Corso di SISTEMI OPERATIVI

La tolleranza ai guasti

Esempi

Professore:
William Fornaciari

Sommario

- Fault tolerance nei sistemi Tandem
- Logged Virtual memory
- Un meccanismo di checkpoint per s.o. real-time
- Leases: un meccanismo FT per la consistenza della cache distribuita
- FT attraverso le comunicazioni di gruppo
- Gruppi di conversazione
- Sincronizzazione e ridondanza
- Servizi di log condiviso per sistemi distribuiti FT
- Memoria transazionale stabile
- Computazione parallela su una rete di workstations
- Dischi RAID

FT nei sistemi Tandem
Introduzione

- Sistema multiprocessore, con CPU connesse in rete locale gerarchica FT
- Le periferiche necessarie alle transazioni sono comandate da controller a doppia porta
- Il sistema software prevede messaggi e processi dedicati all'isolamento dei guasti
- Le applicazioni sono progettate in modo da richiedere processi mediante Remote Procedure Call (RPC) a un processo server. Sono i processi server a sfruttare i multiprocessori
- Il processo complessivo astratto è un sistema distribuito su migliaia di processori con MTBF di anni

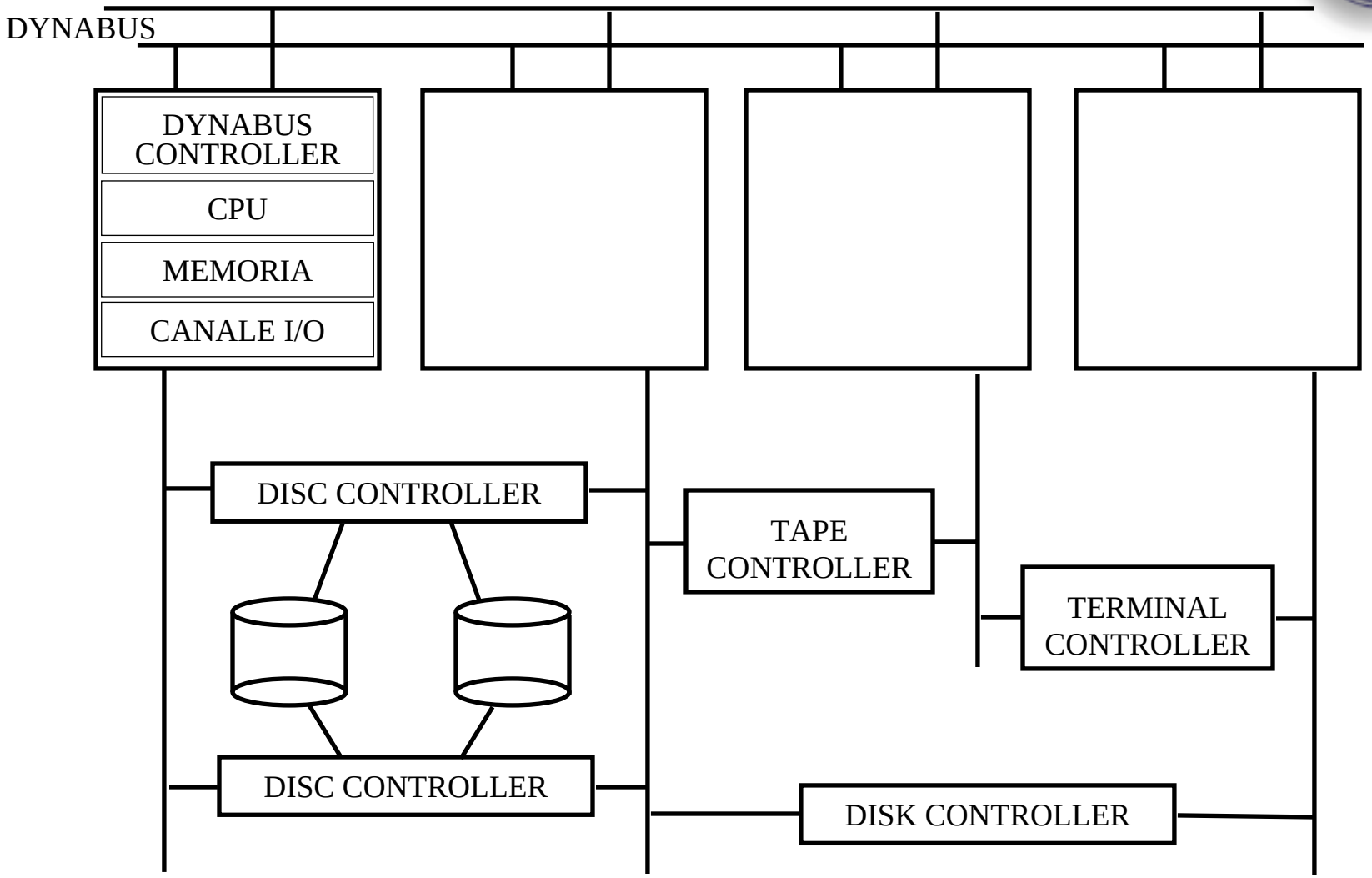
FT nei sistemi Tandem
Principi generali di design

- **modularità**: hw e sw in moduli a granularità fine: unità di servizio, guasto, diagnosi e riparazione.
- **Blocco immediato**: ogni modulo prevede diagnosi interna e in caso di guasto si ferma immediatamente
- **guasto singolo**: il sistema hw e sw non risente di un guasto al singolo modulo; non si blocca durante la sua riparazione; e la reintegrazione del modulo riparato non è motivo di interruzione
- **manutenzione on-line**
- **interfaccia utente semplificata** per evitare che interfacce troppo complesse possano essere fonte di guasti

FT nei sistemi Tandem
Architettura Tandem

- Vediamo il sistema *Tandem Non Stop* (1976)
- Il sistema è costituito da 2-16 processori collegati tra loro da un bus a 13 MB/sec. Ogni processore possiede memoria (con una copia del sistema operativo) e bus di I/O.
- Ogni controller ha due porte, ciascuna collegata a un processore diverso: in caso di guasto sulla linea o sul processore si attiva l'altra linea o l'altro processore.
- Tutti i moduli (controller, dischi, ecc.) sono duplicati, di modo che il guasto di uno di essi venga mascherato dalla presenza del gemello
- In caso di guasto di un processore, il carico di lavoro viene diviso sugli altri processori

FT nei sistemi Tandem
Architettura Tandem



FT nei sistemi Tandem
Architettura Tandem - CPU

- Sostanzialmente simile ad una CPU tradizionale
- Ogni processore opera in modo indipendente dagli altri e in modo asincrono
- Si richiede che un guasto non si propaghi, quindi in caso di guasto un processore deve disabilitarsi, impedendo la trasmissione di informazioni errate sia attraverso dynabus, sia attraverso il bus di I/O
- In alcuni sistemi Tandem è possibile ritentare l'esecuzione di un'istruzione dopo il suo fallimento, ripartendo dalla cache di primo livello, che ha una propria RAM nel caso la RAM del processore si guastasse

FT nei sistemi Tandem
Architettura Tandem - Periferiche

- Con disco duplicato e doppio controllore a doppia porta si hanno 8 vie d'accesso per l'I/O di dati
- Per gestire queste possibilità si è passati da un set di switch in dotazione a ciascun processore a un processore Z80 dedicato (Operation and Service Processor, OSP)
- Se è necessaria FT anche sull'OSP, si passa al sistema CHECK, ossia un dual-68000 che comunica col resto del sistema con un doppio bus
- Ogni guasto è registrato in un log dal CHECK, e viene da esso comunicato ad un sistema esperto che gira sul processore principale, che gestisce il guasto

FT nei sistemi Tandem
Architettura Tandem - Periferiche

- In caso di guasto, i controller devono:
  - avere gli stessi meccanismi di blocco immediato come i processori
  - se possibile notificare al loro processore il fallimento dell'I/O
  - supportare la rilevazione automatica degli errori, al di là dei controlli software a livello superiore
- Il progettista deve tenere a mente che la duplicazione, nei sistemi Tandem è molto costosa, quindi bisogna tenere basso il costo delle componenti

Kernel

- Supporta processi multipli, che condividono memoria fisica solo in read-only (i guasti non si propagano). I processi comunicano con messaggi
- Si occupa della trasmissione dei messaggi tra processi e della riconfigurazione in caso di avaria
- Può spedire messaggi a processi di un altro processore, ossia ad altri kernel che poi consegnano il messaggio, occupandosi lui stesso del *routing*
- Nasconde errori di memoria read-only ricaricando la pagina corrispondente da disco
- Si accorge della caduta di un altro processore in 2 sec

Coppie di processi

- La tolleranza a guasto singolo vuole che in caso di bug o caduta del processore, l'applicazione proceda
- Esiste quindi una coppia di processi: ogni processo ha un suo gemello di back-up, che gira su un altro processore, con risorse completamente duplicate
- In determinati istanti, il processo primario manda un messaggio di checkpoint a quello di back-up, che raggiunge così lo stesso stato del processo primario
- In caso di fallimento del processo primario, il kernel ridirige tutti i messaggi al processo di backup a partire dall'ultimo checkpoint

Classi di processi server

- Definizione: è una collezione di processi, ripartiti su diversi processori, che svolgono tutti la stessa funzione
- Le richieste di particolari servizi sono fatte **alla classe** corrispondente anziché al singolo processo
- Se il carico di quel servizio aumenta, vengono aggiunti nuovi membri alla classe
- Introdotte per evitare che un processo, invocato (per la modularità del software) da diverse applicazioni, diventi un collo di bottiglia

FT nei sistemi Tandem
Architettura Tandem – Software di sistema

File

- I file possono essere partizionati su più dischi e ogni partizione è duplicata su un mirror. Una classe di processi supervisiona ogni disco
- In lettura il supervisore accede al disco più veloce
- In scrittura il supervisore scrive su entrambi i dischi

Transazioni

- Nei Tandem esiste il Transaction Monitor Facility (TMF), che etichetta ogni job, compresi i record di undo e redo con un identificatore di transazione.
- I processi doppi servono quindi solo per implementare il commit, risparmiando risorse

FT nei sistemi Tandem
Architettura Tandem – Software di sistema

Applicazioni software

- Massima semplicità dell'interfaccia per evitare errori dell'utente
  - interfaccia grafica a menù
  - supporto per basi di dati relazionali
  - supporto a più linguaggi di programmazione

Manutenzione

- Il Tandem cerca di eliminare l'intervento umano
  - Gestisce automaticamente operazioni di routine
  - Fornisce un diagnostico passo passo per interventi eccezionali

Logged Virtual Memory (LVM)

- Fornisce un log di ogni attività di scrittura su una o più porzioni di memoria virtuale
- Ogni record del log è costituito da
  - data e ora (timestamp)
  - indirizzo di scrittura
  - valore del dato scritto
  - lunghezza del dato scritto
- I record sono scritti sequenzialmente in un segmento di memoria dedicato e accessibile
- Semplificazione del recupero dati in seguito a crash
- Aumento della consistenza in caso di scritture di diversi processi alla stessa area

Checkpoint per s.o. real time
Architettura del protocollo

Il protocollo deve soddisfare le seguenti caratteristiche

- **Supporto real time** per applicazioni composte da task che girano in modo concorrente. Un checkpoint su un task di comunicazione o consistenza dei dati deve forzare un checkpoint su tutti i task ad esso collegati
- **Mascheramento degli errori**: in caso di guasto non deve essere necessario l'intervento di un'applicazione real-time
- **Trasparenza**: esegue il checkpoint il s.o. stesso
- **Portabilità**: si ottiene usando lo standard Posix.1b
- **Unità stabile** per memorizzare i dati di recovery

Checkpoint per s.o. real time
Analisi del protocollo

- Bisognerebbe eseguire un checkpoint ad ogni chiamata di sistema, ma così facendo si riduce la performance, a causa dell'enorme mole di dati
- Per ridurne il numero, si salvano i checkpoint quando:
  - termina un ciclo di controllo
  - vengono eseguite alcune operazioni di scrittura
  - viene creato un nuovo processo (FORK)
- La cancellazione di un checkpoint si esegue quando:
  - un processo finisce la propria esecuzione (EXIT)
  - cambia l'immagine di esecuzione (EXEC): gli indirizzi virtuali prima e dopo l'EXEC non sono più gli stessi

Checkpoint per s.o. real time
Informazioni contenute nel checkpoint

- Tutte le pagine dell'utente modificate dopo il precedente checkpoint. Quindi ogni pagina necessita di un flag apposito
- L'entry della tabella dei processi del processo in uso, insieme ad altre informazioni aggiuntive
  - files aperti
  - stato di schedulazione
  - ...
- Strutture interne al s.o. relative ad eventuali comunicazioni del processo controllato
  - semafori
  - memoria condivisa
  - code di messaggi aperte

Checkpoint per s.o. real time
Tracciamento delle dipendenze

- Forza bruta: checkpoint globale di tutti i processi in atto, anche non comunicanti. Comporta ritardi e perdita di deadline
- Si può allora eseguire un checkpoint quando due processi comunicano, ma anche questo crea ritardi
- Metodo migliore: tracciare le dipendenze tra processi di una stessa applicazione (=> deadline mancate ridotte del 90%). Si sfruttano
  - code di messaggi
  - semafori Posix.1b
  - oggetti di memoria condivisa
  - pipes
  - segnali

FT per la consistenza di cache distribuita
Introduzione

- In qualunque sistema i vantaggi del caching sono sempre ridotti dagli algoritmi per la consistenza dei dati
- I protocolli di consistenza per sistemi multiprocessore non sono adatti alla FT per sistemi distribuiti, con canali di comunicazione molto più insicuri di un bus di sistema.
- Protocollo **Leases** (letteralmente “contratti”): mantiene la consistenza usando dei contratti di breve durata, basati su clock fisici.

FT per la consistenza di cache distribuita
I Leases

- **Lease**: contratto che dà alla cache che lo possiede diritti di scrittura di un dato per un certo periodo
- Una cache che carichi un dato, prima di renderlo leggibile, chiede al server un lease sul dato stesso
- Il lease concesso dal server garantisce l'integrità del dato fino alla scadenza del contratto.
- Se un client tenta di scrivere un dato, il server può chiedere alla cache titolare del lease il permesso di scrittura.
  - in caso di assenso, il server scrive il dato e la cache invalida il dato in suo possesso
  - altrimenti il server attende la scadenza del lease

FT per la consistenza di cache distribuita
I Leases

- Contratti a breve scadenza hanno come vantaggi
  - minimizzare i ritardi dovuti alla caduta di un client o di un server
  - minimizzare i conflitti *di contratto*, che si creano nonostante non ci siano effettivi conflitti di accesso
- Contratti a lunga scadenza hanno come vantaggio
  - diminuire l'overhead dei client per gestire una minore quantità di lease
- Si può formalizzare matematicamente questo trade-off, e calcolare così la durata ottimale di un contratto.

FT per la consistenza di cache distribuita
FT nel protocollo *Leases*

- Se le scritture sono persistenti sul server, Leases assicura la consistenza dei dati, se i guasti sono:
  - perdita di messaggi
  - errori del client o del server
- Leases dipende dalle velocità dei clock del server e dei client
  - se il clock del server è più rapido di quello del client si possono verificare inconsistenze (il server considera scaduto il contratto prima del client e viceversa)
  - se il clock del server è più lento di quello del client non si verificano inconsistenze, ma aumenta il traffico
- È sufficiente introdurre un protocollo di sincronizzazione

FT tramite comunicazioni di gruppo
Introduzione

- Se la FT si realizza replicando i dati (ridondanza), allora è cruciale la consistenza dei dati replicati.
- Per questo si possono utilizzare comunicazioni broadcast (un solo trasmittente, *n* riceventi), ma:
  - Esiste l'hardware che supporta il broadcast
  - Esistono pochi s.o. distribuiti che supportano il broadcast, per la lentezza che la sua sicurezza imporrebbe.
- Illustriamo quindi un nuovo sistema che riduce mediamente a 2 i messaggi che servono ad avere la certezza che ***tutti*** abbiano ricevuto il messaggio

FT tramite comunicazioni di gruppo
Primitive del protocollo

- Ipotesi:
  - sistema distribuito comunicante via LAN
  - i processi non mandano messaggi contraddittori
- Il protocollo è compilato nel Kernel
- Primitive (indicate senza parametri):
  - **CreateGroup**: crea un nuovo gruppo e specifica quanti membri guasti sono tollerati senza perdere messaggi
  - **JoinGroup**: costruisce un processo membro del gruppo
  - **LeaveGroup**: l'ultimo che abbandona cancella il gruppo
  - **SendToGroup**: spedisce un messaggio a tutti i membri
  - **ReceiveFromGroup**: il membro aspetta il messaggio
  - **ResetGroup**: ripristino dopo un guasto in un processo

FT tramite comunicazioni di gruppo
Fallimento delle comunicazioni

- Se un processo membro esegue *SendToGroup*, il suo kernel lo blocca e invia il messaggio *M*, in point-to-point, al **sequencer**, un membro speciale
- Il sequencer
  - dà ad *M* il suo numero di sequenza *s*
  - salva copia del messaggio
  - avvia il broadcast di (M,s)
- Solo quando un broadcast è terminato può partirne un altro. (Ogni broadcast ha *s* aumentato di 1 rispetto al precedente)
- Quando il kernel mittente riceve (M,s), sblocca il processo chiamante restituendogli *s*

FT tramite comunicazioni di gruppo
Fallimento delle comunicazioni

- Supponiamo che un nodo non abbia ricevuto un pacchetto: gli arriva una comunicazione numerata *s+1*, mentre doveva ricevere *s*
- Il kernel del nodo chiede al sequencer (in point-to-point) la ritrasmissione (sempre point-to-point) di *s*
- La finitezza del buffer del sequencer si risolve cancellando i messaggi arrivati con successo
  - ogni kernel memorizza una tabella di messaggi ricevuti, contenenti anche il numero di sequenza
  - trascorso un certo tempo *t* dall'ultimo messaggio invia un pacchetto di ACK sui broadcast ricevuti
  - Se il sequencer ha bisogno di spazio, cancella il messaggio con numero di sequenza più vecchio

FT tramite comunicazioni di gruppo
Fallimento di un sequencer

- Se il kernel di un processo mittente non riceve il broadcast di ritorno del suo messaggio, ritenta per *n* volte, poi assume che il sequencer è caduto
- Può quindi invocare *ResetGroup* che ricostruisce un nuovo gruppo dal precedente
  - invita tutti gli altri membri nel nuovo gruppo
  - ottiene in risposta da tutti gli altri sopravvissuti il più alto numero *s* di sequenza da loro ricevuto
  - elegge a nuovo sequencer chi ha *s* più alto
  - Il nuovo sequencer manda a tutti l'informazione della sua elezione, ogni membro manda un ACK e si riparte
- Si devono verificare le tabelle dei kernel per vedere se tutti tutti i messaggi spediti siano stati ricevuti

FT tramite gruppi di conversazione
Introduzione

- Ipotesi: il sistema consiste di processi concorrenti che possono contemporaneamente comunicare. I processi possono essere
  - **indipendenti**
  - **competitivi**: due o più processi sfruttano risorse di sistema condivise ***separatamente*** (s.o., DBMS, …)
  - **cooperativi**: più processi hanno un fine comune e si aiutano ***sincronizzandosi*** e comunicandosi anche computazioni intermedie (ad es. sistemi di controllo)
- Le operazioni di conversazioni sono atomiche
- problema dell'isolamento: il recupero di una transazione non deve disturbare le altre

FT tramite gruppi di conversazione
Proprietà delle conversazioni

- I processi entrano in conversazione in modo asincrono
- Si comunica solo all'interno di una conversazione
- Quando i processi in conversazione terminano, si esegue un test di accettazione, reso noto a tutti:
  - se positivo, abbandonano *insieme* la conversazione
  - se negativo, ripristinano *insieme* lo stato al punto di recovery
- Se un processo cade durante una conversazione, tutti i processi eseguono *insieme* un rollback al punto di recovery
- All'interno di una conversazione si posso creare dei sottogruppi di conversazione

FT tramite gruppi di conversazione
Conversazioni e comunicazioni

- Le comunicazioni di gruppo sono altamente tolleranti ai guasti hw, ma poco a quelli sw (non atomicità dei messaggi)
- L'introduzione dei gruppi di conversazione, invisibili all'esterno, risolve anche la FT a livello sw
- La coesistenza dei due metodi è garantita dalle seguenti regole
  - Una conversazione è un'unità atomica e non può ricevere o mandare messaggi al mondo esterno
  - nessuno può mandare messaggi multicast ai membri di una conversazione
  - Al termine di una conversazione i partecipanti possono rimanere nello stesso gruppo

Sincronizzazione e ridondanza
Introduzione

- Un sistema distribuito completamente FT si può basare su sincronizzazione a più livelli e ridondanza dei componenti
- Ricerca errori, voting, riconfigurazione, operazioni real time, tempi minimi di recupero, numero di errori tollerati ecc. necessitano un'interazione tra componenti che richiede sincronia
- Sistemi complessi si basano su livelli gerarchici, su cui è naturale ricalcare analoghi livelli di sincronizzazione, tramite cui componenti ridondanti realizzano ogni successivo schema di FT

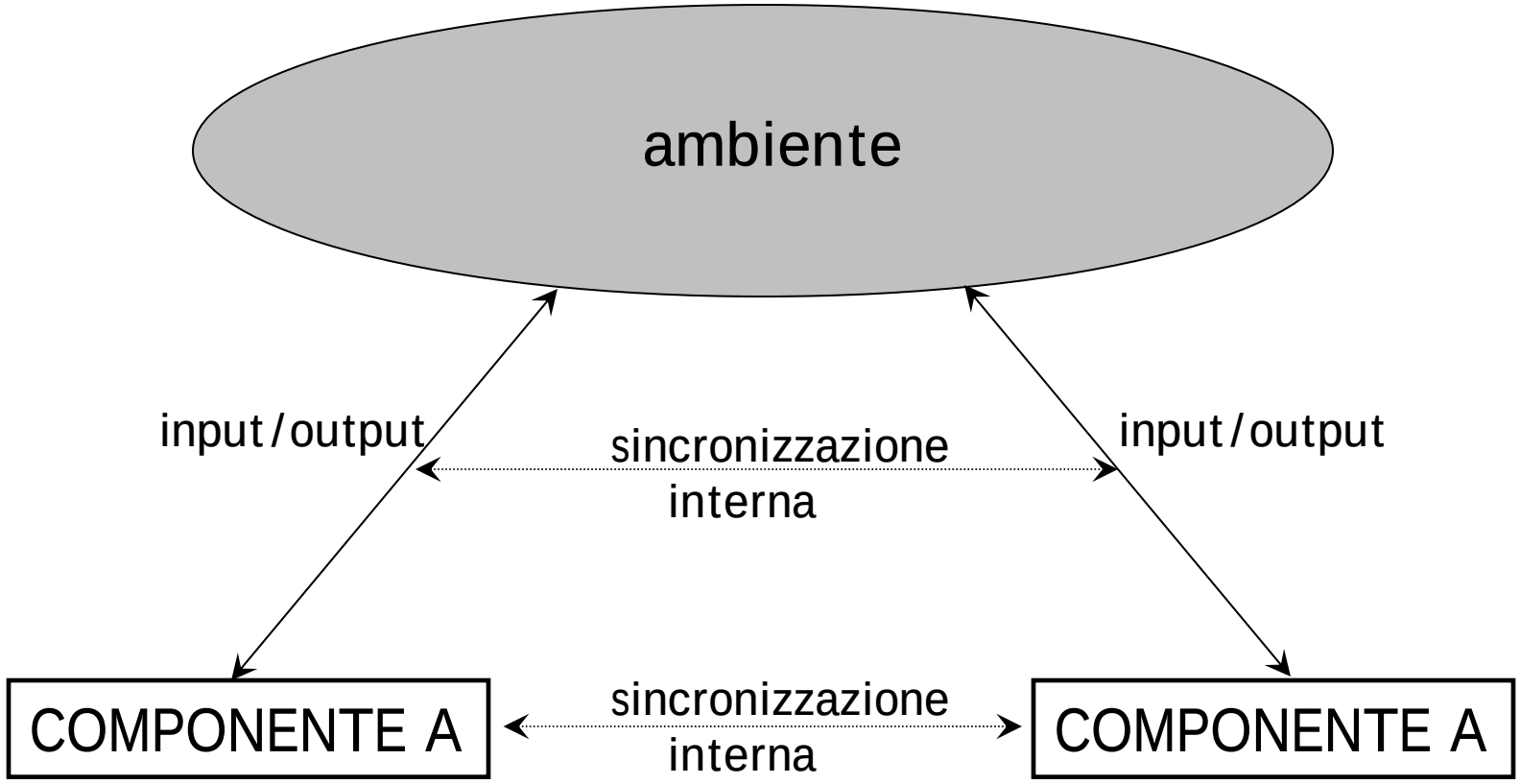
Sincronizzazione e ridondanza
Definizione

- Chiamiamo ***RS*** *(Redundancy synchronization)* la sincronizzazione di operazioni hw o sw ridondanti
- Non è richiesto che i componenti ridondanti siano identici, ma solo che svolgano lo stesso servizio
- Scopo della RS è sincronizzare tutte le risorse di FT a disposizione di un componente di alto livello (per esempio un utente DOS (Distributed Operating System). È essenziale per:
  - eliminare gli sfasamenti temporali tra operazioni
  - riconoscere guasti (è possibile confrontare diversi stati solo se sono sincronizzati)
  - recupero in caso di guasto

Sincronizzazione e ridondanza
Implementazione RS

- **Controllo processi e CPU ridondanti** (detta anche *replicazione dei processi*): sincronizzazione tramite messaggi
- **Controllo nodi ridondanti**: i risultati comuni (finali o intermedi) sincronizzati tramite clock comune
- **Controllo periferiche ridondanti***:* ad es. dischi shadow
- **Controllo di diversi input o output**: sincronizzazione sull'input e voting sull'output
- **Controllo files ridondanti**
- **Controllo timer ridondanti**: è lo stesso che sincronizzare dati in input, ma complicato dal fatto di dover avere un tempo comune a tutto il DOS

Sincronizzazione e ridondanza
Implementazione RS



Sincronizzazione e ridondanza
Implementazione RS

- Si hanno due tipi di sincronizzazione:
  - interna: confronto risultati intermedi, ecc.
  - esterna: procedure chiamate, dati input e output, ecc.
- La RS ad un certo livello è efficiente quando i diversi elementi hanno uno “stato dei lavori” simile. Se ciò non accade, si passa a RS ad un livello superiore
- Il sistema di RS è normalmente gestito da un server (RSS) che stabilisce
  - i punti di sincronizzazione (SP) con ogni risorsa
  - quale risorsa è primaria e quale ridondante
  - i lassi di tempo necessari e gli scarti accettabili
  - quali devono essere i tempi di risposta delle applicazioni

Servizi di log condiviso per sistemi distribuiti FT

- L'uso dei log visti nel caso delle basi di dati, crea problemi se applicato ad un s.o.
  - interpretabilità dei log: si hanno diversi manager di risorse con i loro algoritmi di recupero
  - consistenza: i log condivisi vanno protetti da intrusioni di log locali
- Soluzione: il log condiviso deve avere uno spazio di indirizzamento separato da quello dei log locali. Ma
  - è più complicata l'assegnazione dei Log Sequence Number (LSN) (sui DBMS, il byte di indirizzo relativo): si crea overhead per gestire le comunicazioni tra processi e le chiamate incrociate
  - lo spazio di indirizzamento non è infinito

Servizi di log condiviso per sistemi distribuiti FT

Sistema Quick Silver (IBM)

- Una routine sempre attiva assegna una sequenza di interi come LSN, non i byte di indirizzi relativi
- Gestione dei log:
  - Un log locale inizialmente è registrato nello spazio proprio del gestore locale
  - successivamente viene inviato al buffer del LM oppure forzato in scrittura su una memoria stabile
  - quando un log è forzato in scrittura, ***tutti*** i log precedentemente generati da quella risorsa vengono forzati
  - Forzare in scrittura un log ne provoca il *commit* (e quindi a cascata il *commit* di tutti i log precedenti)

Servizi di log condiviso per sistemi distribuiti FT

- Per risolvere il problema dello spazio di log condiviso, Quick Silver utilizza per il recupero speciali archivi con filtri gestiti direttamente dai gestori locali
- Esiste uno speciale archivio on-line per abortire transazioni a lungo termine. Quando un gestore non risponde per un certo tempo, i suoi record vengono spostati off-line per liberare spazio
- Il sistema supporta anche macchine diskless

Memoria transazionale stabile
Introduzione

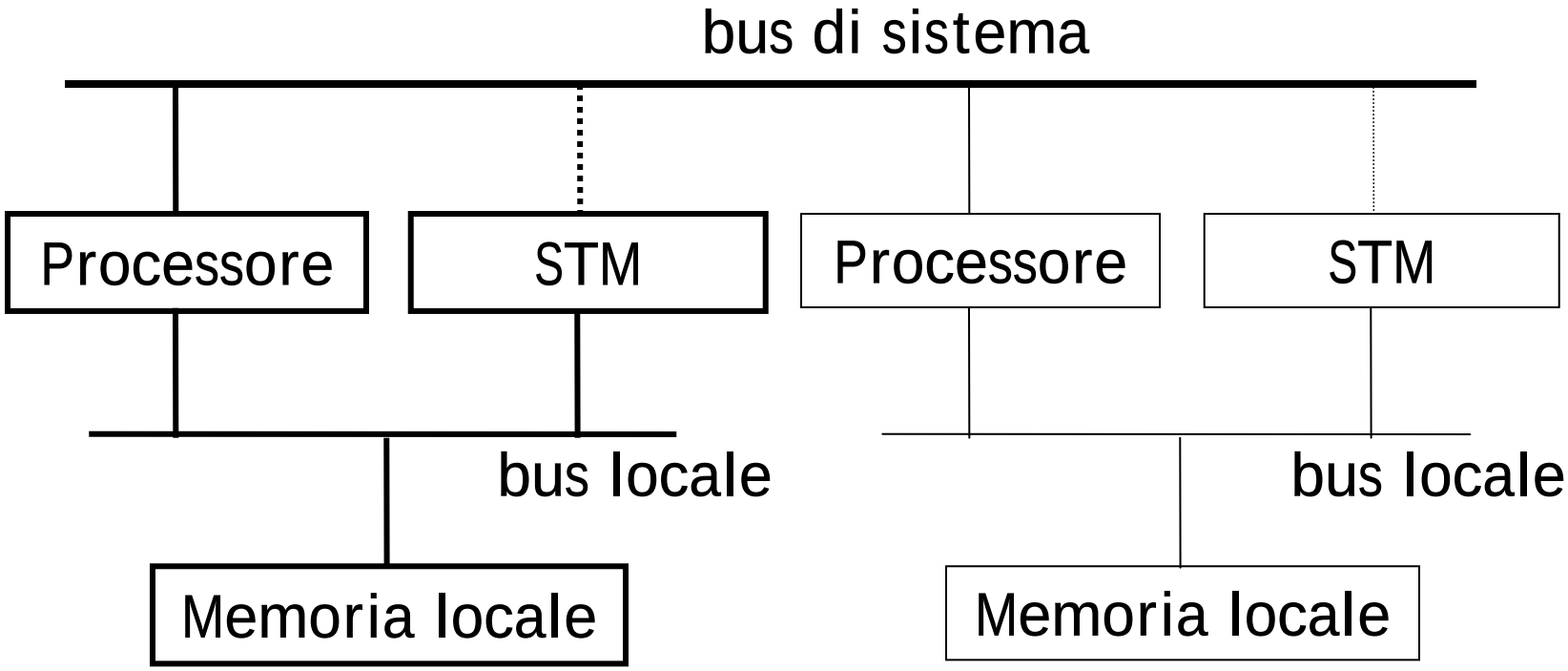
- Le architetture FT degli ultimi anni possono dividersi sommariamente in due categorie:
  - **blandamente accoppiate**: ogni processore ha risorse private e comunica con gli altri tramite messaggi
  - **strettamente accoppiate**: ogni processore ha accesso a tutte le risorse e comunica con gli altri attraverso memoria condivisa. Ogni processore possiede cache non write-through, il che permette più scritture prima di aggiornare la cache centrale
- L'approccio STM (Stable Transactional Memory) è diverso, e sfrutta le strutture proprie delle transazioni

Memoria transazionale stabile
STM

- Il **primo livello** di STM è una memoria veloce e stabile per realizzare un veloce e affidabile protocollo di *commit*.
- Il **secondo livello** di STM, innestato su un sistema multiprocessore con architettura blandamente accoppiata e s.o. distribuito. Caratteristiche:
  - autonomia: la STM può prendere decisioni (se il suo processore cade, comincia la riconfigurazione)
  - autoprotezione
  - atomicità: le transazioni sono atomiche
- il **terzo livello** di STM, si innesta su un'architettura strettamente accoppiata

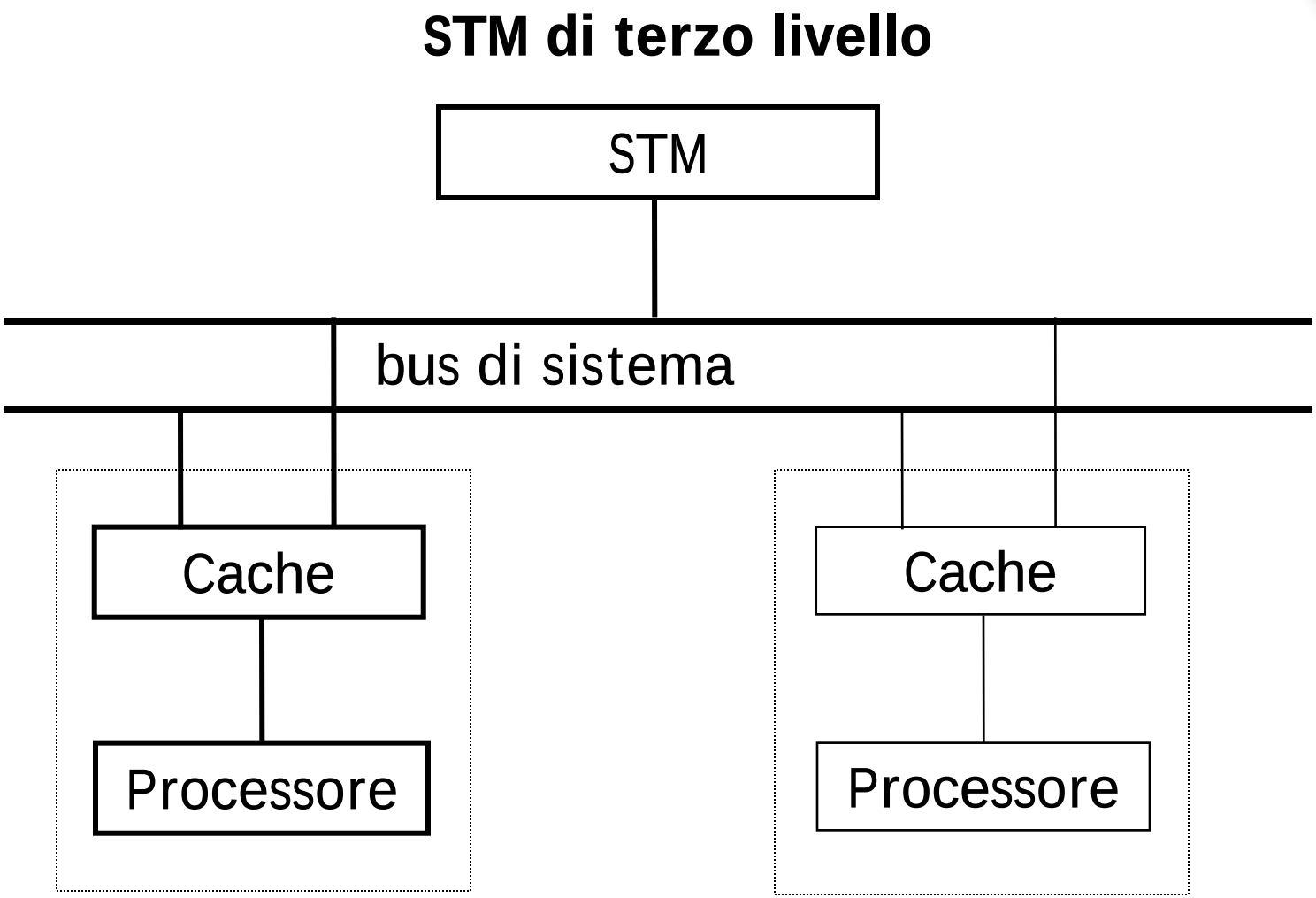
Memoria transazionale stabile
STM

STM di secondo livello



La linea tratteggiata rappresenta il canale di recovery

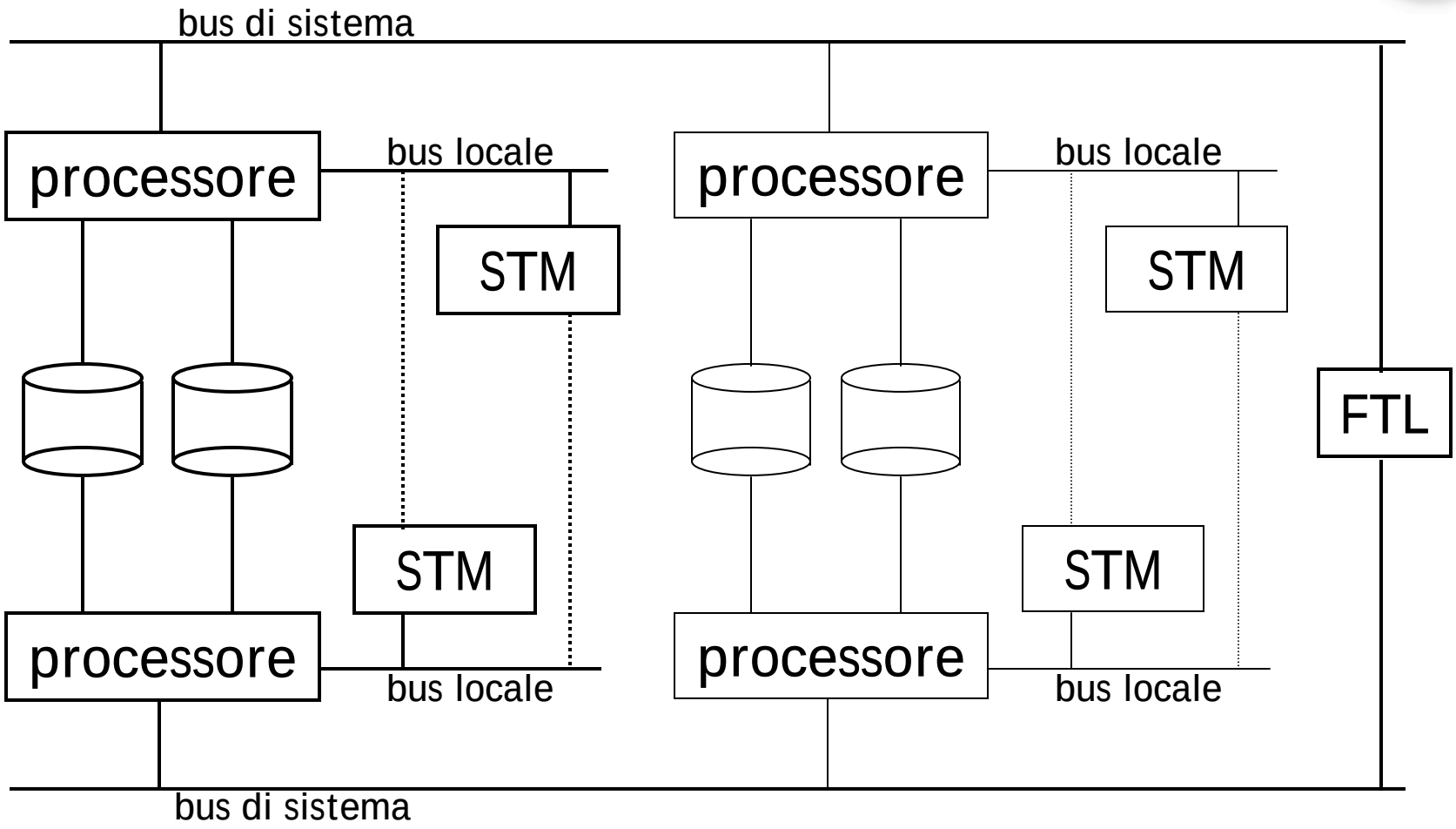
Memoria transazionale stabile
STM



Memoria transazionale stabile
Esempio

- STM nel sistema operativo FT Multiprocessor (FTM) ideato da Banâtre, nel 1991.
- Sistema a due processori blandamente accoppiato da un FT Link (FTL) con risorse replicate (STM inclusa)
  - guasti hw e di s.o. mascherati all'utente
  - veloce ripristino
  - portabilità
- Prima di accedere ad una risorsa, si deve passare per la STM, che
  - comunica la richiesta al kernel
  - imposta una transazione per gestire le operazioni richieste

Memoria transazionale stabile
Esempio



Computazione parallela su rete di workstation
Introduzione

- Il problema della computazione parallela è l'overhead introdotto per le comunicazioni tra processori. Soluzione: sistemi hw progettati *ad hoc*
- Qui vediamo un sistema che consente a una rete di workstation convenzionali e a basso costo di eseguire efficientemente codice parallelo già esistente con poche modifiche.
- Si scinde quindi la **macchina virtuale** su cui opera il programmatore dalla **macchina reale** che esegue il codice scritto
- Tolleranza ai guasti e load balancing sono intrinseci al sistema

Computazione parallela su rete di workstation
La macchina virtuale

- Il programmatore opera su una **macchina virtuale** con le seguenti caratteristiche:
  - processori sincroni
  - memoria condivisa
  - numero illimitato di processori virtuali
- Il programmatore scrive un programma consistente in una sequenza di passi paralleli, costituiti ognuno da diversi thread
  - ogni thread di un passo parallelo viene eseguito su un singolo processore virtuale
  - un passo parallelo termina quando tutti i thread hanno terminato

Computazione parallela su rete di workstation
La macchina reale

- La **macchina reale** viceversa è dotata di:
  - processori completamente asincroni
  - memoria condivisa
  - numero finito di processori
- I passi paralleli sono numerati sequenzialmente. Per quanto riguarda il passo corrente:
  - un contatore logico identifica qual è il passo corrente
  - un processore libero si schedula da solo (***eager scheduling***), catturando la copia di un thread non ancora portato a termine e lo esegue.
  - quando il passo termina si incrementa il contatore
  - ogni thread deve essere eseguito *dal punto di vista logico* una sola volta

Computazione parallela su rete di workstation
Progetto del sistema

- Assunzioni:
  - il programma è una sequenza di passi paralleli **non** annidati
  - se un thread scrive una variabile nessun altro thread dello stesso passo può scrivere la stessa variabile
- L'architettura:
  - rete di workstations convenzionali con s.o. standard
  - alcune ws sono *memory server* e *progress manager*
  - le altre ws sono *compute server*
- Il programma parallelo si basa su punti di sincronizzazione (cobegin-coend)
- Sono tollerati guasti su qualunque tipo di ws

Computazione parallela su rete di workstation
Precompilazione

- Ogni passo è individuato dal blocco cobegin-coend
- Il precompilatore elimina cobegin-coend e trasforma ogni thread in una procedura a sé
- La notazione, in pseudo-Pascal, è Sxyz
  - x identifica il passo
  - y identifica il thread
  - z identifica l'istruzione

```
begin
  cobegin
     begin S111, S112, S113,... end
     begin S121, S122, S123,... end
  coend
  cobegin
     begin S211, S212, S213,... end
  coend
end
```

```
procedure P11
   begin S111, S112, S113,... end
procedure P12
   begin S121, S122, S123,... end
procedure P21
   begin S121, S122, S123,... end
```

Computazione parallela su rete di workstation
Compilazione

Il compilatore (sequenziale) opera sul codice precompilato, memorizzando sul memory server:

- codice oggetto
- *Progress Table*, creata estraendo l'indirizzo fisico di partenza di ogni procedura dalla tabella dei simboli

Thread ID.	Start Address	Started?	Done?
P11	0x0ppp	FALSE	FALSE
P12	0x0qqq	FALSE	FALSE
P21	0x0rrr	FALSE	FALSE
...	...	...	...

Computazione parallela su rete di workstation
Memory service ed esecuzione

- Ogni *compute server* ha un demone interfacciato con la *progress table.* Nell'ordine, il demone
  - cerca un thread P_{ij} non ancora iniziato e setta TRUE il campo "Started?" del suo record
  - cerca un thread P_{ij} iniziato ma non ancora completato
  - scarica dal *memory server* la procedura in memoria a partire dallo "Start Address" indicato
  - inizia l'esecuzione
- La workstation lavora su una copia locale della procedura e solo al termine si aggiorna la memoria condivisa sul memory server e la *progress table*
- Ogni ws sottoutilizzata può eseguire lo stesso thread, attraverso l'eager scheduling

Computazione parallela su rete di workstation

Memory service ed esecuzione

- La FT e il load balancing si realizzano attraverso la ridondanza indotta dall'eager scheduling
  - nessun *compute server* attivo aspetta che un altro *compute server* termini
  - se un compute server si guasta non c'è bisogno di coordinamento globale per allocare un'altra macchina
- Gli inconvenienti da risolvere sono:
  - troppi *compute server* sullo stesso thread
  - Supponiamo che il *compute server* A sia più lento di B; e che B inizi lo stesso thread di A, finendo prima. La macchina A cercherà di sovrascrivere il risultato di B, annullando la ridondanza.
  - Non c'è FT sul memory server

Computazione parallela su rete di workstation

Memory service ed esecuzione

Possibili soluzioni ai precedenti problemi:

- Un campo aggiuntivo sulla Progress Table, che indichi quanti *compute server* stanno eseguendo il thread. Il demone di una workstation libera non inizia se ce ne sono già n
- Per non sovrascrivere la memoria condivisa si assegnano indirizzi sparsi: l'indirizzo di una variabile cambia ogni volta che viene scritta. Si deve allora introdurre un *progress manager*
- Si rende FT il *memory server* tramite replicazione. Si introduce però overhead su ogni operazione di scrittura

Dischi RAID
Introduzione

- Definizione: Redundant Array of Independent Disks (alternativamente Redundant Array of *Inexpensive* Disks) che operano in parallelo
- Diversi livelli, numerati da 0 a 5
  - RAID è un insieme di dischi fisici visti dal s.o. come un unico disco logico
  - I dati sono sempre distribuiti su più dischi
  - La ridondanza introdotta si usa per memorizzare informazioni utili al ripristino in caso di guasto
- A seconda del livello, le precedenti caratteristiche differiscono (ad esempio Raid 0 non ha ridondanza)
- Esistono livelli aggiuntivi ancora sperimentali

Dischi RAID
Livello 0

- In realtà non è un vero livello RAID, perchè non prevede ridondanza. Non c'è tolleranza ai guasti
- Tutti i dati (utente e di sistema) sono distribuiti sull'array di dischi tramite **striping**.
- Dati sistemati *a fette* (stripe), realizzando una distribuzione uniforme. Esempio: array di 3 dischi
  - le prime 3 stripe del disco logico vengono mappate una per disco fisico.
    - La stripe 0 sarà la prima stripe del disco 0
    - La stripe 1 sarà la prima stripe del disco 1
    - La stripe 2 sarà la prima stripe del disco 2
    - La stripe *n* verrà impilata nel disco *n mod 3*
  - si occupa della mappatura il sw di gestione del RAID

Dischi RAID
Livello 0

- Per alti transfer rate si deve sfruttare efficientemente l'architettura fisica dell'array. Supponendo un array di *n* dischi:
  - sono ottimali richieste di quantità di dati contigui *n* volte più grandi di una stripe.
  - Stripe piccole portano a transfer rate alti
- In ambiente transazionale, interessa piuttosto il tempo di risposta ad una richiesta di I/O
  - *load balancing* (implica numerose richieste accodate)
    - molti processi fanno poche richieste diverse
    - un solo processo fa molte richieste asincrone
  - stripe grandi richiedono per ogni richiesta di I/O un solo accesso al disco, situazione ottimale

Dischi RAID
Livello 1

- Rispetto al livello 0 si inizia ad aggiungere ridondanza
- Ridondanza raggiunta attraverso **mirroring.** Ogni stripe logica viene mappata su due diversi dischi: ogni disco dell'array ha un disco gemello
- Vantaggi:
  - Ogni richiesta di lettura può essere soddisfatta da ognuno dei due dischi: raddoppio delle prestazioni in ambito transazionale
  - Ogni scrittura è doppia ma parallela
  - Guasti mascherati
- Svantaggi
  - raddoppio dei costi. RAID 1 si usa quindi solo per software e dati di sistema e file critici

Dischi RAID
Livello 2

- L'array diventa ad accesso parallelo: **tutti** i dischi partecipano ad **ogni** richiesta di I/O. Ci sono dischi di dati e di ridondanza. Accesso **simultaneo** ai dischi
- Stripe molto piccole (un *byte* o una *word*)
- Rilevazione e correzione errori tramite codice Hamming (correzione singolo errore, rilevazione errore doppio)
  - bit di dati e bit ridondanti inviati al controller
  - se c'è singolo errore il controller lo corregge senza richiedere la ripetizione dell'operazione
- Numero dischi ridondanti minore di RAID 1 ma porzionale al logaritmo del numero di dischi di dati
- RAID 2 utile con molti errori di disco (quindi inutile)

Dischi RAID
Livello 3

- Simile a RAID 2, ma c'è solo un disco di ridondanza, indipendentemente dalla dimensione dell'array
- Anzichè usare codici correttori, il disco di ridondanza è di pura parità
  - se un disco cade, si entra in **modalità ridotta**, si ricostruiscono al volo le informazioni sulla base degli altri dischi e di quello di parità, sfruttando l'Ex-OR
  - appena il disco viene ripristinato si reinseriscono i dati e si riparte in modalità normale
- Transfer rate ottimo, per la piccolezza delle stripe
- Tempi di risposta per sistemi transazionali scadenti: l'accesso contemporaneo a tutti i dischi implica che si possa soddisfare una sola richiesta I/O alla volta

Dischi RAID
Livello 4

- La differenza rispetto a RAID 2 e 3 è l'accesso **indipendente** ai dischi: richieste di diversi I/O gestibili in parallelo. Migliore tempo di risposta per richieste di I/O, peggiore transfer rate
- Le stripe sono relativamente grandi e sul disco di parità si calcola su un'intera stripe di parità bit a bit basata sulle corrispondenti stripe di dati
- Quando si scrive, si introduce overhead:
  - leggere i dati vecchi con la vecchia parità
  - calcolare la nuova parità
  - aggiornare dischi dati e disco di parità
- Il disco di parità diventa un collo di bottiglia

Dischi RAID
Livello 5

- La differenza rispetto a RAID 4 è che le stripe di parità sono distribuite con algoritmo *round-robin* su tutti i dischi dell'array, eliminando il problema del collo di bottiglia del disco di parità

Dischi RAID
Tabella comparativa

liv.	categoria (striping ovunque)	descrizione	frequenti richieste I/O read/write	transfer rate read/write
0		Non ridondante	Grandi stripe: eccellente	Piccole stripe: eccellente
1	mirroring indipendente	duplicazione	discreto/buono	discreto/discreto
2	accesso parallelo	Ridondante con codice Hamming	scarso	eccellente
3	accesso parallelo	Parità bit a bit	scarso	eccellente
4	accesso indipendente	Parità stripe a stripe	eccellente/discreto	discreto/scarso
5	accesso indipendente	Parità stripe a stripe distribuita	eccellente/discreto	discreto/scarso

Dischi RAID
Recenti sviluppi - RAID di Livello 6

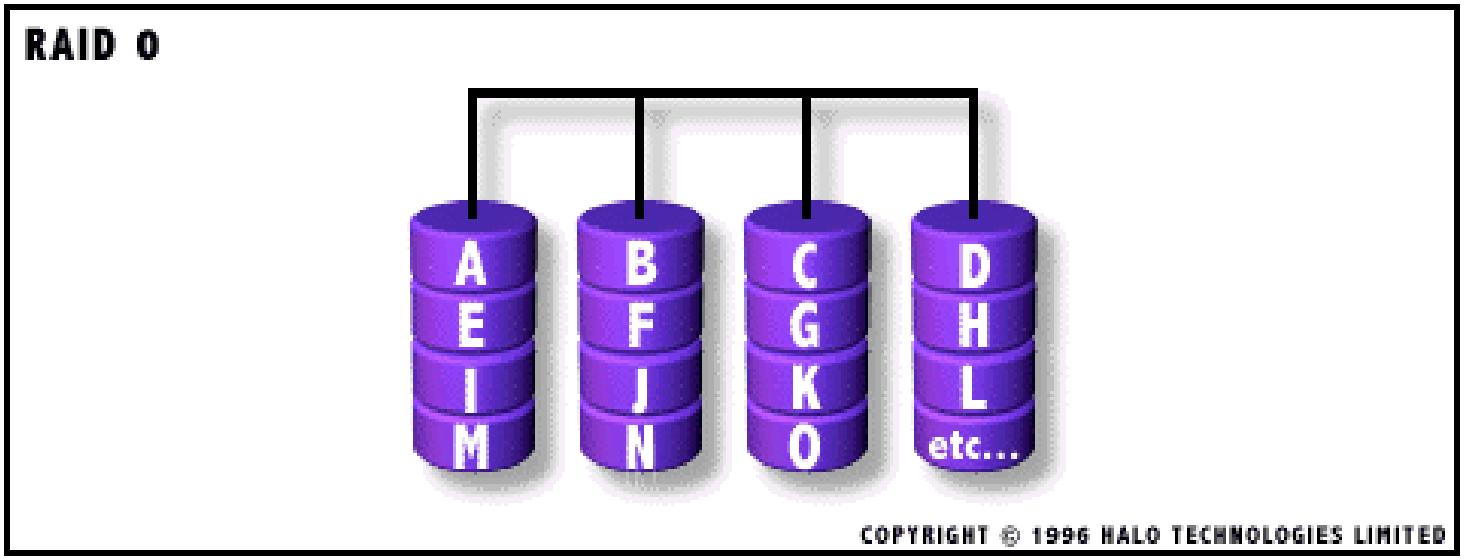
- Sfrutta il codice Reed-Solomon (RS)
- Segue l'architettura di RAID 5, ma usa due stripe di parità anziché una (**ridondanza P+Q**)
- Le stripe di parità sono ancora disposte sull'array di dischi con algoritmo *round-robin.*
- Il codice RS corregge fino a **due guasti** contemporanei sull'array di dischi. Svantaggi
  - Occorre un disco in più (la parità è doppia rispetto a RAID 5)
  - Overhead dovuto alla complessa implementazione dell'algoritmo di recovery
  - Costi elevati

Dischi RAID
Recenti sviluppi - RAID di Livello 7

- Le stripe di parità possono andare da una a oltre tre
- I guasti contemporanei gestibili sono 4
- Lo stesso dato non viene necessariamente riscritto nella stessa posizione fisica come in tutti i livelli precedenti
- La complicazione della gestione dell'array di dischi richiede un s.o. *real-time* dedicato, che:
  - si occupa di generare la parità e di tutta la logica
  - gestisce la ricostruzione delle informazioni
  - rende altamente indipendenti tutti i dischi
- Molto più veloce in scrittura di RAID 5 (da 1,5 a 6 volte) ma estremamente costoso

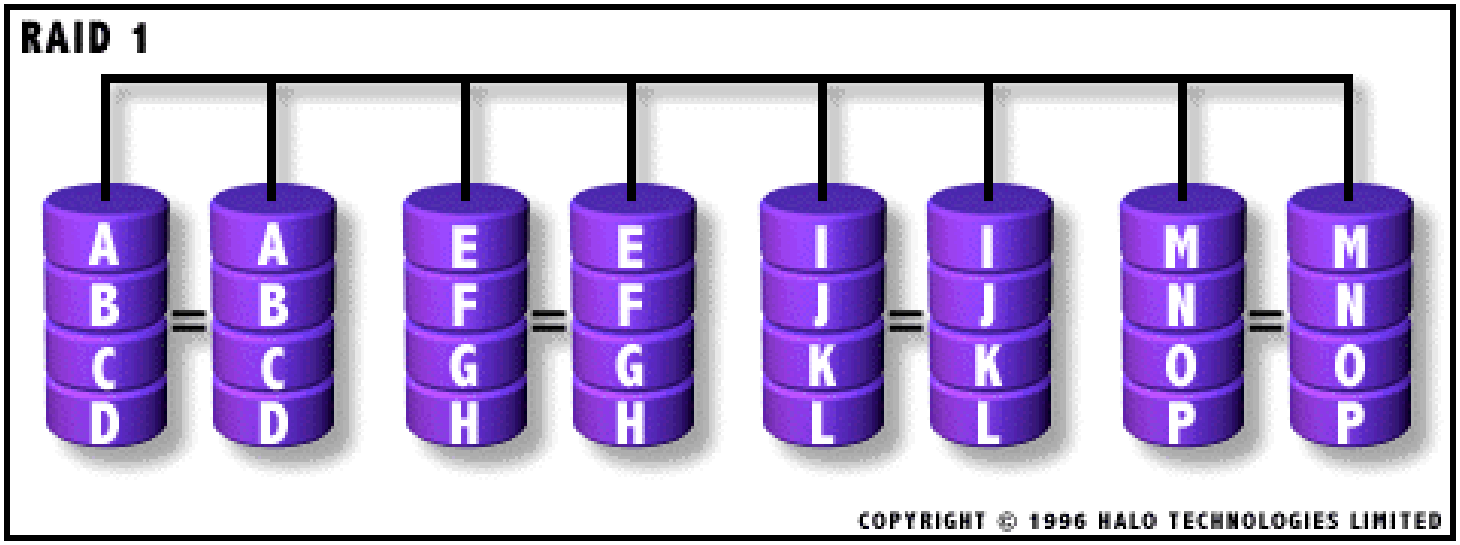
Dischi RAID
Schemi

- RAID 0: nessuna ridondanza.



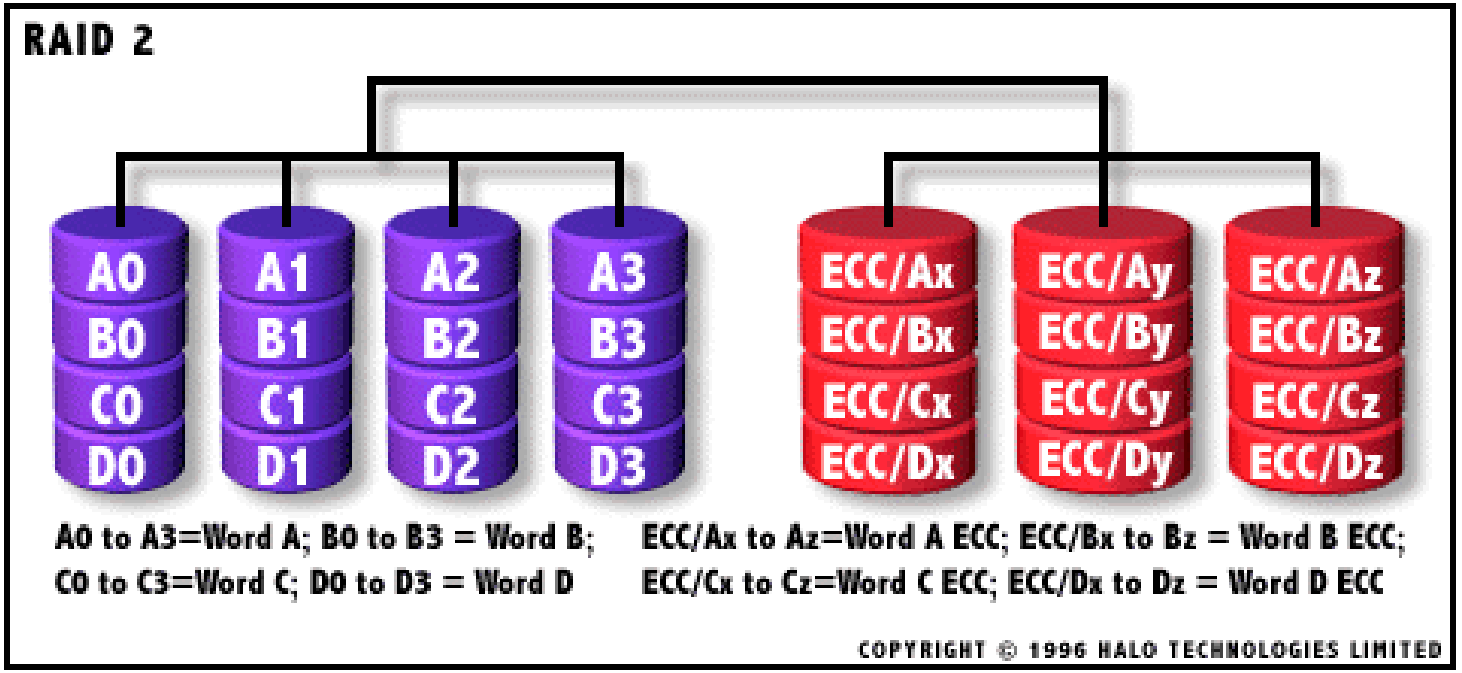
Dischi RAID
Schemi

- RAID 1: mirroring



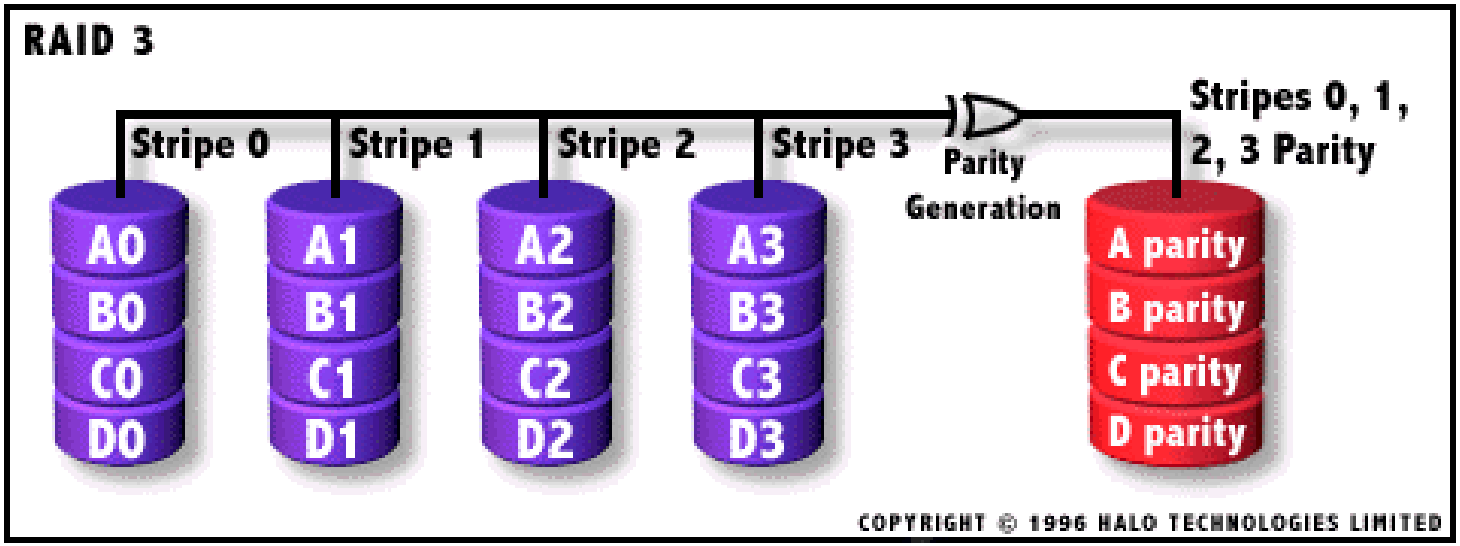
Dischi RAID
Schemi

- RAID 2: Parità e codice Hamming



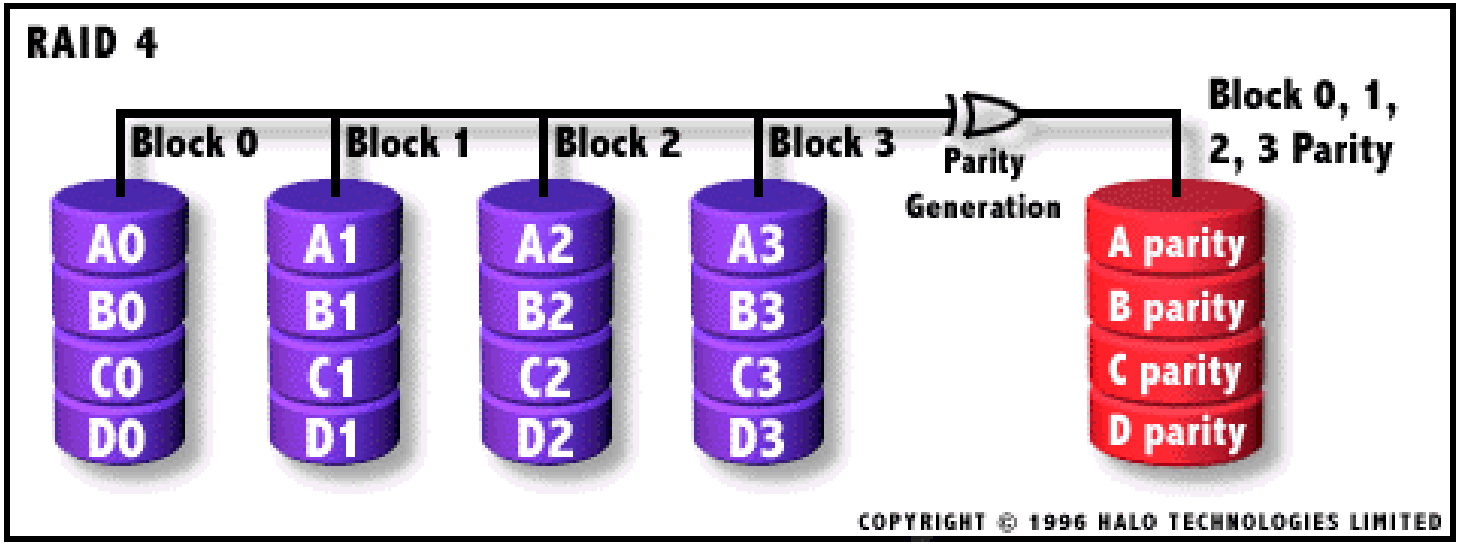
Dischi RAID
Schemi

- RAID 3: Parità bit a bit



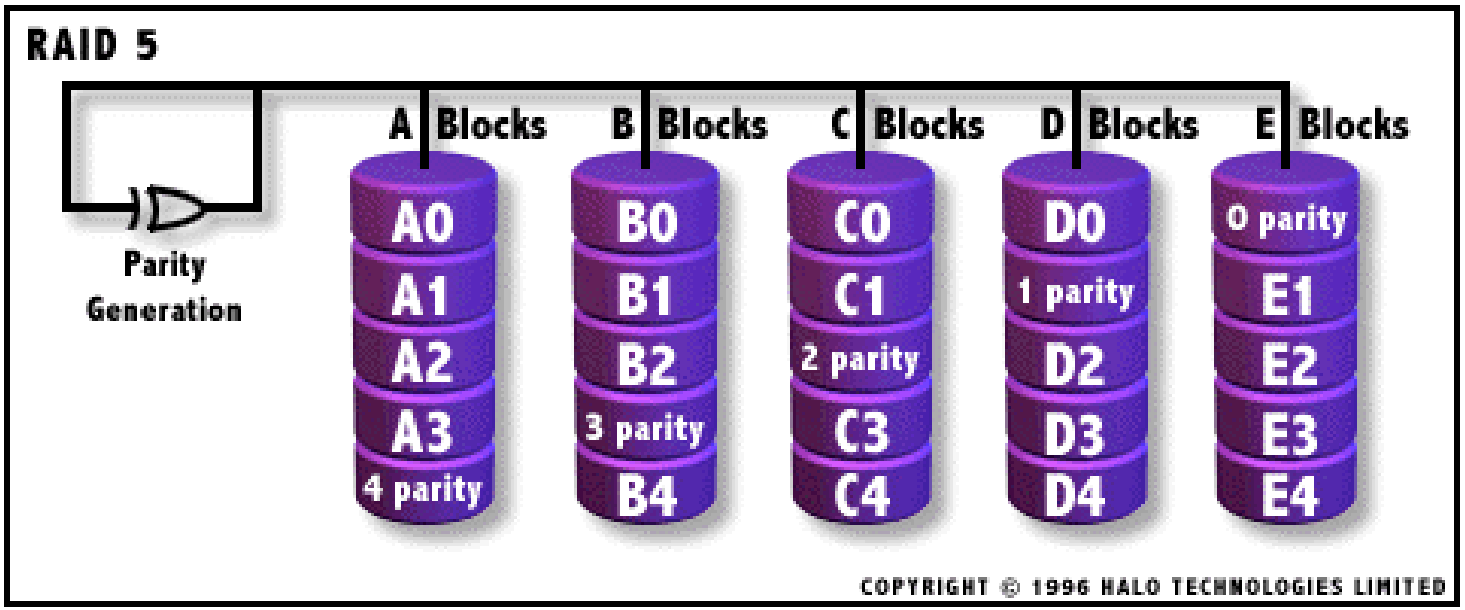
Dischi RAID
Schemi

- RAID 4: Parità sulle stripe



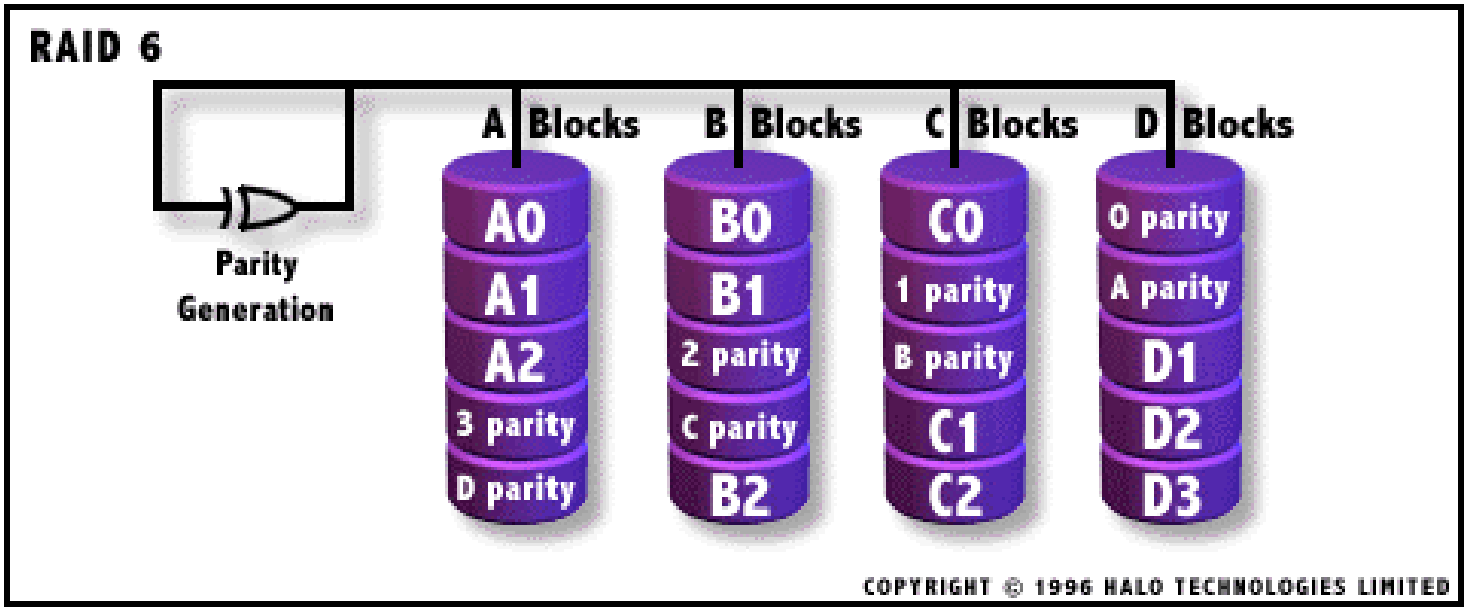
Dischi RAID
Schemi

- RAID 5: Parità sulle stripe distribuita



Dischi RAID
Schemi

- RAID 6: Parità doppia e codice Reed-Solomon



Dischi RAID
Schemi

- RAID 7: Parità multipla