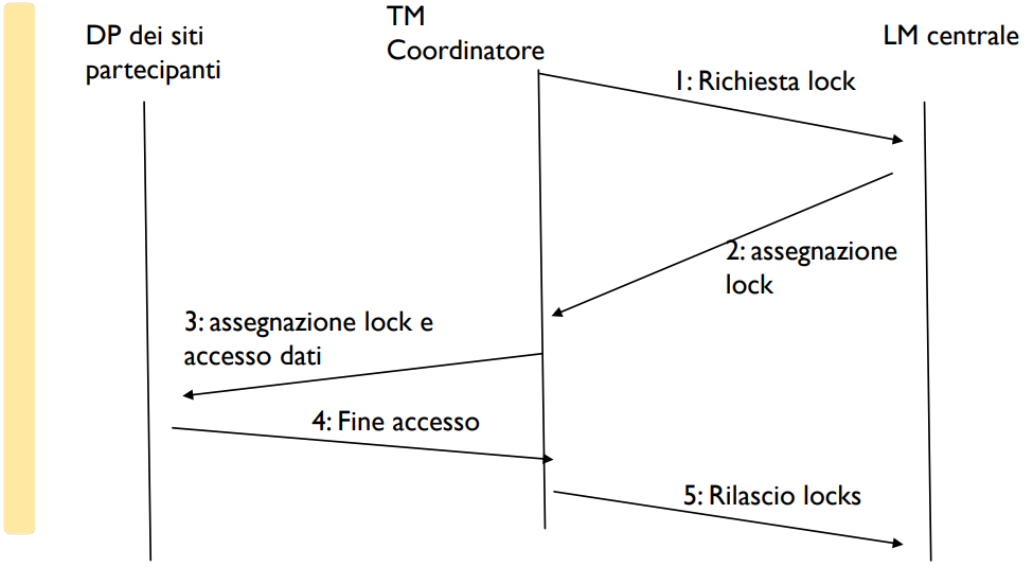
Lezione 3 13/03/2024

Strategia centralized 2PL

- Attori
  - Ogni nodo ha un Lock Manager, uno viene eletto LM coordinatore
    - Gestisce i locks per l'intero DDB
  - Il Transaction Manager del nodo dove inizia la transazione e' considerato TM coordinatore
  - La transazione e' anche eseguita su altri Data Processor e corrispondenti nodi
- Strategia
  - Il TM coordinatore formula al LM coordinatore le richieste di lock
  - Il LM le concede, utilizzando un 2PL
  - Il TM le comunica ai DP
  - I DP comunicano al TM e il TM al LM la fine delle operazioni



In questo modo introduco un elemento che sa tutto (il lock manager), che è centralizzato, così come nello schema centralizzato c'è lo scheduler. In questo caso creo un collo di bottiglia, perché tutte le transazioni chiederanno le risorse al lock manager. Questo diventa anche un single point of failure.

Per mitigare questo problema potrei avere un nodo di backup (vice lock manager) che sia pronto nel caso il lock manager principale cada. Bisogna essere sicuri che anche il lock manager secondario rimanga aggiornato sui locking però, per esempio usando il protocollo ROWA per comunicare con il secondario, però rallentando il sistema perché il lock manager principale deve aspettare le conferma dagli altri.

Strategia primary locking 2PL

Un'altra modalità è quella di dire, distribuiamo il potere di scelta. C'è un concetto di risorsa unica, ma dato che le risorse sono su ciascun nodo il lock manager di ciascun nodo decide, però tutti i nodi condividono le scelte. Quindi non ho più un lock manager centralizzato ma ho una directory centralizzata per memorizzare i lock. Il vantaggio è che in questo modo devo solamente avere la tabella condivisa dei lock, anche per controllare la presenza di deadlock.

Questo diventa però il nuovo single point of failure.

Fix:

- Per ogni risorsa e' individuata una copia primaria
- La copia primaria e' individuata prima della assegnazione dei lock
- Diversi nodi hanno lock managers attivi, ognuno gestisce una partizione dei lock complessivi, relativi alle risorse primarie residenti nel nodo
- Per ogni risorsa nella transazione, il TM comunica le richieste di lock al LM responsabile della copia primaria, che assegna i lock
- Conseguenze
  - Evita il bottleneck
  - Complicazione: e' necessario determinare a priori il lock manager che gestisce ciascuna risorsa.
  - E' necessaria una directory globale

Deadlock distribuito

Come è possibile accorgersi di situazioni di deadlock, a livello globale (esempio di deadlock precedente con le due transazioni)? La transazione 1 può comunicare il fatto che è in attesa che un'altra transazione. Se nel grafo di attivazione è presente un ciclo, allora (probabilmente) è presente un deadlock.

Possiamo avere un algoritmo distribuito dove non c'è un single point of failure, ma che sia basato sulla collaborazione dei nodi? I nodi devono comunicare tra di loro per scoprire i deadlock.

Posso fare delle fotografie periodicamente, anticipando il timeout.

La decisione di come risolvere un deadlock deve essere presa da un singolo nodo, altrimenti si rischia che entrambe le transizioni vengano uccise, e poi ripartendo il deadlock avverrà di nuovo. Per esempio dando una priorità diversa ai diversi nodi, e quello con priorità più alta risolverà il deadlock.

- Causato da un' attesa circolare tra due o piu' nodi
- Gestito comunemente nei DDBMS tramite time-out
- Vediamo un' algoritmo di rilevazione del deadlock in ambiente distribuito
- L'algoritmo e' asincrono e distribuito

- Notazione
- t_{ij} sottotransazione della transazione t_i al nodo j

- Assumiamo che le sotto-transazioni siano attivate in modo sincrono (tramite Remote Procedure Call bloccante): t_1 attende t_2
- Questo puo' dare origine a due tipi di attesa:
  - *ATTESA DA REMOTE PROCEDURE CALL*
  - t_{11} sul nodo 1 (secondo pedice) attende t_{12} sul nodo 2 perche' aspetta la sua terminazione
  - *ATTESA DA RILASCIO DI RISORSA*
  - t_{11} sul nodo 1 attende t_{21} sullo stesso nodo perche' attende il rilascio di una risorsa
- La composizione dei due tipi di attesa puo' dar luogo a uno stato di deadlock globale

Condizioni di attesa

- E' possibile caratterizzare le condizioni di attesa su ciascun nodo tramite condizioni di precedenza
- Notazione
  - EXT_i: external, chiamata da un nodo remoto i
  - X < Y: X attende il rilascio di una risorsa da Y (puo' essere EXT)
- La sequenza di attesa generale al nodo k e' della forma:

  $EXT < t_{ik} < t_{jk} < EXT$
- Esempio
- Su DBMS1: $EXT2 < t_{21} < t_{11} < EXT\ 2$
- Su DBMS2: $EXT1 < t_{12} < t_{22} < EXT1$

Una transazione che è nata in un certo nodo, è in attesa che un'altra transazione finisca, che a sua volta è in attesa di un'altra. (ultime righe dell'immagine sopra)

Esempio:

Nodo 1: $EXT2 < t_{21} < t_{11} < EXT\ 2$ *Nodo 2*: $EXT1 < t_{12} < t_{22} < EXT1$

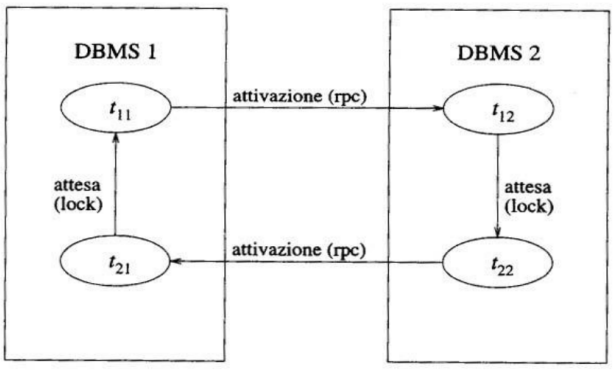


Figura 10.7 Esempio di un deadlock distribuito

Periodicamente il nodo con l'indice più basso farà un'analisi per il deadlock, e manderà questo schema al nodo 2, che mettendo insieme le informazioni si accorgerà del deadlock. Visto che il nodo 2 può agire solo sulle transazioni del suo nodo, uccide la sua transazione, dando strada libera all'altra.

Algoritmo di distribuzione del deadlock distribuito

- Attivato periodicamente sui diversi nodi del DDBMS:
  - 1. In ogni nodo, integra la sequenza di attesa con le condizioni di attesa locale degli altri nodi logicamente legati da condizioni EXT
  - 2. Analizza le condizioni di attesa sul nodo e rileva i deadlock locali
  - 3. Comunica le sequenze di attesa ad altre istanze dello stesso algoritmo (cioe' agli altri nodi)

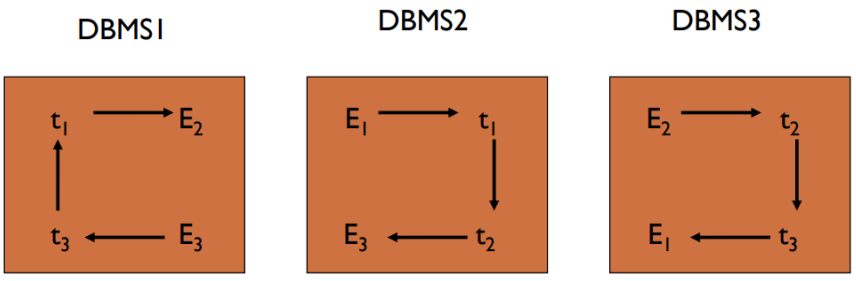
	Nodo 1	Nodo2	Nodo3
T1			
T2			
T3			

- E' possibile che lo stesso deadlock venga riscoperto piu' volte. Per evitare questo problema, e rendere piu' efficiente l'algoritmo, l'algoritmo invia le sequenze di attesa:
  - in avanti, verso il nodo ove e' attiva la sottotransazione t_i attesa da t_j
  - Solamente quando $i > j$ dove i e j sono gli identificatori dei nodi

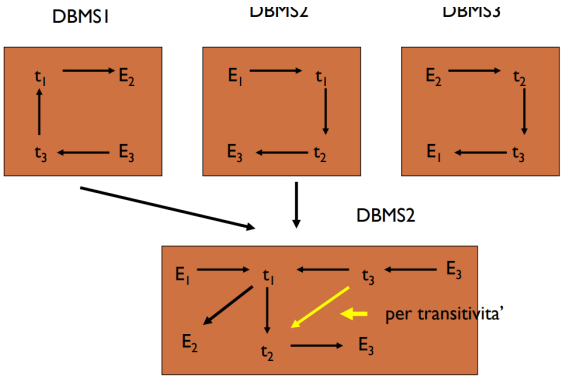
Ogni nodo non ha visibilità globale, questa si crea tramite la comunicazione tra i nodi. Quindi finché rimane il concetto di ordine, questo algoritmo non dà problemi, non c'è un single point of failure.

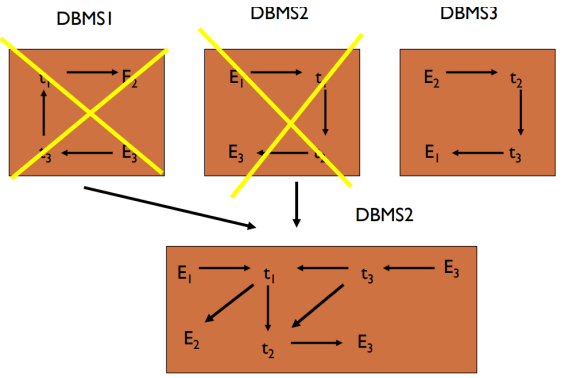
Esempio

situazione di partenza:

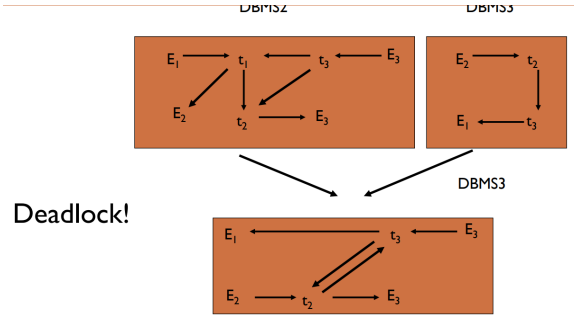


primo passo, il DMBS 1 passa l'informazione al DBMS 2 perché localmente non vede problemi. Le frecce forse sono le attese.





secondo passo: il nodo 2 non vede problemi e quindi passa l'informazione al nodo 3, che trova un ciclo. Quindi il DBMS 3 può uccidere la transazione 2 o 3 perché sono operate su quel nodo, non può toccare la transazione 1.



Quindi il DBMS 3 sa tutto solamente perché ha ricevuto le informazioni dagli altri, e solo perché il problema non è già stato risolto precedentemente.

L'ordine di comunicazione tra i nodi è deciso a priori, a design time.

Recovery management (atomicità)

In un sistema distribuito la cosa più pericolosa sono i **messaggi che si perdono**. Oppure c'è un **partizionamento della rete**, ovvero alcuni nodi si vedono, ma altri no. In questo caso il nodo non muore, ma rimane invisibile (per esempio se viene interrotto il cavo di rete).

I protocolli di commit consentono ad una transazione di giungere ad una decisione di **abort/commit** su ciascuno dei nodi che partecipano ad una transazione.

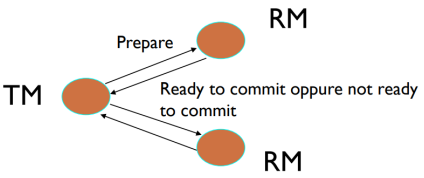
Nel 2PC esiste un **transaction manager**, che di solito è il nodo da cui parte la transazione, che coordina questa attività. Questa è una soluzione centralizzata.

- I server sono chiamati **resource managers (RM)**
- Il coordinatore e' chiamato **transaction manager (TM)**

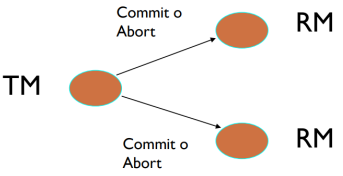
Il protocollo 2PC si basa sullo scambio di messaggi tra TM e RM, i quali mantengono ognuno il proprio log.

Protocollo 2PC (in assenza di guasti)

Prima fase: c'è in transaction manager che chiede a tutti se possono fare commit. Ogni nodo decide autonomamente se commit o abort e comunica unilateralmente la sua decisione irrevocabile. se anche solo uno risponde di no, allora si fa abort.



Una volta che il TM ha raccolto i consensi, allora prende una decisione globale e poi comunica la decisione ai nodi per le azioni locali (commit/abort)



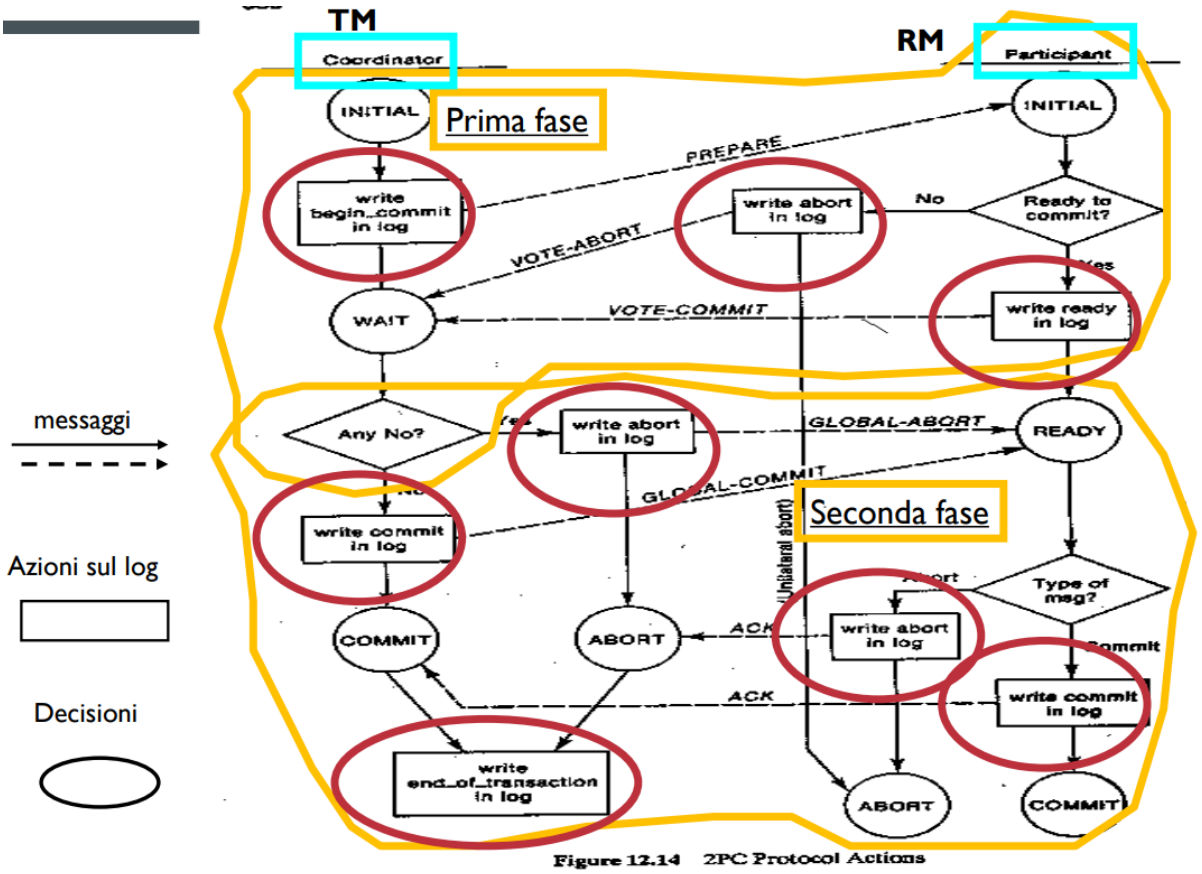


Figure 12.14 2PC Protocol Actions

In questo algoritmo sono importanti i file di log. La logica è che prima scrivo sul file di log quello che voglio fare, e poi lo faccio. Questo è importante perché se l'ordine fosse opposto, potrebbe capitare un problema prima della scrittura. Invece scrivendo prima di fare, in fase di recovery si può recuperare tutto, ripartendo.

Scritture sui file di log:

- `Prepare` record (`begin_commit` in figura): contiene l'identita' di tutti i RM (nodi + transazioni)
- `global commit` o `global abort` record: descrive la decisione globale. La decisione del TM diventa esecutiva quando il TM scrive nel proprio log il record `global commit` o `global abort`
- `Complete (end of transaction` in figura) record: scritto alla fine del protocollo

- `ready` record: disponibilita' irrevocabile del RM a partecipare alla fase di commit
  - Puo' assumere diverse politiche sul protocollo di 2PL: recoverable, 2PL, ACR, strict 2PL.
  - Contiene anche l'identificatore del TM

  Inoltre, come nel caso centralizzato vengono scritti anche i records `begin`, `insert`, `delete`, `update`, `commit`
- `not ready (abort` in figura) record: indisponibilita' del RM al commit

Perché questo è un sistema efficace? Ragioniamo sui casi peggiori.

Nella prima fase c'è un timeout.

Gestione dei timeout

Sia nella prima fase che nella seconda fase possono avvenire guasti. In entrambe le fasi tutti i partecipanti devono poter prendere delle decisioni connesse allo stato in cui si trovano.

Questo avviene utilizzando timers e stabilendo un intervallo di tempo di timeout.

Nella seconda parte serve avere un timeout? Un resource manager potrebbe esplodere dopo che ha comunicato di voler fare commit. Quindi il resource manager, dopo che ha ricevuto la decisione globale, deve dare conferma, e

quindi il TM deve aspettare di ricevere tutti gli ACK. In questo modo se non riceve risposta, ripeterà l'indicazione di fare commit al RM.

Prima fase del 2PC

- TM scrive `prepare` nel suo log e invia un messaggio `prepare` a tutti i RM. Fissa un timeout per indicare il massimo intervallo di tempo di attesa per le risposte
- Gli RM che sono recoverable, cioe' sono pronti a fare il commit, scrivono `ready` nel loro log record e inviano un messaggio `ready` al TM
- Gli RM che *non* sono recoverable, perche', ad esempio devono abortire per un deadlock, inviano un messaggio `not-ready` e terminano il protocollo, effettuando un abort unilaterale

- Il TM raccoglie i messaggi di risposta dagli RM:
  - Se *tutti* gli RM rispondono positivamente, scrive `global commit` nel suo log
  - Se riceve almeno un messaggio `not-ready` o scatta il timeout, scrive `global abort` nel suo log

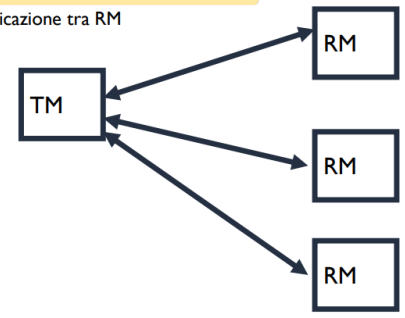
Seconda fase del 2PC

- Il TM trasmette la decisione globale agli RM e fissa un nuovo timeout
- Gli RMs che sono ready ricevono il messaggio, scrivono `commit` o `abort` nel loro log, e inviano un acknowledgment al TM. Poi eseguono il loro commit o abort locale
- Il TM raccoglie tutti i messaggi di *acknowledgement* dagli RM. Se scatta il time-out, poiche' ha gia' deciso in modo irrevocabile, fissa un nuovo time-out e ripete la trasmissione a tutti i RMs dai quali non ha ancora ricevuto un *ack*
- Quando tutti gli *acknowledgement* sono arrivati, il TM scrive `complete` nel suo log

Paradigmi di comunicazione tra TM e RMs

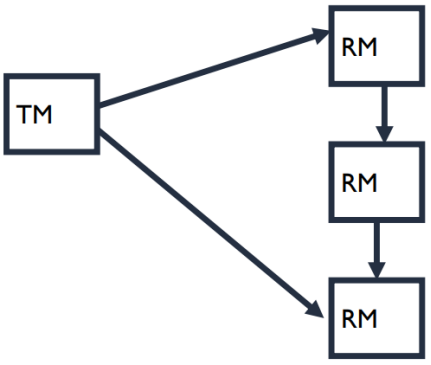
Centralizzato (e' quello che abbiamo visto)

- La comunicazione avviene solo tra TM e ogni RM
- Nessuna comunicazione tra RM



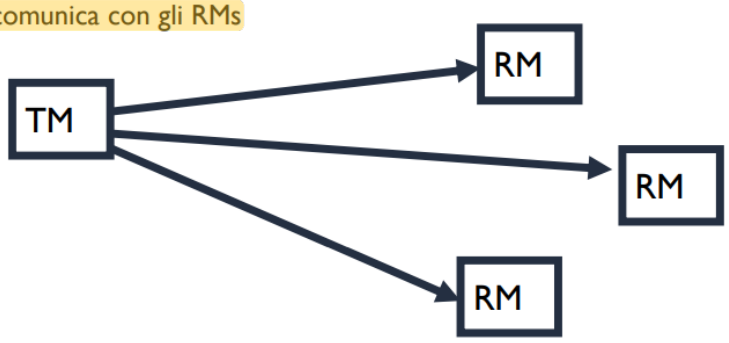
Lineare:

- Gli RM comunicano tra loro secondo un ordine prestabilito
- Il TM e' il primo nell'ordine
- Utile solo per reti senza possibilita' di broadcast



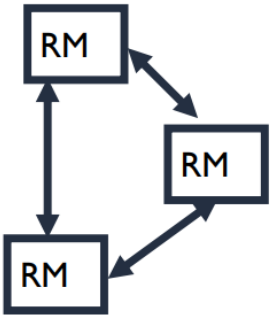
Distribuito:

- Nella prima fase, il TM comunica con gli RMs



Distribuito:

- Nella prima fase, il TM comunica con gli RMs
- Gli RMs inviano le loro decisioni a tutti gli altri partecipanti
- Ogni RM decide in base ai voti che "ascolta" dagli altri
- Non occorre la seconda fase di 2PC

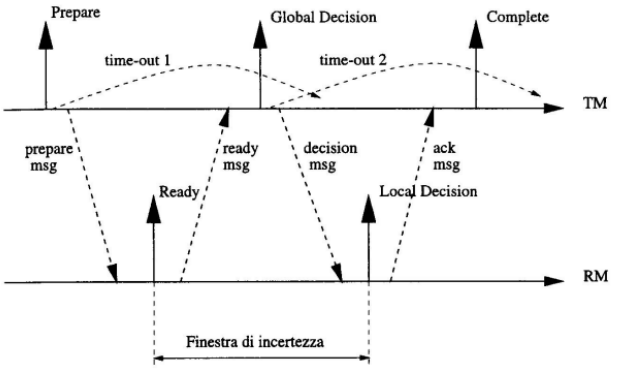


Guasti

Cosa succede se il TM aspetta le risposte (commit/abort) ma non riceve risposta da uno? farà abort.

Cosa succede se il RM dice not ready ma poi non riceve risposta dal TM? farà direttamente abort perché ne basta uno che faccia abort. Se invece avesse risposto commit dovrà aspettare perché non sapendo cosa succede sugli altri RM non può prendere una decisione.

L'intervallo tra la scrittura di ready nel log dei RMs e la scrittura di commit o abort è detta **finestra di incertezza**.



Guasti di componenti

Devono essere utilizzati protocolli con due diversi compiti:

- Assicurare la terminazione della procedure (**PT, protocolli di terminazione**)
- Assicurare il ripristino (**PR, protocolli di recovery**)

Caduta del coordinatore (TM)

Se i RM si accorgono che il TM è caduto, potrebbero eleggere un nuovo TM. Questo specialmente se accade prima dell'inizio del protocollo.

Quello che può capitare è che non tutti i RM siano collegati, e che quindi a gruppi eleggano TM diversi, ma in questo caso poi alcuni RM non risponderanno e quindi la decisione sarà abort.

- Casi possibili
- 1. l'ultimo record del log e' `prepare`
  - Il guasto del TM puo' avere bloccato alcuni RM
  - Due opzioni di recovery:
    - Decidere `global abort`, e procedere con la seconda fase di 2PC
    - Ripetere la prima fase, sperando di giungere a un global commit
- 2. l'ultimo record nel log e' `global-commit` o `global-abort`
  - alcuni RMs potrebbero non essere stati informati, e altri possono essere bloccati
  - Il TM deve ripetere la seconda fase
- 3. l'ultimo record nel log e' una `complete`
  - la caduta del coordinatore non ha effetto

Ottimizzazioni

Se un RM tra quelli coinvolti fa solo operazioni di lettura, questo non è influente nelle decisioni di commit/abort, quindi lui stesso, quando riceve l'informazione, comunica di non essere interessato.

Regola "scordarsi gli abort, ricordarsi i commit" (**presumed abort**).

Se almeno un resource manager mi dice che c'è abort, non aspetto la risposta degli altri, posso dire a tutti subito che c'è abort. Quindi non scrivo neanche abort nel file di log, perché se non c'è commit allora è a prescindere abort.

fin qua abbiamo considerato lo use case bancario, dove quindi è importante non perdere dati etc etc