Università degli Studi di Roma “Tor Vergata”

Facoltà di Ingegneria

Tolleranza ai Guasti nei Sistemi Distribuiti

Corso di Sistemi Distribuiti

Valeria Cardellini

Anno accademico 2008/09

Dependability

- Per comprendere il concetto di tolleranza ai guasti, analizziamo cosa si intende con **dependability**
  - Abilità di un sistema di fornire un servizio che può essere considerato fidato in maniera giustificata
  - Abilità di un sistema di evitare interruzioni di servizio più frequenti ed importanti di quanto accettabile
- Un *componente* fornisce *servizi* ai suoi *clienti*; per fornire tali servizi, il componente può richiedere i servizi di altri componenti → un componente può **dipendere** da altri componenti
  - Un componente *C* dipende da *C** se la correttezza del comportamento di *C* dipende dalla correttezza del comportamento di *C**
  - Nei SD un componente può essere un *processo* o un *canale di comunicazione*
- Un sistema *dependable* mostra le seguenti proprietà:
  - Disponibilità, affidabilità, sicurezza, manutenibilità, integrità

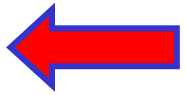
Dependability (2)

- **Disponibilità** (availability)
  - Sistema pronto per essere usato
    - Probabilità in funzione del tempo che il sistema sia correttamente operativo all'istante *t*
- **Affidabilità** (reliability)
  - Sistema funzionante senza guasti in maniera continuativa
    - Probabilità in funzione del tempo che il sistema sia correttamente funzionante all'istante *t* se il sistema stesso era funzionante all'istante 0
- **Sicurezza** (safety)
  - Se il sistema smette di operare correttamente, non succede nulla di catastrofico per l'utente e l'ambiente
- **Manutenibilità** (maintainability)
  - Facilità con cui il sistema può essere riparato dopo un guasto
- **Integrità** (integrity)
  - Assenza di alterazioni improprie del sistema

Terminologia

- **Failure** (*fallimento*): quando il comportamento di un componente non è conforme alle sue specifiche
- **Error** (*errore*): quella parte di un componente che può determinare una failure
- **Fault** (*guasto*): la causa di un errore
  - Guasti transienti, intermittenti o permanenti

fault $\rightarrow$ error $\rightarrow$ failure

- **Prevenzione di guasti**
  - Prevenire l'occorrenza dei guasti
- **Tolleranza ai guasti** (fault tolerance) 
  - Costruire un componente in modo tale che sia conforme alle sue specifiche anche in presenza di guasti (mascherare i guasti)
- **Rimozione dei guasti**
  - Ridurre la presenza, il numero, la serietà dei guasti
- **Predizione dei guasti**
  - Stimare l'incidenza futura e le conseguenza dei guasti

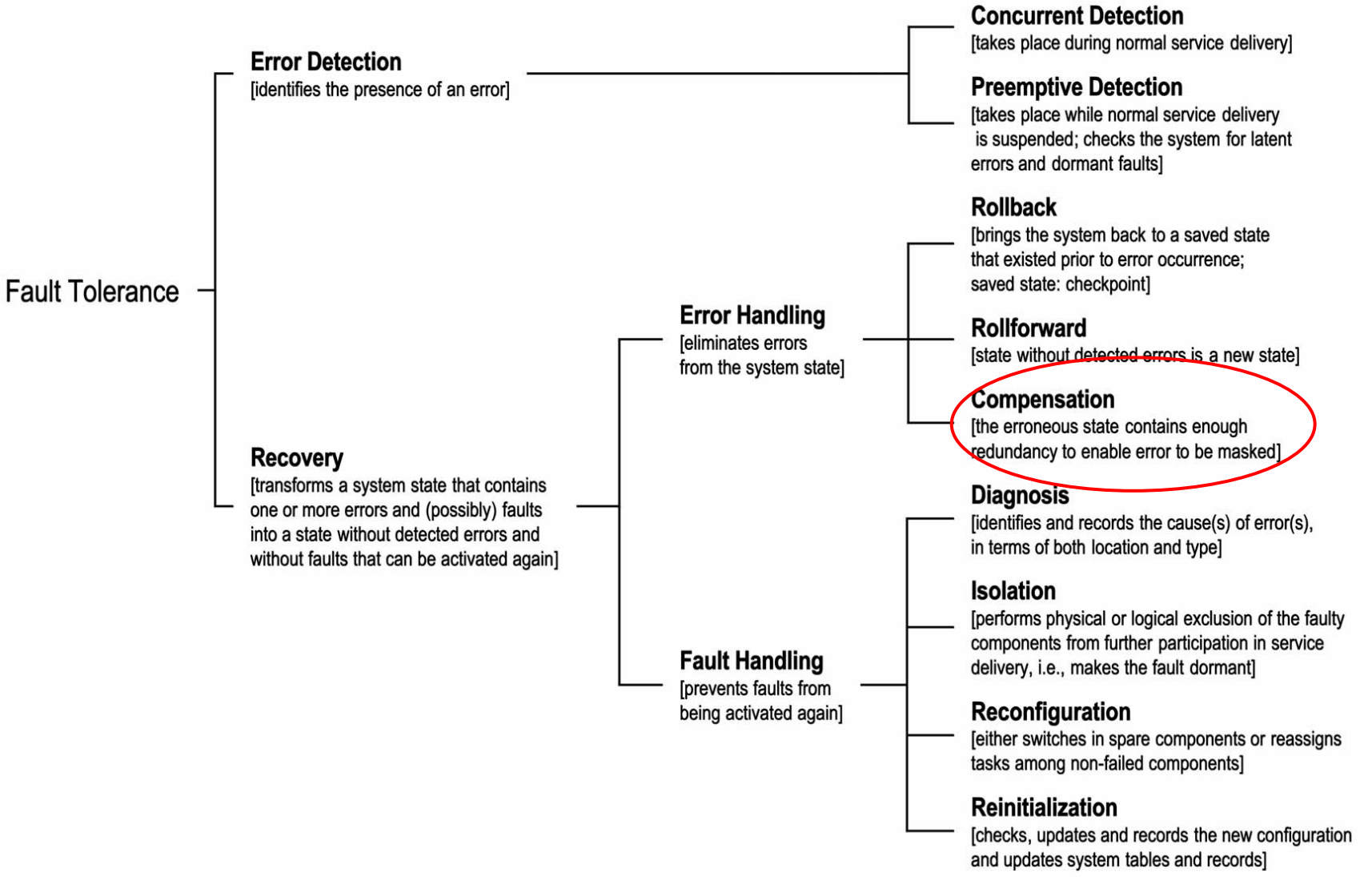
Modelli di failure

- Quali sono i diversi tipi di failure?
  - Crash: il componente si arresta, ma funziona correttamente fino a quel momento
  - Omissione: il componente non risponde alle richieste
  - Fallimento nella temporizzazione: la risposta del componente è corretta, ma il tempo di risposta è al di fuori dell'intervallo specificato
  - Fallimento nella risposta: la risposta del componente non è corretta
    - Fallimento nel valore
    - Fallimento nella transizione di stato
  - Fallimenti arbitrari (o *bizantini*): il componente può produrre una risposta arbitraria ed essere soggetto a fallimenti arbitrari nella temporizzazione
- I crash sono i fallimenti più innocui, quelli bizantini i più gravi

Modelli di failure (2)

- *Problema*: i client non possono distinguere tra un componente che ha subito un crash ed uno che è solo troppo lento
  - Esempio: consideriamo un server dal quale un client sta aspettando una risposta
    - Il server è soggetto ad un fallimento nella temporizzazione o ad una omissione?
    - Il canale di comunicazione tra client e server è soggetto ad un guasto?
- Fallimento fail-silent: il componente ha subito un crash od una omissione; i client non possono saperlo
- Fallimento fail-stop: il componente ha subito un crash, ma il suo fallimento può essere scoperto (tramite un timeout o un preannuncio)
- Fallimento fail-safe: il componente ha subito un fallimento arbitrario, ma senza conseguenze serie

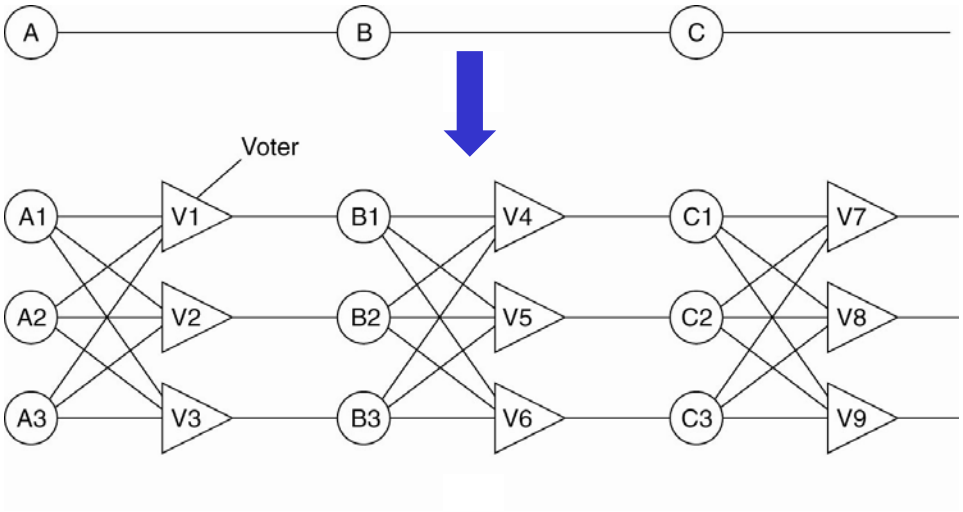
Tecniche per la tolleranza ai guasti



Fonte: Avizienis et al, “Basic concepts and taxonomy of dependable and secure computing”

Ridondanza

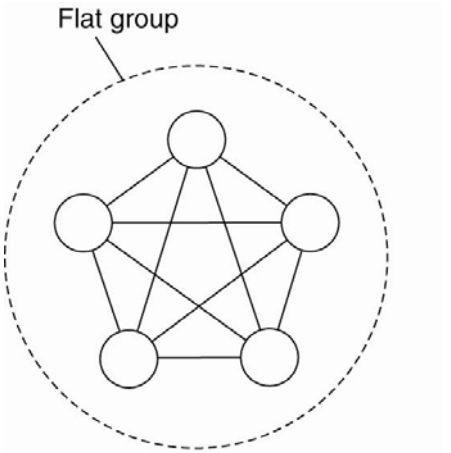
- Tipologie di ridondanza
  - Ridondanza delle informazioni
  - Ridondanza nel tempo
  - Ridondanza fisica
    - A livello hardware o software
- Esempio di ridondanza fisica: Triple Modular Redundancy
  - 3 componenti replicati eseguono un'operazione, il cui risultato viene sottoposto ad un sistema di voting per produrre un unico output
  - Se uno dei tre componenti fallisce, gli altri due possono mascherare e correggere il guasto

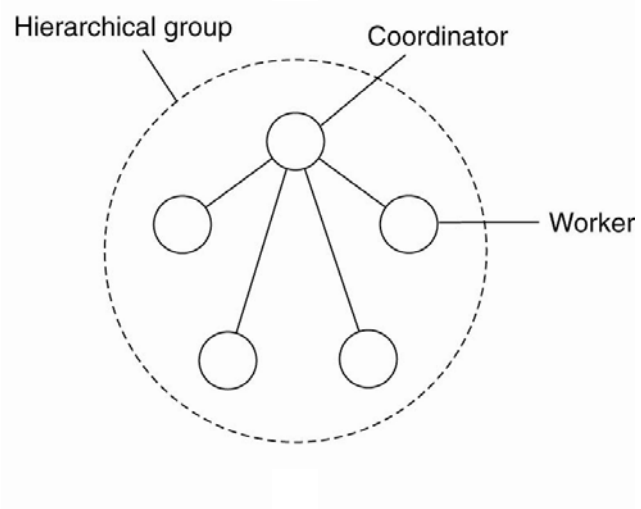


Perché 3 voter e non uno solo?

Resilienza dei processi

- Per tollerare il guasto di un processo, occorre replicare e distribuire la computazione in un gruppo di processi identici
- Gruppo flat (o semplice)
  - Adatto per la tolleranza ai guasti (simmetrico e no single point of failure)
  - Maggiore overhead in quanto il controllo è completamente distribuito (difficile da implementare)
- Gruppo gerarchico: tutte le comunicazioni attraverso un singolo coordinatore
  - No veramente tollerante ai guasti, ma relativamente semplice da implementare





Modelli di replicazione

- Replicazione passiva (protocolli primary-based nella consistenza)
  - Gruppo di processi organizzati in modo gerarchico
    - Un solo processo (attivo) esegue le azioni sui dati; le altre repliche (passive) servono in caso di guasto
    - Una sola replica corretta, le altre possono anche non essere aggiornate (repliche calde o fredde)
    - Possibile scollamento dello stato tra il coordinatore ed i worker
      - Il coordinatore aggiorna lo stato dei worker (checkpoint)
    - Se il coordinatore subisce un crash, le altre repliche eseguono un algoritmo di elezione per individuare il nuovo coordinatore
- Replicazione attiva (protocolli replicated-write nella consistenza)
  - Gruppo di processi organizzati in modo flat
  - Occorre il coordinamento tra le repliche attive
    - Costi accettabili solo per gradi di replicazione limitati e per politiche di coordinamento semplici

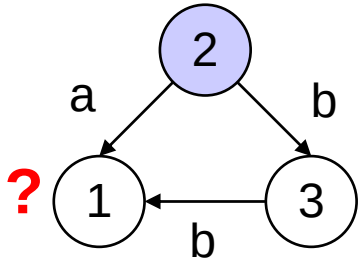
Gruppi e mascheramento dei guasti

- Consideriamo per semplicità solo gruppi flat
- Un gruppo è **k-fault tolerant** se può mascherare *k* fallimenti concorrenti
  - *k* è il grado di tolleranza ai guasti
- Quanto deve essere grande un gruppo *k*-fault tolerant?
  - Se il fallimento è fail-silent $\rightarrow$ *k*+1 processi
  - Se il fallimento è arbitrario ed il risultato del gruppo è definito tramite un meccanismo di voto $\rightarrow$ 2*k*+1 processi
- *Assunzione* implicita: tutti i processi sono identici e processano tutti gli input nello stesso ordine (problema del *multicasting atomico*)
  - Per essere certi che tutti i processi facciano esattamente la stessa cosa

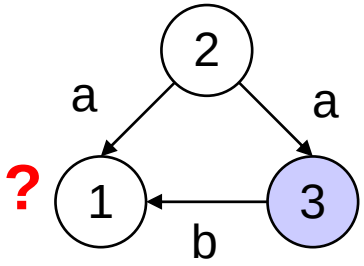
Gruppi e mascheramento dei guasti (2)

- *Assunzione*: i processi del gruppo *non* sono identici, ovvero c'è una computazione distribuita
- *Obiettivo*: i processi non guasti del gruppo devono raggiungere un **consenso** sullo stesso valore

Il processo 2 dice cose diverse



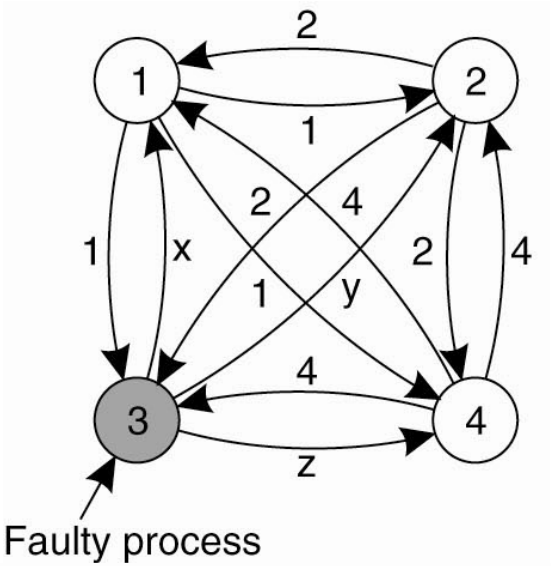
Il processo 3 passa un valore diverso



- Assumendo fallimenti arbitrari, occorrono $3k+1$ processi nel gruppo per sopravvivere ad attacchi di k processi guasti
  - E' il problema dei generali bizantini (definito da Lamport)
  - *Idea*: stiamo cercando di raggiungere un voto di maggioranza tra un gruppo di generali fedeli, essendoci k generali traditori→ occorrono $2k+1$ generali fedeli
    - Se non ci sono più dei 2/3 di generali fedeli, non è possibile raggiungere il consenso

Problema del consenso bizantino

- *Assunzioni*: processi sincroni, comunicazione unicast con ordinamento dei messaggi e ritardi limitati
- Come raggiungere un accordo nel caso di 3 processi che funzionano correttamente ed 1 no (*N*=4, *k*=1)?



		1 Got	2 Got	4 Got
1	Got(1, 2, x, 4)	(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
2	Got(1, 2, y, 4)	(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
3	Got(1, 2, 3, 4)	(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)
4	Got(1, 2, z, 4)			

Problema del consenso bizantino (2)

Algorithm OM(0)

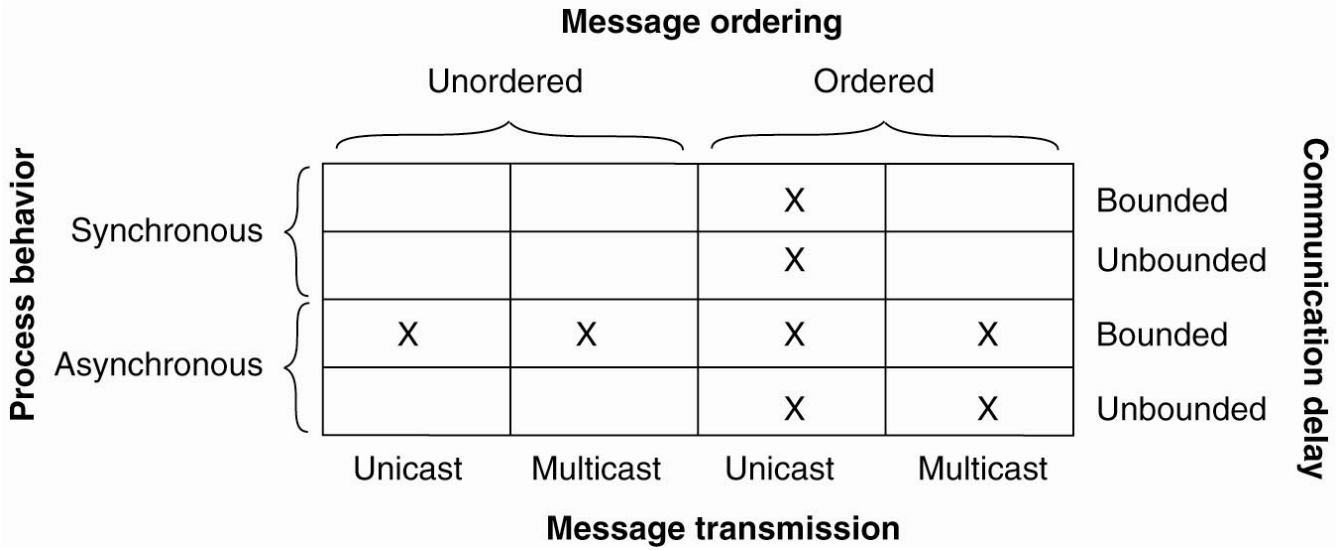
1. The commander sends his value to every lieutenant
2. Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value

Algorithm OM(k), k>0

1. The commander sends his value to every lieutenant
2. For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM(k-1) to send the value v_i to each of the N-2 other lieutenants
3. For each i, and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step 2 (using Algorithm OM(k-1)), or else RETREAT if he received no such value. Lieutenant i uses the value majority (v_1, …, v_{N-1})

Consenso nei sistemi guasti

- Quali sono le condizioni necessarie per raggiungere il consenso nei sistemi guasti?



- Processi: sincroni → operano in modalità lockstep
- Ritardi: il ritardo nella comunicazione è limitato?
- Ordinamento: i messaggi sono consegnati nell'ordine i cui sono stati inviati?
- Trasmissione: i messaggi sono trasmessi in unicast o multicast?

Rilevamento di fallimenti

- Due meccanismi per rilevare un fallimento

1. Invio di un messaggio e uso di un meccanismo di timeout per rilevare se un processo è fallito
   - Soluzione più usata
2. Attesa passiva di un messaggio

- Meccanismo di timeout
  - Difficoltà nell'impostazione del valore del timeout e dipendenza dall'applicazione
  - Difficoltà nel distinguere tra fallimenti dei processi o fallimenti della rete di comunicazione
- Il rilevamento dei fallimenti può anche essere ottenuto in modo proattivo come effetto collaterale dello scambio delle informazioni coi i vicini
  - Ad es. tramite disseminazione delle informazioni basat su protocolli epidemici o di gossiping

Comunicazione affidabile

- Finora ci siamo concentrati sulla resilienza dei processi (mediante i gruppi di processi). Occorre considerare anche i fallimenti della comunicazione
  - I modelli di fallimenti analizzati si applicano anche ai canali di comunicazione
- **Rilevamento di errori**
  - Frammentazione dei pacchetti per permettere di individuare errori a livello di bit
  - Uso della numerazione per rilevare la perdita di un pacchetto
- **Correzione di errori**
  - Aggiungere ridondanza di informazione in modo tale che i pacchetti corrotti possano essere automaticamente corretti
  - Richiedere la ritrasmissione dei pacchetti persi o degli ultimi N pacchetti
- *Osservazione*: la maggior parte di queste tecniche tradizionali considerano una comunicazione punto-punto

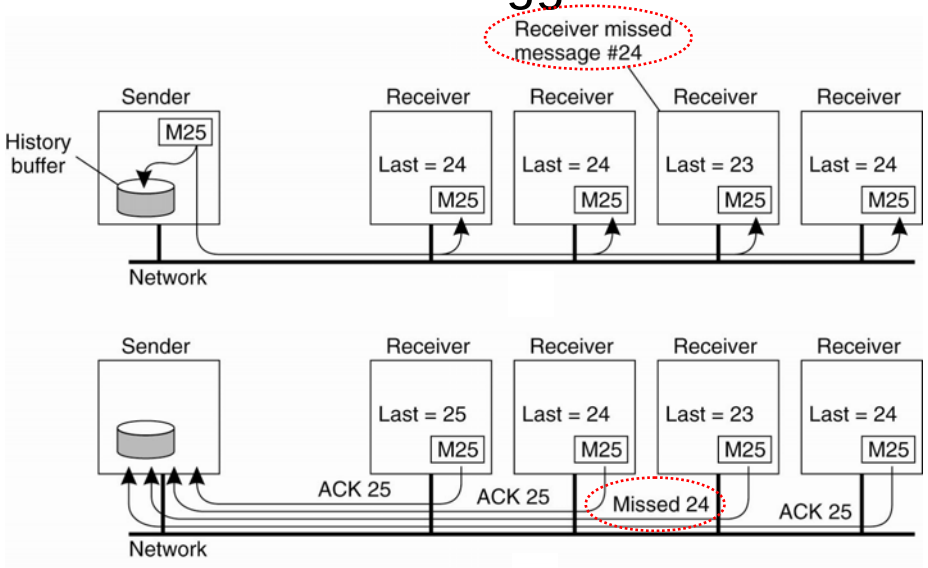
Multicasting affidabile

- Come realizzare la comunicazione affidabile nei gruppi?
- *Modello di base*: un canale di multicast con due gruppi (possibilmente sovrapposti):
  - Il gruppo mittente *SND*(*c*) di processi che sottomettono messaggi al canale *c*
  - Il gruppo destinatario *RCV*(*c*) di processi che possono ricevere messaggi dal canale *c*
- **Affidabilità semplice**: se il processo $P \in RCV(c)$ al tempo in cui il messaggio *m* è stato sottomesso a *c* e *P* non esce da *RCV*(*c*), *m* deve essere consegnato a *P*
- **Multicast atomico**: come assicurare che un messaggio *m* sottomesso al canale *c* sia consegnato al processo $P \in RCV(c)$ solo se *m* viene consegnato a tutti i processi membri di *RCV*(*c*)

Multicasting affidabile (2)

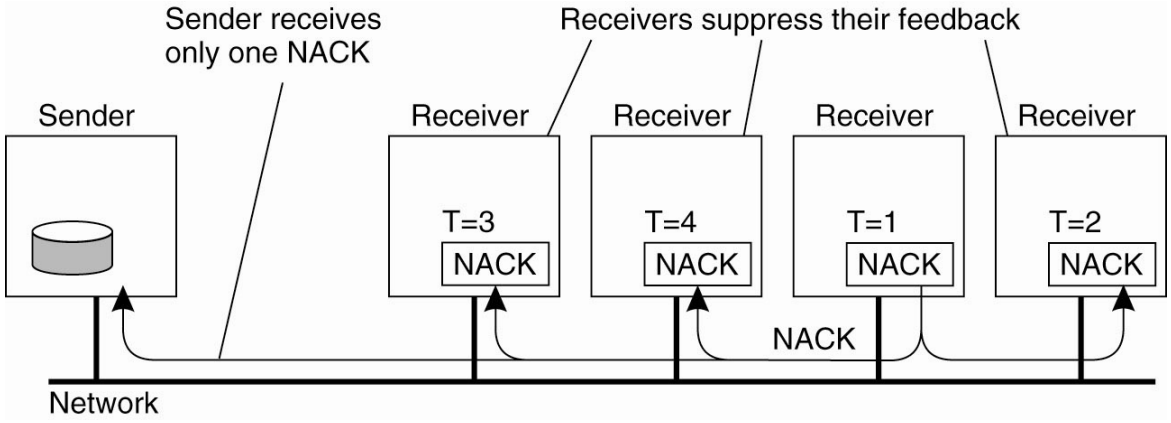
- In una LAN il multicasting affidabile semplice è relativamente semplice da realizzare
- Il mittente registra in un log i messaggi sottomessi al canale *c*
  - Se *P* invia il messaggio *m*, *m* è memorizzato in un history buffer
  - Ogni destinatario invia un acknowledgment (ACK) della ricezione di *m*, o richiede la ritrasmissione (NACK) a *P* quando nota la perdita del messaggio
  - Il mittente *P* cancella *m* dall'history buffer quando tutti i destinatari di *m* hanno ricevuto il messaggio

Perché questa soluzione non è scalabile?



Multicast affidabile scalabile

- Soluzione: soppressione dei feedback
  - Per evitare il fenomeno *dell'implosione dei feedback* sul mittente, il processo *Q* cancella il proprio feedback (NACK) quando nota che un altro processo *R* sta già chiedendo la ritrasmissione
- Assunzioni
  - Tutti i destinatari sono in ascolto su un canale di feedback comune su cui vengono inviati i messaggi di feedback
  - Il processo *Q* schedula il suo messaggio di feedback *casualmente* e lo cancella quando osserva un altro messaggio di feedback



Argomenti non trattati

- Argomenti importanti per la problematica di questa lezione non trattati per limiti temporali del corso
  - Multicast atomico
    - Come affrontare il multicast affidabile quando sono presenti fallimenti sui processi in termini di gruppi di processi e cambiamenti di appartenenza ai gruppi?
  - Commit distribuite
    - Data una computazione distribuita su un gruppo di processi, come possiamo assicurare che tutti i processi eseguano il commit del risultato finale o che nessuno di loro lo esegua (atomicità)?
  - Recupero dai fallimenti
    - Come riportare il SD in uno stato esente da errori e consistente in seguito ad un fallimento?