# LA COORDINAZIONE DISTRIBUITA

Smettiamo ora di occuparci di sistemi EMBEDDED (dedicati ad una specifica applicazione) e prendiamo in esame la gestione di sistemi GENERAL PURPOSE a più processori o a più computer.

Studiamo il software di gestione per sistemi multiprocessore, ovvero in pratica le macchina virtuali progettate allo scopo di virtualizzare l'hardware di simili sistemi. Ci interesseremo inoltre non solo delle architetture, ma anche di problematiche di affidabilità (cosa accade al sistema in caso di guasti hardware) e sicurezza (riservatezza e affini).

PROBLEMI CHE CONSIDEREREMO:

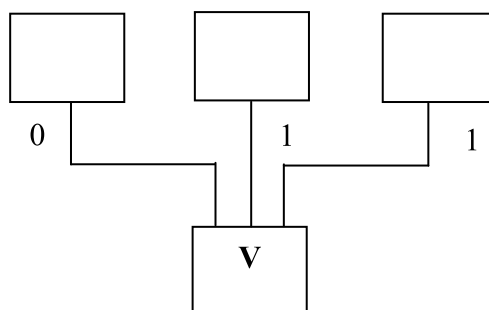
1. ORDINAMENTO DEGLI EVENTI. Il funzionamento di un SO tradizionale si basa sul fatto che delle richieste pervengono in ordine ad uno schedulatore, che le esaudisce secondo opportune politiche. I processi vengono eseguiti secondo un certo ordinamento, scandito dall'orologio macchina, che è unico. Questo non vale più nei sistemi distribuiti, *nei quali ogni nodo ha un proprio orologio*. L'ordinamento globale rispetto ad una serie di orologi, che non sono mai perfettamente sincronizzati, non è facile da gestire. Un ordinamento, tuttavia dev'essere necessariamente stabilito; quando al SO distribuito arrivano due richieste da parte di due diversi processori, occorre decidere quale delle due debba essere servita per prima.

2. MUTUA ESCLUSIONE. Due nodi del sistema possono richiedere simultaneamente l'accesso ad una risorsa che è in possesso di un terzo nodo.

3. ATOMICITÀ. Le azioni di un SO distribuito devono essere *atomiche*: o vengono eseguite per intero, o non vengono eseguite affatto. Una tecnica per garantire l'atomicità è proprio quella del roll-back che abbiamo considerato poco fa.

4. CONTROLLO DELLO STATO. Se (p.e.) tre nodi generano richieste che formano un *ciclo* (il primo inoltra una richiesta al secondo, il secondo una richiesta al terzo ed il terzo al primo) si corre il rischio di mandare il sistema in stallo. Questo problema, già difficile da gestire nel caso monoprocessore, diviene ancora più complicato nel caso distribuito.

5. ALGORITMI DI ELEZIONE. Se uno dei nodi collegati si guasta (magari un nodo che ha un ruolo cruciale, ad esempio da coordinatore) desideriamo continuare a lavorare con i nodi rimanenti. Questo lo si può fare con gli algoritmi di *elezione*, o algoritmi di AGREEMENT (per 'raggiungere un accordo' fra i vari sistemi).

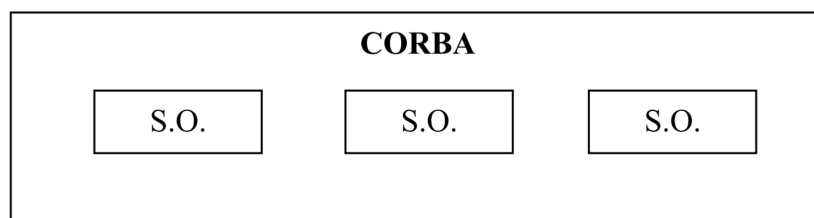


I sistemi distribuiti possono essere progettati per farvi girare applicazioni distribuite, o anche perché si cerca di avere con essi migliori prestazioni o maggior sicurezza.

Consideriamo per esempio il caso di un sistema 'vitale'. Tutti i dati e le istruzioni, come pure i risultati delle operazioni, sono corredati di opportuni codici di controllo per verificare la loro correttezza. Come può essere implementato un sistema vitale? Supponiamo di avere a disposizione un solo processore altamente affidabile, e due porzioni di memoria $M_1$ e $M_2$ per contenere i dati. Possiamo ricopiare *gli stessi dati* nelle due memorie per essere sicuri della correttezza dei dati. In un sistema multiprocessore invece, possiamo predisporre tre processori per eseguire lo stesso calcolo, più un processore che fa da *voter*, ovvero che preleva i risultati in uscita e decide 'a maggioranza'. Se, come nell'esempio, due processori danno 1 come risultato e il terzo dà 0, è naturale ipotizzare che l'uscita corretta sia 1; questo è un esempio - banale – di *agreement*. Bisogna però anche cercare di ripristinare il funzionamento del nodo 'dissidente' per evitare che continui a sbagliare nel futuro.

Nei corsi precedenti non abbiamo mai esaminato SO distribuiti (ovvero kernel distribuiti); al più abbiamo fatto cenno a SO aventi il file system distribuito (NFS, Network File System).

Una realtà commerciale piuttosto diffusa è quella in cui si ha un certo numero di macchine, ciascuno con il proprio SO, anche di tipi differente, e i vari SO sono integrati da una macchina virtuale, come CORBA, che implementa le tecniche cui abbiamo appena fatto cenno, o un loro sottoinsieme. Il ricorso alla macchina virtuale è reso necessario dal fatto che ciascun nodo possiede il proprio SO e quindi non siamo in presenza di un unico SO distribuito.




Una macchina virtuale è perciò una sorta di integratore di SO locali, in grado di risolvere problemi come quelli considerati (es. ordinamento eventi, mutua esclusione ecc.); e però naturale pensare di risolvere solo quel **sottoinsieme di problemi** inerenti lo scopo per cui viene costruita la macchina virtuale. Il software aggiuntivo CORBA, che viene installato su ciascun nodo, mette a disposizione una serie di funzionalità che potrebbero non servire e risulta pesante dal punto di vista computazionale.

In considerazione di ciò, in certe reti di calcolatori non si monta la macchina virtuale CORBA su tutti i nodi, ma piuttosto ogni nodo offre *servizi singoli* disponibili attraverso una porta d'accesso. Quindi è possibile accedere ai servizi

offerti dal nodo attraverso un *portale d'accesso ai servizi*. Per ciascun servizio saranno installate solo le proprietà che effettivamente servono: per un servizio potrebbe essere necessaria l'atomicità, ma inutile l'ordinamento degli eventi, ecc.

Ad esempio, nell'eventualità dei *tre processori più il voter*, l'ordinamento degli eventi sembra una proprietà inutile; è il voter che di fatto **sincronizza** i tre nodi, poiché costituisce una *barriera* (l'elaborazione non può proseguire finché non pervengono tutti e tre i risultati).

Un nodo che monta il *SO Linux* potrebbe avere invece l'esigenza di supportare l'ordinamento degli eventi; per conseguenza, incorporerà un task (in pratica, un demon) che gestisce tale algoritmo. Tale task permetterà la comunicazione con gli altri sistemi, sia verso l'interno (accesso alle risorse del sistema in esame) che verso l'esterno (risorse della rete), secondo i meccanismi di Inter Process Communication considerati nel corso di Sistemi Operativi. Da un nodo è possibile attivare il demon di un altro nodo se si possiede un *account* (diritto di accesso) per quest'ultimo e ricorrendo quindi ad una RPC (chiamata a procedura remota)

Per un *sistema di transazioni bancarie*, ad esempio accesso da parte di più persone ad uno stesso conto corrente, sono essenziali le proprietà di mutua esclusione (non possibili due operazioni simultanee) e l'atomicità (l'eventualità di un guasto alla rete non deve causare problemi) mentre non sembra essenziale, anche se sarebbe preferibile, l'ordinamento degli eventi. La prevenzione del deadlock in contesto applicativo bancario, come nel Bancomat, sembra argomento superfluo, ma può diventare importante se la transazione coinvolge vari enti bancari che si chiedono delle garanzie a vicenda.

È importante ricordare che una certa funzionalità (mutua esclusione) può essere garantita nei nodi di un sistema distribuito solo fintantoché tutti i nodi usufruiscono dello stesso software. Un ulteriore ambiente che si aggiunge in secondo tempo deve montare lo stesso software per poter interagire con gli altri secondo il medesimo algoritmo. Deve cioè essere *interoperabile* con gli altri. L'interoperabilità in questo contesto è la proprietà per cui se abbiamo un insieme di N nodi ciascuno con il proprio SO, è possibile realizzare un'unica macchina virtuale che mette a disposizione l'insieme delle funzionalità componendo i servizi messi a disposizione dai singoli nodi.

Ad esempio, il ministero delle finanze fornisce alle università la possibilità di fare un'analisi della denuncia dei redditi per verificare se gli studenti dichiarano in modo corretto il pagamento delle tasse. Questo pone ovvi problemi non solo di sicurezza (non tutti possono accedere ai dati riservati) ma anche di interoperabilità (vari sistemi informativi devono interfacciarsi: ministero delle finanze e università). In generale nel campo dell'e-government occorre, partendo da determinati sistemi distribuiti:

- installarvi sopra delle macchine virtuali;
- creare dei portali di accesso ai servizi, e
- far comunicare fra di loro i portali.

**MAZZOCCA 12/04/02**

L'**ORDINAMENTO DEGLI EVENTI** è il criterio fondamentale della coordinazione distribuita, perché la gestione delle priorità e quella dei conflitti si basano su di esso. Quindi se occorrono due eventi $e_1$ e $e_2$ che accadono agli istanti $t_1$ e $t_2$ è fondamentale comprendere e controllare la relazione d'ordine che c'è fra i due istanti di tempo (quale dei due eventi si è verificato per primo). Ad esempio se due programmi *che interagiscono* all'interno di un sistema distribuito stanno effettuando il debugging, è necessario sapere, mentre viene eseguita una certa istruzione del primo programma, quale istruzione del secondo viene eseguita (*correlazione delle tracce*).

Stabilire con precisione l'ordinamento degli eventi è fondamentale soprattutto nei **sistemi reattivi**, che devono onorare determinati vincoli di causa ed effetto; caso particolare dei sistemi reattivi sono i **sistemi real-time**, in cui al predetto vincolo si aggiunge anche quello temporale (gli eventi devono verificarsi entro scadenze prefissate).

Nei sistemi sequenziali la questione dell'ordinamento è banale, perché esiste un unico orologio che scandisce tutti gli eventi. Invece per i sistemi distribuiti possiamo dire che in generale è impossibile risolvere il problema dell'ordinamento perché **i clock sono scorrelati**, e perciò non può mai esserci una correlazione temporale fra due eventi. Non possiamo pensare di usare un CLOCK UNICO per tutti i nodi, perché in tal caso li costringeremmo ad avere la stessa struttura e quindi sarebbero compromessi la proprietà di eterogeneità e il concetto stesso di distribuzione. Inoltre inviando lo stesso segnale di clock su di una intera rete geografica non riusciremmo mai a sincronizzare i vari sistemi, in quanto il segnale nel corso della propagazione sarebbe sottoposto a vari filtraggi indesiderati e i nodi finirebbero col non vedere più alcun segnale. Si potrebbe ovviare con la tecnica del rifasamento, cosa che però porta con sé grandi complicazioni progettuali di ordine sia fisico che informatico.

Dobbiamo quindi considerare come assunto che nei sistemi distribuiti non è ragionevolmente possibile avere un unico ordinatore degli eventi.

Criterio di Lamport: **L'unica relazione che può sussistere in un sistema distribuito fra gli eventi è la relazione HAPPENED BEFORE** ("accaduto prima di"). Se dunque sappiamo che A è accaduto prima di B, e i due eventi interagiscono, possiamo fare in modo che il tempo di B si adegui a quello di A.

Se ad esempio A deve inviare un messaggio a B, quest'ultimo non potrà riceverlo prima che A gliel'abbia inviato. Indichiamo con $t_1$, $t_2$ i relativi istanti di tempo, interpretandoli come tempi *logici*: avremo quindi nei due istanti un eventi di send e di receive rispettivamente.

S $t_1$

R $t_2$

possiamo avere i tre casi:

- $t_1 > t_2$;
- $t_1 = t_2$;
- $t_1 < t_2$.

Le prime due relazioni sono logicamente impossibili, in quanto in contraddizione col significato dell'operazione. Solo se si verifica la terza eventualità, dunque, l'ordinamento degli eventi è mantenuto. Nel caso in cui dovesse verificarsi una delle due prime condizioni, l'evento B modifica il suo orologio logico al tempo dell'istante dell'evento A, più uno:

$t_2 = t_1 + 1$

Effettua cioè una **rifasatura logica** del tempo. Allo scopo è ovviamente indispensabile che i due eventi comunichino fra loro: LA RELAZIONE HAPPENED BIFORE NON PUÒ ESSERE SPECIFICATA SE NON È STABILITA UNA COMUNICAZIONE FRA GLI EVENTI.

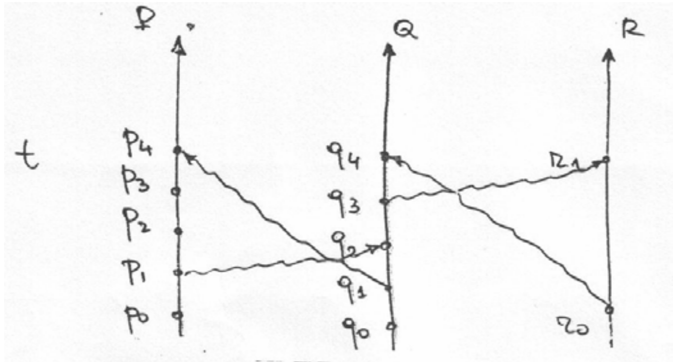
Nei sistemi distribuiti quindi l'ordinamento che si riesce a ottenere non è oggettivo, ma esclusivamente logico, ovvero **riguarda soltanto gli eventi che sono logicamente correlati**; i singoli nodi tornano a "scorrelarsi" non appena gli eventi A e B sono terminati. *L'ordine tra gli eventi scorrelati è cioè affidato al caso*. Gli eventi su Internet seguono questa fenomenologia. È comune ad esempio che una e-mail arrivi prima di un'altra che era stata inviata per prima. Se si vuole impedire tale eventualità, occorre instaurare un *processo di correlazione*, ossia in pratica man mano che la e-mail passa per vari nodi aggiorna il tempo logico, e grazie a ciò è possibile stabilire una relazione d'ordine.

Il sistema illustrato si chiama **logic clock**: processo di aggiornamento logico dei clock. Si può dimostrare matematicamente in quali casi il processo converge e in quali no. Grazie ad esso il corretto funzionamento dei sistemi reattivi è garantito, poiché diventa possibile instaurare un ordinamento globale fra gli eventi.

La relazione happened before può essere così definita:

- **Se A e B sono eventi di uno stesso processo e A è stato eseguito prima di B, si potrà dire che "A implica B" ($A \rightarrow B$).**
- **Se "A = invia messaggio" e "B = ricevi lo stesso messaggio" da parte di un altro processo si ha $A \rightarrow B$.**
- **Se $A \rightarrow B$ e $B \rightarrow C$, si ha $A \rightarrow C$**: 'ordinamento in triangolazione'.

Se si hanno tre eventi, P, Q ed R, in generale non è possibile dire alcunché sui tempi di attivazione, dato che sono generati da clock diversi e scorrelati. Tuttavia, se la situazione è quella descritta dalla figura che segue, possiamo individuare i seguenti ordinamenti parziali tra eventi:




$p_1 \rightarrow q_2$;
$r_0 \rightarrow q_4$;
$q_3 \rightarrow r_1$;
$q_1 \rightarrow p_4$;
$p_1 \rightarrow q_4$ (perché $p_1 \rightarrow q_2$ e $q_2 \rightarrow q_4$).

Ovviamente è ben difficile trovare le implicazioni in sistemi reali, che sono molto più articolati di questo esempio semplice.

Comprendiamo che se esiste un evento di sincronizzazione, tutti gli eventi che sono al di sotto di tale evento implicano quelli che vi si trovano al di sopra. Ciò vuol dire che c'è garanzia che, se un messaggio dev'essere inviato da $r_0$ a $q_4$, possiamo star certi che l'ordine logico sarà verificato, mentre se perviene un messaggio a $r_0$, non c'è garanzia che $q_0$ e $p_0$ vengano serviti in ordine corretto, poiché non c'è implicazione fra di essi (eventi scorrelati). La filosofia di assegnazione agli eventi di un **tempo logico** sotto forma di etichetta (e quindi non di un tempo fisico) è detta **timestamping**. [1]

Considereremo nel proseguimento una serie di algoritmi che si basano sul clock logico, e che convergono proprio in virtù di tale ipotesi. La loro convergenza non sarebbe cioè garantita in assenza del concetto di ordinamento logico globale. Nella tecnica del clock logico, tutti gli eventi partono in un tempo pari a '0 logico'. Ad ogni evento il clock logico viene incrementato; se gli eventi sono scorrelati non si può garantire fra di essi un ordinamento, mentre se sono correlati vengono eseguiti secondo l'ordine definito da tali implicazioni.

**PROBLEMA DELLA MUTUA ESCLUSIONE**. Supponiamo di avere N processi, ciascuno dei quali risiede su uno di N processori collegati in rete. Nel sistema è presente una *sezione critica* che può essere gestita unicamente in mutua esclusione. Mentre nei sistemi sequenziali abbiamo a disposizioni elementi come il *signal* a livello hw ed il SO a livello sw, che gestiscono in maniera abbastanza semplice la mutua esclusione, nel caso distribuito la cosa appare problematica.

Da notare che il problema si presenta non solo nel caso multicomputer ma anche in quello multiprocessore. Tali sistemi hanno infatti una memoria comune e una cache per ogni singolo nodo. Se la regione critica è implementata all'interno della

.

memoria comune non c'è rischio di conflitto, in quanto un processo che intende accedere ad una determinata zona della memoria può porre al rosso un semaforo e impedire l'accesso contemporaneo da parte di altri. Molto più complicato è il caso in cui il dato risiede nella cache di uno dei nodi. L'hardware di tali sistemi prevede a volta dei meccanismo nativi per ottenere la mutua esclusione; in altri casi la questione deve essere affrontata a livello di SO distribuito. Un modo di aggirare l'ostacolo è impedire che un certo segmento di indirizzi fisici possa essere mappato nella memoria cache, ma rimanga sempre e comunque nella memoria comune.

Una certa regione critica può essere eseguita se nessun processo vi accede. Presentiamo tre algoritmi per la gestione della regione critica: quello centralizzato, quello token-passing e quello totalmente distribuito. Nel **caso centralizzato**, ad un processo coordinatore viene affidata la gestione della regione critica; tutte le richieste (messaggi *request*) arrivano a quel processo, che le seleziona opportunamente sulla base dei time-stamp (tale informazione sarebbe inutile in un sistema monoprocessore).

Il coordinatore invia un messaggio di *reply* al processo autorizzato a entrare nella regione critica. Dopo essere uscito dalla regione critica, invia un messaggio di *release* al coordinatore. Per gestire il protocollo centralizzato occorrono quindi soltanto tre tipi di messaggi: request, reply e release. Un notevole problema si presenta nel caso in cui un processo viene abortito prima che abbia fatto in tempo ad inviare la release. Infatti regione critica e processo richiedente sono gestiti da due SO separati. Il problema viene risolto aggiungendo un quarto messaggio: *alive*, per sapere se il processo che aveva effettuato la richiesta 'è vivo'.

Un caso tipico in cui è utile la mutua esclusione è quando si vuole emulare un'architettura a memoria condivisa su di una rete di computer dotati di sola memoria privata. Si ricorre allora ad una *macchina virtuale* che implementi specificamente la proprietà di condivisione della memoria (creazione di un buffer di memoria accessibile simultaneamente da più utenti). In un sistema multiprocessore ciò che gestisce la mutua esclusione è il bus; la macchina virtuale deve quindi svolgere lo stesso compito del bus, emulato appunto in software (*ricorda*: emulare un bus in software vuol dire gestire la mutua esclusione).

Il ruolo di gestore del bus è rivestito da uno dei nodi, quindi se questo si rompe la rete non funziona più (*single point of failure*); possiamo però adottare un *algoritmo di elezione* in modo che all'occorrenza un altro nodo possa essere eletto come gestore della mutua esclusione (emulazione di una memoria condivisa affidabile). Tale nodo dovrà possedere anch'esso un'immagine della memoria condivisa. Ecco allora, come si diceva la scorsa volta, che una macchina virtuale deve possedere tutte le proprietà necessarie all'esecuzione del proprio lavoro: nel nostro esempio mutua esclusione ed elezione. Più evoluta è la macchina, maggiore è il tempo di elaborazione.

L'inconveniente del sistema centralizzato è la scarsa scalabilità, a motivo dell'elevato numero di richieste che può arrivare al processo in questione. Si può allora approcciare il modello **distribuito a token**: si fa circolare fra tutti i nodi un token; il nodo che di volta in volta possiede il token possiede il diritto di accesso alla regione critica. Anche in questo caso si ha poca scalabilità, poiché per mandare il token a tutti si è costretti a conoscere la totalità delle stazioni che fanno parte dell'anello. Da notare che in questo caso non occorre utilizzare i time-stamp.

Terzo modello è quello **totalmente distribuito**. In tal caso non esiste un singolo coordinatore della risorsa. Quando un processo $P_i$ vuole entrare nella regione critica genera un time-stamp Ts, e invia il messaggio request ($P_i$, $T_s$) a tutti i processi. Quando un altro processo $P_j$ riceve il messaggio, può rispondere immediatamente con un *reply* (se non intende entrare nella regione critica) o differire il tempo di replica (ha già chiesto l'accesso alla regione ed ha un time-stamp più vecchio, oppure è già nella sezione critica). Se $P_i$ e $P_j$ si erano correlati in un momento passato, viene mantenuto l'ordinamento reciproco. Il processo $P_i$ può entrare nella regione critica nel momento in cui riceve messaggi di reply da tutti gli altri. Durante l'uso della risorsa accoda eventuali messaggi di request da parte degli altri processi, ai quali risponderà con altrettanti *reply* all'atto di uscire dalla regione critica. Sui messaggi di reply viene imposto un time-out, di modo che se il messaggio di replica tarda ad arrivare, ad esempio a causa di un guasto, allo scadere del tempo fissato ci si comporta come se tale messaggio fosse stato inviato.

In questi due ultimi approcci non c'è il 'single point of failure': se un nodo si rompe, il resto della rete continua a funzionare in modo corretto. Si tratterebbe quindi di modelli maggiormente affidabili. Tuttavia non possiamo dirlo in generale. Esistono sistemi distribuiti (ad esempio quelli 'safety critical') nei quali la rottura di un singolo nodo può essere un evento catastrofico, nonostante il fatto che in questi casi non esista un nodo coordinatore.

Si noti anche che se esiste un solo punto di rottura, si ha anche un solo punto che dev'essere protetto. Ad esempio nell'architettura con 3 nodi che danno i risultati ad un voter, il quale poi decide 'a maggioranza', è di vitale importanza che il voter sia protetto sia da guasti che da errori casuali poiché un suo malfunzionamento renderebbe l'intera rete inutilizzabile, mentre se è uno dei tre nodi a sbagliare la rete continua ad essere utilizzabile nel breve periodo. Inoltre, è molto facile rendere affidabile il voter perché si tratta di un circuito semplicissimo (poche porte logiche). In un sistema distribuito potremmo invece essere costretti per varie ragioni a proteggere più sistemi, spesso ben più complessi di un voter, dall'eventualità di guasti, con conseguente aumento dei costi. Non è detto quindi in generale che gli approcci distribuiti siano da preferire a quelli centralizzati.

**IL DEADLOCK**. Altro problema dei sistemi distribuiti si ha quando l'allocazione delle risorse tra i vari nodi è tale che nessun nodo può proseguire nell'elaborazione perché necessita di una risorsa che è in possesso di qualcun altro (allocazione ciclica) e la rete va in stallo. Esistono due grandi famiglie di tecniche per aggirare tale situazione: quella del DEADLOCK PREVENTION e quella del DEADLOCK DETECTION. Si cerca cioè rispettivamente di impedire l'eventualità dello stallo o di rilevarne la presenza per poi rimuoverlo.

Deadlock prevention. Nel caso dei sistemi monoprocessore, abbiamo visto nel corso di Sistemi Operativi l'algoritmo del banchiere, il cui funzionamento potrebbe essere riassunto con la frase "presta la risorse solo a chi può restituirle". Tale algoritmo è stato studiato per il caso centralizzato e quindi non è troppo comodo da implementare nel caso distribuito. Causa un notevole overhead di comunicazione, in quanto consiste nell'inviare tutte le richieste di risorse ad un singolo processo che le assegna in modo opportuno. Non presenta invece particolari difficoltà di implementazione, come sempre avviene quando si demanda l'intero lavoro ad un solo nodo anziché tentare un approccio distribuito.

Maggiormente usato è l'algoritmo basato sull'ordinamento delle risorse: in tal caso, ad esempio l*a risorsa 3 può essere assegnata solamente ad un sistema che detiene già le risorse 1 e 2*. Dualmente rispetto al caso del banchiere, questo algoritmo trova enormi difficoltà organizzative nelle reti, perché richiede di numerare le risorse e non è una cosa ovvia se le risorse sono distribuite. La difficoltà si presenta specialmente in quelle eventualità in cui un nuovo nodo si aggiunge alla rete con le proprie risorse. Non si ha invece un significativo overhead di comunicazione.

Nella maggioranza dei casi si opta comunque per un diverso algoritmo, detto Timestamped Deadlock Prevention. Ad ogni processo viene assegnata una **priorità** *unica*. Un processo $P_i$ può ottenere una risorsa posseduta da $P_j$ solo se ha una priorità più elevata. Se questo è il caso, $P_j$ subisce un roll-back, altrimenti è $P_i$ che subisce un roll-back. Il processo che subisce il roll-back perde con effetto immediato l'uso delle **proprie** risorse.

L'algoritmo si basa sulla filosofia secondo cui un processo che ha necessità di essere eseguito velocemente non deve 'trovare ostacoli lungo il suo cammino'. Perché ciò avvenga dev'essere però possibile sottrarre una risorsa ad un processo che ne è in possesso. Si dice perciò che la tecnica è *pre-emptive*.

Nel caso in cui un processo venga rolled-back, esso deve poter tornare al suo stato precedente, di cui occorre quindi conservare memoria. Si apre quindi una problematica simile a quella del caso delle interruzioni precise. Si tiene memoria di un checkpoint fino al punto in cui si entra nella regione critica; nel momento in cui si esce dalla regione critica, lo stato memorizzato nell'ultimo checkpoint viene imposto come nuovo stato del processo. Per la coerenza del sistema è indispensabile che le azioni eseguite sulle risorse siano **atomiche**: proprio perché c'è la possibilità di togliere con

la forza ad un processo una risorsa, non deve essere consentito esercitare su di esse delle azioni 'frazionarie'.

L'eventualità di attese cicliche infinite è scongiurata, poiché nel wait-for-graph non possiamo sicuramente avere cicli. Un arco $P_i \rightarrow P_j$ in tale grafo può esistere solo se $P_i$ ha priorità più elevata; quindi chi ha una priorità più elevata 'rompe il ciclo' a proprio vantaggio.

L'ovvio problema legato a questa procedura è che un processo con bassa priorità può subire continui roll-back, non riuscendo mai ad aggiudicarsi una risorsa o perdendone continuamente l'usufrutto (magari un attimo prima di terminare).

Per ovviare a tale inconveniente si possono usare i time-stamp in luogo delle priorità. In tal modo il processo più prioritario è il 'più vecchio'. Il vantaggio sta nel fatto che il time-stamp è un concetto dinamico, poiché viene continuamente incrementato. In tal modo il time-stamp di un processo può finire col raggiungere quello di un altro. Si noti che ciò rende possibile che due processi siano ugualmente prioritari (stesso time-stamp) e questa situazione va opportunamente gestita.[2]

Un primo schema che usa i time-stamp è il **WAIT-DIE** che è di tipo NON PRE-EMPTIVE, cioè non prevede la possibilità di sottrarre una risorsa ad un processo. Se $P_i$ vuole una risorsa posseduta da $P_j$, può *mettersi in attesa* su di essa solo se ha un time-stamp più piccolo (è più vecchio[3]). Se invece $P_i$ è più giovane, subisce roll-back. È importante che alla 'morte' di $P_i$ *non* gli venga assegnato un nuovo time-stamp. $P_i$ può quindi ripresentare la sua richiesta, ma, finché la risorsa è occupata da $P_j$, non riuscirà mai ad ottenerla e continuerà a subire roll-back. Questo sistema può causare quindi un gran numero di roll-back. [4]

Supponiamo per esempio di avere tre processi con i seguenti time-stamp:

$P_1 \rightarrow 5$
$P_2 \rightarrow 10$
$P_3 \rightarrow 15$

Se $P_2$ è in possesso di una risorsa, e $P_1$ la richiede, viene posto in attesa; se poi anche $P_3$ richiede tale risorsa, viene rolled-back perché il suo time-stamp rivela che è più giovane.

Supponiamo invece che $P_2$ richieda una risorsa a $P_3$ e $P_3$ la richieda a $P_2$ (potenziale ciclo). In tal caso $P_2$ si accoda su $P_3$. Quanto a $P_3$, la sua richiesta non può causare un ciclo perché avendo un time-stamp più elevato è più giovane e quindi viene rolled-back. *Questo significa che $P_3$ perde le proprie risorse* e quindi l'esecuzione di $P_2$ può proseguire. Sottolineiamo quindi che un processo non può sottrarre con la forza le risorse ad un altro (non pre-emptive), però può perdere le eventuali risorse che sta

usando se effettua una richiesta che finisce col causare il suo stesso roll-back. In questo schema, l'attesa di un processo può terminare solo nel momento in cui il processo più giovane ha rilasciato la risorsa.[5]

Il secondo schema è il **WOUND-WAIT** ed è PRE-EMPTIVE[6]. Se $P_i$ vuole una risorsa posseduta da $P_j$, può mettersi in attesa solo se ha un time-stamp più grande ($P_i$ è più giovane di $P_j$). Contrariamente, $P_j$ è rolled-back e la risorsa viene assegnata a $P_i$. Anche in questo caso il roll-back deve comunque preservare il time-stamp originale. A questo punto $P_j$ ripresenta la richiesta della risorsa appena persa, e, essendo $P_i$ più vecchio, $P_j$ si mette in attesa. Si noti che in questo caso un processo anziano non ne attende mai uno giovane.

Nell'esempio di prima:

$P_1 \rightarrow 5$
$P_2 \rightarrow 10$
$P_3 \rightarrow 15$

Se $P_2$ è in possesso di una risorsa, e $P_1$ la richiede, $P_1$ riesce ad ottenerla (più vecchio) e $P_2$ viene rolled-back. Se $P_3$ richiede una risorsa posseduta da $P_2$, viene posto in attesa perché il suo time-stamp rivela che è più giovane.

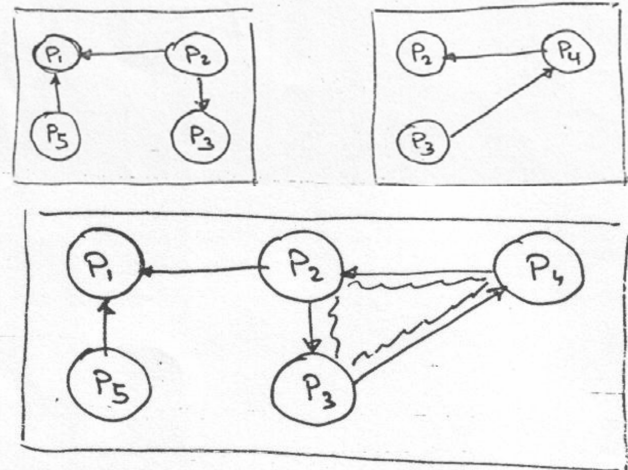
Nelle due tecniche l'eventualità di attesa indefinita è scongiurata dal fatto che prima o poi un processo avrà il time-stamp più basso, e quindi rispettivamente si mette in attesa o si aggiudica la risorsa. Nella modalità wound-wait l'occorrenza di roll-back è inferiore rispetto al caso wait-die. Entrambi gli schemi recano comunque lo svantaggio di poter causare dei roll-back non necessari. In particolare le tecniche pre-emptive possono essere nocive, perché si può causare il roll-back di un processo un attimo prima che il processo stesso abbandoni spontaneamente la risorsa, determinando inutili perdite di tempo.[7]

**MAZZOCCA 19/04/02**

Abbiamo parlato la scorsa volte della deadlock prevention. In un sistema centralizzato si può assumere come fatto scontato che il supervisore sia in grado di monitorare e controllare l'intero stato del sistema, e quindi anche l'alternarsi dei processi nell'utilizzo delle risorse. Nel distribuito, questo concetto viene a cadere, perché lo stato diventa una *collezione* di più stati, ognuno soggetto al proprio

schedulatore, e la *contemporaneità* nel cambio degli stati non è in alcun modo garantita. Il ricorso ai timestamp diviene necessario per gestire le priorità, e consente di gestirle in modo dinamico. Abbiamo visto comunque come questi algoritmi facciano perdere molto tempo in fase decisionale e per di più esiste il rischio di sottrarre le risorse ad un processo quando non ce n'è la necessità.

Consideriamo ora il criterio alternativo per il trattamento del fenomeno del deadlock, la **deadlock detection**. È un sistema potenzialmente più veloce, poiché la risorsa viene data in ogni caso a chi la richiede. Ogni tot di tempo si interviene a verificare se per caso il sistema non si trovi in stallo. Come sempre, ciò che rende difficoltosa la questione è la sua distributività. Su un sol nodo[8] si può esaminare il 'grafo delle attese' e scoprire che esiste un ciclo che genera il deadlock. In questo caso però risorse e richiedenti sono distribuiti. Si noti ad esempio i due grafi sulla destra; rappresentano due differenti siti, dove $P_i$ sono nomi di processi. Così, nel primo sito i due processi $P_2$ e $P_5$ hanno presentato una richiesta a $P_1$, e così via. In un grafo locale possono essere presenti processi locali e non locali e questi possono chiedere risorse appartenenti a qualunque nodo. I processi, dunque, possono essere presenti in più siti, perché possono richiedere risorse appartenenti a siti differenti. Così, i processi $P_2$ e $P_3$ hanno richiesto risorse in almeno due siti differenti.




Nei due grafi locali non risulta alcun ciclo. Un eventuale osservatore globale, però, riscontrerebbe un ciclo fra i processi $P_2$, $P_3$ e $P_4$, come è facile comprendere confrontando i due grafi. L'assenza di cicli nei wait-for-graph locali, cioè, non esclude la presenza di cicli nel wait-for-graph globale.

Come si può progettare un'architettura che consenta di avere informazioni sullo stato globale?

In un sistema distribuito l'ideale è poter riprodurre una *visione globale* del sistema, e questo lo si può fare in due modi differenti: o uno dei nodi ha la prerogativa della gestione centralizzata del sistema, oppure i nodi si scambiano dei messaggi in modo che ciascuno di essi abbia una visione sufficientemente ampia della rete.

Cominciamo a considerare il primo caso, l'**approccio centralizzato**. Il nodo che si rende conto dell'esistenza di un deadlock lo può eliminare togliendo a qualcuno dei processi le risorse di cui è in possesso. È il modello più vicino a quello che si è visto nel corso di Sistemi Operativi. I vari nodi spediscono dei messaggi al nodo

coordinatore, nei quali indicano il loro stato attuale, e il nodo coordinatore costruisce sulla base di tali messaggi un grafo globale. Tale costruzione può avvenire in tre diverse modalità:

1. aggiorniamo il grafo ogni volta che un arco è inserito o rimosso nei grafi locali.
2. costruiamo il grafo periodicamente, dopo un certo numero di variazioni nei grafi locali; possiamo combinare tale metodo con la tecnica impiantistica del *tuning*, in cui aumentiamo ogni volta di uno il numero di archi da modificare prima di costruire il grafo, per trovare il valore ideale di tale parametro.
3. lo costruiamo quando il coordinatore deve invocare l'algoritmo per l'individuazione dei cicli

In generale, per stabilire quale dei tre algoritmi funziona meglio (nel senso della correttezza e dell'efficienza) occorre usare strumenti matematici/probabilistici e simulazioni.

Il principale problema di questo approccio sta nel fatto che i messaggi dei vari nodi viaggiano in un tempo finito, per cui arrivano al coordinatore con qualche ritardo, e nel frattempo il loro stato può evolvere notevolmente: alcune risorse possono essere liberate e altre possono essere occupate nel giro di pochi istanti. Un ciclo può essersi sciolto e un altro può essersene formato. Il nodo cioè potrebbe avere una 'istantanea' non attendibile della rete ed effettuare di conseguenza delle *false detection*: comprometiamo inutilmente la <u>*disponibilità*</u> del sistema. Si potrebbe pensare di interrompere tutti i sistemi per il tempo necessario, da parte del nodo centrale, a conoscere lo stato effettivo della rete, ma ciò è gravemente inefficiente, quando non impossibile.

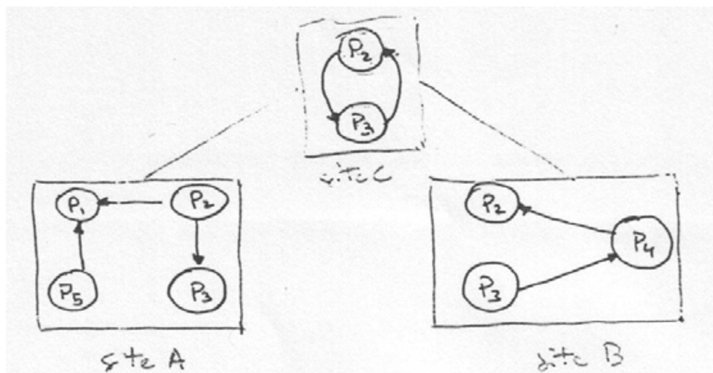
Volendo fare un discorso più formale, a fronte del **grafo reale** della rete, abbiamo quindi un **grafo costruito** che ne è una approssimazione generata dal coordinatore; quando tale grafo evidenzia un deadlock, dovremmo essere sicuri che il sistema è realmente in tale situazione, ma non è semplice garantirlo. È facile fare un esempio in questo senso. Supponiamo che il processo A richieda a B una risorsa $R_1$ nel tempo t. Il processo B libera $R_1$ al tempo $t - \Delta$. In queste condizioni, dall'esterno osserviamo che non c'è contesa di risorsa. Se però l'informazione di risorsa liberata da parte di B, viaggiando nella rete, arriva al coordinatore solo all'istante $t - \Delta + k$, e la richiesta da parte di A arriva all'istante $t + l$, il coordinatore potrebbe osservare un conflitto nel caso in cui si abbia $l < k - \Delta$. La **relazione di corretto funzionamento** è dunque $\mathbf{l > k - \Delta}$, ma, non potendosi questo garantire con sicurezza, l'algoritmo può non funzionare.

La tecnica centralizzata dunque non viene di solito presa in considerazione, a vantaggio dell'**approccio gerarchico** . In tale alternativa, le informazioni per la detection degli stalli non sono centralizzate, ma distribuite.

Nell'approccio gerarchico, ogni sito gestisce un wait-for-graph locale, e il graph globale è distribuito tra una gerarchia di controller organizzati ad albero, nel quale ogni foglia contiene il wait-for-graph di un solo sito.

Ogni nodo che non è foglia è un controller di livello superiore che contiene un grafo con informazioni ricavate dai grafi dei controllers dei sottoalbero che esso domina. In pratica, ogni controller controlla un certo numero di siti per vedere se c'è deadlock, e passa la relativa informazione ai controller sovrastanti.

Se A, B e C sono controller e C è il più basso progenitore comune di A e B, supposto che il processo $P_i$ compaia nel grafo locale di A e B, esso deve comparire anche nei grafi locali del controller C, di ogni controller nel path da C ad A e di ogni controller nel path da C a B. Ogni controllore raggruppa quei nodi che possono interagire fra di loro e quindi occorre conoscere la distribuzione dei processi lungo i vari nodi. Conviene che tale informazione sia pianificata staticamente e non ricavata dinamicamente, alternativa che recherebbe enormi complicazioni algoritmiche.

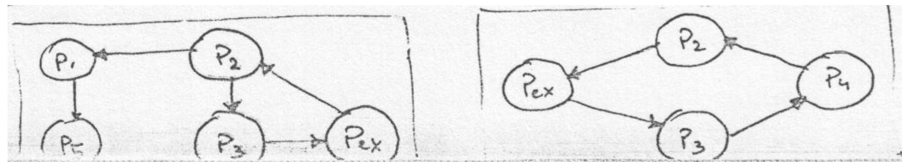



Consideriamo l'esempio sulla sinistra. Il nodo C, come si vede, contiene solamente i nodi comuni ai due grafi sottostanti e quindi $P_2$ e $P_3$. Dalle informazioni che gli pervengono dai siti A e B (che inviano solo quelle informazioni relative ai nodi che potrebbero causare ciclo, e quindi $P_2$ e $P_3$[9]), esso costruisce un grafo e riscontra la presenza di un ciclo. Può quindi decidere di inviare il comando di abortire l'arco o al sito A o al sito B; il sito interpellato prenderà le contromisure necessarie a eliminare il ciclo.

Il vantaggio è che in questa modalità si inviano relativamente pochi messaggi, tra l'altro abbastanza semplici, lungo la rete. Da notare però che peggiora l'affidabilità, poiché avendo più coordinatori, aumenta la probabilità che si rompa uno di essi e che quindi non vengano rilevati dei cicli.

Il terzo e ultimo approccio che consideriamo è l'**approccio completamente distribuito**. Nel grafo di ogni sito può apparire un nodo addizionale esterno. Ogni qual volta un nodo interno interagisce con l'esterno (perché il nodo interno è in attesa di una risorsa presente in un altro sito e che è correntemente posseduta da qualcun altro) o viceversa, si pongono il nodo $P_{EX}$ ed un arco nel grafo per segnalare tale interazione. Si consideri l'esempio riportato sotto a sinistra, in cui

all'interno del nodo $P_2$ chiede una risorsa a $P_3$, ma c'è anche un nodo esterno che interagisce con entrambi nel modo indicato. La presenza di un ciclo simile denota solo che è *possibile* una situazione di stallo. Per saperlo con certezza, occorre inviare al sito su cui $P_{EX}$ è presente (sito di destra) il sottografo relativo ai nodi che fanno ciclo con $P_{EX}$.



Non sono possibili falsi deadlock, perché nel momento in cui si invia il grafo ad un altro sito, quest'ultimo viene impegnato in un'operazione di ingresso/uscita e quindi non può svolgere altre operazioni, come liberare le risorse; né può svolgere altre operazioni il sito che ha inviato il grafo. La 'fotografia' fatta del sistema è, cioè, sempre consistente. Il sito destinatario unisce il grafo ricevuto col proprio grafo e rileva la presenza di eventuali cicli. La risposta potrebbe non essere determinabile se non prima di aver interpellato un terzo grafo, poi un quarto e così via. Possono essere cioè coinvolti in tale scambio di informazioni più di due siti. Il procedimento termina finché si rileva il deadlock (in tal caso occorre abortire uno dei processi che fanno ciclo), oppure non si trovano cicli e quindi non si ha deadlock.

L'ovvio inconveniente di tale sistema è che può essere necessario inviare una gran quantità di messaggi sulla rete, diminuendo il grado di parallelismo, anche se magari nella maggioranza dei casi non si ha deadlock. Tale problema può essere migliorato? Sì, se esiste la possibilità di allocare i processi nel sistema in modo da minimizzare la conflittualità.

Anche questa procedura potrebbe essere portata avanti mediante il tuning. L'allocazione dei processi può essere programmata attraverso una *macchina virtuale*. Ad esempio: progettiamo un sistema Web con una grande base di dati. Quando arrivano le richieste, si creano dei task che le servono. Il sistema potrebbe essere suddiviso su più server per diminuire il sovraccarico, cosicché anche le risorse sono suddivise su più nodi, e lo sono per conseguenza i task. Al solito, scegliendo la strada centralizzata, ovvero ponendo i task tutti su di uno stesso nodo, si hanno notevole conflittualità ma maggiore controllabilità; se si opta per la decentralizzazione è esattamente il contrario. Il tuning permette di determinare la migliore politica di allocazione. La macchina virtuale (*broker*) fa sì (attraverso un esteso impiego dell'indirizzamento indiretto) che i task indirizzino le risorse mediante indirizzi non fisici, ma logici, in modo che possano essere opportunamente spostati fra i nodi (per poter fare il tuning).

**Di un sistema distribuito, a questo punto**, sappiamo ordinare gli eventi, gestire la mutua esclusione, prevenire o eliminare il deadlock.

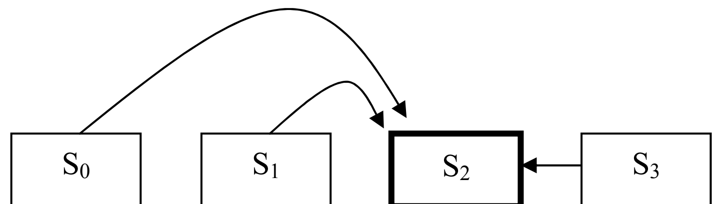
Nel corso di Sistemi Operativi abbiamo trattato anche il File System; la sua gestione in un sistema distribuito è meno complessa di quanto possa sembrare, poiché si tratta in definitiva di gestire strutture dati in mutua esclusione.

C'è poi la schedulazione di un sistema distribuito. È un aspetto che presenta maggiori difficoltà, poiché nel considerare le reti abbiamo supposto che su ogni singolo nodo c'è un SO tradizionale (NT, Linux …), che schedula per conto suo i processi (gestione LIFO, cambio di contesto ecc.). Immaginare la presenza di uno schedulatore centralizzato, che interagisca con gli schedulatori locali, introduce problematiche molto particolareggiate che vanno al di là dei limiti di questo corso. Diciamo solo che gli schedulatori distribuiti generalmente non entrano nel merito dei singoli SO residenti nei nodi, ma forniscono delle funzioni globali per lo *scambio di messaggi* (primitive di comunicazione) fra di essi.

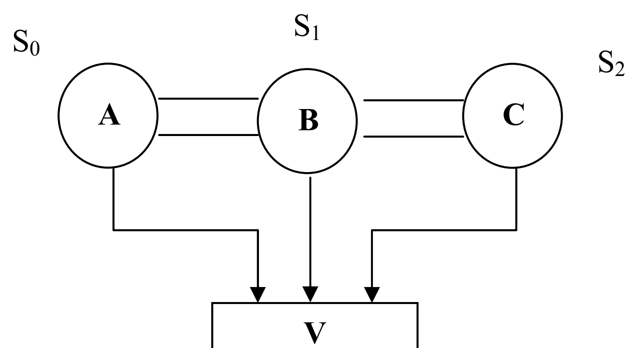
Passiamo invece ad approfondire l'argomento del **distributed agreement** (accordi distribuiti), che è il più complesso di quelli che considereremo sui sistemi distribuiti.

Se vogliamo che lo stato di un sistema centralizzato non sia corrotto, possiamo proteggere i dati con tecniche di *codifica*. In pratica si aggiungono dei bit di controllo che garantiscono la coerenza dello stato. Altra tecnica alternativa (o che si aggiunge) alla precedente consiste nel *duplicare* nella memoria i dati da proteggere.

In una rete, se si vuole che tutti i nodi conoscano con esattezza il suo stato (il che è come dire, che ogni nodo conosca lo stato di tutti gli altri) si apre una problematica di *agreement distribuito*. Ad esempio, si consideri l'architettura a 4 nodi stilizzata qui di seguito. L'elaborazione di $S_2$ non può proseguire finché tale nodo non ha ricavato l'informazione sullo stato dell'intero sistema. Gli altri tre nodi quindi dovranno inviargli ciascuno il proprio stato, come indicato in figura. Se tale informazione non perviene, l'intero sistema rimane bloccato; quindi il protocollo deve prevedere la possibilità di trasmettere lo stato del singolo nodo. Da notare che per la coerenza è indispensabile che tale trasmissione sia *sincrona*.




Caso particolare ed esempio classico è quello già considerato del voter su 3 nodi, in cui A, B e C eseguono la stessa attività. Questa architettura la si può usare al posto della precedente perché non rende necessaria una codifica dei dati; se uno dei tre nodi sbaglia a effettuare il calcolo, tale errore è ovviato dal fatto che V decide



per la maggioranza di 2 su 3. Nella figura abbiamo anche voluto indicare i tre stati ($S_i$). Supponiamo che il nodo C sbagli: lo stato $S_2$ è errato. Se non si interviene su C, tale nodo potrebbe continuare a sbagliare nelle elaborazioni future. Se in seguito, per qualche motivo, anche B dovesse cominciare a sbagliare, il voter inizierebbe anche lui a prendere decisioni errate perché è la maggioranza dei nodi a essere in errore. È allora indispensabile che al termine di ogni elaborazione i tre nodi si scambino gli stati per verificare che *lo stato sia lo stesso* per tutti. Essi devono perciò pervenire ad un *agreement* (accordo). Il nodo che si ritrova in minoranza (cioè con lo stato diverso dagli altri due) modificherà il proprio stato per renderlo uguale a quello della maggioranza. È evidente che i tre nodi devono essere opportunamente istruiti in proposito.

Quando un componente all'interno di una rete prende a non funzionare, fornendo cioè risultati sbagliati, e tale malfunzionamento può indurre un malfunzionamento anche negli altri nodi, si dice che si ha a che fare con un **problema dei generali bizantini** ovvero che si è verificato un 'errore bizantino'. Risolverlo significa ottenere l'agreement fra *n* nodi per ricavare una informazione globale consistente, anche nell'ipotesi che uno di essi abbia subito un guasto.

Un concetto intuitivo, su cui torneremo, è che l'agreement sia possibile solo a determinate condizioni sul numero di nodi coinvolti. Ciò è evidente già nell'architettura col voter: un accordo può essere raggiunto solo se si hanno almeno tre nodi (più un 'voter' in grado di decidere a maggioranza; si ipotizza che il voter sia certamente affidabile), in quanto in presenza di due soli nodi contrastanti *non* è possibile sapere quale di essi stia funzionando correttamente.

Le configurazioni di errore possono essere le più disparate, anche in presenza di pochi nodi. Supponiamo ad esempio che su 4 generali uno sia mentitore, fornendo risultati errati agli altri tre, e per di più tre risultati diversi fra loro. Ancora più sottilmente, può fornire il risultato corretto a 2 nodi e un risultato errato al terzo nodo. Nel caso generale sappiamo che almeno uno dei componenti 'mente' e che di conseguenza non si è raggiunto un agreement distribuito. L'algoritmo dei generali bizantini permette di sapere se c'è almeno un nodo 'mendace'. In molti casi è possibile, oltre che determinare l'errore, anche correggerlo, ma l'algoritmo non lo garantisce nel caso generale. In altri casi si può solo rilevare l'errore senza correggerlo, ossia constatare che non c'è stato agreement, senza riuscire a determinarlo. Vedremo esempi nell'uno e nell'altro senso.

Formulazione generale del problema dei generali bizantini: supponiamo di avere N processi, ciascuno dei quali ha un valore privato $V_i$. Sia M il numero massimo di nodi faulty. Si vuole costruire un algoritmo che consenta ad ogni processo $P_i$ (non-faulty) di individuare un vettore $X_i = (A_{i,1}, A_{i,2}, \dots A_{i,N})$ tale che:

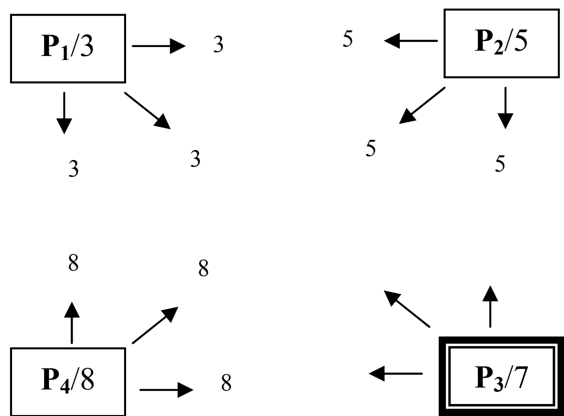
- se $P_j$ è un processo non-faulty, allora $A_{i,j} = V_j$ (cioè lo stato di $P_j$ visto dal processo $P_i$ è quello reale)

- se $P_i$ e $P_j$ sono non-faulty, $X_i = X_j$ (i due processi hanno la stessa informazione sullo stato globale).

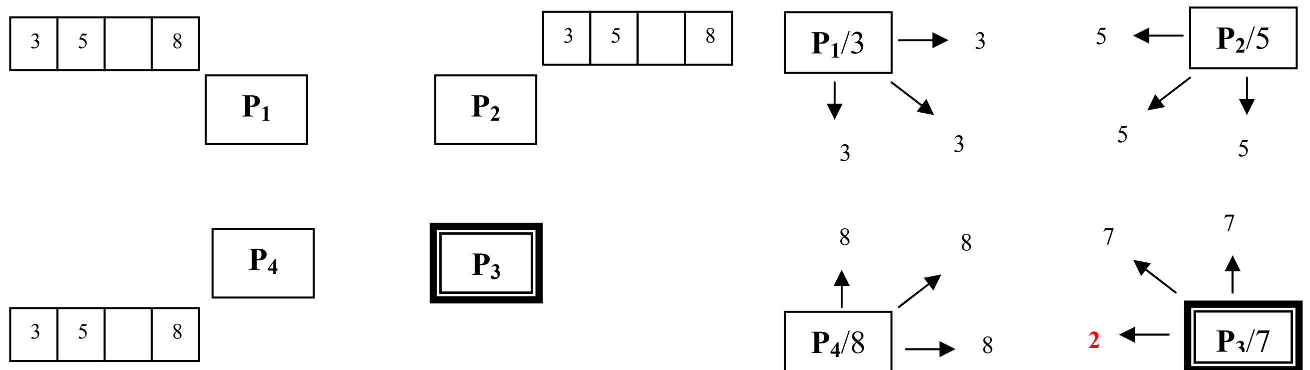
Nel caso in cui non si raggiunga l'accordo per un particolare stato, si potrò segnare in corrispondenza un valore 'null' (non specificato).

Il problema è **risolvibile** (è possibile *rilevare* l'esistenza di un errore) solo se $N \geq 3*M + 1$. Nel peggiore dei casi, il ritardo per raggiungere l'accordo è proporzionale al ritardo di (M+1) messaggi. Il numero di messaggi richiesto per raggiungere l'accordo è molto elevato. Poiché nessun processo è sicuramente affidabile, tutti i processi devono raccogliere tutte le informazioni possibili e prendere di conseguenza le proprie decisioni.

**Un esempio**. Se uno dei nodi è mendace, servono sicuramente due round. Nell'esempio indicato, i processi P, $P_2$ e $P_4$ sono veritieri, e $P_3$ è mendace.



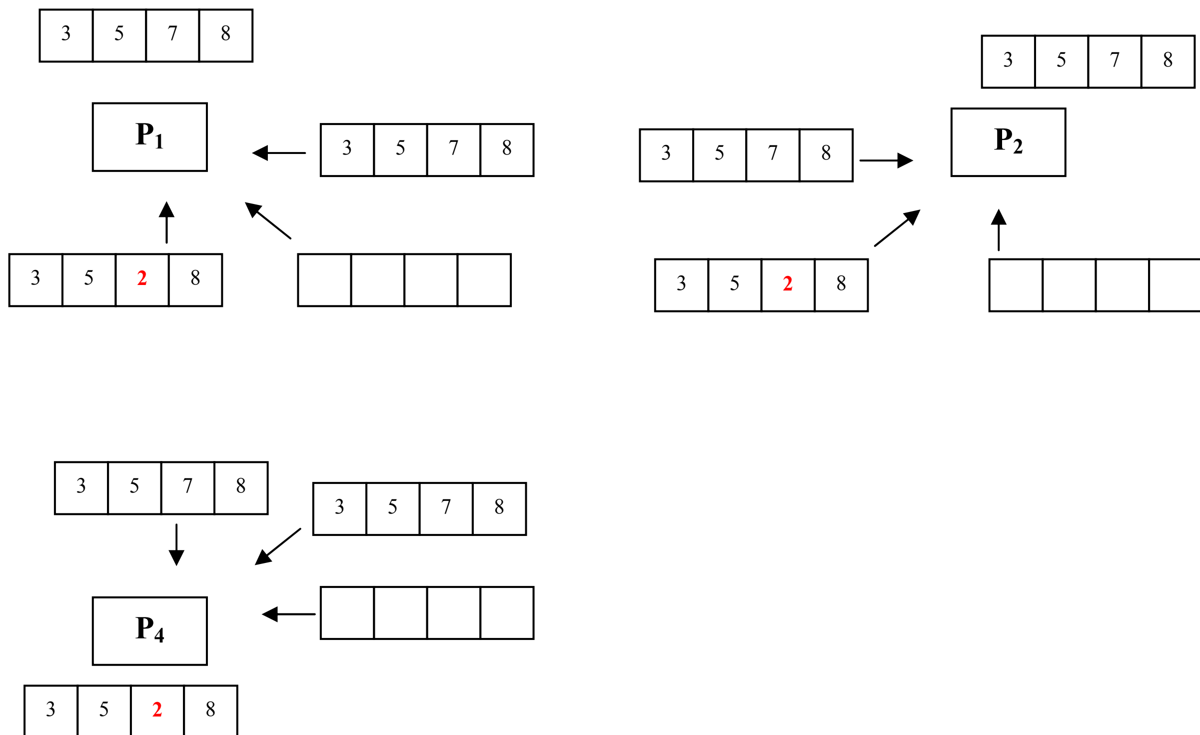

Di conseguenza, tre processi su quattro possono scambiarsi informazioni esatte sullo stato del sistema (figura sotto a sinistra).



Nella figura di destra possiamo notare come il nodo $P_3$ abbia un comportamento bizantino: invia a due degli altri nodi il valore corretto, 7, e al terzo (il nodo $P_4$) il valore errato 2.

Se facessimo un solo round, $P_4$ avrebbe una immagine discordante ed errata dello stato globale del sistema. Non si è cioè raggiunto un agreement distribuito.

Al SECONDO ROUND, tutti i nodi mandano ad ognuno degli altri il proprio stato globale. Ignoriamo il punto di vista di $P_3$, che sappiamo essere guasto.



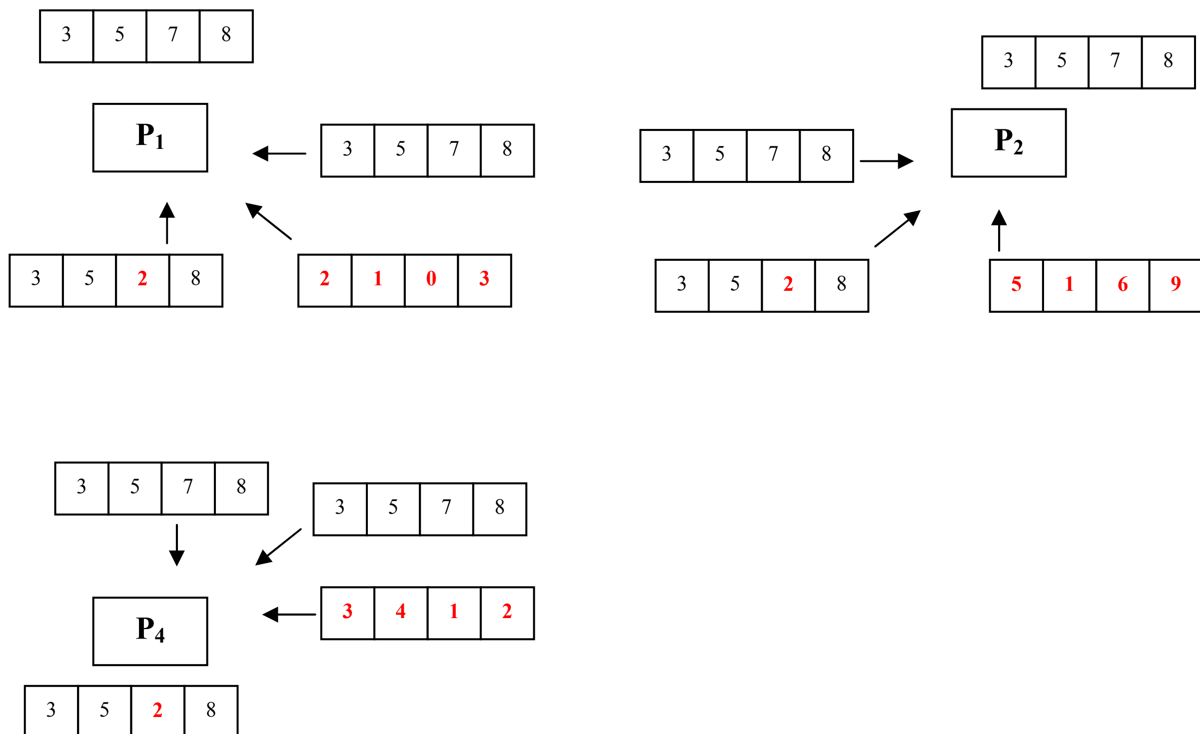

A questo punto, l'algoritmo deve votare. Guardiamo la cosa dal punto di vista di $P_1$. Sul primo valore (3) non ci possono essere dubbi, e l'algoritmo deve solo rilevare se tale stato è visto correttamente dagli altri 3 sistemi.

Per il secondo valore, 5, si noti che $P_1$ non può basarsi sull'immagine di tale stato fornitagli da $P_2$, perché tale informazione gli era stata già data in precedenza proprio da $P_2$ che potrebbe teoricamente mentire (altrimenti è come se si desse a $P_2$ il diritto di votare due volte). $P_1$ deve cioè decidere sulla base del proprio valore e di quello degli altri due nodi. Poiché sulla base di tale scelta emerge una maggioranza di 2 su 3, anche il secondo stato è assestato al valore 5. A questo passo dell'algoritmo dovrebbe essere chiaro perché servono altri 3 nodi (3M+1 = 4).

Per il terzo valore si riscontra una difformità. A questo punto si può procedere in maniera ottimista, ragionando che esiste comunque una maggioranza 2 su 3 (il valore registrato, cioè quello che gli era prevenuto da $P_4$ – il cui voto quindi non deve essere più preso in considerazione – più il valore che gli viene da $P_2$), o in maniera cautelativa. Nel primo caso si sceglie 7 come valore presumibilmente esatto, nel secondo caso tale valore viene scartato. Supponiamo per ora di essere ottimisti; torneremo fra breve su questa seconda eventualità.

L'ultimo valore (8) può essere accettato sulla base delle stesse considerazioni viste per il secondo (5).

Torniamo ora al secondo valore (valore corretto: 5). Supponiamo che il nodo guasto mandi agli altri nodi un'informazione errata anche sulla propria versione dello stato globale, come nel caso di seguito indicato (il problema comincia a diventare *fortemente bizantino*).




In questo caso, $P_1$ vede giungere da $P_4$ il valore 5 e da $P_3$ il valore 1. Il voto di $P_2$, già usato, non può essere preso in considerazione. Se si opta per la strategia 'cautelativa' di cui si diceva prima, anche questo valore dovrebbe essere scartato poiché si nota un disaccordo fra due nodi. Lo stesso ragionamento si può fare per il quarto valore (8 secondo $P_1$ e $P_2$, ma 3 secondo $P_3$).

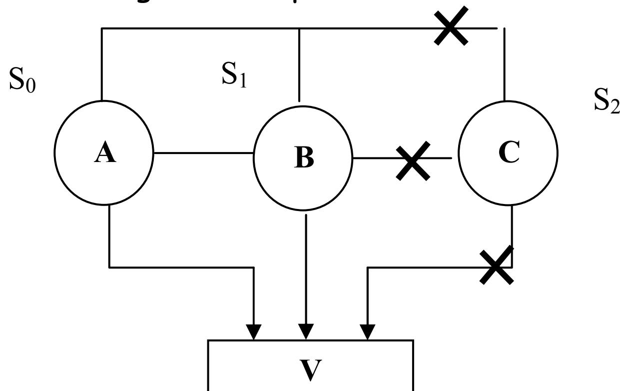
A questo punto può subentrare un criterio relativo alla *storia passata* del sistema. Nel caso del quarto valore, 8, se $P_2$ non ha mai sbagliato in precedenza (o ha sbagliato al massimo proprio lungo la linea di comunicazione con lo stesso $P_3$) e $P_3$ ha

già sbagliato, si può assumere che la versione corretta è 8 e correggere lo stato errato di $P_3$. Un caso in cui evidentemente si può *senz'altro* ritenere che sia $P_3$ ad errare è quando quest'ultimo manda valori diversi a tutti gli altri, come avviene, si noti, nell'esempio indicato.

Nei sistemi in cui lo stato di tutti i nodi è lo stesso (es: sistemi di sicurezza dell'Ansaldo per i trasporti ferroviari) l'algoritmo prevede evidentemente delle semplificazioni: ad esempio non sono necessari 4 nodi ma ne bastano sicuramente 3. Rimane comunque la strategia a due round, perché non si può essere sicuri sull'affidabilità dei *canali di comunicazione* fra i vari nodi. Un nodo può funzionare correttamente di per sé, ma il canale di comunicazione può avere dei problemi e corrompere il dato in uscita.

**MAZZOCCA 23/04/02**

Abbiamo considerato la scorsa volta l'algoritmo dei generali bizantini, necessario per raggiungere l'agreement per lo stato di un sistema distribuito. Caso particolarmente semplice è quello del sistema con il voter, in cui la semplificazione è costituita dal fatto che gli stati devono essere uguali per tutti i sistemi: $S_0 = S_1 = S_2$. Basilarmente questo *non è* un problema di agreement, ma lo diventa nel momento in cui nasce l'esigenza da parte dei tre nodi di scambiarsi lo stato. Se uno dei sistemi, poniamo sia C, mostra uno stato sbagliato, per un errore di elaborazione interno, o perché c'è un errore sulle linee di trasmissione, o sul driver, o per qualunque altra ragione, è inutile continuare a prenderlo in considerazione: gli si potrà dare al massimo una 'seconda possibilità', scambiando ancora lo stato con lui, nella speranza che si sia trattato di un errore transitorio, ma se la cosa si ripete occorrerà escludere sia la sua linea di uscita verso il voter che le linee di scambio con gli altri due processori. Si passa quindi, mediante una *logica di esclusione*, dal sistema 2 su 3 al sistema 2 su 2. Occorrerà passare ai nodi superstiti l'informazione sulla mutata configurazione di rete. Si noti anche che, in questa evenienza, *non è più necessario scambiare lo stato* perché una eventuale discordanza degli stati non permette in nessun modo di stabilire quale dei due può aver ragione.



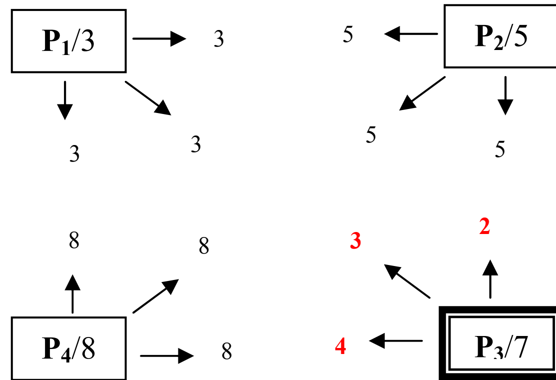

Nel caso più generale, abbiamo fatto nell'ultima lezione un esempio di problema bizantino. Il nodo $P_3$ comunicava un dato errato (2 anziché 7) presumibilmente a motivo di un problema nel canale di comunicazione. Prima di proseguire, sottolineiamo un aspetto abbastanza importante. Se il nodo $P_3$ commette un *errore* nel calcolare il valore dello stato, supponiamo che tale valore errato sia 7, e trasmette agli altri tre questo valore, *non si ha un problema bizantino*. Tale errore di elaborazione potrebbe essere risolto solo utilizzando architetture come quella vista sopra, con il voter. L'errore bizantino si verifica quando il nodo invia, per lo stato, un valore errato a uno degli altri processi.

Abbiamo visto che se un solo nodo mente, occorrono due round. Nel primo round ogni nodo riceve lo stato da tutti gli altri. Nel secondo ciascun nodo invia la propria immagine dello stato globale a tutti gli altri e si possono registrare eventuali discordanze. In generale, se M nodi mentono, occorrono (M+1) round.

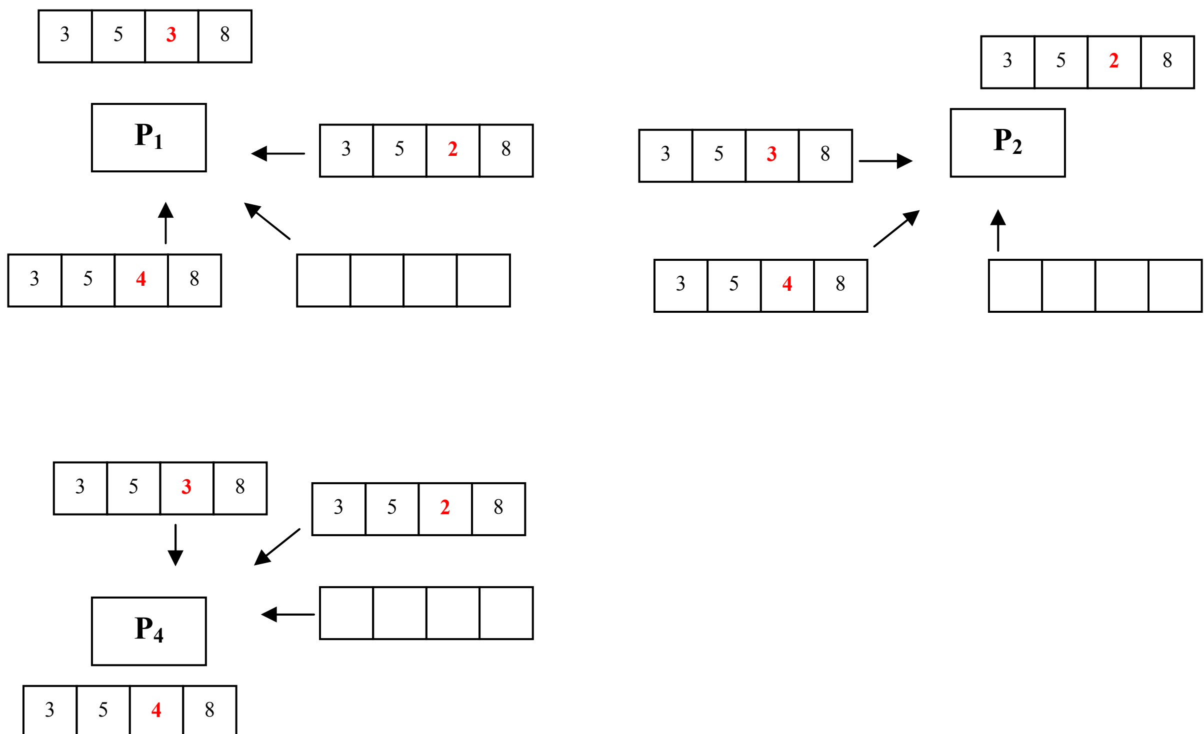
Se il nodo mendace invia tre valori diversi del proprio stato agli altri nodi (7, 7 e 2) , $P_1$ riesce a ricostruire uno stato corretto mediante semplici scelte a maggioranza. Anche $P_4$ riesce a farlo. Il primo, secondo e quarto valore sono posti correttamente a 3, 5 e 8. Per il terzo valore, la sua 'salvezza' sta nel fatto che è

proprio il nodo mendace a non dover essere preso in considerazione, per la solita considerazione sul doppio voto. $P_4$ vede dunque per tale stato che il suo valore è 2, mentre gli altri due nodi $P_1$ e $P_2$ hanno 7, e quindi ricostruisce a maggioranza il corretto stato (7).

Consideriamo ora un altro caso, peggiore, nel quale il nodo mendace invia valori diversi a tutti gli altri. Primo stadio:




Secondo stadio:




**Non è possibile ricostruire lo stato globale del sistema!** Non ci può essere quindi correzione dello stato, e in corrispondenza della variabile indefinita si porrà un simbolo di 'non definito'. Siamo cioè in grado di riscontrare l'errore, ma non di correggerlo. Osserviamo un'altra cosa: supponiamo che lo stato corretto di $P_3$ sia 7. Il nodo invia la seguente terna: 2, 2, 7 e cioè un solo valore esatto e due errori. Questo

caso è ancora più pernicioso del precedente, perché **il sistema ricostruisce uno stato errato**. Quindi: a conferma di quello che si diceva ieri, mentre l'algoritmo dei generali bizantini permette *sempre* di scoprire se manca l'agreement sullo stato globale, è possibile una correzione solo se il nodo mendace *commette un solo errore*. Inoltre, come si è visto, *il tentativo di correggere uno stato in assenza di agreement può essere rischioso*. È giusto osservare comunque che l'eventualità che l'errore si propaghi su due canali e che sia lo stesso per entrambi, come nel caso suggerito, è estremamente improbabile: ecco perché generalmente si opta per la correzione.

Abbiamo considerato i problemi di ordinamento a mezzo di logic clock, mutua esclusione, stallo (tecnica predittiva o di detection) e agreement su di uno stato noto a tutti i nodi in presenza di errori bizantini. Rimangono sostanzialmente tre problematiche: *elezione* (il coordinatore centrale può essere *eletto* dalla rete e dobbiamo cercare di capire come avviene questo), *atomicità* e *robustezza*.[10]

**Robustezza**. In un sistema distribuito uno dei nodi può subire un guasto hardware. I più comuni sono link failure, site failure o perdita di un messaggio. Un sistema robusto deve individuare il guasto, riconfigurare il sistema per permettere al resto della rete di continuare a funzionare e ripristinare il sistema quando il link o il site vengono riparati.

L'unico problema per cui esiste soluzione nota è quello di riscontrare la presenza di un guasto e si risolve inviando periodicamente dei messaggi ai processi. Il messaggio chiede in pratica al processo: "sei vivo?" (*alive*) e quest'ultimo può rispondere: "sono vivo" entro un certo tempo. Se si ha il time-out prima che arrivi la

---

[10] In seguito alla domanda di uno studente, il Prof. ha aperto a titolo di mera curiosità un lungo discorso su di una problematica, inerente sempre i sistemi distribuiti, non trattata nel corso (come è accaduto per la schedulazione) ma abbastanza significativa: la **migrazione**. Ne riporto gli aspetti salienti (non è argomento di esame).

La migrazione è il problema di spostare un processo P dalla macchina $M_1$ alla macchina $M_2$ (anche quando è stato già posto in esecuzione) perché sia eseguito su quest'ultima. I motivi possono essere vari: quello più tipico è il bilanciamento del carico. Nei kernel dei nodi collegati in rete devono essere previsti allo scopo degli *agenti mobili* che risolvano i complessi problemi sollevati da questa opportunità.

Alcuni di questi problemi sono facilmente intuibili. Ogni processo ha un contesto, formato da codice, memoria (incluse eventuali aree di stack utente) e stato (registri del processore, file aperti ecc.) e sono tre informazioni indispensabili alla sua esecuzione. Si tratta quindi di inviare un messaggio alla macchina $M_2$ contenente il contesto completo del processo migrante. In considerazione della grande mole di dati che occorre trasferire, è ovvio che tale procedura causa un grosso overhead di comunicazione. Bisogna quindi sapere se e quando conviene implementare la migrazione.

Si può essere costretti a imporre dei limiti alla migrazione dei processi nella rete. Ad esempio, se non c'è un NFS distribuito, si può precludere la migrazione ai processi che hanno aperto dei file (il processo non può vedere lo stesso file aperto nel nuovo contesto). Inoltre, il processo può essere spostato solo se usa indirizzi relativi anziché assoluti: si deve quindi implementare un meccanismo distribuito per la conversione fra le due tipologie di indirizzi. Anche questi aspetti devono essere tenuti in conto dagli agenti mobili.

Esempio di applicazione moderna: gli *applicativi su rete per i telefoni cellulari* non vengono eseguiti sul cellulare stesso (poca memoria) ma sul server di rete o sui sistemi (es. bancari) che comunicano col server. Dal cellulare parte solo un messaggio di richiesta con i dati necessari per l'autenticazione da parte del server. I processi quindi migrano fra il server e gli altri nodi fornendo ai cellulari il risultato della richiesta elaborazione. Durante ogni migrazione, il processo si arricchisce di una serie di informazioni inerenti per lo più la sicurezza.

risposta, il processo in questione viene dato per morto. Questo potrebbe far sorgere la necessità di far partire un algoritmo di elezione (vedi oltre).

Il sistema si presta purtroppo a vari inconvenienti. Il principale sta nel fatto che basilarmente è solo possibile accorgersi che c'è stato un guasto, senza poterne individuare il tipo. Ad esempio se in una rete si verifica un guasto in uno o più collegamenti, il messaggio *alive* può non avere la possibilità materiale di raggiungere tutti i nodi. Di conseguenza un processo può essere non guasto, ma piuttosto isolato (parzialmente o totalmente) dal resto della rete, e se non si adottano ulteriori meccanismi non si riesce a distinguere tra le due situazioni. Il modo più diffuso per ovviare a tale inconveniente consiste nell'utilizzare collegamenti *multipli* (ridondanti) fra i nodi.

**Elezione**. L'algoritmo di elezione viene solitamente avviato per eleggere un nodo coordinatore (in quelle applicazioni distribuite in cui ciò è previsto) e viene avviato in determinati frangenti: ad esempio in fase di boot, o quando si verifica un guasto come quelli di cui si è poc'anzi discusso.

Gli algoritmi di elezione determinano una [nuova] *istanza* del coordinatore e il sito in cui deve essere attivata. Il coordinatore gestisce le funzioni richieste dagli altri processi (mutua esclusione, costruzione del grafo wait-for globale e così via). Si può prevedere l'elezione, finalizzata a nominare un nuovo coordinatore, solo nelle reti *stateless* (senza stato, il che implica che *i nodi sono tutti uguali*) perché se è previsto uno stato globale e il coordinatore si rompe si perde memoria dello stato, e non c'è modo di proseguire oltre; oppure, se è previsto in uno stato globale, questo dev'essere salvato su di un supporto stabile di memoria (stiamo quindi parlando di un sistema a memoria condivisa con la quale lavorano molti nodi, e un nodo in particolare fa la funzione di coordinatore per l'accesso alla risorsa critica. La sua rottura non determina la perdita dello stato globale del sistema).

Assumiamo che ad ogni processo attivo $P_i$ sia associato un univoco indice i di priorità. Per semplicità, supponiamo che ci sia un unico processo su ogni sito (processore). **Il coordinatore è sempre il processo con il numero di priorità più elevato**. Quando il coordinatore fallisce, l'algoritmo deve eleggere come coordinatore il processo attivo con la priorità più alta.

Proponiamo due diversi algoritmi: bully ('spaccone') e ring (anello). Un ruolo importante in questo contesto è rivestito dal <u>partizionamento</u> della rete. Può succedere che se c'è un guasto da qualche parte (ad es. si rompe un link), una porzione della rete non riesce a raggiungere il coordinatore, che però è vivo e vegeto. In questo caso si finirebbe con l'eleggere un <u>secondo coordinatore</u> per quella porzione di rete. Questa è una situazione poco appropriata, e si cerca di limitarla il più possibile aumentando la connettività della rete, ovvero diminuendo il partizionamento.

1. **algoritmo bully**: il processo $P_i$ invia una richiesta al coordinatore, che però *non* gli risponde entro T secondi. In tal caso, $P_i$ tenta di eleggersi come nuovo coordinatore. Invia allora un messaggio di elezione a tutti gli altri processi che abbiamo *priorità più alta* e attende una risposta entro il tempo T.

Se non c'è risposta, il processo 'bully' assume che tutti i processi in questione sono falliti, e si autoproclama nuovo coordinatore. In particolare, esso manda a tutti i nodi con priorità inferiore un messaggio col quale notifica di essere il nuovo coordinatore. Si osservi come possa avere luogo l'anomalia del doppio coordinatore, come si diceva prima. In seguito, infatti, uno dei processi che era stato dato per morto da $P_i$ e che invece era attivo invia una richiesta al vecchio coordinatore che potrebbe essere anche lui, in realtà, ancora attivo (evidentemente i due processi non avevano risposto a $P_i$ perché c'era un link guasto).

Se $P_i$ invece riceve risposta, attende un tempo T' che un processo con maggiore priorità si elegga come nuovo coordinatore. Il processo $P_j$, maggiormente prioritario, che ha ricevuto da $P_i$ la richiesta di divenire coordinatore, invia un messaggio di ritorno a $P_i$ con il quale lo pone in attesa e propone esso stesso la propria candidatura esattamente come aveva fatto $P_i$. In presenza di link guasti, questo procedimento può impedire l'eventualità di doppio coordinatore. Può darsi infatti che $P_i$ sia collegato a $P_j$, e $P_j$ lo sia al resto della rete (incluso quindi il vecchio coordinatore) e in tal caso il vecchio coordinatore rimane tale. Se in kernel è abbastanza intelligente, si può instaurare un diverso *routing* (instradamento) dei messaggi da $P_i$ verso il vecchio coordinatore in modo da non reiterare l'errore. L'altra possibilità è che $P_j$ sia effettivamente eletto nuovo coordinatore; questo, per quanto è stato detto, non esclude che il vecchio coordinatore sia ancora in vita. Il partizionamento non può essere scongiurato in generale.

Se passa il tempo T' e non si ha risposta ($P_j$ potrebbe essere caduto nel frattempo), $P_i$ ripete l'algoritmo.

Mettiamoci ora nei panni di un processo $P_i$ che non sia né il coordinatore, né il processo che ne ha rilevato il fallimento. $P_i$ può ricevere uno fra due diversi messaggi da parte di un altro processo $P_j$.

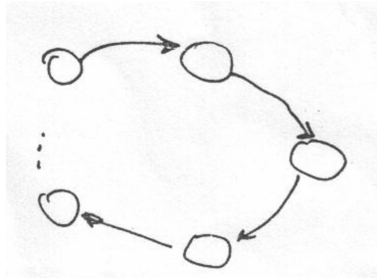
- $P_j$ è il nuovo coordinatore (il che può voler solo dire che è maggiormente prioritario di $P_i$). $P_i$ ne prende atto: invierà future richieste a $P_j$.
- $P_j$ ha fatto partire una elezione. Questo può voler solo dire che $P_i$ è maggiormente prioritario di lui, dato che un messaggio simile viene inviato ai processi più prioritari. In tal caso, $P_i$ blocca con un messaggio l'elezione di $P_j$ e fa partire un proprio algoritmo di elezione.

Quando un processo entra nel sistema, *ex novo* oppure dopo il proprio fallimento (ripartenza), esso 'indice le elezioni', e il risultato può essere che diviene il

coordinatore (se è più prioritario di quelli che erano prima presenti nella rete) oppure che viene a conoscenza di chi sia il coordinatore.

Requisito irrinunciabile di questo algoritmo è la corretta assegnazione delle priorità (i processi devono essere 'battezzati' man mano che entrano in un modo che risulti globalmente coerente. Questo viene realizzato mediante complessi protocolli di broadcast che devono sincronizzarsi con precisione).

2. **algoritmo ad anello**[11]: applicabile per link sia logici che fisici. Si assumono *unidirezionali* i link, e i processi possono ricevere messaggi dai processi alla loro sinistra, e inviarli a quella alla loro destra, come nella figura a sinistra. Ogni processo mantiene una lista di link attivi, consistente di tutti i processi attivi nel sistema con la relativa priorità.



Quando un processo $P_i$ si accorge del fallimento del coordinatore (non riesce a parlargli), crea una nuova lista attiva inizialmente vuota e manda in giro il messaggio-token elect(i) cominciando dal nodo alla sua destra. Aggiunge quindi i alla sua lista attiva.

Ogni nodo $P_i$ che si vede recapitare un 'certificato elettorale' elect(j) dalla propria sinistra può reagire in tre modi differenti:

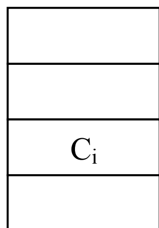
- se quello ricevuto [o inviato] è il <u>primo</u> token elettorale, crea una nuova lista attiva con i numeri i e j e invia in successione i messaggi elect(i) ed elect(j);
- se $i \neq j$, $P_i$ aggiunge j alla propria lista attiva e ritrasmette il messaggio al suo vicino di destra;
- se $i = j$, significa che $P_i$ riceve il messaggio che lui stesso aveva originariamente spedito. A questo punto la sua lista attiva contiene i numeri di tutti i processi attivi del sistema. $P_i$ ora è in grado di determinare il numero più elevato nella lista attiva per identificare il nuovo coordinatore.

Un processo che, reduce da un fallimento, riparte, deve ovviamente sapere chi è il coordinatore, e può farlo mandando un messaggio di interrogazione lungo l'anello. Il coordinatore attuale risponderà, identificandosi.

**MAZZOCCA 07/05/02**

**ATOMICITÀ**. Pensiamo ad un sistema client-server, con un solo server e più client. È lecito chiedersi se le operazioni richieste in un sistema del genere <u>si sono concluse o meno</u>, e, nel caso in cui non siano state concluse, <u>ritenere mai fatte quelle operazioni</u>. Alla base delle tecniche che trattano questa problematica c'è ovviamente un'operazione di roll-back.

Indichiamo con $C_i$ il coordinatore delle transazioni e con $S_i$ il generico sito-client; a seguito dell'attivazione del manager, $S_i$ deve passare da uno stato $S^*_i$ ad uno stato $S^{*\prime}_i$. Quest'ultimo è effettivamente il nuovo stato raggiunto dalla macchina se tutti i processi $S_k$ coinvolti nell'esecuzione di quell'azione atomica hanno chiuso in modo corretto il proprio sistema. Tutti migrano cioè verso i rispetti stati ' ("primo") e lo stato $S^*_i$ non è più significativo. Però è sufficiente che anche uno solo di tali processi fallisca, perché sia necessario tornare allo stato precedente $S^*_i$. È necessario disporre di un *log* di sistema che permetta di congelare lo stato di tutti i processi coinvolti, così da poter tornare indietro.




**OK**

L'idea basilare consiste nel creare una lista di processi attivi. Man mano che i processi terminano, comunicano l'avvenuta operazione al sistema C. C viene a sapere ad esempio che il processo $S_i$ ha terminato la propria elaborazione. C, tuttavia, non può dare l'OK alla conservazione dell'informazione elaborata da parte di $S_i$ se non dopo aver saputo che tutti gli altri processi hanno terminato in modo corretto.

Il protocollo si svolge cioè in due fasi: nella prima, C attiva $S_i$ e quest'ultimo comunica ad C l'avvenuta operazione. Nella seconda, C comunica a tutti che l'operazione è stata completamente eseguita (*committed*) e quindi si può cambiare stato, acquisendo come nuovo stato $S^{*\prime}$.[12]

Dal punto di vista logico, che siano necessarie due fasi anziché una sola deriva dal fatto che è impossibile stabilire un *agreement* fra due sole unità. Inoltre, quando si ha un handshaking chi conosce lo stato dell'altra unità è chi ha attivato l'handshaking. Nel nostro caso, chi attiva i vari client è il server e quindi è C che conosce lo stato dell'intera rete, informazione che però deve essere resa disponibile anche ai client. Questo avviene nel modo di seguito indicato. La risposta del client ('ho eseguito la mia operazione') diventa in pratica una nuova richiesta di handshaking a cui il server risponde con un segnale di commit.

Tale semplice protocollo comporta diversi problemi che meritano attenzione. Innanzitutto, <u>si può rompere il client quando ha già comunicato di aver eseguito la propria operazione</u>. In tal caso il server invia il segnale di commit senza aspettare altre risposte dal client: dal suo punto di vista, la transazione è chiusa. Tuttavia, quando il client ritorna, funzionante, nel sistema, trova la transazione ancora aperta; può allora richiedere al server lo stato della transazione per poterla chiudere da parte sua. Il server avrà infatti congelato lo stato globale del sistema, ottenuto grazie alle informazioni che gli sono state trasmesse da tutti i client.

Mentre un client può senz'altro gettare via il suo log quando gli perviene il commit da parte del server, il server non può mai farlo, perché se nel frattempo

qualche client muore, quando il client in questione si riattiva gli chiederà lo stato; il server deve conoscere una 'storia più lunga' rispetto ai client.

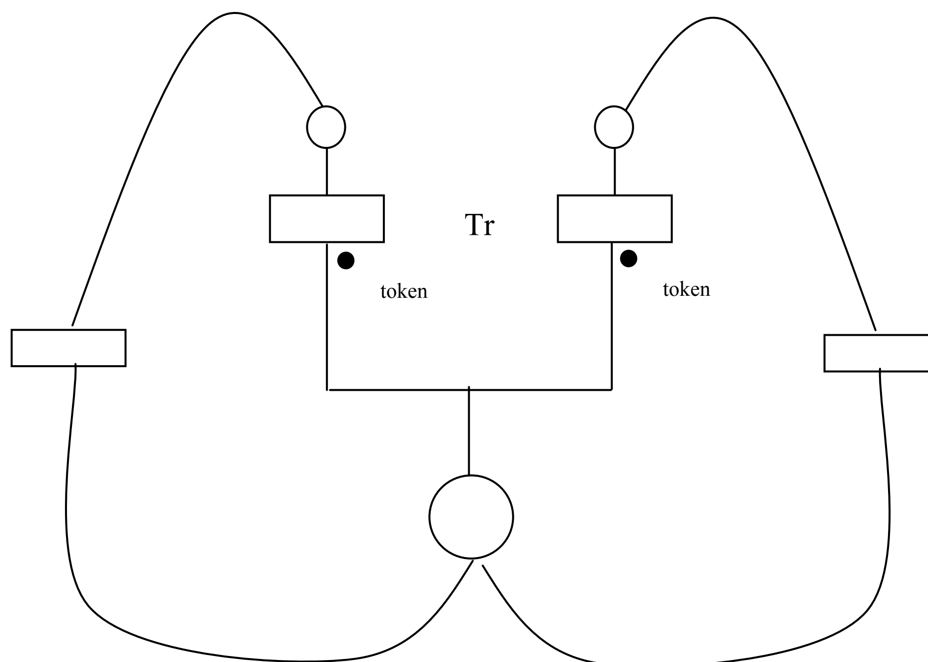
Più semplice è il caso in cui un client si rompa quando non ha ancora dato l'OK della propria operazione. In tal caso, lo stato globale non è stato ancora raggiunto; il server può fermare tutti gli altri client, riportandoli in $S^*_i$.

Il caso in cui si rompa il server non può essere gestito, poiché la rete prevede uno stato globale che è memorizzato proprio in C. Il server avrà un certo grado di *affidabilità* ed un certo grado di *disponibilità*. Ricordiamo che la disponibilità è una misura di frequenza (sistema attivo in un istante t), l'affidabilità (o sicurezza) di un intervallo temporale (sistema attivo in un certo intervallo). Per poter misurare questi dati, dobbiamo eseguire delle operazioni di campionamento.

Il sistema C sarà stato costruito in base ad un certo indice di qualità (tutte le sottoparti componenti del sistema devono essere affidabili). L'affidabilità delle componenti è caratterizzata grazie alla definizione di un modello, oppure viene misurata iniettando dei guasti nel sistema e monitorandone la reazione in esercizio (tecnica del *dizionario dei guasti*). Nel caso del ***modello***, si ha basilarmente che i messaggi scambiati tra i vari nodi subiscono dei ritardi o dei guasti dovuti alla rete di interconnessione. La rete può così subire un *guasto transiente*, come la corruzione di un messaggio. Possiamo caratterizzare tale guasto mediante una *funzione di guasto*: ad esempio, la frequenza di occorrenza di tale guasto (uniformemente distribuita) sia $f = 10^{-8}$. C'è da dire tuttavia che il modello matematico diviene sempre più complicato al crescere degli elementi che vengono ad essere perturbati. Ad esempio, la radiazione elettromagnetica che può essere alla base del guasto della rete può coinvolgere anche il SO di un nodo: la distribuzione totale di probabilità non è facile da calcolare, neanche conoscendo la distribuzione dei singoli componenti (a meno che gli eventi non siano scorrelati).

La matematica quindi viene incontro solo fino ad un certo punto; non può fornire modelli accurati in contesti abbastanza complicati. In casi come questi, si può a volte ricorrere alla ***simulazione***. Si sviluppa cioè una piattaforma simulativa in cui si iniettano deliberatamente dei guasti. Comprensibilmente, questa alternativa va incontro a difficoltà di ordine progettuale (la simulazione deve essere molto accurata e non è facile garantirlo), e inoltre c'è il problema tempo. La misurazione di ciascun evento della simulazione può costare un sacco di tempo, in quanto, come è noto, per disporre di un buon modello probabilistico occorre un elevato numero di prove sperimentali (teoricamente infinito. Per dimostrare che con un dado non truccato la probabilità che esca un numero è di 1/6, occorrerebbero infinite prove). Ad esempio, se iniettiamo un guasto tipo corruzione di un messaggio, non è detto che si abbia un fault: il guasto può essere corrotto in una zona inessenziale, e sottoponendolo al sistema si ha comunque l'OK. Il sistema dovrebbe quindi girare per ore per poter dare un modello matematico degno di fiducia.

Infine, neanche l'approccio simulativo rende invulnerabili alla complessità della realtà con cui si tratta. Ci si può cioè trovare nella condizione di dover analizzare troppi eventi. Esista una terza possibilità: quella dei ***metodi formali***. Il sistema può per esempio essere descritto con una *rete di Petri*, uno strumento apposito per la rappresentazione degli eventi. Si noti la figura sottostante, che propone una semplice rete di Petri. Due nodi elaborativi si trovano in un certo stato. Eseguire una transizione (che avviene nel tempo Tr) significa portarli in un nuovo stato. Una comunicazione sincrona può quindi essere descritta come una transizione in un nuovo stato-evento (il pallino grosso). Tale evento è possibile solo se le predette transizioni sono abilitate da un *token* ciascuna. L'evento comunicazione, una volta avvenuto, abilita a sua volta l'evento chiusura della comunicazione e si ritorna da capo.

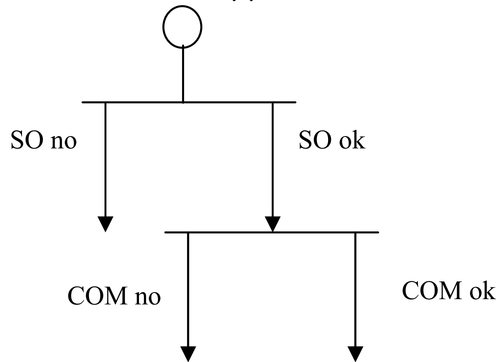



Poiché allora questo strumento consente di modellare eventi e tempi, permetterebbe di rappresentare il sistema client-server di cui si parlava prima. Il vantaggio ulteriore è che usando questa rappresentazione (e purché la distribuzione degli eventi sia *uniforme*) si arriva ad una CATENA DI MARKOV. Se la catena è risolvibile (non troppo grande), si può determinare la distribuzione stazionaria di probabilità, ovvero la probabilità che un token si trovi in una determinata posizione. Attribuendo dei pesi alle varie transizioni, è quindi possibile determinare l'evoluzione del sistema attraverso i vari stati (ad esempio, che probabilità che si ha di stare in uno stato di errore).

Il metodo simulativo dunque lascia a desiderare perché non consente di tirare conclusioni generali, a meno di eseguire un notevolissimo numero di esperimenti. Si ricorre allora a taluni metodi matematici che in particolari condizioni, e sempre che lo spazio degli eventi sia contenuto, permettono di trovare una buona

rappresentazione analitica, e infatti importanti aziende come l'Ansaldo usano correntemente le reti di Petri.

Riportiamo di seguito un possibile inizio di rappresentazione mediante Reti di Petri del problema della simulazione di errori nella trasmissione su di un canale. Un Sistema Operativo può commettere un errore nell'attivare un canale di comunicazione, oppure non commetterlo. La comunicazione a sua volta può avere esito positivo o negativo. Il SO viene cioè modellato non in tutti gli eventi possibili e immaginabili, ma solo nei riguardi di una particolare azione. Possiamo iniettare dei guasti e studiare il comportamento conseguente del sistema, modellato statisticamente, in un approccio che combina il metodo simulativo con quello descrittivo ed è perciò detto 'ibrido'. Le transizioni hanno una certa probabilità di abilitare un ramo o un altro; la struttura che ne risulta è quella tipica di un albero, o più in generale, essendo possibile tornare indietro in particolari transizioni, quella di un grafo.




Esempio applicativo: vogliamo simulare un'applicazione di calcolo nell'ambiente PVM. È dato un programma che deve girare su 100 nodi, ovviamente senza avere a disposizioni l'effettiva rete di 100 nodi. Vogliamo stimare il tempo complessivo di elaborazione. Quello che è soprattutto importante rappresentare è lo scambio dei messaggi, cosa che può essere tranquillamente fatta su di un solo nodo. Lo scambio dei messaggi dipende dalla tipologia di calcolo (es.: ci sarà una modalità di scambio dei messaggi per il prodotto di matrici). Considerato allora il processo PVM $P_i$, quest'ultimo eseguirà sessioni alternate di calcolo e comunicazione. Il tempo impiegato dal calcolo può essere stimato, mentre la comunicazione dipende unicamente dal sistema.

Quindi quello che ci serve non è simulare l'intero algoritmo PVM, ma piuttosto basta considerare un insieme di *tracce di esecuzione*, date da un tempo di calcolo $t_c$ e una richiesta di comunicazione $R_{com}$. Il simulatore sarà quindi un insieme di task, ciascuno dei quali lascia trascorrere un certo tempo fissato, simulando così il tempo di calcolo, e lancia dei messaggi in una rete. Si danno in ingresso al simulatore i tempi di comunicazione e le tracce di calcolo; le tracce vengono analizzate in parallelo e se ne ricava il tempo totale di elaborazione. Per migliorare la precisione, conviene aumentare il tempo di calcolo rispetto a quello di comunicazione, che è più difficile da stimare[13].

In applicazioni in cui il calcolo dipende dall'allocazione sui singoli nodi (es.: visita di un albero), tale approccio evidentemente non va più bene. L'elaborazione deve

cioè essere strutturata in maniera deterministica, come avviene nel caso del prodotto delle matrici.

Inoltre, con riferimento all'esempio dell'attivazione del canale, evidentemente abbiamo simulato solo errori che richiedono la ritrasmissione di un messaggio, e non quelli per i quali il messaggio vada perso, che necessiterebbero di un modello più sofisticato. Il modello quindi è adatto soltanto ad esempio per rappresentare reti congestionate, ma diversamente dovremmo in effetti rappresentare la rete a tutti i livelli, demon PVM, SO, socket etc., in un panorama decisamente più complesso. Quando il modello simulativo risulta troppo semplificato rispetto alla realtà, che è composta da una combinazione molto complessa di eventi non scorrelati, possiamo servircene per effettuare almeno delle 'stime peggiorative' della realtà stessa, dopo aver dimostrato la correttezza di tale procedimento.

Torniamo ora al problema vero e proprio dell'atomicità. Tutte le operazioni associate ad un'unità di programmazione o vengono eseguite o completate, o non sono mai state fatte. È richiesta la presenza di un **coordinatore di transazioni**[14], che è responsabile delle seguenti azioni: fa partire l'esecuzione della transazione attiva, scompone ciascuna transazione in diverse sottotransazioni, le distribuisce tra i vari siti (quelli che abbiamo chiamato 'client', i $S_i$), e può ordinare l'interruzione di ciascuna transazione, che può così solo risultare o fatta da tutti i siti, o abortita da tutti i siti.

Il protocollo più usato è naturalmente in **two-phase commit protocol**. L'esecuzione del protocollo a due fasi è iniziata dal coordinatore. Bisogna aver completato tutte le transazioni precedenti prima di iniziare delle nuove transazioni, che vengono completate solo quando tutti i siti hanno dato risposta. Quando il protocollo è inizializzato, la transazione dev'essere ancora eseguita sui siti locali. Il protocollo coinvolge tutti i siti che stanno eseguendo la transazione.

Nella prima fase il gestore $C_i$ inizia la transazione $T_s$, inviando i relativi messaggi ai siti, i processi $S_i$. Il Transation Manager sul sito S, il modulo che gestisce localmente le transazioni, determina se la transazione può essere eseguita. Se no, invia un messaggio NO_T al coordinatore, il quale gli risponde a sua volta con un acknowledgement di tipo ABORT_T.

Se la può fare, aggiunge un record T al proprio log e *forza il record (così come tutti gli altri che caratterizzano lo stato del sistema) in una stable memory*. Il Transition Manager invia al coordinatore un segnale di *ready*. Si ha quindi che sia il coordinatore che ciascun sito hanno un proprio log; quello del coordinatore tiene conto di tutte le transizioni attivate, quello del singolo sito memorizza lo stato del sistema.

Il coordinatore raccoglie tutte le risposte. Quando gli pervengono tutte le risposte *ready*, la transizione può essere 'committed' e viene inviato a tutti il relativo messaggio (seconda fase). Altrimenti invia a tutti un messaggio di abort,

---

messaggio che può arrivare anche se almeno un sito non risponde all'interno del time-out predefinito. In caso di commit, si può essere sicuri dell'agreement distribuito e ciascun sito svolge le proprie azioni in modo appropriato (e, per inciso, non è tenuto a svolgerle immediatamente). L'evento commit viene pure memorizzato in ciascuna stable memory ed è irrevocabile.

L'utilizzo di una tecnica ad agenti mobili risulterebbe vantaggiosa per il fatto che un agente mobile, migrando, porta con sé il codice, ma non può portare il log né i dati. All'atto pratico si preferisce seguire una filosofia di replicazione dei dati, distribuendo copie dei dati su alcuni punti della rete in modo che se si verifica un fault in un sito le informazioni in esso contenute non vadano irrimediabilmente perdute (si pensi al contesto bancario); comunque, per ragioni di coerenza e di altra natura si tende a limitare i punti di concentrazione dei dati.