



*Lezione 2 -
Comunicazione di
Gruppo e
Consistenza*

Prof. Esposito Christian

*Corso di
Sicurezza dei Dati*

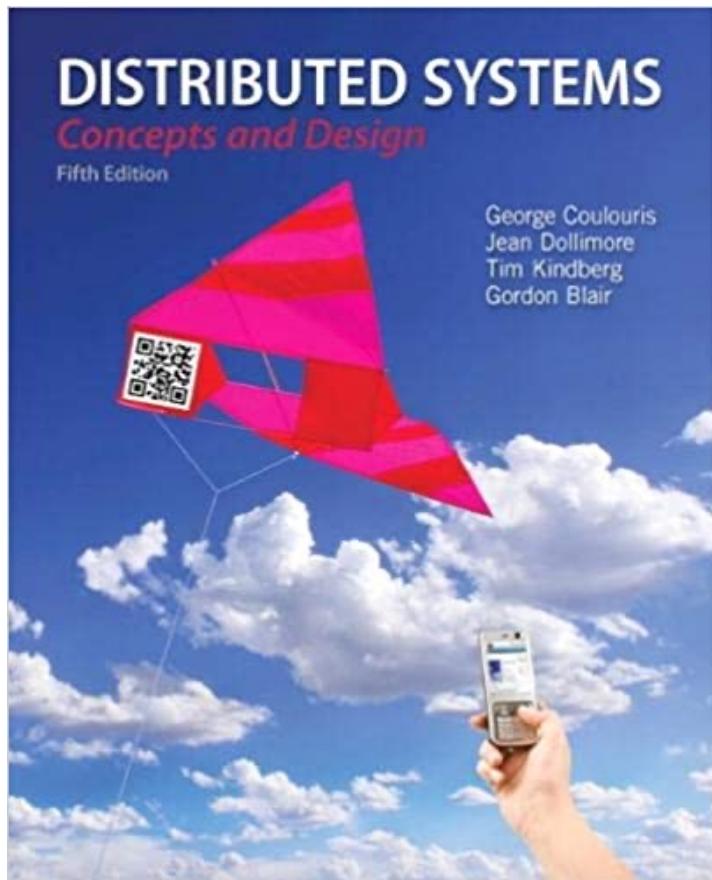


::: Sommario

- Primitive di Comunicazione di Gruppo
 - Reliable multicast ed ordinamenti;
 - Flooding & Gossiping.
- Failure Detectors
 - Classificazione ed implementazioni;
 - Consenso con failure detectors.
- Consistenza dei dati
 - Replicazione e Consistenza;
 - Modelli di Consistenza e Teorema CAP.

::: References

- G. Coulouris et al., “Distributed Systems: Concepts and Design”, V Edizione (Aprile 27, 2011) capitoli 15, 18.



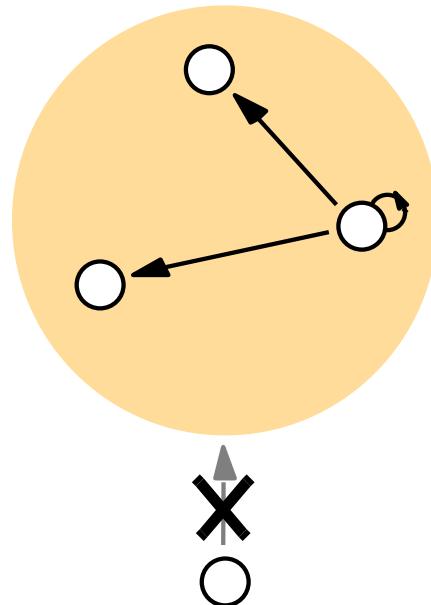


Primitive di Comunicazione di Gruppo

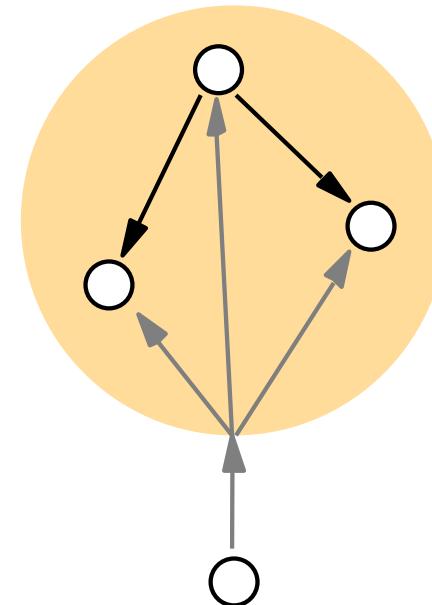
::: Comunicazioni di Gruppo (1/2)

Gruppo chiuso: solo i membri possono inviare messaggi in *multicast* al gruppo

Gruppo aperto: anche i non membri possono inviare messaggi in *multicast* al gruppo



Gruppo Chiuso



Gruppo Aperto

::: Comunicazioni di Gruppo (2/2)

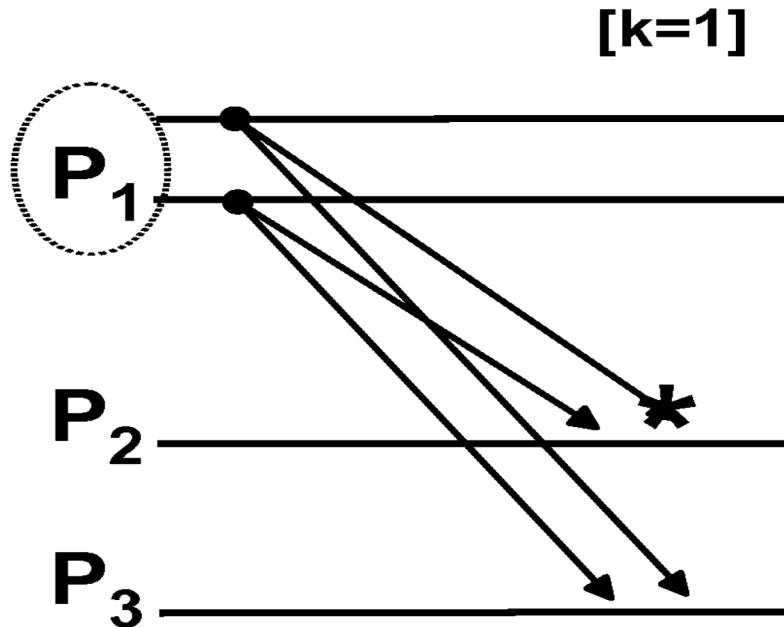
■ Obiettivo

- Assicurare che i processi possano scambiare messaggi di gruppo anche in presenza di fallimenti dei canali di comunicazione e/o dei processi stessi.
e/o
- Assicurare che i messaggi inviati a un gruppo siano ricevuti in maniera ordinata

Consegna affidabile dei messaggi

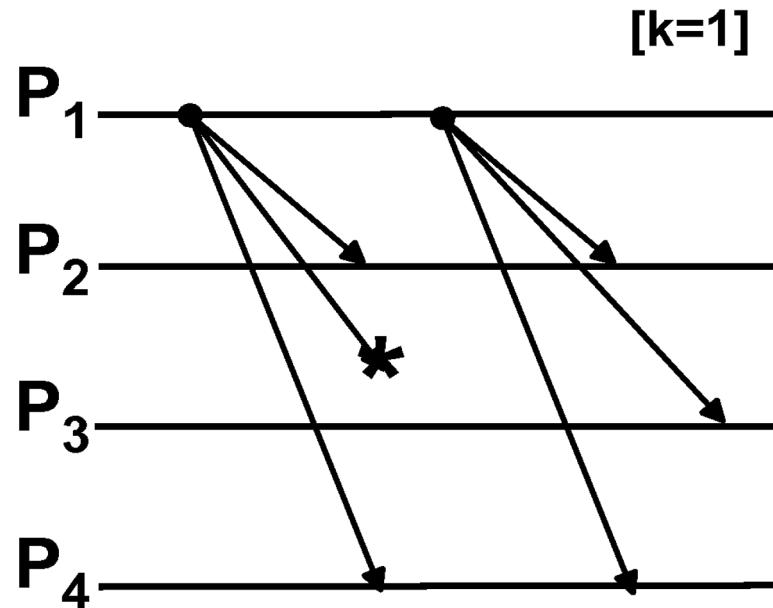
- Principali strategie:
 - ***Error Masking***
 - ridondanza spaziale, ridondanza temporale
 - ***Error Detection and Recovery***
 - basata su *acknowledgements* (acks) e *timeout*

::: Error Masking (1/3)



- **Ridondanza Spaziale:** I processi sono connessi con collegamenti ridondanti.
- Per mascherare k omissioni, occorrono $k+1$ collegamenti.

::: Error Masking (2/3)

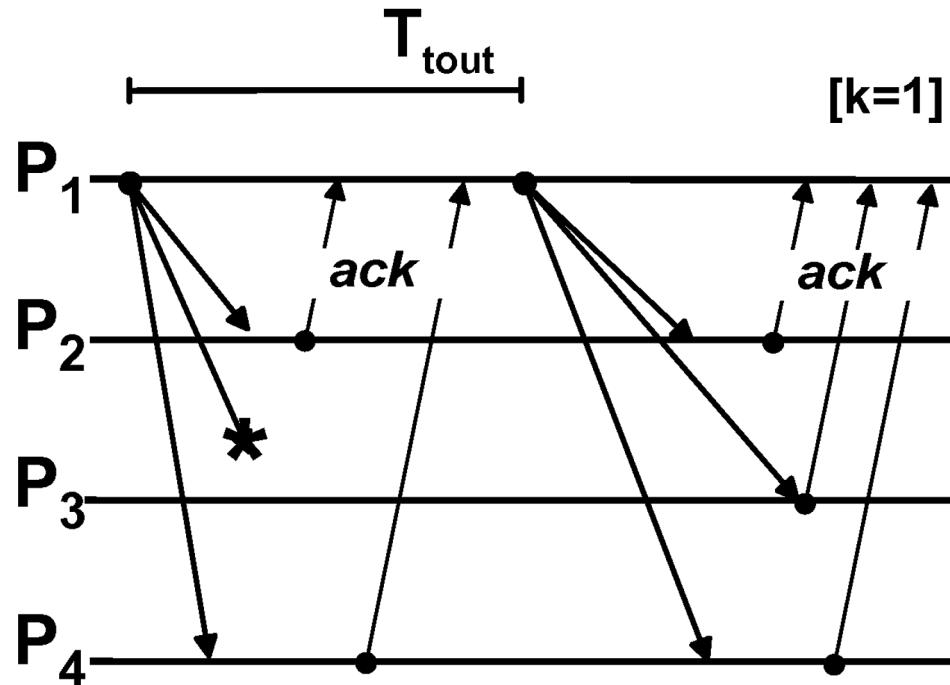


- **Ridondanza temporale:** il messaggio viene inviato più volte
- Per mascherare k omissioni, il messaggio deve essere inviato $k+1$ volte

::: Error Masking (3/3)

- In entrambi i casi si pone il problema dei messaggi duplicati: essi devono essere scartati dal ricevente
- È difficile in generale trovare il giusto compromesso tra semplicità di *recovery* e consumo di banda.

::: Error Detection and Recovery



- Gli *ack* possono essere inviati quando:
 - Un messaggio viene ricevuto (*positive ack*, v. figura)
 - Un *timeout* scade al ricevente, che decreta la perdita di un messaggio (*negative ack*)

::: Schemi per ack dei messaggi

- Nello **schema con *positive ack***, un messaggio viene ritrasmesso se non viene ricevuta una conferma al mittente entro un *timeout* predefinito.
 - La *failure* viene individuata più rapidamente in caso di traffico sporadico.
- Nello **schema con *negative acks***, il ricevente richiede una ritrasmissione inviando un ack negativo
 - Minimizza il traffico di rete ma richiede l'implementazione di uno dei seguenti meccanismi:
 - Numerazione dei messaggi, in maniera tale da poter richiedere la ritrasmissione di un particolare messaggio (se ad esempio viene ricevuto il messaggio i ma non il messaggio i-1)
 - Una strategia time-triggered in maniera tale che il ricevente sappia quando deve ricevere un messaggio.

::: Basic Multicast

- 2 primitive: B-multicast e B-deliver
- Garantiscono che un processo corretto prima o poi riceverà il messaggio, a patto che il mittente non fallisca
- La primitiva di B-multicast può essere implementata utilizzando una primitiva per la comunicazione punto-punto su canale affidabile:
 - $B\text{-multicast}(g, m)$: for each process $p \in g$, $\text{send}(p, m)$
 - $B\text{-receive}(m)$ at p : $B\text{-deliver}(m)$ at p
- Soffre del problema dell'ack-implosion, spreca larghezza di banda, e comunque il mittente può fallire in ogni istante durante l'esecuzione del B-multicast!

::: Reliable Multicast

- Hadzilacos e Toueg (1994), Chandra e Toueg (1996)
- Soddisfa le seguenti proprietà:
 - **Integrity** – un processo corretto p riceve m al più una volta. Inoltre, $p \in \text{group}(m)$ e m è stato inviato via multicast da $\text{sender}(m)$
 - **Validity** – se un processo corretto p invia m in multicast, prima o poi riceverà m
 - **Agreement** – se un processo corretto riceve m , allora tutti i processi corretti in $\text{group}(m)$ prima o poi riceveranno m

::: Uniform Reliable Multicast

- Le definizioni delle proprietà del *reliable multicast* fanno riferimento a processi corretti (cioè che non falliscono mai).
- Una proprietà valida a prescindere dal fatto che i processi siano corretti o meno si dice **uniforme**.
- In particolare si ha:
 - **Uniform agreement** – se un processo (corretto o meno) riceve m , allora tutti i processi corretti in $\text{group}(m)$ prima o poi riceveranno m

::: R-Multicast con B-Multicast (1/3)

On initialization

Received := {};

For process p to R-multicast message m to group g

B-multicast(g, m); // p ∈ g is included as a destination

On B-deliver(m) at process q with g = group(m)

if (m ∈ Received)

then

Received := Received ∪ {m};

if (q ≠ p) then B-multicast(g, m); end if

R-deliver m;

end if

::: R-Multicast con B-Multicast (1/3)

L'implementazione soddisfa le proprietà del *reliable multicast*:

Integrità: discende dall'integrità dei canali di comunicazione

Validità: è evidente che un processo corretto prima o poi fa il *B-deliver* a sé stesso

Accordo: discende dal fatto che ogni processo corretto esegue *B-multicast* dopo *B-deliver*. Se dunque un processo corretto non effettua *R-deliver*, ciò può essere dovuto solo al fatto che non ha fatto *B-deliver*; ciò può sussistere solo se nessun altro processo corretto ha fatto il *B-deliver*: in tal caso nessuno farà *R-deliver*

- **Questa implementazione è corretta in un sistema asincrono, ma inefficiente (il messaggio è inviato lgl volte a ciascun processo)**

::: R-Multicast con B-Multicast (3/3)

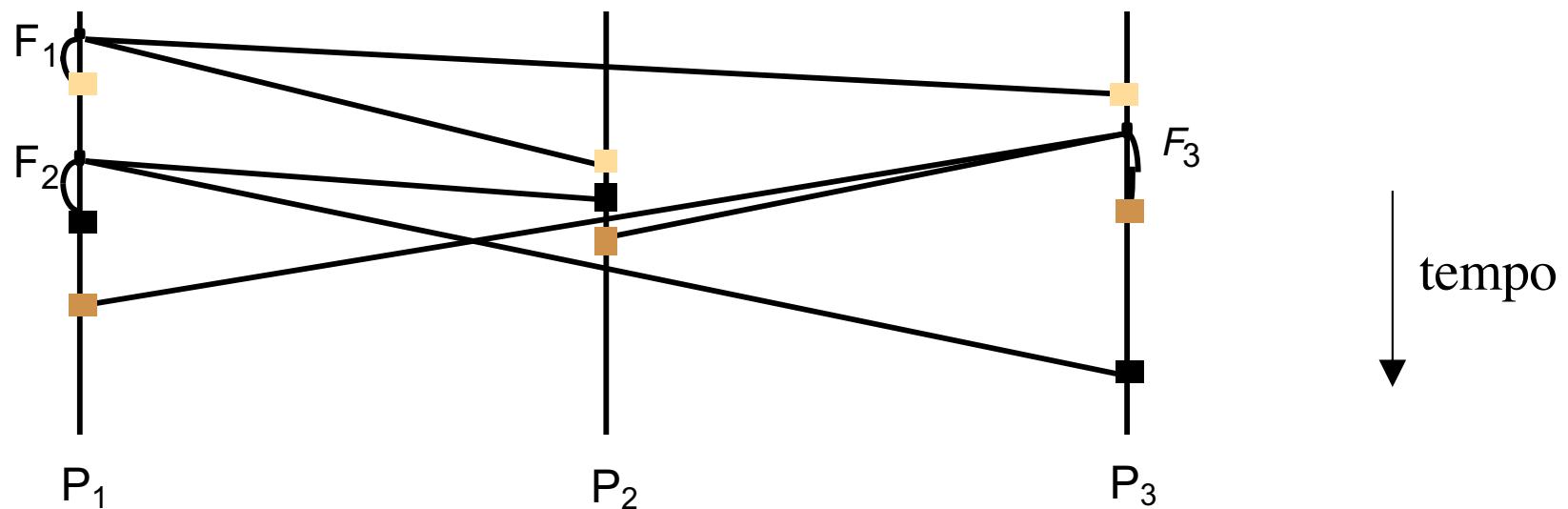
L'implementazione soddisfa anche la proprietà di *uniform agreement*

Se ad esempio un processo non è corretto e va in *crash* dopo aver effettuato *R-deliver*, poiché in precedenza ha sicuramente effettuato *B-deliver*, nondimeno dunque tutti i processi corretti riceveranno il messaggio.

N.B.: Se si invertono le istruzioni ‘*R-deliver m*’ e ‘*if($q \neq p$) then B-multicast(g, m) endif*’, l’implementazione non soddisfa più la proprietà di *uniform agreement*.

::: Ordinamento FIFO

- Se un processo corretto invia in *multicast* m e poi m' , allora ogni processo corretto che riceverà m' avrà già ricevuto m
- Versione uniforme: idem senza ‘corretto’

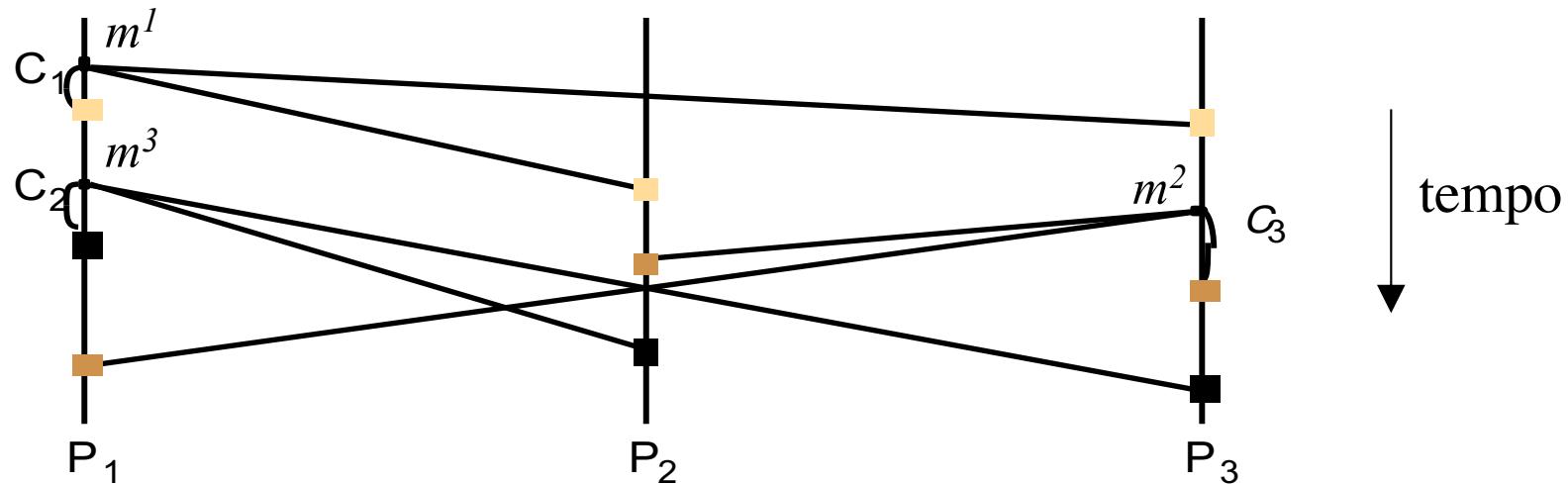


::: Ordinamento Causale (1/2)

- L'ordinamento causale implica quello FIFO, poiché gli invii di uno stesso processo sono in relazione Happened Before (HB): ordinamento degli eventi in base alla potenziale relazione causale di coppie di eventi in un sistema asincrono.
 - Se gli eventi a e b si verificano sullo stesso processo, se il verificarsi dell'evento a ha preceduto il verificarsi dell'evento b , allora $a \rightarrow b$ o $a \text{ HB } b$.
 - Se a è l'invio di un messaggio e b è la ricezione del messaggio, allora $a \rightarrow b$ o $a \text{ HB } b$.
 - Se due eventi accadono in diversi processi isolati, allora si dice che i due processi sono concorrenti, allora è vero sia $a \rightarrow b$ che $b \rightarrow a$.
- L'ordinamento causale estende l'ordinamento FIFO con l'ordinamento della ricezione dei messaggi che, pur inviati da processi diversi, sono in relazione di potenziale causalità
 - Se un processo p invia in *multicast* un messaggio m ed un processo $p' \neq p$ riceve m prima di inviare in multicast m' , allora nessun processo corretto riceve m' a meno che non abbia già ricevuto m

::: Ordinamento Causale (2/2)

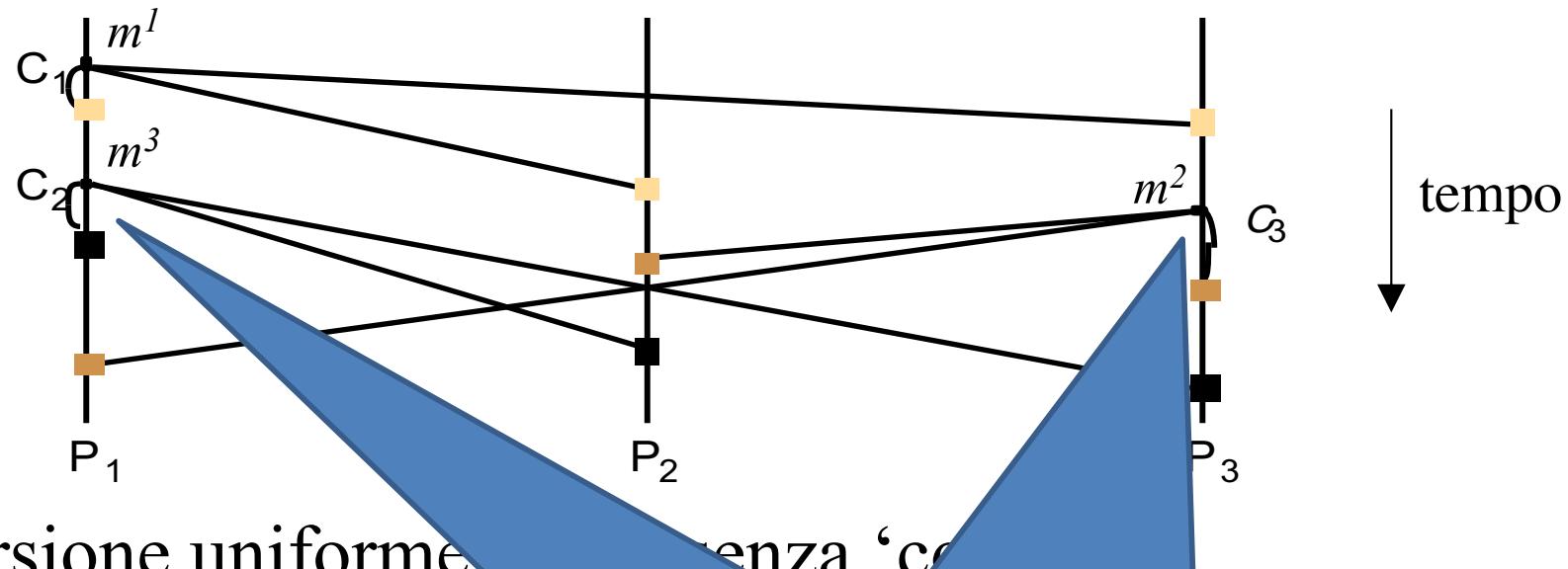
- Se $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$, dove \rightarrow è la relazione *happened-before* nell'ambito del gruppo g , allora ogni processo corretto che riceverà m' avrà già ricevuto m



- Versione uniforme: idem senza ‘corretto’

::: Ordinamento Causale (2/2)

- Se $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$, dove \rightarrow è la relazione *happened-before* nell'ambito del gruppo g , allora ogni processo corretto che riceverà m' avrà già ricevuto m

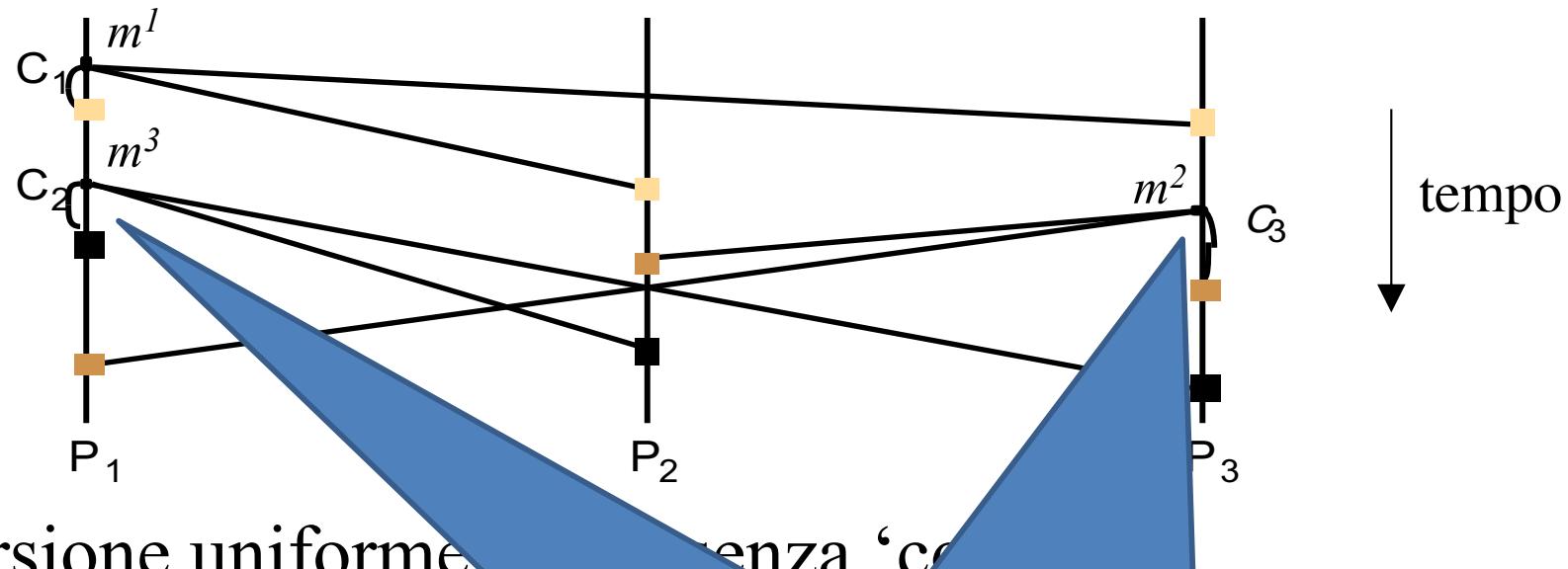


- Versione uniforme senza ‘causalità’

Se non sussiste sincronismo dei clock delle macchine, non è possibile sempre essere d'accordo sull'ordine in cui le cose accadono.

::: Ordinamento Causale (2/2)

- Se $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$, dove \rightarrow è la relazione *happened-before* nell'ambito del gruppo g , allora ogni processo corretto che riceverà m' avrà già ricevuto m

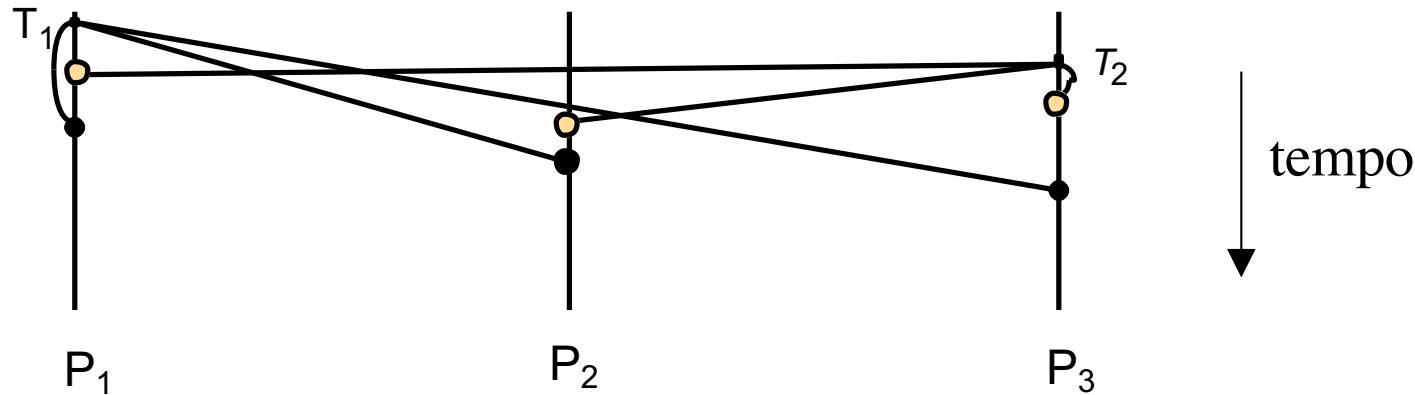


- Versione uniforme: senza ‘causalità’

L'unico modo per sapere con certezza è se qualcosa collega questi due eventi.

::: Ordinamento Totale

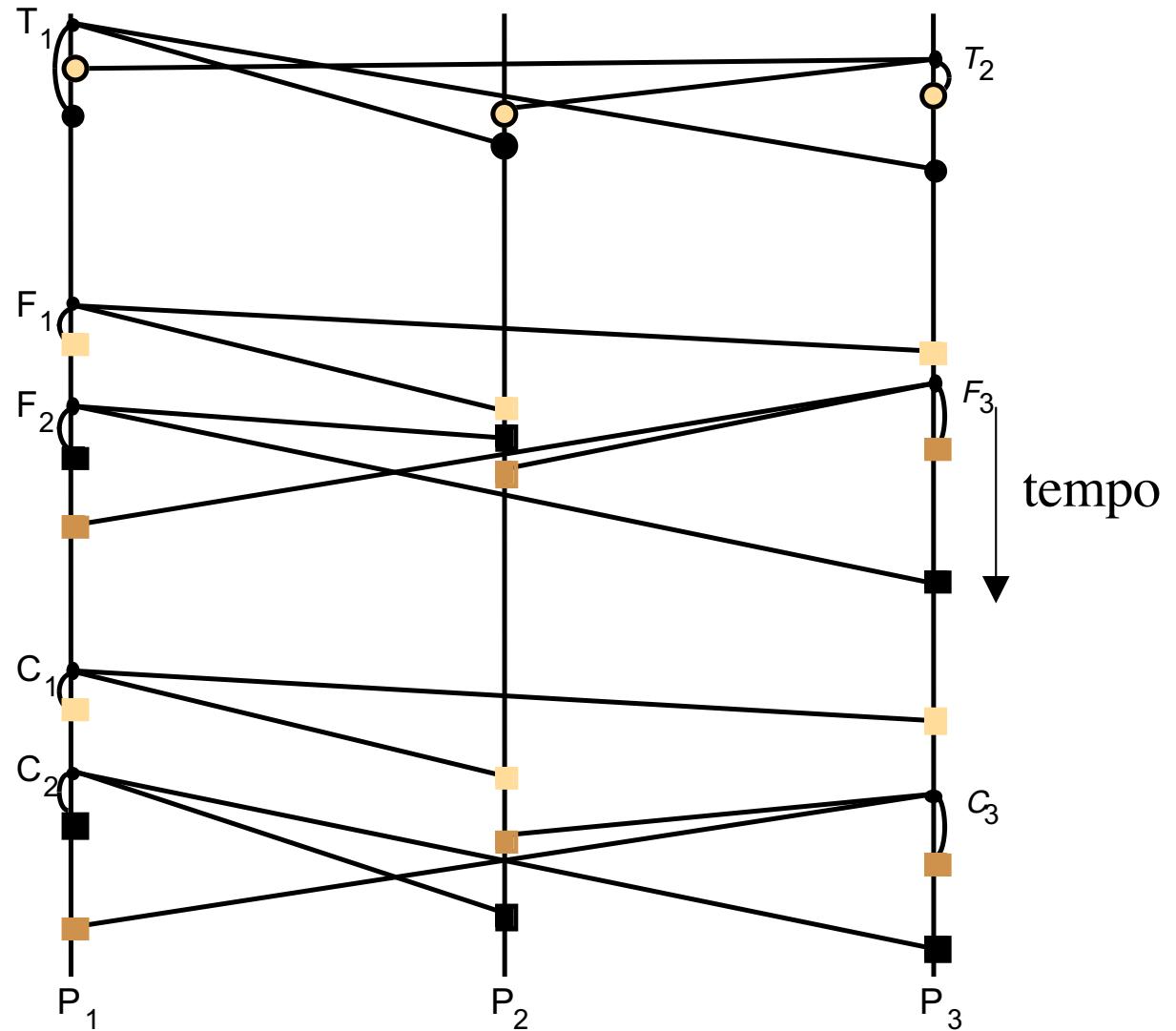
- Se un processo corretto riceve i messaggi m e poi m' , allora ogni altro processo corretto che riceve m' avrà già ricevuto m



- *Uniform Version*: idem senza ‘corretto’
- L’ordinamento totale non implica quello FIFO né quello causale

::: Confronto degli Ordinamenti

- NB: nell'esempio i messaggi TO sono ricevuti nell'ordine opposto rispetto all'invio: la definizione di TOM richiede infatti che l'ordine sia lo stesso per tutti, ma può essere arbitrario



::: IP Multicast (1/2)

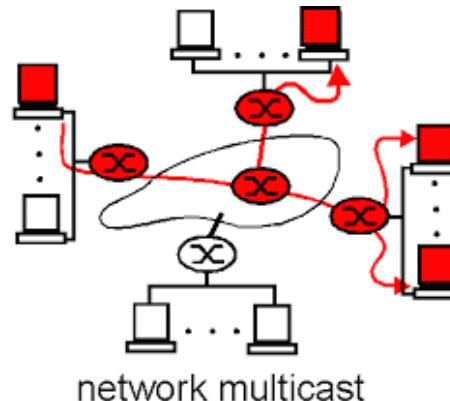
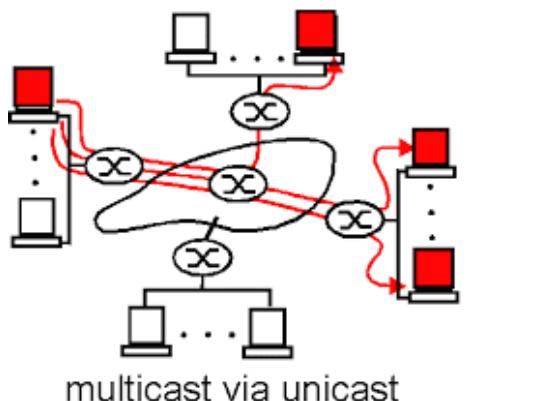
La soluzione più comune per realizzare un sistema di comunicazione di gruppo è di impiegare IP Multicast.

IP Multicast è definito al di sopra del protocollo IP, ed un'applicazione può effettuare comunicazioni multicast solo attraverso l'invio di datagrammi UDP.

IP Multicast

IP

Un gruppo multicast viene identificato per mezzo di indirizzi di classe D assegnati dalla Internet authority nell'intervallo 224.0.0.1 – 224.0.0.255.



I router sono responsabili dell'opportuna replicazione e istradamento dei pacchetti per la consegna ai membri attraverso il Protocol-Independent Multicast.

::: IP Multicast (2/2)

IP Multicast rappresenta un'ottima soluzione per comunicazioni di gruppo in LAN, ma presenta limiti su WAN:

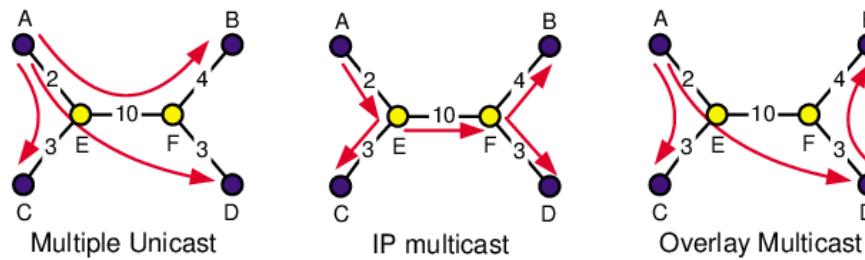
1. Gli ISP non consentono traffico IP Multicast per ridurre il carico sui router e proteggersi da traffico indesiderato;
2. IP Multicast richiede aggiornamenti all'attuale infrastruttura di rete, che non sono facilmente sostenibili dal business model degli attuali ISP;
3. IP Multicast viola il principio di stateless del protocollo IP, siccome i router devono mantenere informazioni sulla membership dei gruppi;
4. IP Multicast introduce forti complessità e limiti di scala su scenari di ampia scala come Internet.

Tali motivi hanno portato alla definizione di un nuovo approccio alla comunicazione di gruppo.

::: Application-Level Multicast (1/3)

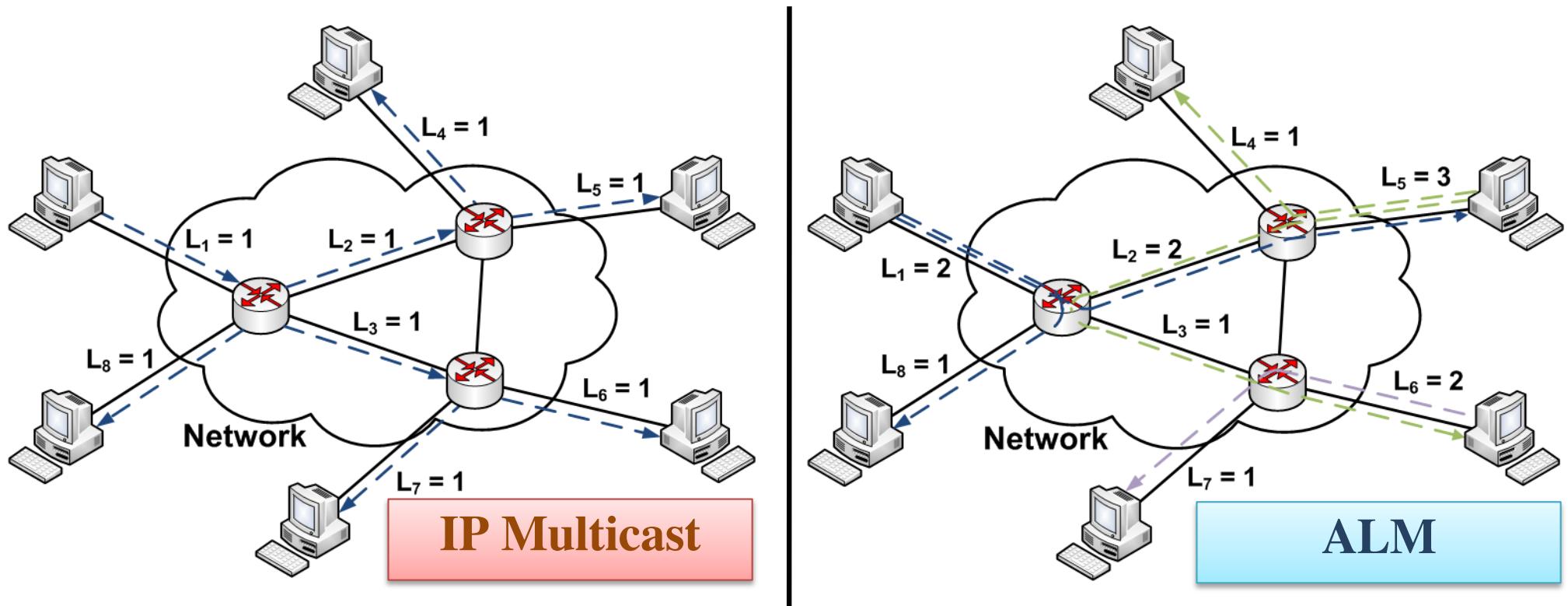
Una soluzione per ovviare agli inconvenienti di IP Multicast è quello di implementare le funzionalità di multicasting a livello applicativo piuttosto che a livello trasporto dello stack protocolare ISO/OSI:

Le applicazioni sono interconnesse per mezzo di una overlay network, gestiscono i meccanismi di membership al gruppo e si fanno carico delle operazioni di replicazione e disseminazione dei pacchetti.



L'interconnessione al livello overlay può essere strutturata come un albero, oppure non strutturata come una mesh. L'uso di una struttura ad albero implica una maggiore efficienza nella disseminazione, ma anche più vulnerabilità ai fallimenti dei nodi overlay.

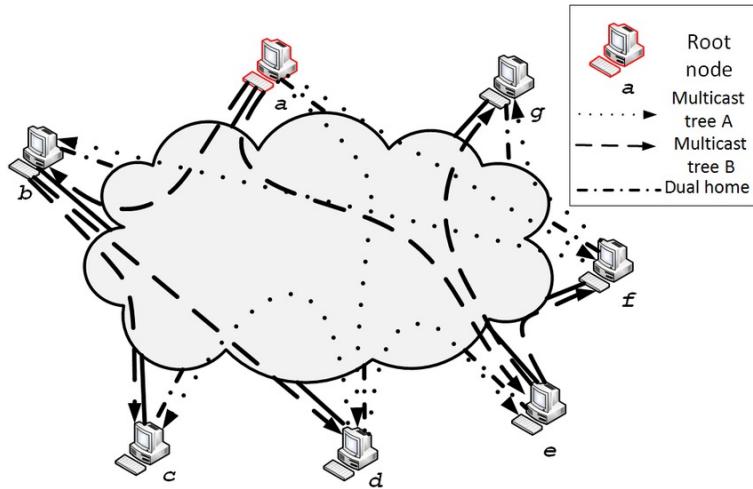
::: Application-Level Multicast (2/3)



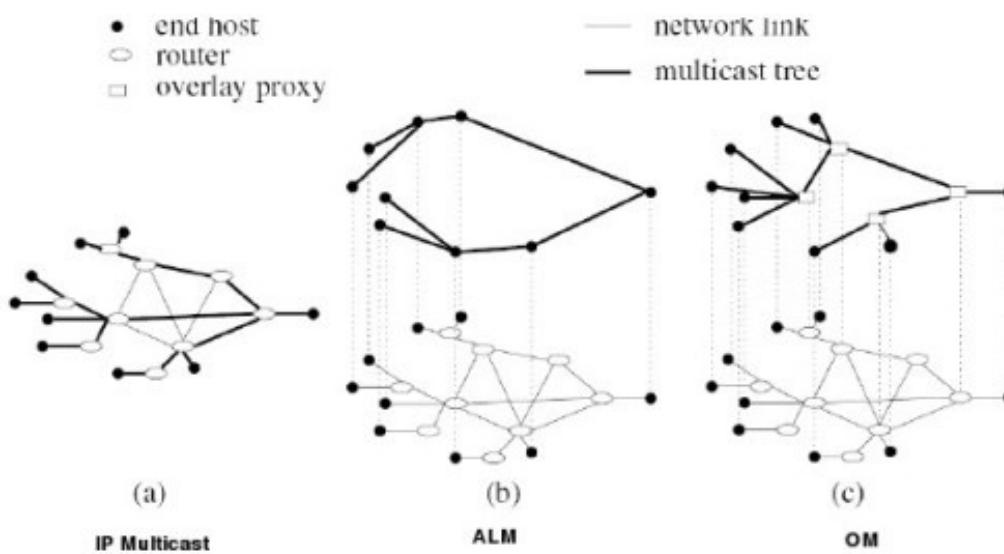
L'efficienza (η) è misurata come il numero medio di pacchetti scambianti sui link durante la disseminazione di un messaggio in multicast.

$$\eta_{\text{IP_Multicast}} = 1 \text{ mentre } \eta_{\text{ALM}} < 1$$

... Application-Level Multicast (3/3)



ALM indica il caso in cui i nodi dell'Overlay sono realizzati dagli stessi computer situati presso gli utenti finali. Il risultato è che la distribuzione ottenibile è caratterizzata da una topologia spesso molto lontana dal caso ideale.



Il termine Overlay Multicast è usato per identificare il caso in cui i nodi dell'overlay sono disposti direttamente nella core network, permettendo di realizzare un albero di distribuzione molto più efficiente.

::: Implementazione R-Multicast

- Il protocollo multicast affidabile può essere eseguito su UDP e utilizza il servizio multicast IP per la consegna dei pacchetti.
- Poiché sia IP multicast che UDP sono protocolli inaffidabili, l'affidabilità si ottiene eseguendo un protocollo affidabile end-to-end a livello di applicazione. Per ottenere l'affidabilità, i pacchetti con errori o pacchetti persi verranno ritrasmessi, utilizzando un protocollo a finestra scorrevole basato sul feedback delle stazioni.
- Nel contesto dell'ALM oltre alla ridondanza temporale realizzata dalle ritrasmissioni, è possibile adottare anche ridondanza spaziale:
 - Impiegando percorsi multipli con copie dei messaggi istradati nei vari percorsi;
 - Usando tecniche di codifica per ottenere informazioni addizionali e ricostruire i pacchetti persi dalla rete.

::: Implementazione Ordinamento FIFO

- L'ordinamento si realizza mediante l'utilizzo di *numeri di sequenza*
- Assunzione: i gruppi non si sovrappongono
- S^p – contatore dei messaggi inviati dal processo p al gruppo g
 R^q - numero di sequenza dell'ultimo messaggio delivered da p , inviato a g da q .
- $Hold\text{-}Back\text{-}Queue(p)$ – coda dei messaggi fuori sequenza in attesa di essere ricevuti da p

To FO-multicast(m,g):

piggy-.back S^p onto m ;

$B\text{-}multicast(m,g)$;

$S^p = S^p + 1$;

To FO-deliver(q,m) with m bearing
 $S = R^q + 1$:

$FO\text{-}deliver(q,m)$;

$R^q = S$;

To FO-deliver(q,m) with m bearing
 $S > R^q + 1$:

put m in $Hold\text{-}Back\text{-}Queue$;

wait until $S(m) = R^q + 1$;

$FO\text{-}deliver(q,m)$;

::: Implementazione FIFO R-Multicast

- Si può usare la stessa implementazione del FIFO multicast, usando *R-multicast* al posto di *B-multicast*

To FO-multicast(m,g):

piggy-.back S^p onto m ;

R-multicast(m,g);

$S^p = S^p + 1$;

**To FO-deliver(q,m) with m bearing
 $S = R^q + 1$:**

FO-deliver(q,m);

$R^q = S$;

**To FO-deliver(q,m) with m bearing
 $S > R^q + 1$:**

put m in Hold-Back-Queue;

wait until $S(m) = R^q + 1$;

FO-deliver(q,m);

.... Implementazione Ordinamento Causale (1/3)

- L'implementazione tiene conto solo delle relazioni HB tra i messaggi *multicast*, non di quelle tra messaggi diretti (*one-to-one*) scambiati tra i processi
- Si può far uso di orologi vettoriali (*Vector Clocks*)
 - l'elemento *i-esimo* conta il numero di messaggi inviati dal processo *i-esimo* in relazione HB con il prossimo messaggio
- Assunzione: gruppi chiusi non sovrapposti
- Per effettuare il *CO-multicast*, un processo incrementa il “proprio” elemento nel vettore ed esegue *B-multicast* al gruppo *g* del messaggio marcato col *vector clock*

.... Implementazione Ordinamento Causale (2/3)

- Quando p_i fa il *B-deliver* di un messaggio inviato da p_j , prima di effettuare il *CO-deliver* deve inserirlo nella *hold-back queue* finché non ha fatto il *CO-deliver* di tutti i messaggi in relazione HB con esso
- P_i deve perciò verificare di aver fatto:
 - il *CO-deliver* di tutti i messaggi precedenti inviati da p_j , e:
 - il *CO-deliver* di tutti i messaggi di cui p_j aveva fatto il *CO-deliver* prima di inviare il messaggio in questione
- Queste condizioni possono essere verificate esaminando i *vector timestamps* dei messaggi ricevuti

.... Implementazione Ordinamento Causale (3/3)

Algorithm for group member p_i ($i = 1, 2, \dots, N$)

On initialization

$$V_i^g[j] := 0 \quad (j = 1, 2, \dots, N);$$

To CO-multicast message m to group g

$$V_i^g[i] := V_i^g[i] + 1;$$

B-multicast($g, <V_i^g, m>$);

NB: il *CO-deliver* di un proprio messaggio può essere effettuato immediatamente

On B-deliver($<V_j^g, m>$) from p_j , with $g = \text{group}(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ ($k \neq j$);

CO-deliver m ; // after removing it from the hold-back queue

$$V_i^g[j] := V_i^g[j] + 1;$$

::: Implementazione Causal R-Multicast

- Il multicast con ordinamento causale affidabile può essere implementato a partire dalla implementazione del *Causal Multicast*, usando *R-multicast* al posto di *B-multicast*

.... Implementazione Ordinamento Totale (1/4)

- Basata sulla marcatura dei messaggi con identificatori totalmente ordinati, in modo che tutti i processi possano prendere le stesse decisioni
- Il *delivery* è come per l'ordinamento FIFO, ma i numeri di sequenza si riferiscono al gruppo, non al processo
- Primo metodo: implementazione mediante processo sequenziatore
 - un messaggio m è inviato sia a g sia a $sequencer(g)$, etichettato con un id univoco $id(m)$
 - $sequencer(g)$ può coincidere con un membro di g
 - $sequencer(g)$ gestisce un numero di sequenza per il gruppo s_g , con cui marca i messaggi di cui effettua *B-deliver*
 - $sequencer(g)$ annuncia i numeri di sequenza inviando messaggi di ordinamento a g tramite *B-multicast*
 - un messaggio resta nella *hold-back queue* finché non può esserne effettuato il *TO-deliver*, in base al suo numero di sequenza

.... Implementazione Ordinamento Totale (2/4)

1. Algorithm for group member p

On initialization: $r_g := 0;$

To TO-multicast message m to group g

$B\text{-multicast}(g \cup \{\text{sequencer}(g)\}, \langle m, i \rangle);$

On B-deliver($\langle m, i \rangle$) with $g = \text{group}(m)$

Place $\langle m, i \rangle$ in hold-back queue;

On B-deliver($m_{\text{order}} = \langle \text{"order"}, i, S \rangle$) with $g = \text{group}(m_{\text{order}})$

wait until $\langle m, i \rangle$ in hold-back queue and $S = r_g$;

TO-deliver m ; // (after deleting it from the hold-back queue)

$r_g = S + 1;$

TO-multicast
con sequenziatore

2. Algorithm for sequencer of g

On initialization: $s_g := 0;$

On B-deliver($\langle m, i \rangle$) with $g = \text{group}(m)$

$B\text{-multicast}(g, \langle \text{"order"}, i, s_g \rangle);$

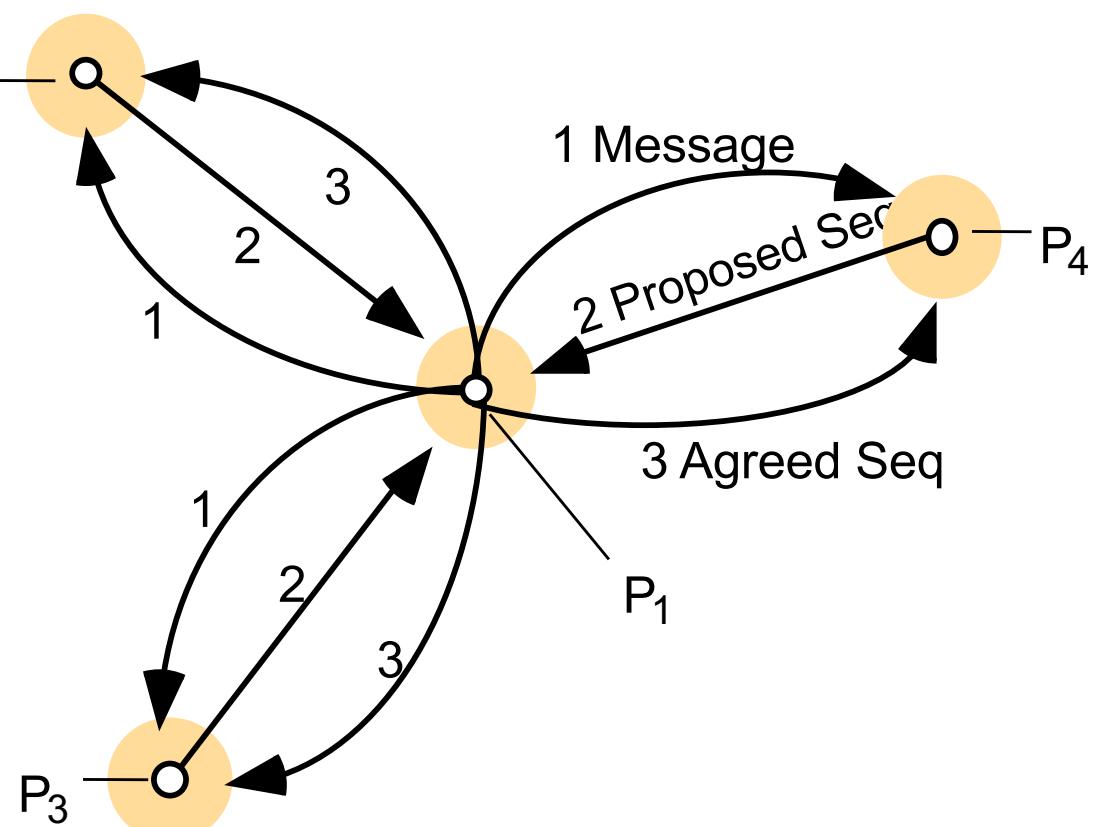
$s_g := s_g + 1;$

.... Implementazione Ordinamento Totale (3/4)

- Secondo metodo: Algoritmo ISIS (Birman, 1987), valido per gruppi aperti o chiusi
- I processi concordano collettivamente sui numeri di sequenza, con un algoritmo distribuito

Ogni ricevente propone il numero di sequenza al *multicaster*, che genera il numero “*agreed*”

Ogni processo q di g possiede:
 A^q_g : il più alto numero di sequenza finora convenuto
 P^q_g : il più alto numero di sequenza finora proposto dallo stesso q



.... Implementazione Ordinamento Totale (4/4)

Ogni ricevente propone:

$$P_g^q := \text{Max}(A_g^q, P_g^q) + 1$$

e assegna temporaneamente al messaggio il numero proposto, inserendolo nella *hold back queue*; questa è ordinata con il valore più piccolo in testa.

Il *multicaster* raccoglie le proposte e seleziona il valore maggiore a , quindi effettua $B\text{-multicast} < i, a >$ (i è l'id univoco con cui il multicaster ha etichettato m)

Ogni ricevente assegna:

$$A_g^q := \text{Max}(A_g^q, a)$$

e associa il valore a al messaggio, quindi riordina la *hold-back queue* se il valore convenuto è diverso da quello proposto.

Quando al messaggio in testa alla coda viene assegnato il numero *agreed*, viene inserito in una *delivery queue*.

::: Implementazione Causal Total Multicast

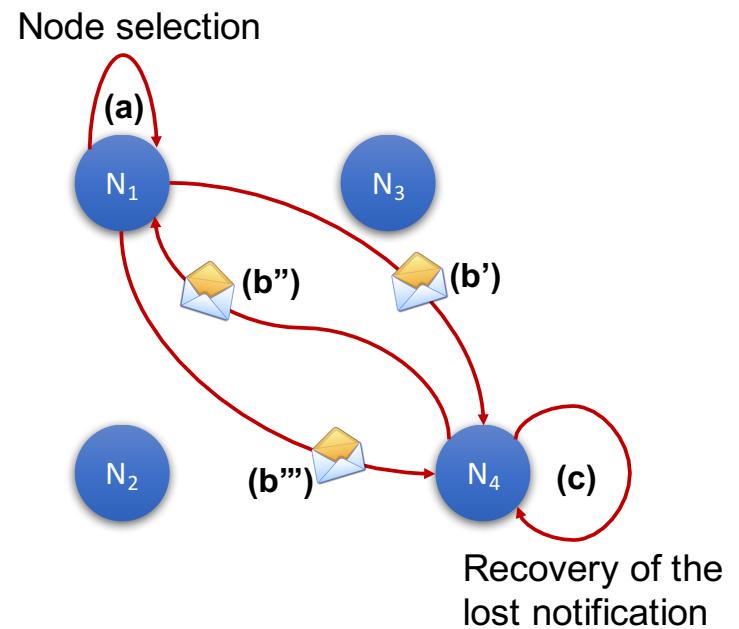
- Se si combina l'algoritmo per l'ordinamento causale (CO) con quello per il *multicast* totalmente ordinato (TO) basato sul *sequencer* si ottiene un ordinamento che è ordinato sia causalmente sia totalmente (C&TO)
 - il sequenziatore deve fare il *delivery* secondo l'ordinamento causale, e il *multicast* dei numeri di sequenza dei messaggi nell'ordine con cui li riceve
 - i processi nel gruppo di destinazione non effettuano il *deliver* prima di aver ricevuto un messaggio *order* dal sequenziatore e il messaggio è il prossimo nella sequenza
 - in tal modo, poiché il *sequencer* effettua il *deliver* in ordine causale, e tutti gli altri processi lo effettuano nell'ordine del *sequencer*, l'ordinamento è sia totale sia causale

::: Gossiping (1/3)



Il gossiping rappresenta un algoritmo distribuito per poter garantire la consegna di messaggi sebbene fallimenti di link, processo e di rete si possano verificare nel sistema. Realizza una soluzione di flooding selettivo.

- A- Periodicamente, ogni processo decide di inviare un messaggio a un insieme k di interlocutori scelti in maniera casuale (k è un parametro dell'algoritmo detto fan-out).
- B - Sulla base dei messaggi scambiati, ritrasmissioni vengono attivate.
- C- Messaggi persi vengono recuperati.



::: Gossiping (2/3)

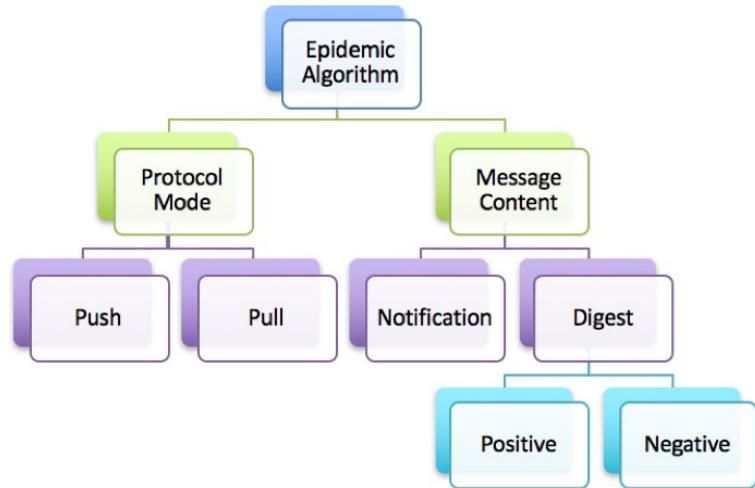
Esistono due modalità di interazione tra i partecipanti a un algoritmo di gossiping:

- Push style: il gossiper invia l'informazione a uno o più destinatari scelti casualmente;
- Pull style: il gossiper interroga uno o più destinatari richiedendo l'invio di un'informazione.

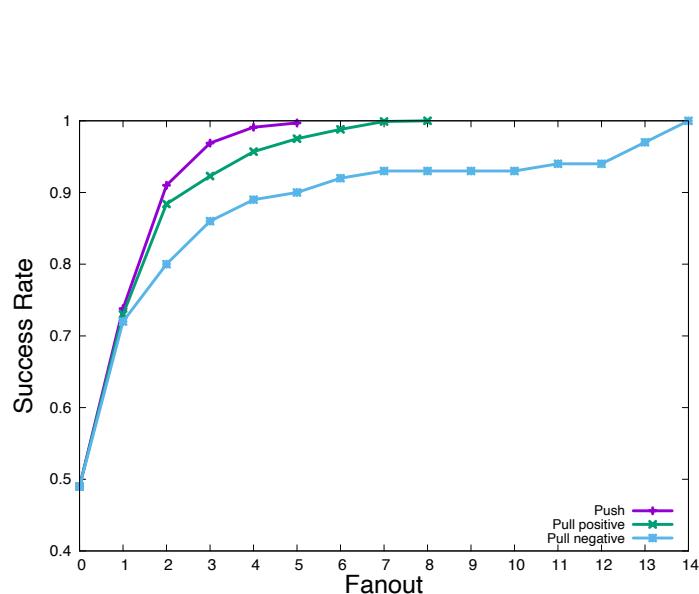
L'informazione che viene scambiata, o digest, può essere di due tipi:

- Positive: il messaggio di gossiping contiene gli identificativi dei messaggi correttamente ricevuti;
- Negative: il messaggio di gossiping contiene gli identificativi dei messaggi noti per essere stati persi.

::: Gossiping (3/3)



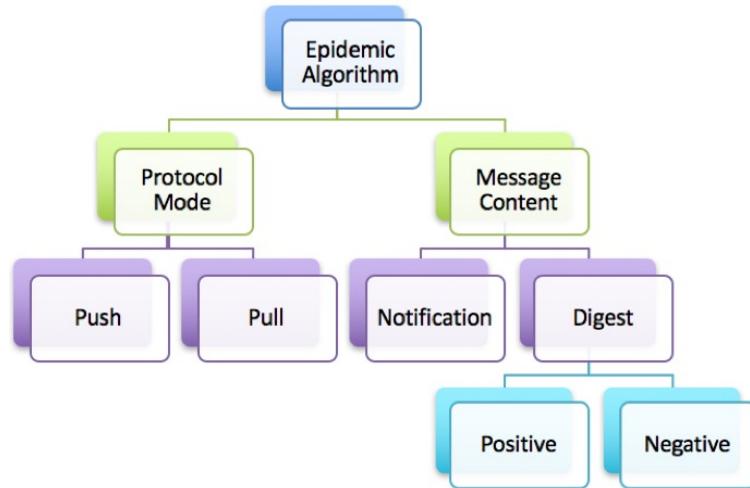
Sono consentite tutte le possibili combinazioni, ma nella pratica nel caso di una strategia di pull si usa un digest negativo, mentre per quella di push uno positivo.



Lo schema pull/negative è intrinsecamente reattivo, mentre quello push/positive è implementato in accordo ad uno schema proattivo.

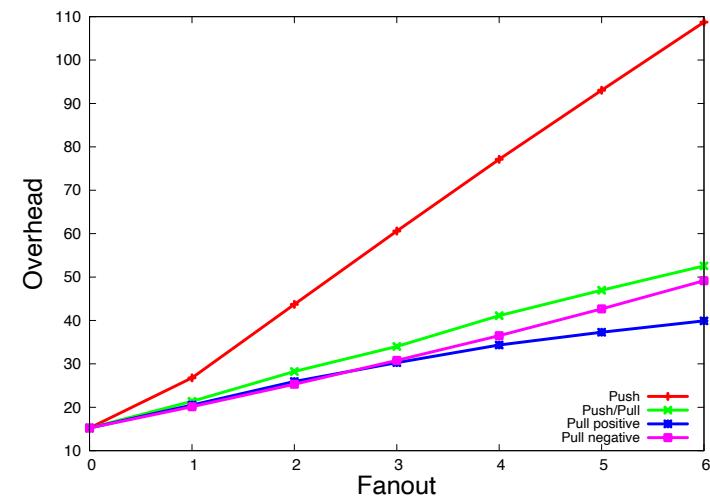
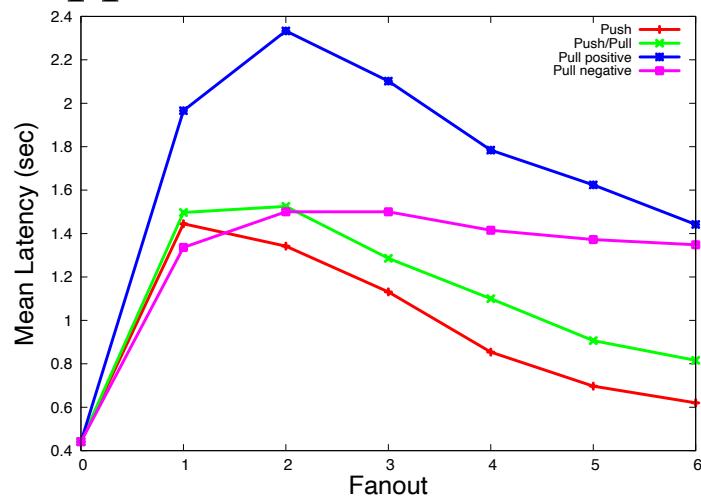
Consente di ottenere la consegna affidabile con una data probability prossima a 1.

::: Gossiping (3/3)



Sono consentite tutte le possibili combinazioni, ma nella pratica nel caso di una strategia di pull si usa un digest negativo, mentre per quella di push uno positivo.

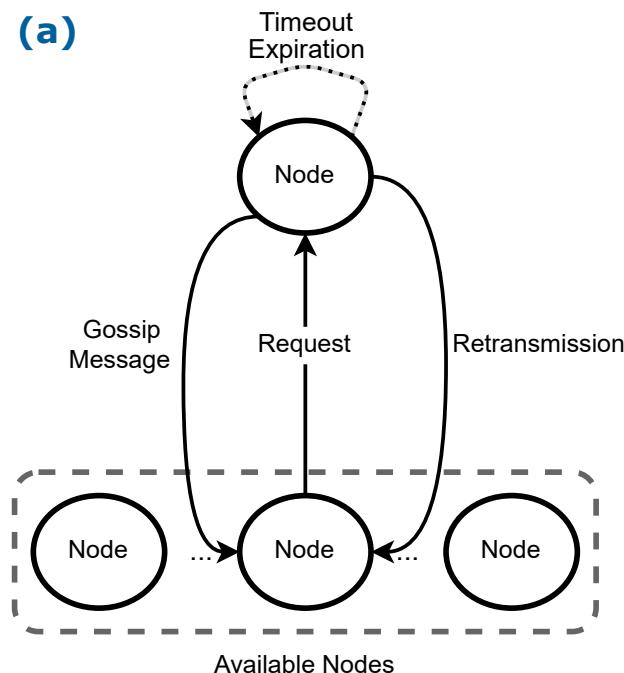
I tempi di latenza e l'overhead di rete dipendono dallo schema applicato.



::: Sicurezza nel Gossiping (1/2)

Uno schema gossiping convenzionale mostra vulnerabilità che possono essere sfruttate in determinati attacchi o per compromettere le applicazioni basate sul gossiping.

Gossiping può essere compromesso al fine di realizzare attacchi di tipo Denial of Service (DoS) per congestionare una porzione di rete o un determinato nodo.

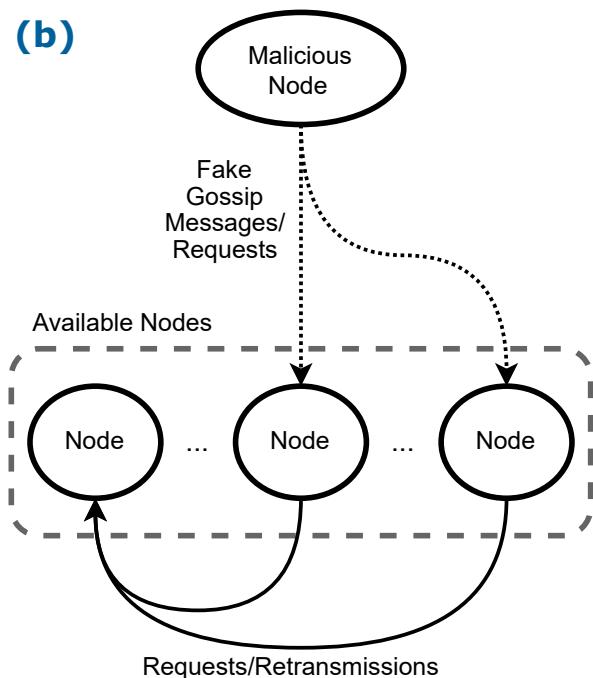


Normalmente la scelta dei nodi verso cui inviare messaggi di gossiping è casuale, oppure in base a determinate metriche di qualità.

::: Sicurezza nel Gossiping (1/2)

Uno schema gossiping convenzionale mostra vulnerabilità che possono essere sfruttate in determinati attacchi o per compromettere le applicazioni basate sul gossiping.

Gossiping può essere compromesso al fine di realizzare attacchi di tipo Denial of Service (DoS) per congestionare una porzione di rete o un determinato nodo.



Una compromissione può causare l'invio di molti messaggi di gossiping verso un nodo, o un sotto-insieme di nodi, con un effetto di «bombardamento».

::: Sicurezza nel Gossiping (2/2)

La protezione contro compromissioni del gossiping è di fondamentale importanza.

- Gli attacchi Sybil vengono evitati usando un servizio di Certificate Authority (CA) che assegna identità ai partecipanti al gossiping in modo che gli avversari malintenzionati non possano falsificare le identità.
- Un'altra soluzione è usare primitive crittografiche per proteggere integrità e autenticità dei messaggi scambiati.
- Un approccio diverso è considerare validi i messaggi di gossiping se ricevuti da un numero elevato di peer.
- Un'alternativa è inviare i messaggi di gossiping a porte conosciute pubblicamente mentre si ritrasmettono a porte selezionate casualmente, e scegliendo casualmente i messaggi da scambiare dai digest per evitare di essere sopraffatti da messaggi fasulli.



Failure Detectors

::: Failure Detector

Uno dei problemi che si riscontra nel progetto di un algoritmo distribuito è decidere *quando* un determinato processo fallisce (con riferimento a fallimenti per *crash*).

Un **failure detector** è un servizio in grado di stabilire se un processo è fallito.

Tale servizio è tipicamente implementato da un insieme di oggetti, ciascuno locale ad uno degli N processi del sistema distribuito.

Ciascun oggetto (detto *local failure detector*) esegue un algoritmo di detection in congiunzione con le sue controparti eseguite localmente ad altri processi.

N.B.: si assume che i processi falliscano solo per *crash*, che abbiano canali di comunicazione affidabili, e che il fallimento di un processo non impedisca ad altri di comunicare.

::: Failure Detector Inaffidabili

In un sistema asincrono, il rilevamento di fallimenti è soggetto ad errori poiché un processo può essere considerato come fallito sebbene questo sia correttamente in esecuzione.

La maggior parte dei *failure detector* sono **inaffidabili**, restituendo uno dei due risultati sullo stato di fallimento di un processo q :

- *Unsuspected*: il detector ha di recente avuto prova che il processo q non è fallito.

Un messaggio è stato ricevuto di recente dal processo; tuttavia, ciò non esclude la possibilità che il processo sia fallito da allora.

- *Suspected*: il detector ha indicazioni sul fatto che il processo q potrebbe essere fallito.

Il processo non riceve messaggi per un tempo maggiore del massimo nominale; tuttavia, il processo potrebbe funzionare correttamente, ma non rispondere a causa ad es. di partizionamenti di rete o eccessivi rallentamenti nell'esecuzione.

Entrambi i risultati sono da considerarsi *suggerimenti*: potrebbero non accuratamente riflettere il reale stato del processo q .

::: Failure Detector Affidabili

Un *failure detector* **affidabile** è sempre accurato nel determinare il *crash* di un processo.

Può restituire i risultati:

- *Unsuspected*: come nel caso di detector inaffidabili, rappresenta solo un suggerimento.
- *Failed*: il detector determina con certezza che il processo q è fallito. Il processo non eseguirà alcuna azione ulteriore (per definizione, un processo fallito per *crash* permane nel suo stato, senza eseguire altri passi di elaborazione).

::: Completezza ed Accuratezza (1/2)

Chandra e Toueg caratterizzano i *failure detector* in termini di completezza e accuratezza.

Completezza (*completeness*)

C’è un istante di tempo dopo il quale ogni processo andato in *crash* è permanentemente sospettato da un processo corretto

Accuratezza (*accuracy*)

C’è un istante dopo il quale nessun processo corretto è sospettato da un processo corretto

::: Completezza ed Accuratezza (2/2)

La **completezza** indica la capacità del detector di rilevare i fallimenti, evitando che processi falliti siano considerati corretti (*falsi negativi*)

L'**accuratezza** indica, invece, la capacità di non rilevare un fallimento in modo errato, cioè di non considerare falliti processi che sono invece corretti (*falsi positivi*)

Si noti che la completezza da sola non serve a molto: basta considerare tutti i processi come *falliti* per piena completezza.

::: Livelli di Completezza

Chandra e Toueg individuano due livelli di completezza ...

Completezza forte (*strong completeness*)

Prima o poi ogni processo fallito è permanentemente sospettato
da ogni processo corretto.

Completezza debole (*weak completeness*)

Prima o poi ogni processo fallito è permanentemente sospettato
da qualche processo corretto.

::: Livelli di Accuratezza (1/2)

... e quattro livelli di accuratezza

I primi due sono detti di accuratezza perpetua (*perpetual accuracy*)

Accuratezza forte (*strong accuracy*)

I *processi corretti* non sono mai sospettati da alcun processo corretto.

Accuratezza debole (*weak accuracy*)

C'è *almeno un processo corretto* che non è mai sospettato da alcun processo corretto.

::: Livelli di Accuratezza (2/2)

Tali livelli sono difficili da soddisfare. Si può rendere i requisiti meno stringenti (e più facili da soddisfare) consentendo a un *detector* di sospettare di un processo corretto in un qualsiasi istante della esecuzione, ma *prima o poi (eventually)* deve soddisfare le proprietà di accuratezza forte o debole.

Accuratezza forte eventuale (*eventual strong accuracy*)

C’è un istante di tempo dopo il quale *i processi corretti* non sono sospettati da alcun processo corretto.

Accuratezza debole eventuale (*eventual weak accuracy*)

C’è un istante di tempo dopo il quale *almeno un processo corretto* non è sospettato da alcun processo corretto.

::: Classificazione Failure Detector (1/3)

Dai livelli appena introdotti, è possibile definire otto tipi di *failure detector*.

Completezza	Accuratezza			
	Strong	Eventually Strong	Weak	Eventually Weak
Strong	Perfect \mathcal{P}	Eventually Perfect $\diamond\mathcal{P}$	Strong S	Eventually Strong $\diamond S$
Weak	\mathcal{L}	$\diamond\mathcal{L}$	Weak \mathcal{W}	Eventually Weak $\diamond\mathcal{W}$

Relazioni:

$$\mathcal{P} \subseteq \mathcal{L}; \quad \diamond\mathcal{P} \subseteq \diamond\mathcal{L}; \quad S \subseteq \mathcal{W}; \quad \diamond S \subseteq \diamond\mathcal{W}.$$

::: Classificazione Failure Detector (2/3)

Perfect failure detector \mathcal{P} (*strongly complete, strongly accurate*): tutti i processi falliti prima o poi sono sospettati permanentemente da ogni processo corretto, e tutti i processi corretti non sono mai sospettati da alcun processo corretto.

Eventually perfect failure detector $\Diamond\mathcal{P}$ (*strongly complete, eventually strongly accurate*): tutti i processi falliti prima o poi sono sospettati permanentemente da ogni processo corretto, e tutti i processi corretti sono non più sospettati da un istante t in poi da alcun processo corretto.

Strong failure detector \mathcal{S} (*strongly complete, weakly accurate*): tutti i processi falliti prima o poi sono sospettati permanentemente da ogni processo corretto, e almeno un processo corretto non è mai sospettato da alcun processo corretto.

Eventually strong failure detector $\Diamond\mathcal{S}$ (*strongly complete, eventually weakly accurate*): tutti i processi falliti prima o poi sono sospettati permanentemente da ogni processo corretto, e almeno un processo corretto è non più sospettato da un istante t in poi da alcun processo corretto.

::: Classificazione Failure Detector (3/3)

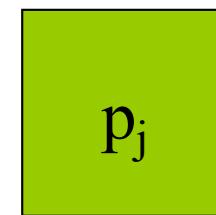
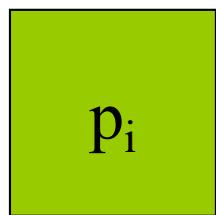
Weakly complete, strongly accurate failure detector \mathcal{L} : tutti i processi falliti prima o poi sono sospettati permanentemente da qualche processo corretto, e tutti i processi corretti non sono mai sospettati da alcun processo corretto.

Weakly complete, eventually strongly accurate failure detector $\Diamond\mathcal{L}$: tutti i processi falliti prima o poi sono sospettati permanentemente da qualche processo corretto, e tutti i processi corretti sono non più sospettati da un istante t in poi da alcun processo corretto.

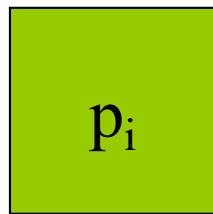
Weak failure detector \mathcal{W} (weakly complete, weakly accurate): tutti i processi falliti prima o poi sono sospettati permanentemente da qualche processo corretto, e almeno un processo corretto non è mai sospettato da alcun processo corretto.

Eventually weak failure detector $\Diamond\mathcal{W}$ (weakly complete, eventually weakly accurate): tutti i processi falliti prima o poi sono sospettati permanentemente da qualche processo corretto, e almeno un processo corretto è non più sospettato da un istante t in poi da alcun processo corretto.

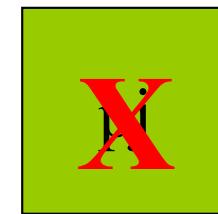
::: Implementazione FD



::: Implementazione FD

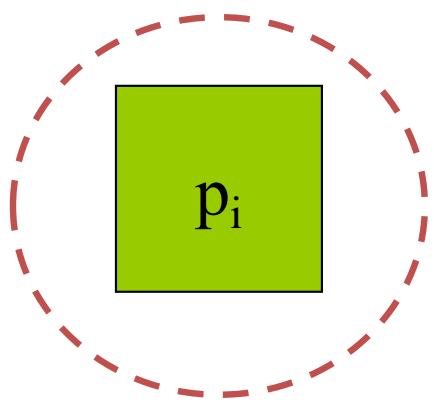


Crash-stop failure
(p_j è un processo fallito)

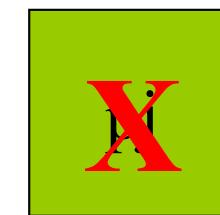


::: Implementazione FD

Deve conoscere se p_j è fallito
(p_i è un processo corretto o vivo)



Crash-stop failure
(p_j è un processo fallito)

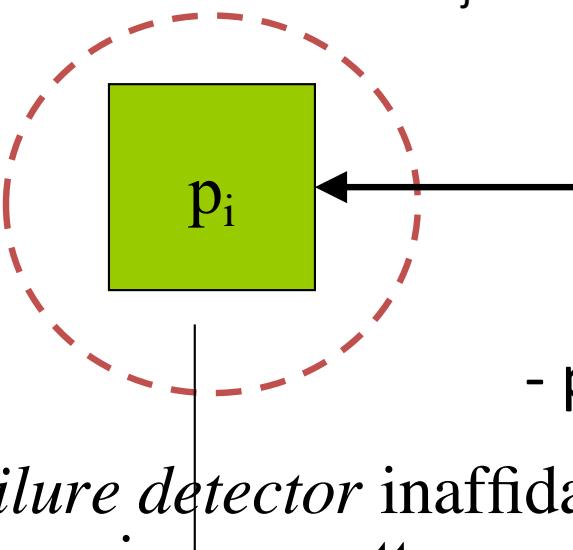


Ci sono due tipi di realizzazioni:

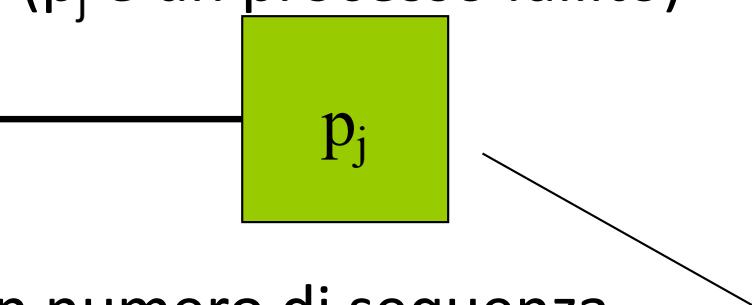
1. Ping-Ack (proattivo)
2. Heartbeat (reattivo)

::: Implem. Detector Inaffidabile (1/3)

p_i deve conoscere se p_j è fallito



Crash-stop failure
(p_j è un processo fallito)



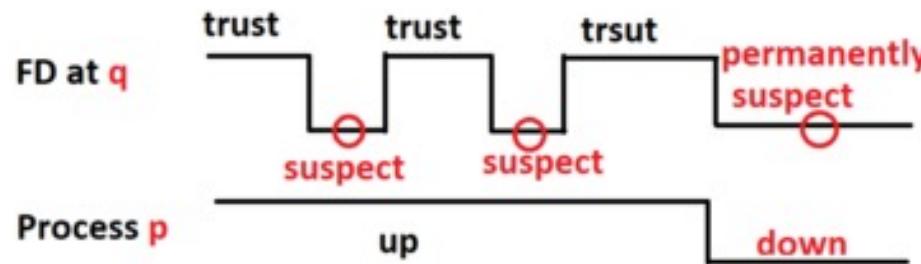
- p_j mantiene un numero di sequenza

Un *failure detector* inaffidabile può essere implementato in un sistema asincrono attraverso la tecnica di *heart beat*:

- Ogni processo p_j invia un messaggio “ p_j -is-alive” a ciascun altro processo nel sistema distribuito, ogni T secondi.
- Il failure detector adotta una stima del massimo tempo di trasmissione di un messaggio, pari a Δ secondi.
- Se il failure detector locale al processo p_i non riceve un “ p_j -is-alive” entro $T+\Delta$ secondi, allora riporta che il processo p_j è *Suspected*.

::: Implem. Detector Inaffidabile (2/3)

- Se successivamente riceve un “ p_j -is-alive”, corregge la sua stima, e riporta a p_i che il processo p_j è *Unsuspected*. Altrimenti, assume il processo come fallito.



La bontà della risposta fornita ad un processo dipende dall'informazione disponibile localmente. Un *failure detector* potrebbe dare suggerimenti errati a causa di condizioni di comunicazione estremamente variabili.

Un processo che fallisce per *crash* inevitabilmente non invierà più messaggi *p-is-alive*; quindi, prima o poi, ogni altro processo corretto, non ricevendo i messaggi, lo sospetterà di fallimento. Ciò implica ***strong completeness***.

::: Implem. Detector Inaffidabile (3/3)

Tuttavia, per l'accuratezza possiamo solo pensare che prima o poi i messaggi p_j -*is-alive* di un processo corretto arriveranno, quindi ci sarà un processo corretto che non sarà sospettato da un altro processo corretto. Ciò implica *eventual weak accuracy*.

La scelta dei tempi T e Δ incide sulle prestazioni del *detector*.

- Tempi brevi portano ad indicare spesso come *Suspected* processi che in realtà non sono falliti (falsi positivi), e richiedono molta banda di rete per i messaggi “ p_j -*is-alive*”.
- Tempi lunghi portano ad indicare spesso come *Unsuspected* processi che in realtà sono falliti (falsi negativi).

Una soluzione pratica al problema prevede l'uso di valori adattativi per T e Δ , che riflettano le reali condizioni correnti della rete.

Il *failure detector* resta inaffidabile, ma l'accuratezza aumenta.

::: Implem. Per Sistema Parz. Sincro. (1/2)

Un sistema parzialmente sincrono è un sistema inizialmente asincrono ma che dopo un istante (sconosciuto) t diventa sincrono.

Per un sistema parzialmente sincrono, rendendo il timeout adattivo il *detector* può essere reso *eventually perfect* ($\diamond \mathcal{P}$)

La *strong completeness* sussiste per fallimenti di tipo *crash*.

Per la *eventual strong accuracy*, si osservi che il sistema diventa sincrono da un istante t in poi, dopo il quale si conosce l'intervallo di tempo necessario per inviare un messaggio da p a q , per cui, prima o poi, nessun processo corretto sarà sospettato da un altro processo corretto.

::: Implem. Per Sistema Parz. Sincro. (2/2)

```
 $Output_p \leftarrow \emptyset$ 
 $\forall q \in \Pi$ 
 $\Delta_p(q) \leftarrow \text{timeout di default}$  /
begin
```

Task 1: periodicamente

invia p -is-alive a tutti gli altri processi del sistema

Task 2: periodicamente

```
 $\forall q \in \Pi$ 
if  $q \notin Output_p$  &
sospettato
```

p non ha ricevuto q -is-alive negli ultimi $\square \Delta_p(q)$ istanti

```
 $Output_p \leftarrow Output_p \cup \{q\}$  // lo aggiunge ai sospettati
```

Task 3: alla ricezione di q -is-alive da un processo q

```
If  $q \in Output_p$ 
 $Output_p \leftarrow Output_p - q$ 
 $\Delta_p(q) \leftarrow \Delta_p(q) + 1$ 
```

end

// *insieme dei sospettati da p*
// *per ogni altro processo q del sistema*
// *per il processo q*

// *per ogni processo nel sistema*
// *se non riceve il messaggio e q non è già*

// *se riceve il messaggio da un processo sospettato*
// *lo rimuove dalla lista dei sospettati*
// *e incrementa il timeout per quel processo*

::: Implem. Detector Affidabile

L’algoritmo di detection presentato diventa affidabile in un sistema sincrono.

Nell’ipotesi di sistema sincrono, Δ diventa un limite superiore assoluto al ritardo di trasmissione di un messaggio, e non più una stima.

L’assenza di un messaggio “ p_j -is-alive” entro $T+\Delta$ permette al failure detector locale di dichiarare con certezza che p è fallito.

::: Protocollo su Messaggi Applicativi

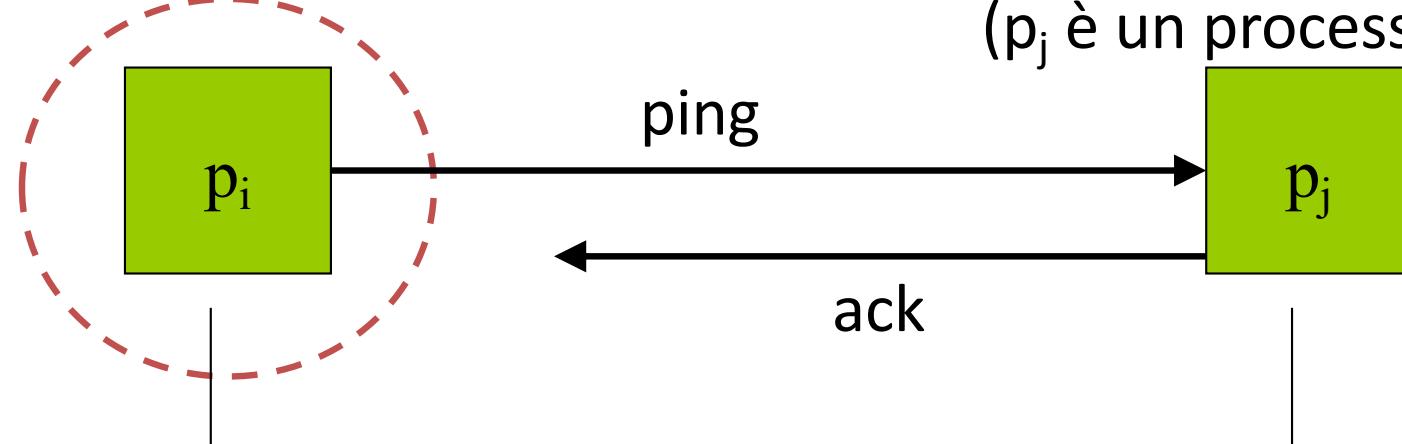
È evidente che l'algoritmo di *detection* presentato, sia esso utilizzato per un sistema asincrono, parzialmente sincrono o sincrono, è molto costoso, vista la richiesta di invio periodico di messaggi di *heart beat*.

Può introdotto un protocollo di *detection* basato principalmente su messaggi applicativi e che fa uso (per la *detection*) di messaggi di controllo solo quando il processo che sta monitorando non invia alcun messaggio applicativo al processo monitorato.

p_i deve conoscere se p_j è fallito

Crash-stop failure

(p_j è un processo fallito)



- p_i interroga p_j ogni T unità di tempo

- p_j risponde

::: Φ Accrual Failure Detector (1/2)

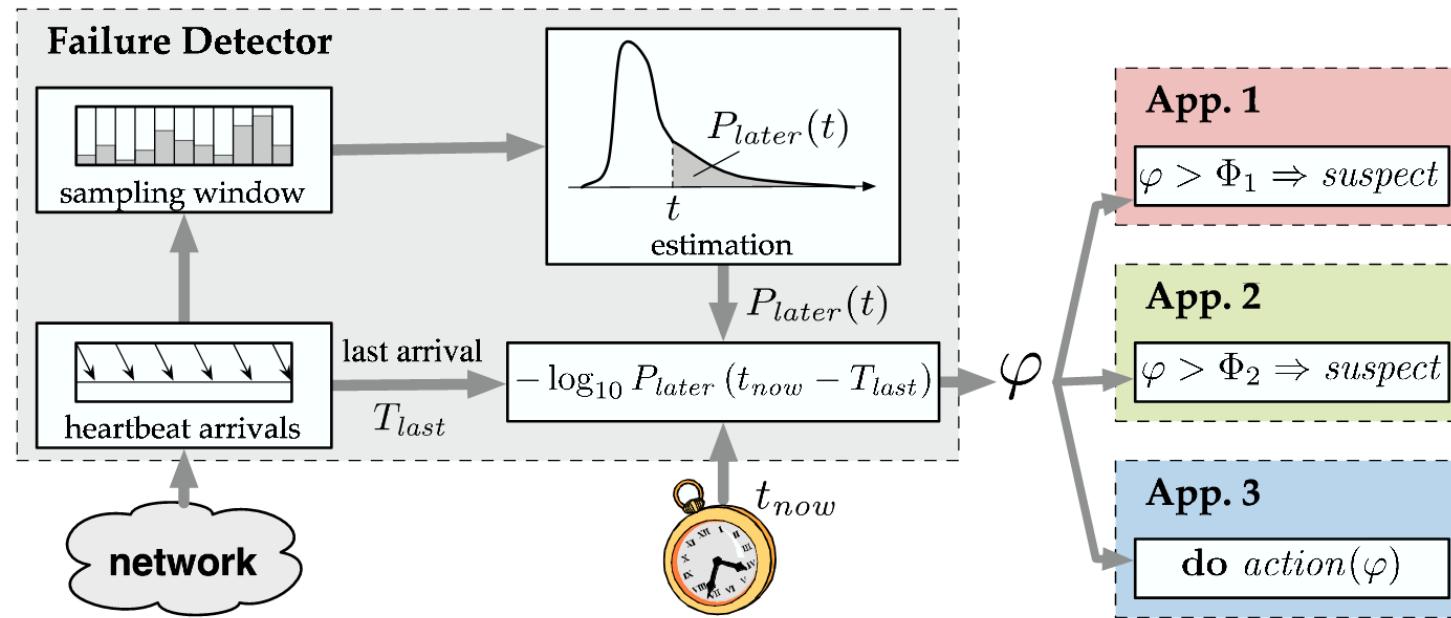
Una nuova astrazione, chiamata accrual failure detectors, promuove flessibilità ed espressività operando con un livello di sospetto Φ su una scala continua, invece di tradizionale informazioni di natura booleana (fiducia vs sospetto).

In parole povere, questo valore Φ rappresenta il grado di certezza che un corrispondente processo monitorato si è arrestato in modo anomalo. Se il processo si blocca effettivamente, è garantito che il valore si accumuli nel tempo e tenda all'infinito.

Viene lasciato alle applicazioni il compito di impostare una soglia di sospetto appropriata in base ai propri requisiti di qualità del servizio.

- Una soglia bassa è propensa a generare molti sospetti errati ma garantisce un rilevamento rapido in caso di incidente reale.
- Al contrario, una soglia alta genera meno errori ma richiede più tempo per rilevare gli arresti anomali effettivi.

::: Φ Accrual Failure Detector (2/2)



Heartbeat arrivano dalla rete e il loro tempo di arrivo viene memorizzato nella finestra di campionamento. I campioni vengono utilizzati per stimare una certa distribuzione degli arrivi. Il tempo dell'ultimo arrivo T_{last} , il tempo attuale e la distribuzione stimata vengono utilizzate per calcolare il valore corrente di ϕ .

Le applicazioni attivano sospetti in base a una certa soglia (Φ_1 per App. 1 e Φ_2 per App. 2), o eseguono alcune azioni in funzione di ϕ .

::: FD nei SD (1/3)

Nelle realizzazioni concrete, chi effettua il monitoraggio e rilevamento dei processi falliti?

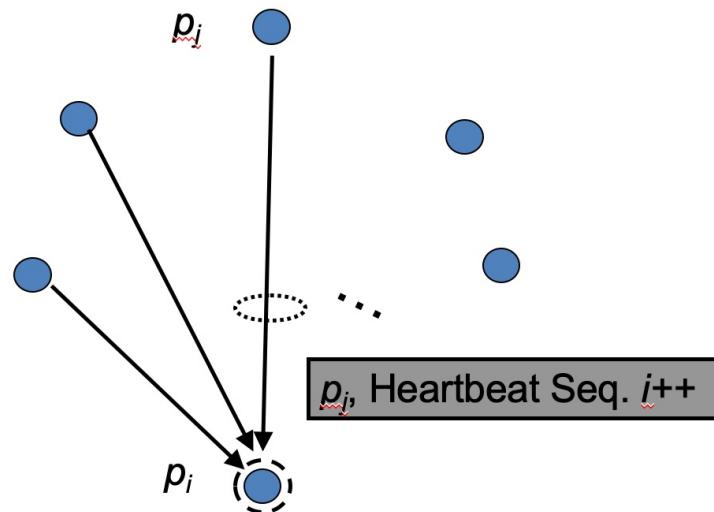
Sono possibili tre diversi approcci:

... FD nei SD (1/3)

Nelle realizzazioni concrete, chi effettua il monitoraggio e rilevamento dei processi falliti?

Sono possibili tre diversi approcci:

- Centralizzato;



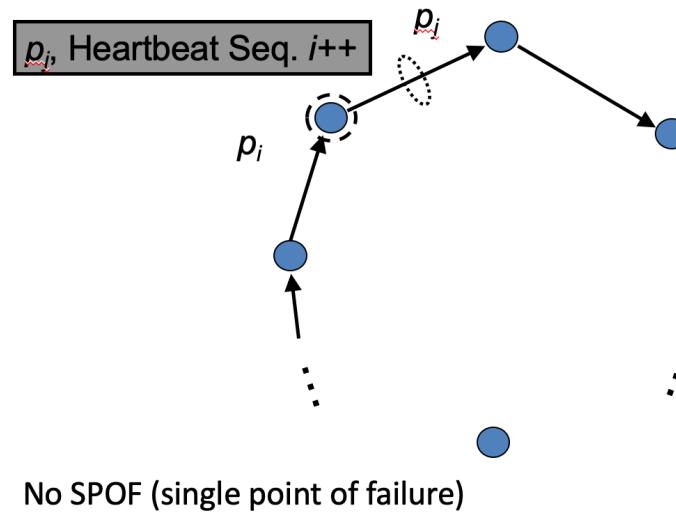
Semplice da realizzare ed in grado di avere una completa visione del sistema, ma può diventare una criticità

::: FD nei SD (1/3)

Nelle realizzazioni concrete, chi effettua il monitoraggio e rilevamento dei processi falliti?

Sono possibili tre diversi approcci:

- Centralizzato;
- Ad anello;



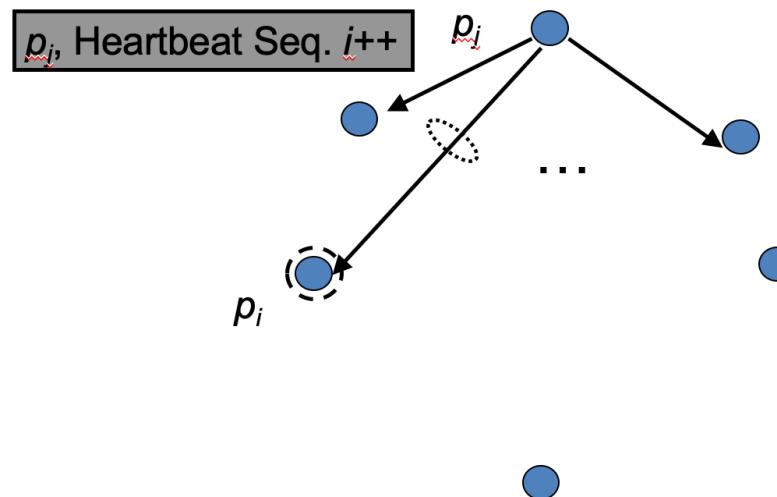
Più complesso da realizzare, non presenta un nodo critico, ma è vulnerabile a possibili problemi sulla rete.

::: FD nei SD (1/3)

Nelle realizzazioni concrete, chi effettua il monitoraggio e rilevamento dei processi falliti?

Sono possibili tre diversi approcci:

- Centralizzato;
- Ad anello;
- Distribuito.



Una soluzione che tollera maggiormente problemi di processo e di rete ma genera un forte carico di rete e consumo energetico.

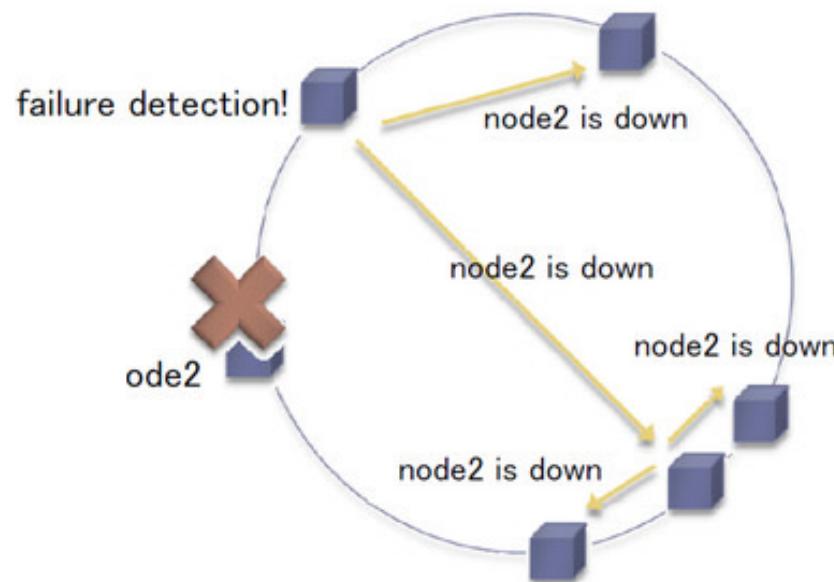
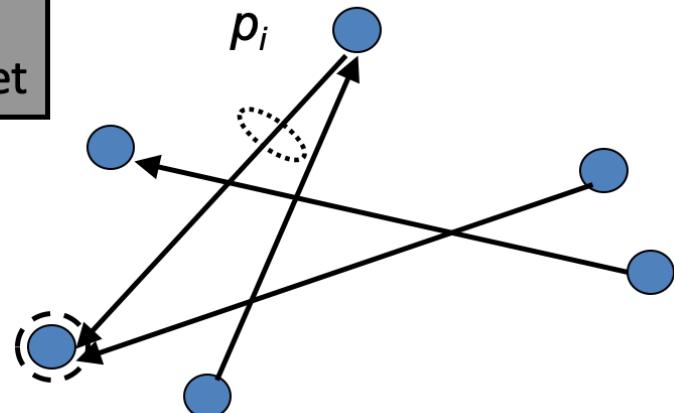
::: FD nei SD (1/3)

Nelle realizzazioni concrete, chi effettua il monitoraggio e rilevamento dei processi falliti?

Sono possibili tre diversi approcci:

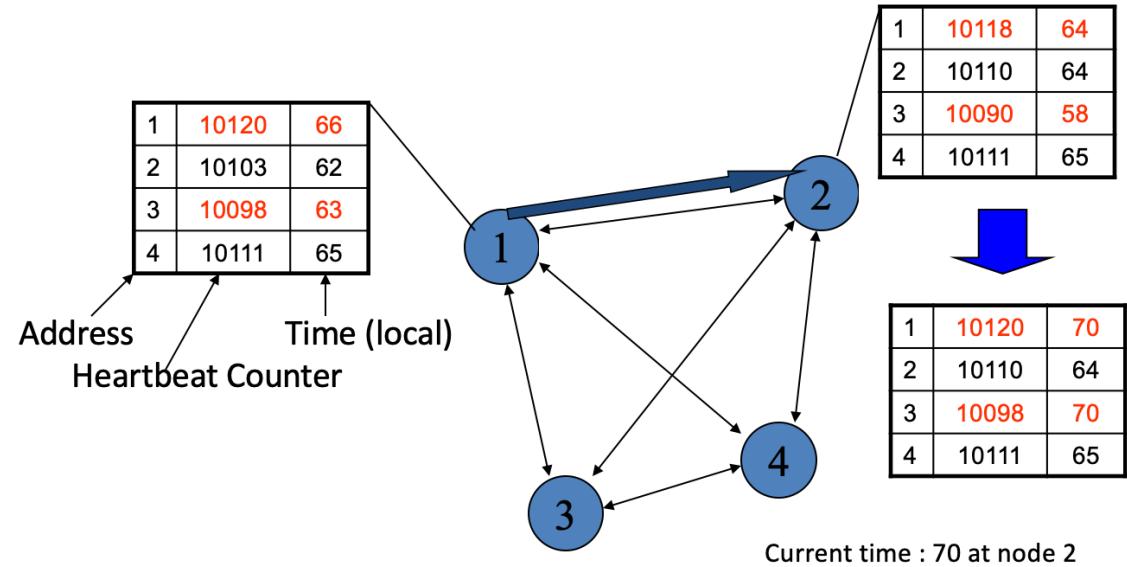
- Centralizzato;
- Ad anello;
- Distribuito.

Array of
Heartbeat Seq. i
for member subset



Una realizzazione più efficiente dell'approccio distribuito si basa sul gossiping.

::: FD nei SD (2/3)



Algoritmo:

- Ogni processo mantiene un elenco di membri;
- Ogni processo incrementa periodicamente il proprio contatore di heartbeat;
- Ogni processo periodicamente invia per mezzo del gossiping la sua lista di membri;
- Al ricevimento, gli heartbeat vengono uniti e l'ora locale viene aggiornata.

::: FD nei SD (3/3)

$O(\log(N))$ è il tempo per un aggiornamento heartbeat di propagarsi e raggiungere tutti i processi del sistema con alta probabilità.

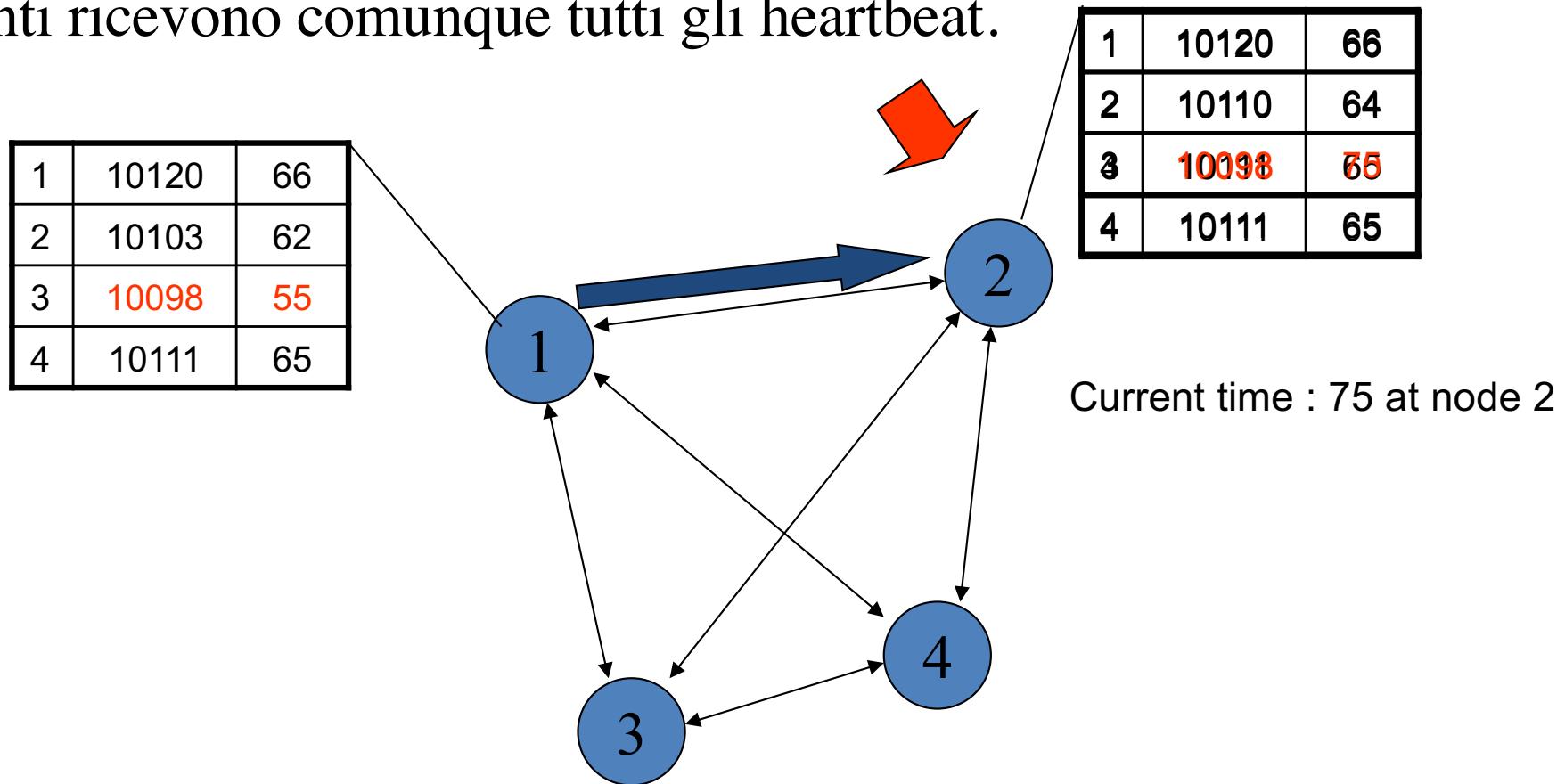
Lo schema è molto robusto contro gli errori: anche se un gran numero di processi si blocca, la maggior parte / tutti i processi rimanenti ricevono comunque tutti gli heartbeat.

- Se l'heartbeat non è aumentato per più di T_{fail} secondi, il processo corrispondente viene considerato non corretto, con T_{fail} solitamente impostato su $O(\log(N))$.
- L'entry nella lista del processo considerato fallito non + cancellata immediatamente, ma si attende altri $T_{cleanup}$ secondi (di solito pari T_{fail}).

::: FD nei SD (3/3)

$O(\log(N))$ è il tempo per un aggiornamento heartbeat di propagarsi e raggiungere tutti i processi del sistema con alta probabilità.

Lo schema è molto robusto contro gli errori: anche se un gran numero di processi si blocca, la maggior parte / tutti i processi rimanenti ricevono comunque tutti gli heartbeat.



::: Considerazioni

È lecito interrogarsi sull'effettiva utilità dei *failure detector*.

I *failure detector* inaffidabili possono generare falsi positivi (quindi essere *inaccurati*) e falsi negativi (essere *incompleti*).

I *failure detector* affidabili richiedono che il sistema sia sincrono (e pochi sistemi reali lo sono).

D'altra parte, i *failure detector* aiutano a comprendere la natura dei fallimenti in un sistema distribuito.

Qualsiasi sistema reale progettato per gestire i fallimenti deve poterli rilevare, anche se in maniera imperfetta.

::: Consenso con FD (1/2)

Alcuni sistemi reali utilizzano *detectors* “affidabili da progetto” per raggiungere il consenso.

I processi si accordano sull’*indicare* un processo p come *Failed* se p non risponde per più di un certo tempo nominale massimo, anche in un sistema asincrono.

In realtà, p potrebbe non essere fallito, ma i rimanenti processi agiscono come se lo fosse:

- essi fanno diventare p “*fail-silent*”, scartando tutti i messaggi che ricevono da esso.

In pratica, ciò equivale a trasformare un sistema asincrono in un sistema sincrono. La tecnica è usata ad esempio in ISIS, il toolkit di *group communication* per la programmazione distribuita realizzato dalla Cornell University.

::: Consenso con FD (2/2)

Un altro approccio consiste nell'usare *failure detectors* inaffidabili.

Il consenso viene raggiunto consentendo ai processi *Suspected* di agire come *corretti* e partecipare alla decisione, anziché escluderli.

Chandra e Toueg hanno dimostrato **che il consenso può essere raggiunto, anche in un sistema asincrono e usando detectors inaffidabili, se il numero di processi falliti non eccede $N/2$ e la comunicazione è affidabile.**

Il più debole *failure detector* che può essere usato allo scopo è un *eventually weak failure detector*.

Chandra e Toueg hanno dimostrato che **in un sistema asincrono non è possibile implementare un *eventually weak failure detector* solo attraverso scambio di messaggi.**

Tuttavia un detector adattativo (con timeout variabili) va bene in molti casi pratici.



Consistenza dei Dati

::: Replicazione e Consistenza (1/2)

- Sussistono due motivi per adoperare la replicazione nei sistemi distribuiti:
 - Per migliorare la reliability del sistema
 - Per ragione prestazionali, ma anche per scalare rispetto all'area geografica.
- Dati i suoi vantaggi, perché va anche temuta? Perché bisogna pagare un prezzo quando i dati sono replicati.
- Avere più repliche di determinati dati pone un problema di consistenza: se una copia viene modificata, le repliche possono avere stati diversi. Conseguentemente, le modifiche devono essere effettuate atomicamente su tutte le repliche e questo ha un costo.

::: Replicazione e Consistenza (2/2)

Two processes accessing shared variables

Process 1

```
br := b;  
ar := a;  
if(ar ≥ br) then  
    print ("OK");
```

Process 2

```
a := a + 1;  
b := b + 1;
```

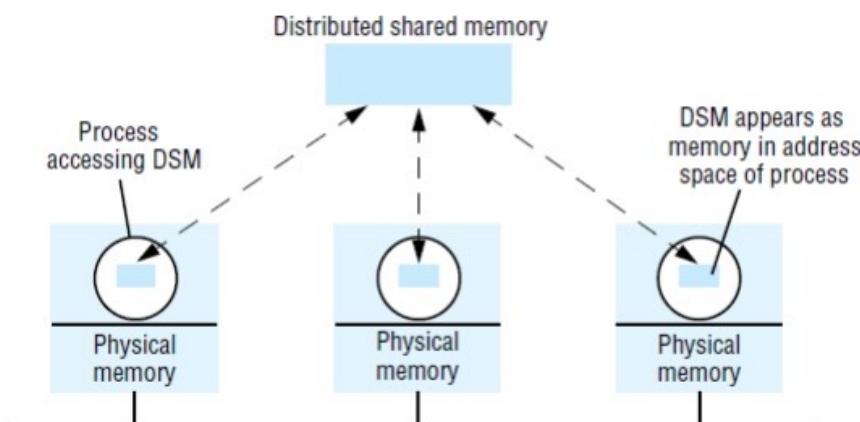
- Tradizionalmente, la consistenza è stata studiata nel contesto di operazioni di read e write su dati condivisi, disponibili per mezzo di un'astrazione di memoria condivisa (distribuita), un database condiviso (distribuito) o un file system (distribuito).
- Una operazione è classificata come una write se ne modifica il valore, altrimenti viene classificata come una read.

::: Distributed Shared Memory

DSM è un'astrazione utilizzata per condividere dati tra computer che non condividono la memoria fisica.

- I processi accedono alla memoria tramite letture e scrittura a quella che sembra essere una normale memoria all'interno del loro spazio di indirizzamento.
- Un sistema sottostante garantisce in modo trasparente che i processi in esecuzione su computer diversi osservino gli aggiornamenti effettuati l'uno dall'altro.

The distributed shared memory abstraction



- I sistemi DSM gestiscono i dati replicati: ogni computer ha una copia locale degli elementi di dati a cui si è avuto accesso di recente archiviata in DSM, per velocità di accesso.

::: Modelli di Consistenza (1/4)

- **Modello di consistenza** (o semantica della consistenza):
 - **Contratto** tra i processi e l'archivio di dati (distribuito): se i processi rispettano un certo **insieme di regole**, l'archivio conterrà valori corretti,
 - Processi concorrenti possono aggiornare simultaneamente un archivio di dati.
- Gli accessi (lettura/scrittura) richiedono un tempo finito, ma sono operazioni diverse operate su processori diversi, che si possono sovrapporre nel tempo.
- Bisogna garantire che le operazioni conflittuali siano eseguite sulle varie repliche con un ordinamento con una chiara semantica.
- Normalmente, un processo che effettua una read su un valore, si attende di ottenere il valore risultante dall'ultima operazione di write. In assenza di un orologio globale è difficile definire precisamente quale operazione di write è l'ultima.

::: Modelli di Consistenza (2/4)

- Modelli di consistenza **data-centrati**
 - Fornire una vista di un archivio di dati consistente **a livello di sistema**
- Modelli di consistenza **client-centrati**
 - Fornire una vista di un archivio di dati consistente **a livello di un singolo client**
 - Più veloce, ma meno accurata della consistenza data-centrica

::: Modelli di Consistenza (3/4)

- Principali modelli
 - Stretta, Linearizzabile, Sequenziale, Causale, Finale
 - I modelli descrivono come e quando le diverse repliche del sistema distribuito vedono l'ordine delle operazioni
 - Le repliche devono accordarsi su quale sia l'ordinamento globale delle operazioni prima di renderle permanenti
- 
- Modelli più deboli, letture/scritture più veloci*

... Modelli di Consistenza (4/4)

Process P ₁	Process P ₂	Process P ₃
$x \leftarrow 1;$ <code>print(y,z);</code>	$y \leftarrow 1;$ <code>print(x,z);</code>	$z \leftarrow 1;$ <code>print(x,y);</code>

Consideriamo 3 processi concorrenti in ambito distribuito, con tre variabili condivise, ognuna inizializzata a 0.

Sono possibili varie sequenze di esecuzioni, potenzialmente 720 ($6!$), e alcune di esse violano l'ordine del programma. Quattro di queste sequenze sono le seguenti:

Execution 1	Execution 2	Execution 3	Execution 4
P ₁ : x ← 1; P ₁ : print(y,z); P ₂ : y ← 1; P ₂ : print(x,z); P ₃ : z ← 1; P ₃ : print(x,y);	P ₁ : x ← 1; P ₂ : y ← 1; P ₂ : print(x,z); P ₁ : print(y,z); P ₃ : z ← 1; P ₃ : print(x,y);	P ₂ : y ← 1; P ₃ : z ← 1; P ₂ : print(x,z); P ₁ : x ← 1; P ₁ : print(y,z);	P ₂ : y ← 1; P ₁ : x ← 1; P ₃ : z ← 1; P ₂ : print(x,z); P ₁ : print(y,z); P ₃ : print(x,y);
Prints: 001011 Signature: 00 10 11	Prints: 101011 Signature: 10 10 11	Prints: 010111 Signature: 11 01 01	Prints: 111111 Signature: 11 11 11
(a)	(b)	(c)	(d)

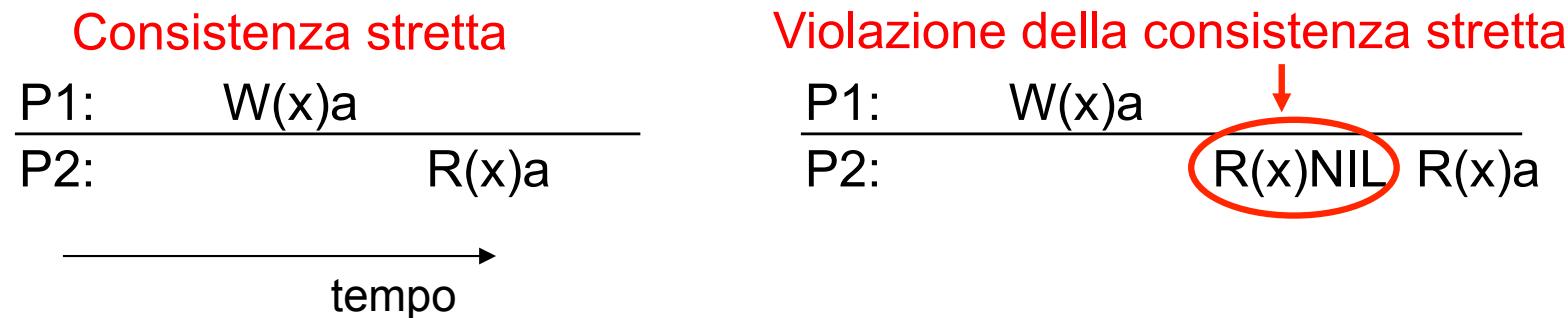
Quale signature è valida? Il modello di consistenza ce lo dice.

::: Modelli di Consistenza Data-Centrici

Consistenza	Descrizione
Stretta	Tutti i processi vedono gli accessi condivisi nello stesso ordine assoluto di tempo
Linearizzabile	Tutti i processi vedono gli accessi condivisi nello stesso ordine : gli accessi sono ordinati in base ad un timestamp globale (non unico)
Sequenziale	Tutti i processi vedono gli accessi condivisi nello stesso ordine ; gli accessi non sono ordinati temporalmente
Causale	Tutti i processi vedono gli accessi condivisi correlati causalmente nello stesso ordine

::: Strict Consistency: Modello Ideale

- Corrisponde alla nozione di correttezza nel modello di Von Neumann monoprocessoress
- Qualsiasi *read* su un dato x restituisce un valore corrispondente al risultato più recente della *write* su x , secondo un **tempo globale**
 - $W_i(x)a$: scrittura di P_i sul dato x con valore scritto a
 - $R_i(x)b$: lettura di P_i sul dato x con valore letto b

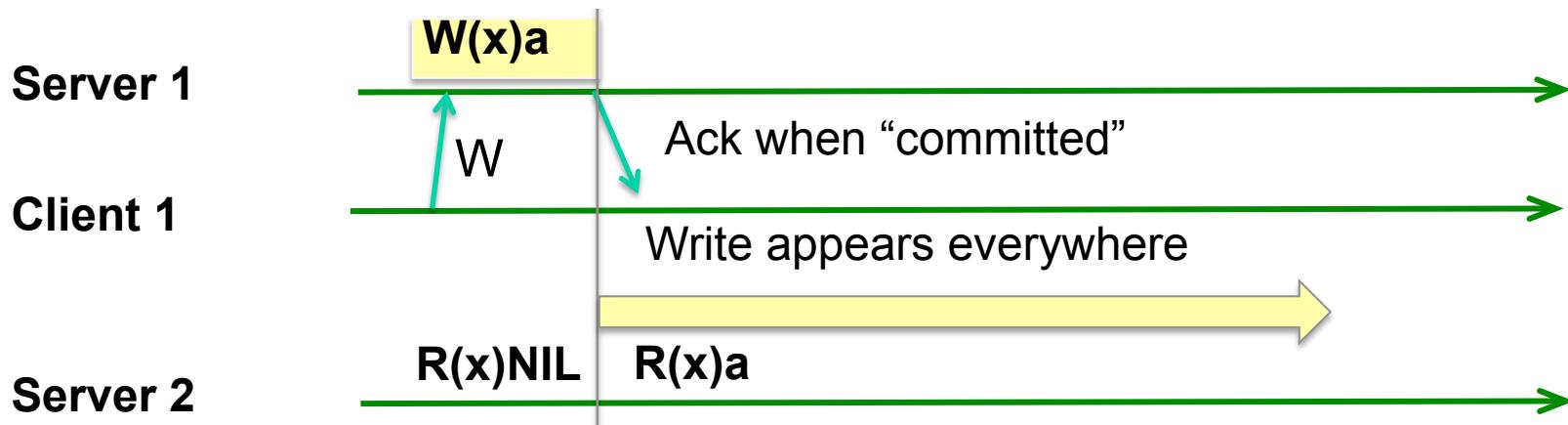


- *Write* vista **istantaneamente** da tutti i processi
- Nessuna ambiguità su “più recente”
- **Richiede un tempo globale**
- Rilassare i vincoli di consistenza => **linearizzabilità e consistenza sequenziale**

::: Linearizzabilità (1/3)

- Tutte le operazioni ricevono un **timestamp** globale usando un clock sincronizzato (e.g., NTP) – simula tempo globale
- **Linearizzabilità**: il risultato di una qualunque esecuzione è uguale a quello ottenuto nel caso in cui:
 - Le operazioni di tutti i processi sono eseguite in qualche ordine sequenziale
 - Le operazioni di ogni singolo processo nella sequenza sono fatte nello stesso ordine indicato dal suo programma
 - Se $ts_{OP1}(x) < ts_{OP2}(y)$, allora OP1(x) deve precedere OP2(y) nella sequenza
- Requisito “real-time”: le operazioni “*appaiono*” come se fossero eseguite istantaneamente su ogni nodo

::: Linearizzabilità (2/3)



- La linearizzabilità non prescrive uno specifico ordine per le operazioni che non si sovrappongono:
 - Si può implementare qualsiasi strategia di ordinamento purché ci sia un singolo ordinamento per le operazioni che si sovrappongono.

::: Linearizzabilità (3/3)

La linearizzabilità si può realizzare ma è molto costosa

- Tutti i processori devono convenire su un ordinamento comune
- Occorre simulare una scala temporale globale
- La linearizzabilità richiede il *total ordering*, che a sua volta può essere implementato con il broadcast

(shared var)

int: x ;

- (1) When the memory manager receives a *Read* or *Write* from application:
 - (1a) **total_order_broadcast** the *Read* or *Write* request to all processors;
 - (1b) **await** own request that was broadcast;
 - (1c) **perform** pending response to the application as follows
 - (1d) **case** *Read*: return value from local replica;
 - (1e) **case** *Write*: write to local replica and return ack to application.
- (2) When the memory manager receives a **total_order_broadcast**(*Write*, x , val) from network:
 - (2a) **write** val to local replica of x .
- (3) When the memory manager receives a **total_order_broadcast**(*Read*, x) from network:
 - (3a) **no operation**.

::: Sequenziale

La consistenza sequenziale assicura la sequenzialità delle azioni tra tutti i processi. Secondo Leslie Lamport:

«il risultato di ogni esecuzione è lo stesso come se le operazioni nel data store di tutti i processi vengano eseguite nello stesso ordine sequenziale, e le operazioni di ogni singolo processo appaiono in questa sequenza nell'ordine specificato dal suo programma.»

Ogni inserimento di operazioni valide di lettura o scrittura è un comportamento accettabile, purché tutti i processi vedano gli stessi inserimenti.

P1: W(x)a

P2: W(x)b

P3: R(x)b R(x)a

P4: R(x)b R(x)a

(a)

P1: W(x)a

P2: W(x)b

P3: R(x)b R(x)a

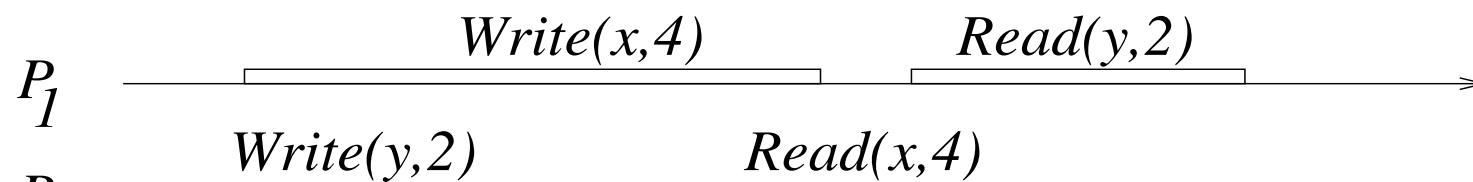
P4: R(x)a R(x)b

(b)

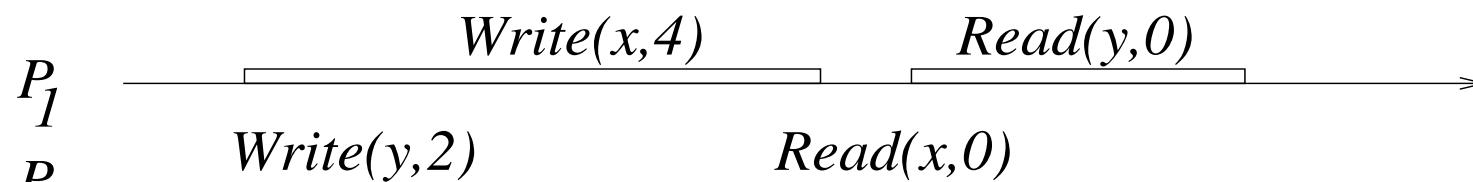
::: Cons. Sequentziale vs Linearizzabilità



(a) Sequentially consistent but not linearizable



(b) Sequentially consistent and linearizable



(c) Not sequentially consistent (and hence not linearizable)

Initial values are zero. (a),(c) not linearizable. (b) is linearizable

::: Cons. Sequenziale vs Linearizzabilità

- **Linearizzabilità** =
Consistenza sequenziale + “una *read* ritorna la *write più recente*, indipendentemente dal client, secondo il loro effettivo ordine temporale”
- Con la **consistenza sequenziale**, il sistema ha libertà rispetto a come alternare le operazioni che vengono da client diversi, purché l’ordine di ogni client sia preservato
- Con la **linearizzabilità**, l’alternanza tra tutti i client è pressoché già determinata in base al tempo

::: Consistenza Causale (1/5)

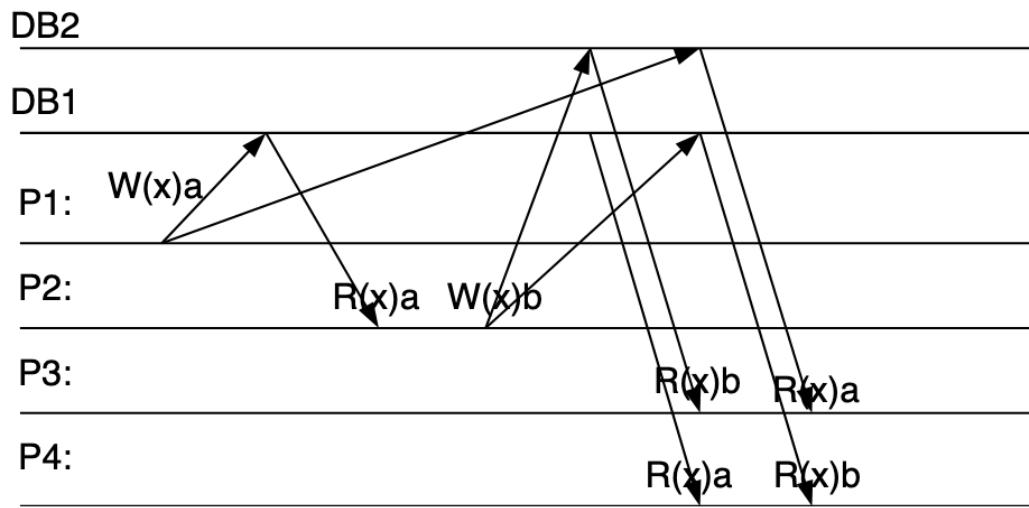
- Rilassamento della consistenza sequenziale, in quanto fa distinzione tra eventi che sono potenzialmente correlati causalmente:
«tutte le scritture che sono potenzialmente correlate causalmente devono essere viste da tutti i processi nello stesso ordine. Scritture concorrenti possono essere viste in ordini diversi su macchine diverse».

P1:	W(x)a		W(x)c
P2:	R(x)a	W(x)b	
P3:	R(x)a		R(x)c R(x)b
P4:	R(x)a		R(x)b R(x)c

Le scritture del processo P1 sono correlate (avvengono nello stesso processo), mentre la scrittura di P2 non è correlata (non sollecita) con W(x)c.

Di conseguenza le letture di P3 e P4 sono giustificabili. Il fatto che il valore b sia letto prima o dopo il valore c non ha importanza. Quel che conta è che il valore a sia sempre letto prima del valore c. Questo comportamento invece sarebbe da respingere se volessimo un modello di consistenza sequenziale.

::: Consistenza Causale (2/5)



Consideriamo ora il seguente scenario, in cui il processo P2 legge l'oggetto x prima di scrivere. La scrittura di P2 quindi può essere condizionata dalla lettura antecedente: di conseguenza $W(x)b$ è correlato con $W(x)a$.

In questo caso la consistenza causale è violata, perché la correlazione tra le due scritture imporrebbe sempre la lettura di a prima di b. Cosa che non avviene. La figura ci permette di ipotizzare una possibile causa. Utilizzando due basi di dati, la scrittura $W(x)a$ verso DB2 può essere ritardata. In tal caso, se P3 legge da DB2, può vedere prima b e poi a. E i due DB sarebbero disallineati.

::: Consistenza Causale (3/5)

Relazione di causalità

- **Local order:** l'ordine locale degli eventi definisce l'ordine causale.
- **Inter-process order:** una *write* precede causalmente una *read* eseguita da un altro processo se la *read* restituisce il valore scritto dalla *write*.
- **Transitive closure:** La chiusura transitiva delle due relazioni precedenti definisce l'ordine causale (globale).

- Per la consistenza causale, solo le *write* correlate causalmente devono essere viste nello stesso ordine.
- Operazioni di *write* che sono potenzialmente in *relazione di causa/effetto* devono essere viste da *tutti* processi nello stesso ordine.
- Operazioni di *write concorrenti* possono essere viste in ordine *differente* da processi differenti.
- Indebolisce la consistenza sequenziale: distingue tra operazioni in relazione causale e quelle che non lo sono.

::: Consistenza Causale (4/5)

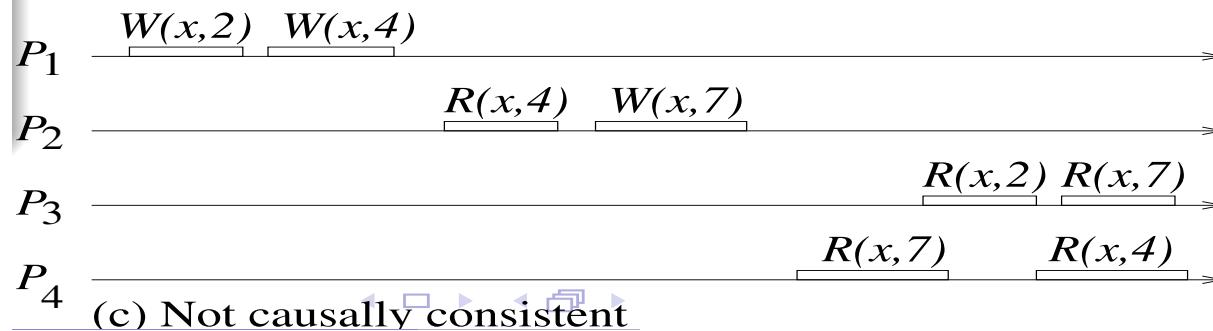
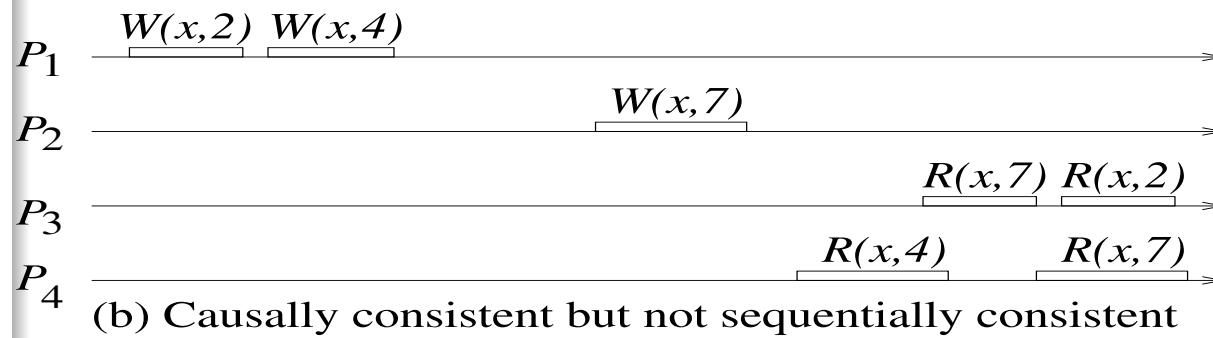
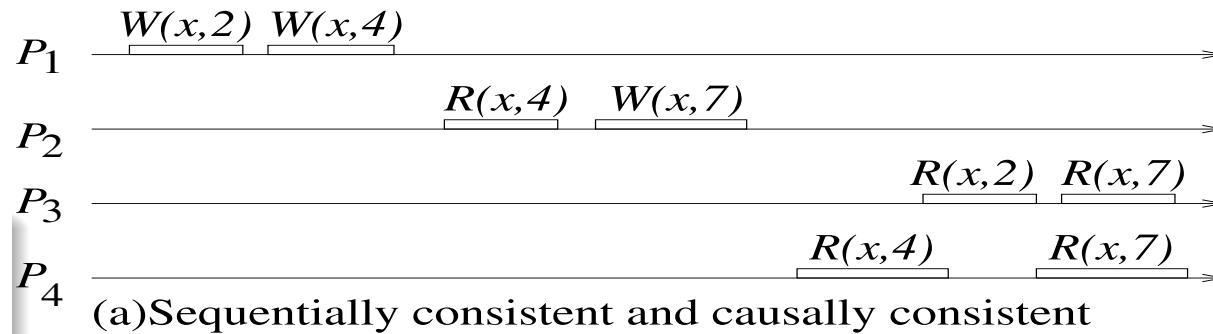
P1:	W(x)a		
P2:	R(x)a	W(x)b	
P3:	R(x)a		R(x)c R(x)b
P4:	R(x)a		R(x)b R(x)c

Sequenza causalmente valida, non sequenzialmente
W₂(x)b e W₁(x)c sono concorrenti: possono essere viste in ordine differente
W₁(x)a e W₂(x)b sono in relazione di causa/effetto

P1:	W(x)a		
P2:	R(x)a	W(x)b	
P3:			R(x)b R(x)a
P4:			R(x)a R(x)b

Sequenza causalmente non valida
W₁(x)a e W₂(x)b sono in relazione di causa/effetto: devono
essere viste da tutti i processi nello stesso ordine.

::: Consistenza Causale (5/5)



::: Eventual Consistency (1/2)

- In un archivio di dati distribuito caratterizzato da:
 - mancanza di conflitti *write-write* o comunque facile soluzione in caso di conflitto
 - Forte prevalenza di letture rispetto alle scritture
- Si adotta spesso un modello di consistenza rilassato, detto **consistenza finale (*eventual consistency*)**
 - Cosa garantisce: **se** non si verificano aggiornamenti, tutte le repliche (distribuite geograficamente) diventano gradualmente consistenti **entro una finestra temporale** (detta ***inconsistency window***)
 - In assenza di *failure*, la dimensione *dell'inconsistency window* dipende da: ritardi di comunicazione, carico del sistema, numero di repliche
 - **Vantaggi:** semplice da realizzare, poco costosa
 - **Svantaggio:** se l'utente accede a repliche diverse in un breve intervallo di tempo può accedere a dati non aggiornati; in tal caso, occorre **risolvere il conflitto (*reconciliation*)**

::: Eventual Consistency (2/2)

- Modello di consistenza spesso adottato nei sistemi distribuiti a larga scala (vedi teorema CAP)
- In particolare per servizi di ***storage*** e ***datastore NoSQL*** (Not Only SQL) **in ambito Cloud**
 - Ad es. **Amazon S3, Apache Cassandra, Apache CouchDB, Dropbox, Google BigTable, Google App Engine HRD**
- Attenzione: il costo di garantire un maggior livello di consistenza ricade sullo sviluppatore dell'applicazione
 - Lo sviluppatore deve sapere quale grado di consistenza viene offerto dal sistema
 - Con la consistenza finale, può accadere che una *read* non restituisca il valore della *write* più recente: lo sviluppatore deve decidere se tale inconsistenza (**staleness** dei dati) è accettabile per l'utente dell'applicazione

::: Consistenza con Sincronizzazione (1/2)

- Operazioni raggruppate.
- Condizioni di consistenza applicate solo a istruzioni di sincronizzazione, ad es. *barrier*.
- Istruzioni non di sincronizzazione possono essere eseguite in ordine diverso dai vari processori.
- E.g.: *consistenza debole*, *consistenza release*, *consistenza entry*.

Consistenza debole:

- ***Tutte le writes sono propagate agli altri processi, e tutte le writes fatte altrove sono riportate localmente, all'occorrenza dell'istruzione di sincronizzazione***
- Gli accessi a variabili di sincronizzazione sono sequenzialmente consistenti.
- L'accesso a variabili di sincronizzazione non è permesso finché tutte le precedenti *writes* non sono state completate su tutte le copie.
- Nessuna *read* o *write* su un dato è permessa finché non siano stati eseguiti tutti i precedenti accessi sulle variabili di sincronizzazione.

::: Consistenza con Sincronizzazione (2/2)

- Due tipi di variabili di sincronizzazione *Acquire* e *Release*.

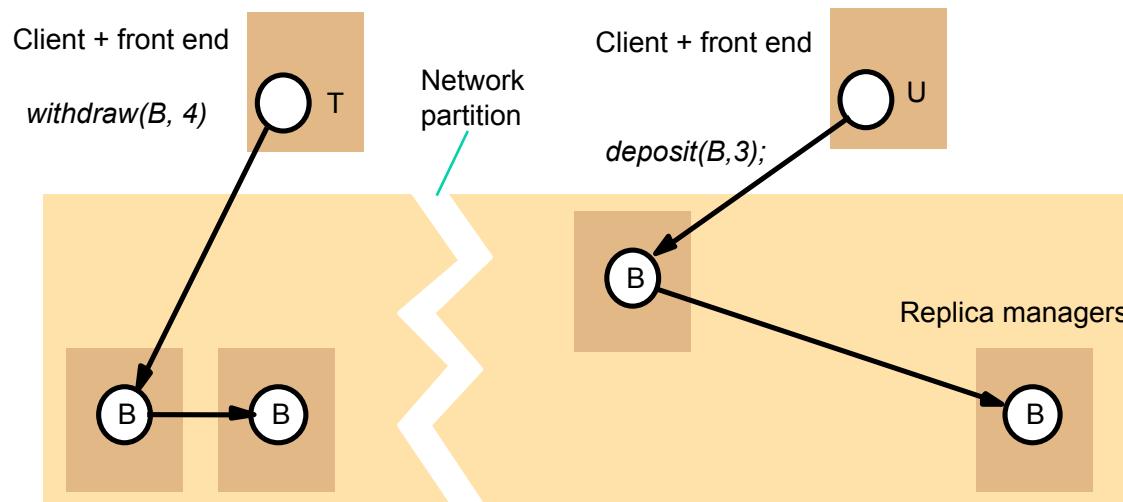
Consistenza *Release*

- *Acquire* indica l'ingresso nella SC. Tutte le write dagli altri processi devono riflettersi localmente all'occorrenza di questa istruzione.
- *Release* indica che l'uscita dalla SC. Tutti gli aggiornamenti fatti localmente devono essere propagati alle repliche.
- *Acquire* e *Release* definibili su un sottoinsieme delle variabili.
- Consistenza *Lazy release*: aggiornamenti propagati on-demand.

Consistenza *Entry*

- Ogni variabile ordinaria condivisa è associata ad una variabile di sincronizzazione (e.g., lock, barrier).
- All'atto *dell'acquire* su una variabile di sincronizzazione, è eseguito l'accesso soltanto alle variabili ordinarie protette da quella variabile di sincronizzazione.

::: Partizionamento



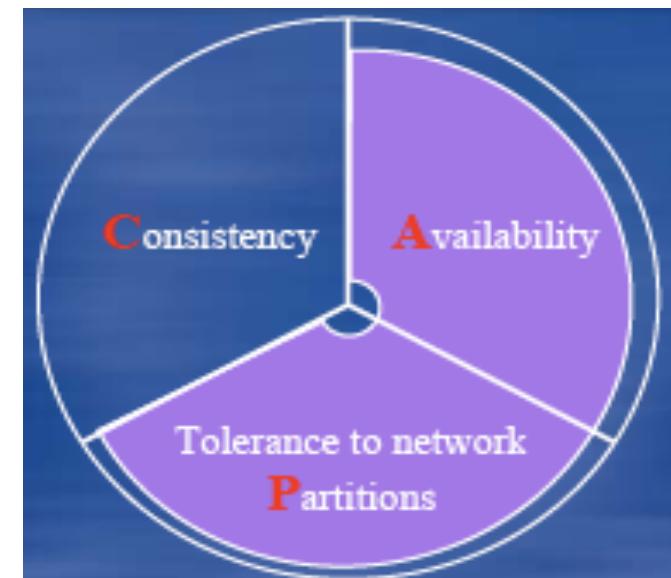
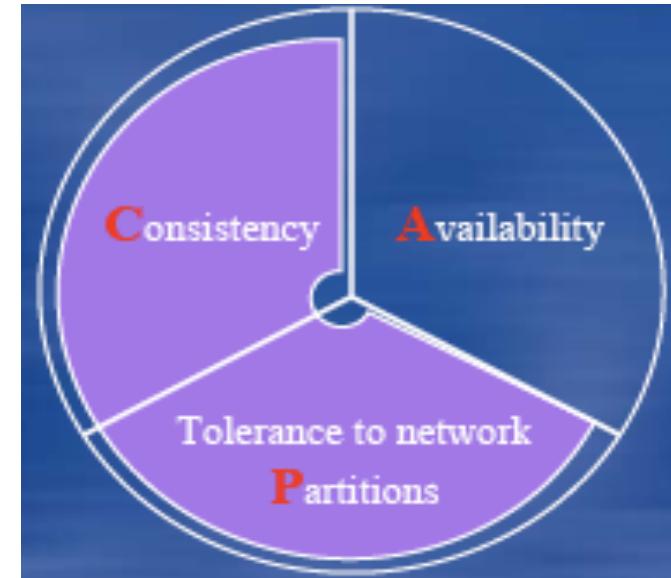
- In presenza di un partizionamento
 - Per mantenere la consistenza, bisogna bloccare il sistema, rendendolo indisponibile
- Se venissero servite le richieste dalle due parti, ci sarebbe inconsistenza
 - Il sistema è **available**, ma non **consistente**
- Il teorema CAP formalizza questa situazione

::: Teorema CAP (1/3)

- Congettura proposta da E. Brewer nel 2000, poi dimostrata da S. Gilbert e N. Lynch nel 2002
- Ogni sistema in rete che condivide dati può avere in un dato istante al più due delle tre proprietà desiderabili:
 - **Consistency (C)**: avere una copia aggiornata dei dati: “*All the clients see the same view, even in presence of updates*”
 - **Availability (A)** – disponibilità dei dati
“*All clients can find some replica of data, even in presence of failure.*”
 - **Tolerance to network partitions (P)**
“*The system property holds even if the system is partitioned.*”

::: Teorema CAP (2/3)

- Se la priorità è la consistenza, si rinuncia all'availability
 - **Sistema CP**
- Se la priorità è l'availability, si rinuncia alla consistenza
 - **Sistema AP**
 - Usando un modello di consistenza più “debole”:
 - **Eventual consistency**



::: Teorema CAP (3/3)

- Quando si adottano sistemi CP e AP, lo sviluppatore deve sapere cosa “offre” il sistema:
- CP: il sistema può non essere disponibile ad eseguire una *write*
 - Lo sviluppatore deve gestire il fallimento di una *write* causato dall’eventuale indisponibilità del sistema
- AP: il sistema può accettare sempre una *write*, ma, sotto certe condizioni, una *read* non riporterà il risultato della *write* più recente
 - Lo sviluppatore deve decidere se il client può richiedere sempre l’accesso all’ultimo aggiornamento

::: ACID vs BASE (1/2)

- ACID e BASE: filosofie di design agli estremi opposti dello spettro *consistency-availability*
- **ACID (Atomicity, Consistency, Isolation, Durability)**
 - Approccio tradizionale per garantire consistenza nei DBMS
 - DBMSs tradizionali (Postgres, MySQL, ...) sono esempi di sistemi **CP**
 - Approccio pessimistico: non scalabile quando bisogna gestire petabytes di dati

::: ACID vs BASE (2/2)

- **BASE**: Basically Available, Soft state, Eventual consistency
 - Approccio ottimistico e per sistemi AP
 - **Basically available**: il sistema è disponibile la maggior parte del tempo, potrebbero esserci sottosistemi temporaneamente non disponibili
 - **Soft state**: la persistenza è gestita dall'utente, che deve preoccuparsi di aggiornare i dati (*refresh*)
 - **Eventually consistent**: il sistema prima o poi converge ad uno stato consistente
- *Soft state ed eventual consistency sono tecniche che funzionano bene in presenza di partizionamenti, favorendo la disponibilità*
- Adottato tipicamente in database NoSQL

::: Risoluzione Conflitti

- Strategie per decidere come risolvere conflitti su copie divergenti a causa di aggiornamenti concorrenti
- Un approccio comune: “***last writer wins***”
 - Etichettare i dati con orologi vettoriali per catturare la causalità tra diverse versioni dei dati
- Un’alternativa è demandare la risoluzione all’applicazione stessa (e.g., Amazon Dynamo) che invoca un gestore dei conflitti (*conflict handler*) specificato dall’utente
- Quando “riconciliare”?
 - Tipicamente, a tempo di *read* (e.g., Amazon Dynamo)
 - Alternative: durante una *write* (*write repair*) e *asynchronous repair*

::: Modelli User-centrati (1/6)

- Consistenza ***monotonic-read*** (o *read-after-read*)
- Consistenza ***monotonic-write*** (o *write-after-write*)
- Consistenza ***read-your-writes*** (o *read-after-write*)
- Consistenza ***writes-follow-reads*** (o *write-after-read*)
- *Obiettivo:* fornire garanzie **ad un singolo client** relative alla consistenza degli accessi da parte di quel client ad un archivio di dati distribuito
 - Nessuna garanzia di consistenza relativamente agli accessi concorrenti da parte di altri client

::: Modelli User-centrati (2/6)

- $x_i[t]$: versione di x sulla copia locale L_i al tempo t
- $WS(x_i[t])$: sequenza di operazioni di scrittura (Write Set, WS) su L_i che hanno portato come risultato a $x_i[t]$
- $WS(x_i[t_1]; x_j[t_2])$: le operazioni in $WS(x_i[t_1])$ sono state eseguite anche su L_j in un tempo successivo t_2
- Significa che $WS(x_i[t_1])$ è parte di $WS(x_j[t_2])$
- Assumiamo che solo il processo proprietario dei dati possa modificarli (assenza di conflitti **write-write**)

::: Modelli User-centrati (3/6)

Monotonic Read

- Se un processo legge il valore di un dato x , qualunque successiva operazione di lettura su x da parte di quel processo restituirà sempre quello stesso valore o un valore più recente
- **Esempio 1:** leggere automaticamente gli aggiornamenti al proprio calendario personale da diverse repliche del servizio
 - La consistenza *monotonic-read* garantisce all’utente di vedere tutti gli aggiornamenti, a prescindere dalla replica su cui avviene la lettura
- **Esempio 2:** leggere (senza modificare) la posta in arrivo mentre ci si sposta
 - Ogni volta che l’utente si connette ad una diversa replica del mail server, la replica carica (almeno) tutti gli aggiornamenti relativi alla mailbox dell’utente dalla replica usata precedentemente

::: Modelli User-centrati (4/6)

Monotonic Write

- Un'operazione di scrittura da parte di un processo su un dato x viene completata prima di qualunque operazione di scrittura successiva su x da parte dello stesso processo
- L'archivio garantisce di serializzare le scritture per il singolo client
- **Esempio:** mantenere versioni di file replicati nell'ordine corretto su ogni server (propagando la versione precedente sul server dove è installata la nuova versione)

::: Modelli User-centrati (5/6)

Read-your-writes

- L'effetto di un'operazione di scrittura da parte di un processo su un dato x sarà sempre visto da una successiva operazione di lettura di x da parte dello stesso processo
- E' un caso speciale di consistenza causale per il singolo client
- **Esempio:** aggiornare la propria pagina Web e garantire che il browser mostri la versione più recente anziché la copia in cache

::: Modelli User-centrati (6/6)

Writes-follow-reads

- Un'operazione di scrittura da parte di un processo su un dato x che segue una precedente operazione di lettura di x da parte dello stesso processo ha luogo sullo stesso valore di x che è stato letto o su un valore più recente
- **Esempio:** vedere i commenti ad un articolo inserito in un *newsgroup* solo se è stato visto l'articolo originale (una lettura “tira” la corrispondente operazione di scrittura)