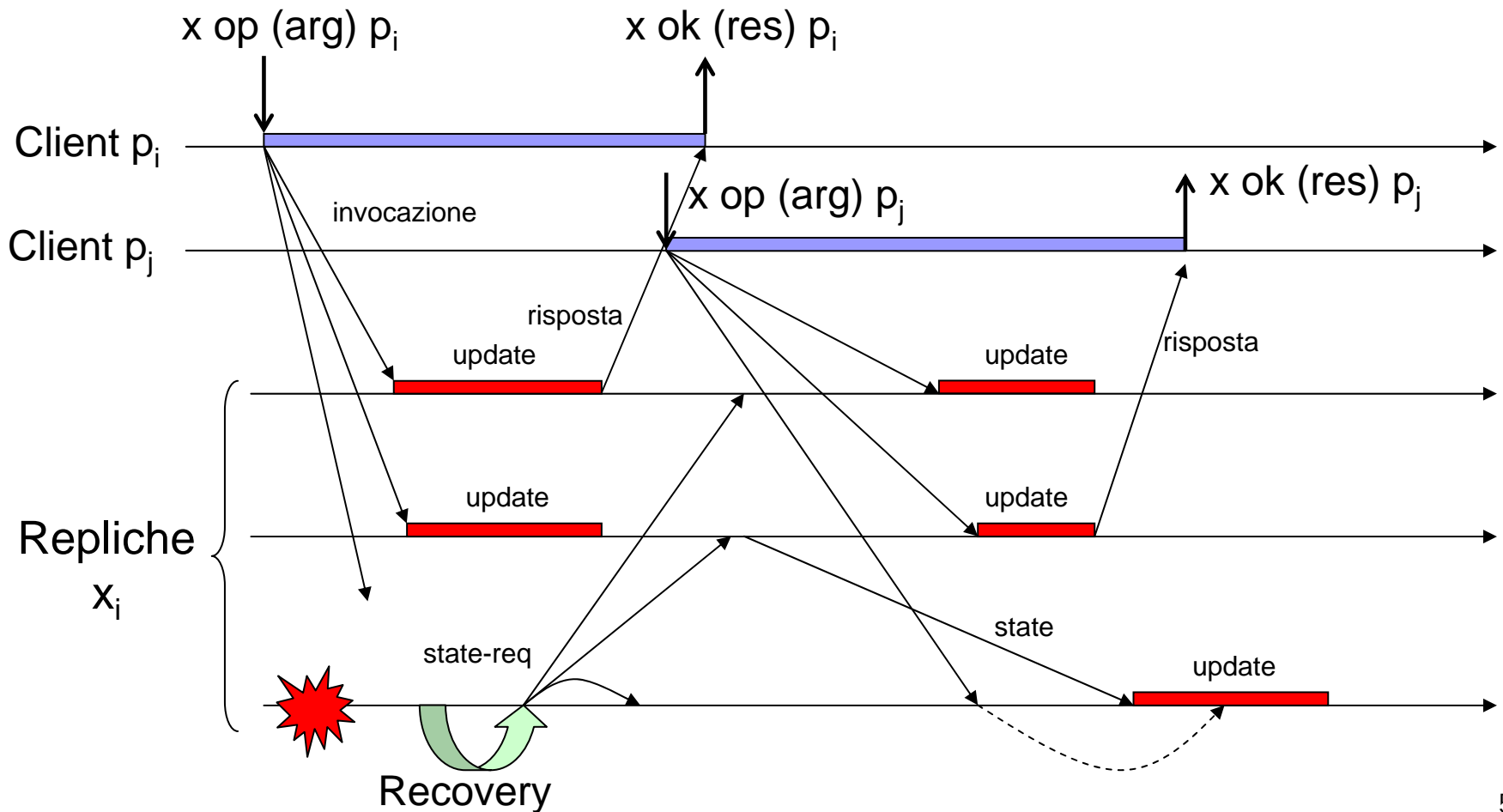
Ripristino di una replica

- Supponiamo che una replica x^k si guasti al tempo t e venga ripristinata all'istante t' ($t < t'$)
- Per l' *atomicità*, x^k dovrebbe aver consegnato tutti i messaggi spediti fino a t'
- Quindi, quando x^k viene ripristinata, viene aggiornata da un'altra replica operativa x^j attraverso la procedura di *State Transfer*

Active Replication: State Transfer

- x^k invia un messaggio di richiesta dello stato (state-req, x^k) attraverso una primitiva di total order multicast.
- Ogni replica dopo aver consegnato il messaggio (state-req, x^k) invia il suo stato a x^k .
- x^k aspetta di aver consegnato il suo messaggio di richiesta dello stato.
- x^k aspetta lo stato corrente da uno dei membri del gruppo g_x .
- Nel frattempo x^k bufferizza i messaggi ricevuti dopo aver inviato (state-req, x^k).
- Aggiorna il suo stato.
- Gestisce ipotetici messaggi bufferizzati.

State Transfer: Esempio



Confronto delle tecniche

	PRIMARY BACKUP	ACTIVE REPLICATION
Approccio	Centralizzato: Primary ha un ruolo di coordinatore	Completamente distribuito: Tutte le repliche hanno lo stesso ruolo
Repliche		deterministiche
Guasto di una replica	<ol style="list-style-type: none"> 1. Trasparente se la replica è un backup 2. Se la replica è il primary allora la latenza sperimentata dal client può essere eccessiva: non accettabile per applicazioni REAL-TIME 	un guasto di una replica è sempre trasparente al client
Uso di risorse	Minore dell'Active Replication	Maggiore del Primary Backup

Esercizio

- Il sistema è costituito da un numero finito di processi
 - Ogni processo è dotato di
 - un Perfect Failure detector (P),
 - una primitiva di BestEffortBroadcast (beb)
 - Primitiva di Uniform Consensus (uc)

 - Modello di guasto: crash-stop (Un processo può guastarsi per crash e non può ripristinarsi)

- Definire un algoritmo che implementi una primitiva di view synchronous multicast che sia uniforme:
 - Nessuna coppia di processi (corretti o no) installa view differenti
 - Ogni messaggio consegnato da un processo (corretto o no) viene consegnato da ogni processo corretto

Soluzione: 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.

Soluzione: 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.

Soluzione: 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.

Soluzione

```
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  $\langle \text{bebBroadcast} \mid [\text{DATA}, \text{current-view.id}, m] \rangle$ ;
```

Quando un processo invoca la primitiva di *vsBroadcast* per inviare un messaggio m:

- m viene inserito nel buffer delivered
- m è inviato agli altri processi attraverso la primitiva di bebBroadcast

Consegna dei msg di una 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  $\langle \text{bebBroadcast} \mid [\text{DATA}, \text{current-view.id}, m] \rangle$ ;
```

- Quando un processo riceve un m inviato nella view corrente, aggiunge il processo mittente p_i in ack_m (p_i ha messo m in delivered)
- Mette m in delivered se già non c'era
- Manda in bebBroadcast m

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

- Quando tutti i processi nella view corrente hanno bufferizzato m , posso consegnarlo rispettando le specifiche di uniformità

Creazione di una nuova view (1)

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

- Quando un processo p si accorge che un altro processo q si è guastato deve invocare la creazione di una nuova vista che escluda q dall'attuale gruppo
- Parte una fase di flushing in cui i processi appartenenti alla vista corrente si scambiano I messaggi bufferizzati in delivered
- Si blocca l'invio di nuovi messaggi nella vista attuale

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

Creazione di una nuova view (2)

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

- ogni processo raccoglie i msg bufferizzati da tutti i processi corretti nella view corrente e poi fa partire un'istanza di uniform consensus
- Il processo propone una view:
 - con id pari a quello della view corrente +1
 - Il cui M è il set di processi corretti secondo la sua conoscenza
 - Propone un set di msg da consegnare prima di consegnare la nuova view

Consegna della nuova 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  $\langle vsView \mid current-view \rangle$ ;
```

- Prima di consegnare la nuova vista, ogni processo deve consegnare ogni messaggio appartenente al delivered set deciso dall'istanza di consenso