UNIVERSITÀ DEGLI STUDI DI MODENA

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Implementazione di modelli di replica per sistemi di gestione di basi di dati distribuite

Relatore:
Prof. Sonia Bergamaschi

Candidato:
Andrea Bisi

Anno accademico 1995/96

Indice

Introduzione

L' interesse nella gestione distribuita delle informazioni ha avuto un notevole incremento negli ultimi anni, dovuto soprattutto alla crescente delocalizzazione delle attività produttive e di ricerca, unita all' evoluzione tecnologica delle telecomunicazioni che consente una maggiore velocità di comunicazione a costi sempre decrescenti. Questo ha favorito la connessione di più siti remoti, ciascuno di questi sede di informazioni che devono essere reperibili anche dalle altre entità della struttura.

I database tradizionali hanno la caratteristica di aver spostato il controllo dei dati dalle applicazioni locali (in cui ogni applicazione controlla e mantiene i propri dati) ad una gestione di tipo centralizzato, nella quale l' applicazione locale è immune alle variazioni della struttura e dell' organizzazione dei dati (*indipendenza dei dati*). Queste variazioni possono essere compiute unicamente dal database centrale e portano a quella che si chiama *integrazione*, cioè i dati di tutte le entità della struttura vengono raccolti in modo centrale e quindi messi a disposizione di tutti gli utenti autorizzati. In questo modo, il controllo degli accessi viene svolto unicamente nel database centrale assicurando quindi un buon livello di sicurezza. Inoltre viene garantita l' integrità e la correttezza di dati che sono la risorsa fondamentale per tutte le entità coinvolte in una data struttura. Il sito in cui risiede il database centrale è il *server* che fornisce ai *client* tutti i servizi richiesti: sia funzioni che dati.

L' obiettivo però è l' integrazione, non la centralizzazione, e questo è possibile utilizzando i database distribuiti.

L' evoluzione dal modello *client/server* al modello distribuito è la risposta all' esigenza sempre maggiore di avere le informazioni "vicine" al luogo in cui si utilizzano, ma contemporaneamente accessibili a tutti. Grazie alla diminuzione dei costi e all' aumento delle prestazioni, è diventato economicamente accettabile lo scambio di consistenti quantità di dati, consentendo la rapida diffusione del modello distribuito.

D' altra parte però il trasferimento di elevate quantità di informazioni è ancora molto oneroso ed è quindi necessaria una considerevole quantità di dati replicati in architetture di gestione di basi di dati distribuite.

La distribuzione dei dati e delle applicazioni ha numerosi vantaggi:

- l' autonomia locale del possessore delle informazioni, che rimangono nel sito in cui sono state create;
- l' incremento delle performance globali delle applicazioni, che accedono ai dati sempre in modo locale (grazie al meccanismo della replica);
- maggiore affidabilità in caso di guasti ai server, infatti ci sono repliche dei dati reperibili anche su altri server;
- una maggiore espandibiltà: è sufficiente aggiungere siti alla rete per avere nuove funzioni e/o nuovi dati su tutti gli altri server.

Il crescente interesse verso la gestione dei dati distribuita non è passato inosservato da parte delle maggiori software house produttrici di DBMS, infatti sono stati introdotti sul mercato in tempi relativamente brevi (e con caratteristiche simili) nuove versioni che consentono la gestione di basi di dati distribuite. Ancora mancano però degli strumenti efficaci e flessibili per la gestione delle repliche.

In questa tesi verranno introdotti alcuni nuovi modelli di replica: static ownership e dynamic ownership. Verrà inoltre implementato il modello più completo (dynamic ownership) mediante la realizzazione di un programma che permette di generare automaticamente i file (dati e programmi SQL) necessari per la gestione della replica. Il modello static ownership verrımplementato come caso particolare del modello precedente.

L' implementazione proposta è prototipale. Un' attività è molto importante, cioè il processo di generazione delle repliche è interamente automatizzato e non richiede interventi da parte dell' utente. Dall' altro lato, la preparazione dei dati richiede però l' interazione con l' utente. Un possibile sviluppo futuro è rappresentato dall' automazione dell' acquisizione della struttura dei database su cui si implementa il modello di replica. Inoltre lo sviluppo di una interfaccia utente maggiormente user-friendly sarebbe auspicabile.

Il contenuto della tesi si articola nel modo seguente: nel primo capitolo

viene fornita una introduzione alle reti di telecomunicazioni ed alle architetture distribuite; nel secondo capitolo vengono descritte le funzionalità dei sistemi di gestione di basi di dati distribuiti (DDBMS). Il terzo capitolo esamina i DDBMS commerciali più diffusi, illustrandone il meccanismo di replica. Nel quarto capitolo introduciamo i nuovi modelli di replica: static ownership e dynamic ownership. Il quinto capitolo è dedicato ad una analisi del sistema software utilizzato per l' implementazione (Microsoft SQL Server), descrivendo in dettaglio i moduli che controllano la replica. Infine nel sesto capitolo è illustrata l' implementazione dei modelli di replica introdotti nel quarto capitolo.

Capitolo 1

Reti di telecomunicazioni e architetture distribuite

In generale possiamo definire una rete di computer come un insieme *interconnesso* di computer *autonomi*. Due computer sono interconnessi se sono in grado di scambiarsi informazioni; il requisito di computer autonomi serve per escludere dalla definizione i sistemi master/slave, dove esiste chiara dipendenza (controllo) di un computer da un' altro.

C' è una distinzione fondamentale tra "rete di computer" e "sistema distribuito": in quest' ultimo l' esistenza di computer autonomi è trasparente all' utente, il quale esegue programmi ed utilizza dati senza preoccuparsi della loro posizione fisica. Con una rete l' utente deve esplicitamente collegarsi ad una macchina, e provvedere personalmente alla gestione dei dati e delle applicazioni.

In realtà un sistema distribuito è un caso particolare di rete: il software di gestione (sistema operativo o DDBMS) si occupa delle connessioni e del trasferimento dei file senza nessuna interazione da parte dell' utente. La differenza sostanziale quindi è solo in "chi" richiede il trasferimento: il sistema o l' utente.

Quella che segue è una classificazione delle reti in base alla loro estensione fisica:

Distanza interprocessore	Elaboratori situati in	Esempio
0.1 m	Piastra circuitale	Macchina di flusso
1 m	Sistema	Multiprocessore
10 m - 1 km	Edificio / Quartiere	Rete locale (LAN)
10 km - 100 km	Città / Nazione	Rete geografica (WAN)
1000 km - 10000 km	Continente / Pianeta	Interconnessione di WAN

1.1 Struttura di una rete

Una rete dal punto di vista tecnico è la connessione di diversi *host* ad una sottorete che si occupa del trasporto dei messaggi, possiamo così separare gli aspetti che riguardano l' applicazione (sugli host) dalla comunicazione pura della rete (la sottorete).

La sottorete può essere:

Point-to-point La rete è piuttosto complessa, vi sono dei nodi di commutazione connessi a coppie attraverso il mezzo fisico (filo di rame, coassiale, fibra ottica, ...), quando un *pacchetto* (messaggio) viene inviato tra due nodi distanti, questo passa attraverso i nodi intermedi sul percorso libero tra l' origine e la destinazione. Esempi di questo tipo di reti sono in figura 1.1.

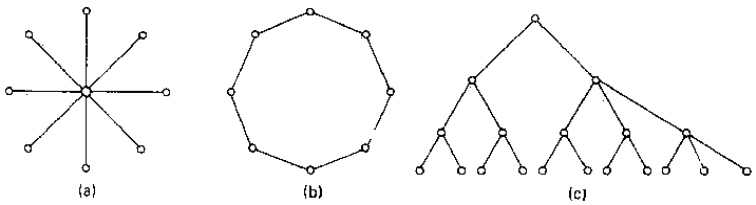


Figura 1.1. Esempi di topologie point-to-point

Broadcast Vi è un unico trasmettitore (che può variare nel tempo) e un singolo canale di comunicazione, e tutti gli altri nodi devono così ricevere tutti i messaggi inviati, all' interno del pacchetto è specificato l' indirizzo del destinatario (uno o più), i nodi non destinatari ignorano il pacchetto. E' possibile anche specificare sottogruppi di destinatari (*multicasting*). Esempi di reti di questo tipo sono in figura 1.2.

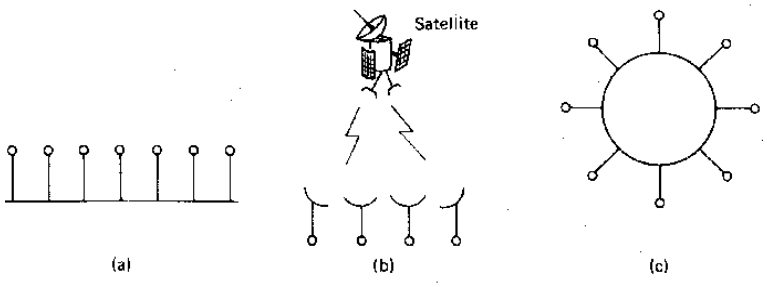


Figura 1.2. Esempi di broadcasting

1.2 Modello ISO/OSI

Gli aspetti fondamentali della comunicazione in rete possono essere suddivisi in vari strati (o livelli) ciascuno costruito sopra il suo predecessore. Ogni strato n comunica con quello immediatamente precedente $(n-1)$ ed immediatamente successivo $(n+1)$, questo tipo di comunicazione deve avvenire attraverso un' interfaccia che definisce le operazioni ed i servizi offerti dallo strato inferiore a quello superiore. Questa tecnica permette di nascondere i dettagli implementativi di ogni strato, ma rendendo nota solo l' interfaccia in modo tale da poter avere le stesse funzioni indipendentemente dall' implementazione di ogni strato, così da permettere l' eterogeneità del sistema. Ad esempio (come vedremo) racchiudendo in uno strato il mezzo fisico di comunicazione, e rendendo disponibile una interfaccia standard di I/O, possiamo scrivere uno strato superiore che risulterà indipendente dal mezzo (sia esso file di rame o fibra ottica o altro ancora).

Gli strati dovranno implementare, oltre al trasporto del segnale sul mezzo fisico, anche i protocolli di comunicazione (simplex, half-duplex, full-duplex), controllo e/o correzione degli errori, gestione dei pacchetti.

Il modello proposto dall' ISO (International Standards Organization) come primo passo verso la standardizzazione dei protocolli è denominato OSI (*Open Systems Interconnection*). E' basato su sette strati (si veda la figura 1.3) in modo tale che:

- Deve essere creato uno strato dove è richiesto un differente livello di astrazione,

- Ogni strato deve svolgere una funzione ben definita,

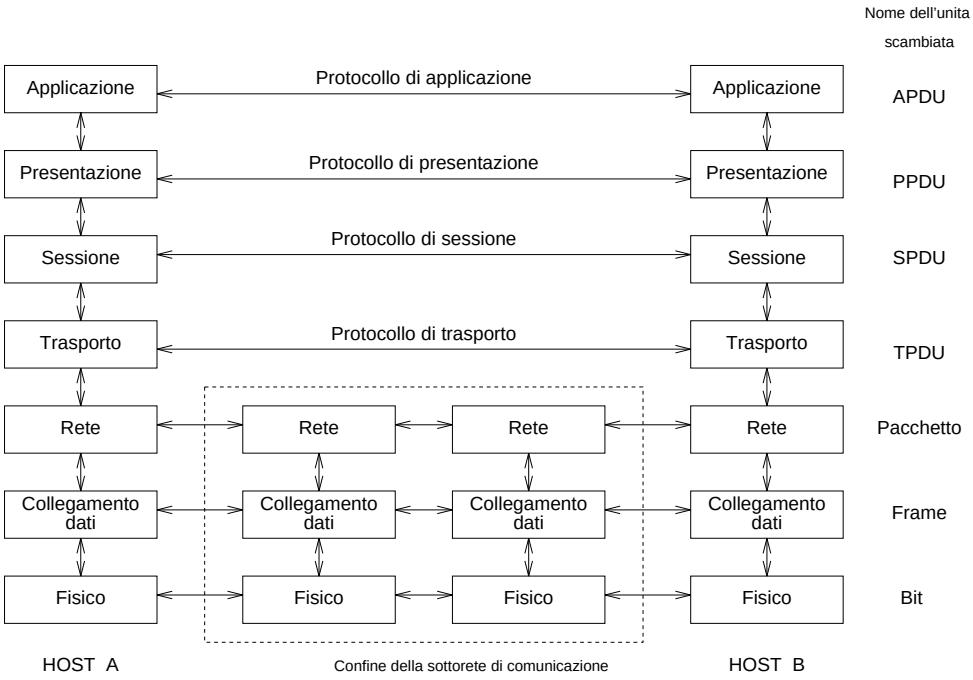


Figura 1.3. Modello di riferimento ISO/OSI

- La funzione di ogni strato deve essere scelta in base ai protocolli standard,
- i confini tra gli strati devono essere tali da minimizzare il flusso di informazioni tra le interfacce,
- numero di strati tale da non permettere il raggruppamento di funzioni distinte nel medesimo strato, ma tale da non appesantire l' architettura.

Vediamo una breve descrizione degli strati secondo il modello OSI:

Physical layer. Riguarda la trasmissione fisica di bit sul canale di comunicazione. Riceve un pacchetto dallo strato superiore, e lo invia bit per bit attraverso il mezzo fisico. Analogamente il ricevitore acquisisce i segnali, li trasforma in valori logici e li trasmette al proprio strato superiore.

Data link layer. Svolge i compiti di ricostruzione dei dati in un frame, e della correzione degli errori nei dati: sia errori di comunicazione (risolvibili con gli appositi algoritmi) che "logici" ad esempio frame che arrivano fuori sequenza.

Network layer. Controlla il funzionamento della sottorete, quindi si occupa dell' instradamento dei pacchetti e di risolvere i problemi dovuti alle reti eterogenee. Nel caso di reti broadcast lo strato di rete è molto semplice o addirittura inesistente.

Transport layer. Deve isolare lo strato superiore dall' implementazione hardware. Si occupa della suddivisione dei messaggi in pacchetti di dimensione minore e deve garantire che la trasmissione avvenga correttamente.

Session layer. Permette la definizione di "sessioni" tra gli utenti di macchine diverse, queste sessioni possono (oltre a trasportare i dati) fornire servizi aggiuntivi come la gestione dei token o la sincronizzazione.

Presentation layer. Esegue le procedure utilizzate molto spesso, e tali da richiedere uno strato dedicato. Un esempio è la codifica dei dati: spedire le stringhe usando il codice ASCII, i numeri interi usando un determinato numero di bit, i numeri floating point attraverso una codifica IEEE 754. In questo modo si assicura che il ricevente interpreti in modo corretto le informazioni contenute nei pacchetti.

Application layer. Contiene i vari protocolli di emulazione di terminale, codici escape, sequenze di controllo, ecc .... Inoltre si occupa del trasferimento di files, e-mail e lavori remoti.

Solo i primi tre strati (verso il mezzo fisico) sono definiti precisamente attraverso standard internazionali.

1.3 LAN

Le LAN (*Local Area Network*) come già detto (tabella 1) sono reti di estensione contenuta, tipicamente a pacchetto e che offrono buone prestazioni in termini di velocità di comunicazione.

Vi sono diversi standard per le LAN, complessivamente noti come IEEE 802, i vari standard differiscono nello strato fisico e nel sottostrato di accesso al mezzo trasmissivo, ma sono compatibili nel data link layer.

1.3.1 IEEE 802.3 e Ethernet

Hanno una topologia a bus (figura 1.4), ed utilizzano un unico cavo su cui trasmettono i pacchetti. Per evitare le collisioni (due trasmettitori che iniziano a trasmettere assieme) si ricorre ad un protocollo CSMA/CD 1-p (*carrier sense multiple access with collision detection 1 persistent*). Quando una stazione intende trasmettere "ascolta" il cavo, se è occupato aspetta finché non lo trova libero, quindi inizia a trasmettere. Se due o più stazioni iniziano a trasmettere contemporaneamente si verifica una collisione, questo provoca l' immediata sospensione delle trasmissioni, poi dopo un periodo di tempo casuale ripeteranno l' intero processo.

Il termine "Ethernet" si riferisce ad un prodotto specifico che implementa lo standard 802.3, esso funziona a 10Mbps (nella versione base) o a 100Mbps (versione fast).

Questi tipi di rete sono tra i più utilizzati nella realizzazione di reti locali non eccessivamente grandi, non multimediali e non real time.

I mezzi fisici utilizzati sono usualmente il cavo coassiale o il doppino, e nel secondo caso occorre la presenza di un HUB (concentratore).

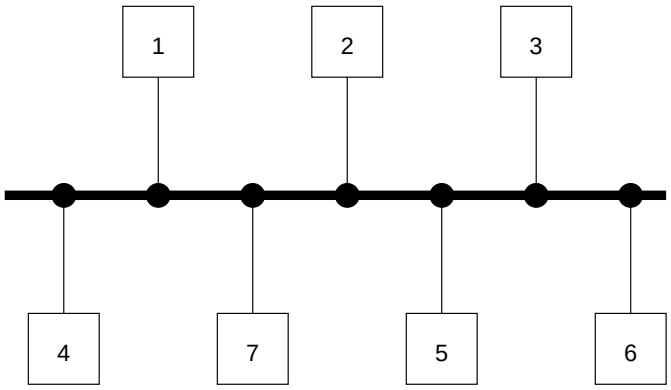


Figura 1.4. Topologia a bus

1.3.2 IEEE 802.4 Token bus

E' un' altro tipo di rete a bus, utilizzato quasi esclusivamente in sistemi real time non ha interesse per applicazioni gestionali.

1.3.3 IEEE 802.5 Token ring

E' una rete con topologia ad anello unidirezionale (figura 1.5), in realtà non è un vero e proprio anello, bensìuna sequenza di collegamenti point-to-point tra stazioni successive, il mezzo fisico è un economico doppino intrecciato.

Le stazioni hanno anche la funzione di ripetitore: ricevono un messaggio verificano l' header il quale contiene l' identificativo delle stazioni destinatarie, lo copiano se è per quella stazione altrimenti ritrasmettono il pacchetto così come lo hanno ricevuto. Una stazione destinataria dopo aver copiato il pacchetto lo ritrasmette con un flag "ricevuto" settato, il trasmettitore riceverà il messaggio, esamina il flag e vede se il proprio messaggio è stato ricevuto o no.

Per trasmettere una stazione dopo aver ricevuto il "token" cioè un messaggio con solo l' header (privo di dati), può accodarvi il messaggio azzerare il flag "ricevuto" e spedire.

Un singolo token in una rete pone grossi limiti, in quanto possiamo spedire un solo messaggio alla volta, pertanto possiamo far circolare più di un token, in questo modo anche se un token risulta occupato potrebbe esserci il successivo libero, possiamo così effettuare due trasmissioni senza aspettare tutto il tempo di giro di un token.

Questo tipo di rete è particolarmente facile da gestire, inoltre ha limite superiore noto per l' accesso al canale, questi motivi hanno spinto l' IBM a sceglierla come topologia per la sua LAN. Queste reti però sono abbastanza lente (circa 4Mbps) con supporti in rame.

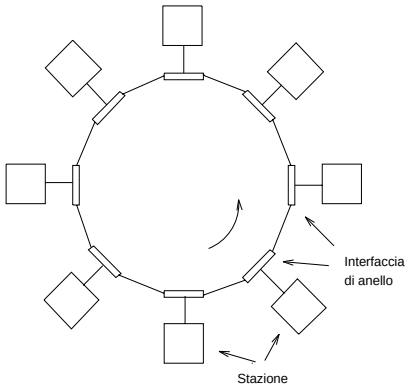


Figura 1.5. Topologia token ring

1.3.4 FDDI

La *Fiber distributed data interface* consente di realizzare LAN particolarmente veloci (oltre 100Mbps) e di dimensione abbastanza ampia (circa 200 km) con oltre 1000 stazioni collegabili, tali da permetterne un utilizzo anche per la realizzazione di MAN (reti metropolitane).

E' basata sul protocollo token ring, con una variante sostanziale: invece di aspettare che il token torni alla stazione trasmittente (il tempo sprecato potrebbe essere notevole), il trasmettitore dopo aver spedito il token col messaggio immette nell' anello un nuovo token vuoto. In questo modo parecchi frame possono essere spediti da più stazioni contemporaneamente.

1.4 WAN broadcast

Le comunicazioni broadcast nel caso delle WAN avvengono soprattutto tramite satellite (figura 1.2b), i quali hanno diversi trasponder, ciascuno di questi copre un' area della superficie terrestre. Da terra si trasmettono i frame alla frequenza di uplink (salita), il satellite converte i frame alla frequenza di downlink (discesa) e trasmette verso la terra, con due frequenze si evitano interferenze tra i frame in salita e quelli in discesa.

Per le comunicazioni point-to-point terrestri, vengono utilizzate le interconnessioni di reti geografiche (di solito nazionali), esempi sono X.25, DATAPAC, INTERNET (ARPANET).

1.5 Architetture client/server

Il paradigma *client/server* è un modello di interazione tra processi software, ove i processi interagenti si suddividono tra *client*, che richiedono servizi, e *server* che offrono servizi. L' interazione client - server richiede la precisa definizione di una *interfaccia* di servizi, che elenca i servizi messi a disposizione dal server.

Il processo client tipicamente si occupa di interagire con l' utente, svolge un ruolo "attivo" in quanto genera autonomamente richieste di servizi. Il server è "reattivo" cioè agisce solo su richiesta precisa da parte di un client. Un client indirizza le proprie richieste verso un singolo server, ogni richiesta appartiene ad una stessa transazione inizializzata alla prima richiesta fatta dal client.

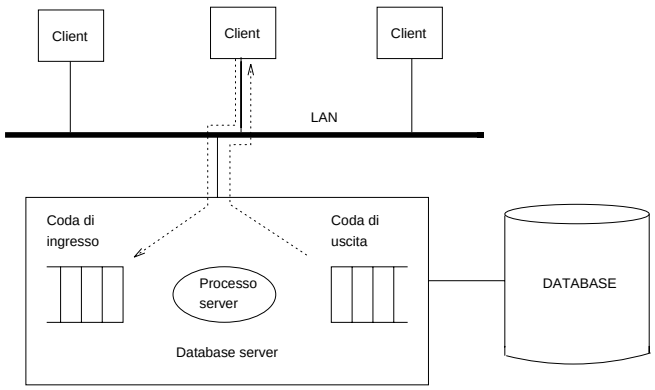


Figura 1.6. Architettura client/server

Vari motivi spingono verso l'uso di architetture client/server per basi di dati:

- Le funzioni di client e server sono ben identificate nel contesto delle basi di dati. Esse corrispondono ad una decomposizione ideale delle competenze e professionalità: il programmatore applicativo ha la responsabilità di gestire il software relativo al client rispondendo a esigenze specifiche, mentre l' amministratore della base di dati ha la responsabilità sul server che è condiviso da varie applicazioni, e deve organizzare la base di dati cosìiche essa garantisca prestazioni ottimali a tutti i processi client.

- Oltre alla decomposizione funzionale dei processi e dei compiti, nelle basi di dati l' utilizzo di elaboratori diversi per client e server è perticolarmente conveniente. L' elaboratore dedicato al client deve essere adatto alla interazione con l' utente; spesso è un personal computer, dotato di strumenti di produttività tipici dell' office automation, tra questi quelli che accedono ai database . L' elaboratore dedicato al server è dimensionato in funzione dei servizi che deve offrire e del carico transazionale, deve gestire ampi buffer in memoria centrale e deve avere elevata capacità di gestire operazioni di I/O.

- Il linguaggio SQL, diffuso in tutte le basi di dati relazionali, offre un paradigma di programmazione ideale per identificare la "frontiera dei servizi". Le interrogazioni SQL vengono infatti formulate dal client e inviate al server, i risultati dell' interrogazione vengono calcolati dal server e restituiti al client. Sulla rete viaggia così una informazione

compatta: l' interrogazione può essere inoltrata come chiamata a un servizio remoto, oppure come una stringa di caratteri. Il server esegue le interrogazioni usando i dati nei file quindi confeziona il risultato. Sulla rete viaggia solo l' informazione utile al processo client, che è una frazione piccola dell' informazione estratta dalla memoria di massa del serve.

L' architettura client/server si adatta sia ad interrogazioni compilate staticamente che interrogazioni con SQL dinamico. Nel primo caso le interrogazioni vengono inviate al server una volta, poi richiamate con procedure e/o servizi remoti. Con un processo dinamico la query viene inviata sottoforma di stringa poi viene compilata ed eseguita dal server. In entrambi i casi l' ottimizzatore e i metodi di accesso risiedono sul server.

Spesso il server che gestisce tali richieste è multi-threaded: ciascuna unità di esecuzione del processo server per conto di una transazione è detto *thread*. I server sono processi permanentemente attivi che controllano due code: una di ingresso (su cui vengono accodate le richieste dei client) e una di uscita (per i risultati).

1.6 Architetture distribuite

Il termine "distribuito" è spesso usato in modo improprio e in contesti non sempre corretti. Nel nostro caso, utilizzeremo la seguente definizione: un sistema distribuito è un insieme di sistemi di elaborazione (o processori), non necessariamente omogenei, interconnessi attraverso una rete che cooperano nell' esecuzione di determinati compiti.

La distribuzione può essere relativa a:

- *Funzioni*: le diverse funzioni di un sistema possono essere delegate a diverse parti della struttura.
- *Dati*: i dati utilizzati dalle applicazioni possono essere distribuite su diversi elaboratori appartenenti alla rete.
- *Controllo*: il controllo dell' esecuzione delle applicazioni può essere svolto da più parti della struttura e non solo da un sistema.

I sistemi distribuiti possono essere classificati seguendo i seguenti criteri:

- *Grado di accoppiamento*: è un indicatore del grado di connessione tra due sistemi di elaborazione, può essere misurato come il rapporto tra la quantità di dati scambiata e la quantità di elaborazione locale in un determinato task. Possiamo avere due livelli di accoppiamento in funzione al tipo di comunicazione tra le diverse entità della struttura:
  - *Debole*: se la comunicazione avviene attraverso una rete di computer.
  - *Forte*: se la comunicazione avviene tramite componenti condivisi (memoria o supporti magnetici).
- *Struttura di interconnessione*: anche in questo caso abbiamo due casi dipendenti dal tipo di collegamento:
  - Point-to-point: i sistemi sono collegati tra loro direttamente (ad esempio una rete ad anello).
  - Con canale di comunicazione comune: i sistemi sono tutti collegati ad un unico mezzo di comunicazione (ad esempio le reti a bus).
- *Indipendenza dei componenti*: i sistemi di elaborazione possono dipendere tra loro, oppure completamente indipendenti. Nel secondo caso l' unico modo per scambiare informazioni è attravero messaggi da inviare all' inizio e alla fine dell' elaborazione rispettivamente con i dati e i risultati.
- *Sincronizzazione* tra i processi: può essere sincrona o asincrona. Questo può coinvolgere anche altri fattori: ad esempio una sincronizzazione sincrona implica dipendenza tra i sistemi e probabilmente un accoppiamento "forte".

Le strutture distribuite permettono di scomporre problemi estesi e complessi in componenti di minor dimensione. Utilizzando adeguati strumenti software per la gestione del sistema distribuito è possibile assegnare ogni componente a sistemi di elaborazione distinti che globalmente risolvono il problema.

1.6.1 Differenze rispetto i sistemi client/server

La differenza principale è nel numero di server per ogni funzione. Nel client/server il server è unico, se sono più di uno, ogni server ha uno specifico insieme di

funzioni, distinto da quello degli altri. Il client che deve utilizzare una funzione deve richiederla al server che la gestisce: aprire la transazione, inoltrare le richieste, quindi attendere i risultati e chiudere la transazione.

Nei sistemi distribuiti i server per la gestione di basi di dati sono molteplici e senza priorità, le transazioni coinvolgono più server, i quali possiedono ciascuno porzioni di dati e funzioni.

Capitolo 2

Sistemi di gestione di basi di dati distribuite

Possiamo definire un *Distributed Database System* (DDBS) come un insieme di più database logicamente intercorrelati e distribuiti attraverso una rete di computer. Di conseguenza un *Distributed Database Management System* (DDBMS) è definito come uno strumento software che permette la gestione di un DDBS e *rende la distribuzione trasparente all' utente.*

Quindi il DDBS non è semplicemente un insieme di file, ma occorre che ci sia una struttura (gestita attraverso il DDBMS) che lega logicamente questi files. Anche il concetto di rete di computer va esteso: non si indica solo l' interconnessione di diversi computer geograficamente distanti tra loro (LAN o WAN), ma ogni sistema in cui lo scambio di dati non avviene attraverso una memoria condivisa ma attraverso una rete (eterogenea) sulla quale passano unicamente messaggi.

Tra i siti della rete non ne esiste uno che contiene tutto il database, infatti se così fosse la struttura sarebbe client/server: la gestione è centralizzata e riservata al server che contiene il database (figura 2.1), al quale arrivano tutte le richieste dai siti periferici. Nell' ambiente distribuito ogni sito della rete possiede e mantiene una porzione del DDBS (figura 2.2) che può rendere disponibile ad un altro sito che ne fa richiesta.

Elenchiamo i vantaggi che la distribuzione dei dati e delle applicazioni possono dare:

Autonomia locale. La struttura distribuita permette al sito proprietario dei dati un controllo maggiore su di essi.

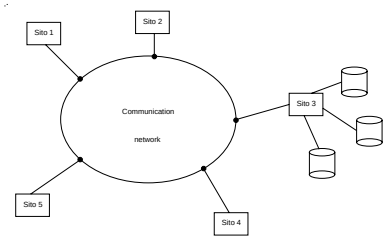


Figura 2.1. Database centralizzato

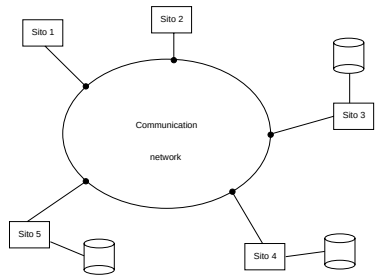


Figura 2.2. Distributed Database System

Performance. I dati più frequentemente utilizzati da un utente sono fisicamente più vicini ad esso, questo porta ad un aumento nelle prestazioni, inoltre ogni sito si occupa della gestione delle informazioni di sua pertinenza, perciò l' utilizzo delle risorse viene anch' esso distribuito tra i vari siti.

Robustezza. I dati sono replicati in più di un sito (tutti quelli che ne fanno richiesta), pertanto in caso di crash esiste sempre una copia dei dati che permette di ricostruire il database. In modo analogo nel caso di problemi sulla linea di comunicazione, è possibile trovare un percorso alternativo per recuperare i dati.

Economicità. Con i sistemi distribuiti è possibile portare i dati da elaborare sul server dell' utente, in questo modo dopo una iniziale fase di sincronizzazione dei database, le comunicazioni possono ridursi notevolmente.

Espandibilità. Espandere un database distribuito è molto semplice, infatti è sufficiente aggiungere siti alla rete, questi nuovi siti si occuperanno della gestione delle nuove funzioni aggiunte, gli altri siti non necessitano di modifiche.

Condivisibilità. La distribuzione dei dati permette una maggiore facilità nel reperimento delle informazioni, in particolare quando le distanze tra i siti sono elevate, in questi casi infatti la distribuzione rende possibile la condivisione dei dati tra i diversi computer ad un costo minore e più semplicemente che nel caso centralizzato.

Osserviamo che i vantaggi elencati devono anche essere visti come obiettivi da raggiungere nello sviluppo di un DDBMS.

Naturalmente la distribuzione non porta solo vantaggi:

Mancanza d' esperienza. I DDBMS commerciali sono stati introdotti sul mercato solo recentemente, questo porta inevitabilmente delle difficoltà nella progettazione ed installazione di nuove applicazioni.

Complessità. Aumentando la complessità della struttura di gestione dei dati, aumenta anche il numero dei problemi nuovi da affrontare, molti dei quali ancora irrisolti.

Costi. A fronte dell' economicità nelle telecomunicazioni vi è un aumento dei costi per le risorse hardware (in parte compensato dalla continua diminuzione dei prezzi) e per il software di gestione sia delle reti che delle applicazioni distribuite.

Sicurezza. Con i sistemi distribuiti il controllo della sicurezza deve essere fatto in ogni sito, inoltre occorre rendere sicura anche la rete. Questo coinvolge quindi più aspetti del sistema, che con un sistema centralizzato erano svolti interamente dal DMBS centrale.

Difficoltà di cambiamento. Il passaggio da un sistema centralizzato (eventualmente già ben collaudato) ad un sistema distribuito, è molto dispendioso sia in termini di adeguamento delle risorse hardware e software che in risorse umane (gli amministratori di sistema). Attualmente non esistono metodi che aiutano a convertire un sistema centralizzato in uno distribuito.

Ci sono però altri aspetti che possono portare a malfunzionamenti, soprattutto legati al meccanismo di replica (che descriveremo nel dettaglio in seguito): le informazioni replicate su un sito, non sono aggiornate istantaneamente quando vengono modificate dal server che le possiede, questo provoca difficoltà nella gestione della sincronizzazione delle transazioni, inoltre le reti di comunicazione diventano un aspetto critico: se qualche malfunzionamento provoca l' interruzione delle comunicazioni è possibile che i siti che possiedono repliche dei dati non abbiano i dati validi. Questi sono problemi che devono essere risolti dal DDBMS e non si presentano nei sistemi centralizzati.

2.1 Livelli di trasparenza nei DDBMS

Uno degli obiettivi dei sistemi distribuiti è la trasparenza, obiettivo comune anche ai database tradizionali, ma che nel caso distribuito assume primaria importanza. E' indispensabile infatti nascondere all' utente tutti i dettagli dell' implementazione del sistema, che possono presentare una complessità di gestione notevole. L' utente deve poter utilizzare tutti i dati e le applicazioni necessarie alla propria funzione, indipendentemente dalla localizzazione fisica all' interno della rete delle informazioni.

In figura 2.3 è rappresentato uno schema dei vari livelli di trasparenza, il nucleo centrale è rappresentato ovviamente dai dati, attorno a questo devono essere forniti gli adeguati supporti fino a permettere l' elaborazione dei dati attraverso linguaggi ad alto livello.

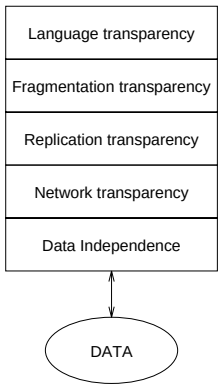


Figura 2.3. Livelli di trasparenza

2.1.1 Data independence

L' indipendenza dei dati è già stata descritta in precedenza, entriamo maggiormente nel dettaglio: possiamo distinguerne due tipi:

- Logical data independence. Riguarda la possibilità per l' applicazione utente di rimanere immune ai cambiamenti nella struttura logica del database. Il subset di attributi delle relazioni utilizzate dall' applicazione deve rimanere disponibile anche dopo eventuali aggiunte di nuovi attributi.

- Physical data independence. La memorizzazione fisica dei dati (ad esempio il tipo di supporto utilizzato) non deve riguardare l' applicazione, questa deve continuare a funzionare anche nel caso di modifiche fisiche (ad esempio il trasferimento di un database da dischi magnetici a dischi ottici).

L' applicazione utente deve essere modificata solo se ci sono variazioni nelle operazioni da eseguire sui dati.

2.1.2 Network transparency

Anche di questo tipo di trasparenza abbiamo già discusso a proposito delle reti di computer, ma entrando nello specifico delle applicazioni sui database, si deve garantire l' indipendenza dalla distribuzione dei dati, cioè l' applicazione non deve essere modificata se i dati non sono più centralizzati ma diventano distribuiti su una rete. Questo implica l' unicità dei nomi utilizzati, infatti le applicazioni li utilizzano come se fossero su un unico database.

2.1.3 Replication transparency

Per ottenere miglioramenti nelle prestazioni, i dati remoti utilizzati frequentemente (in sola lettura) devono essere replicati sul sito locale. Questa operazione viene definita "replica dei dati", e su questo argomento torneremo diffusamente in seguito.

La replica non avviene in modo completamente trasparente all' utente, infatti sarebbe molto complesso per il DDBMS gestire completamente le repliche, si è scelto di lasciare all' utente la responsabilità di decidere se avere copie o no. Questo però implica una riduzione della indipendenza dei dati.

I DDBMS più recenti vanno nella direzione di aumentare la trasparenza anche della replica, questo va a scapito di una maggior complessità del setup iniziale del database.

2.1.4 Fragmentation transparency

Ogni relazione di un database può essere suddivisa in più parti, questo permette di migliorare la gestione aumentando le prestazioni e facilitando la replica. Questa suddivisione implica un cambiamento nelle query immesse

dall' utente, la trasparenza a questo livello permette di utilizzare tabelle frammentate con query globali, il passaggio da query globale a query frammentata è eseguita dal sistema.

2.1.5 Language transparency

Gli utenti accedono ai dati utilizzando linguaggi ad alto livello, questi consentono di utilizzare i dati focalizzando unicamente gli oggetti di interesse e non i dettagli implementativi del database. Questo può avvenire utilizzando linguaggi 4GL, interfacce grafiche, linguaggi naturali, ...

2.1.6 Esempi

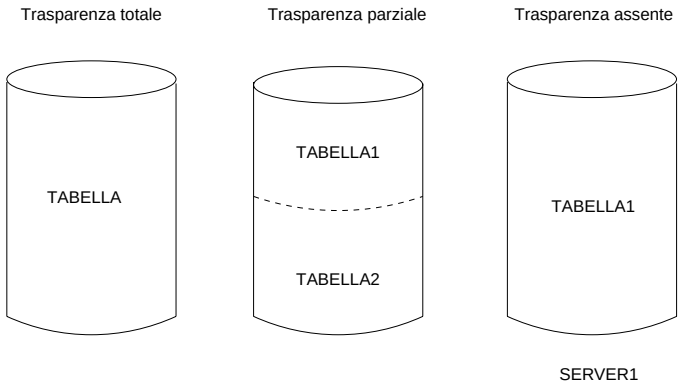


Figura 2.4. Esempi di trasparenza

Vediamo con degli esempi pratici l' impatto dei vari livelli di trasparenza nelle applicazioni dell' utente. Supponiamo di avere 3 server come in figura 2.4, la tabella TABELLA_1 è replicata dal primo server al secondo.

Con massima trasparenza, una query dell' utente non si preoccupa della localizzazione fisica dei dati, sul secondo server possiamo scrivere:

```
select nome
from tabella
where impiego = 'IMPIEGATO'
```

e accediamo a tutti i dati di TABELLA_1 e TABELLA_2 che soddisfano la query. Se dovesse mancare la trasparenza della replica, la query potrebbe trasformarsi nella seguente:

```
select nome
from tabella_1
where impiego = 'IMPIEGATO'
```

abbiamo specificato esplicitamente da quale tabella vogliamo i dati, in questo caso dalla tabella replicata TABELLA_1 sul server centrale.

Eliminiamo anche la trasparenza della rete, ora si dovrà specificare anche il luogo fisico dei dati:

```
select nome
from tabella_1 on site SERVER1
where impiego = 'IMPIEGATO'
```

2.2 Architettura dei DDBMS

Gli usuali DBMS relazionali sono basati sull' architettura ANSI/SPARC (figura 2.5), strutturata in tre livelli di vista sui dati: *esterno*, *concettuale*, *interno*, rispettivamente orientati all' utente, all' azienda e al sistema. Per ognuna di queste viste serve l' appropriata definizione dello schema.

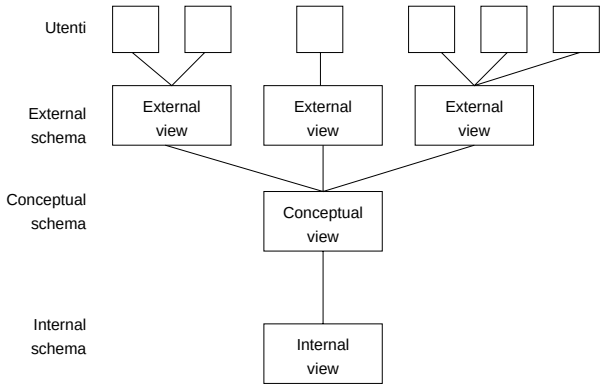


Figura 2.5. Architettura ANSI/SPARC

La trasformazione tra questi livelli viene svolta attraverso un mapping che mette in relazione tra loro le definizioni dei diversi livelli.

La natura del modello ANSI/SPARC porta a ricercarne una estensione da applicare all' ambiente distribuito.

Iniziamo a descrivere l' architettura dei DDBMS dal punto di vista dell' organizzazione dei dati.

Ogni sito può avere una organizzazione fisica dei dati diversa dagli altri, questo impone di avere uno schema interno locale (*LIS*) che permette di definire in modo uniforme le diverse strutture fisiche. Per ogni sito è necessario avere anche uno schema concettuale locale (*LCS*) che permette la definizione dell' organizzazione logica dei dati presenti sul sito.

La vista globale di tutti questi schemi è detto schema concettuale globale (*GCS*), globale in quanto descrive la struttura dei dati di tutti i siti. L' applicazione utente accede ai dati attraverso uno schema esterno (*ES*). Infine il *GD/D* (global directory/dictionary) che permette di effettuare il mapping globale dei dati quando usiamo il GCS. Questa struttura è l' estensione dell' usuale concetto di dizionario dei dati definito per i database relazionali (architettura ANSI/SPARC), è un database che contiene schemi, mapping tra gli schemi, statistiche, controllo degli accessi, ecc ....

Questo modello (figura 2.6) ha il pregio di soddisfare i requisiti di trasparenza descritti in precedenza (2.1), in particolare gli schemi concettuali globali e locali permettono di avere la trasparenza della replica e la frammentazione dei dati.

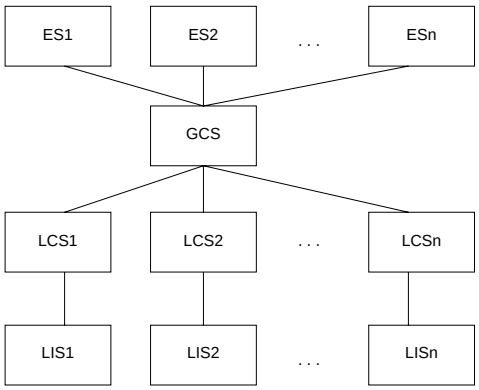


Figura 2.6. DDBS reference

Possiamo ora esaminare i componenti di un generico sistema DDBMS. E' composto da due moduli (figura 2.7):

2.2.1 User processor

L' *user processor* si occupa dell' interazione con l' utente, analizziamo le sue procedure:

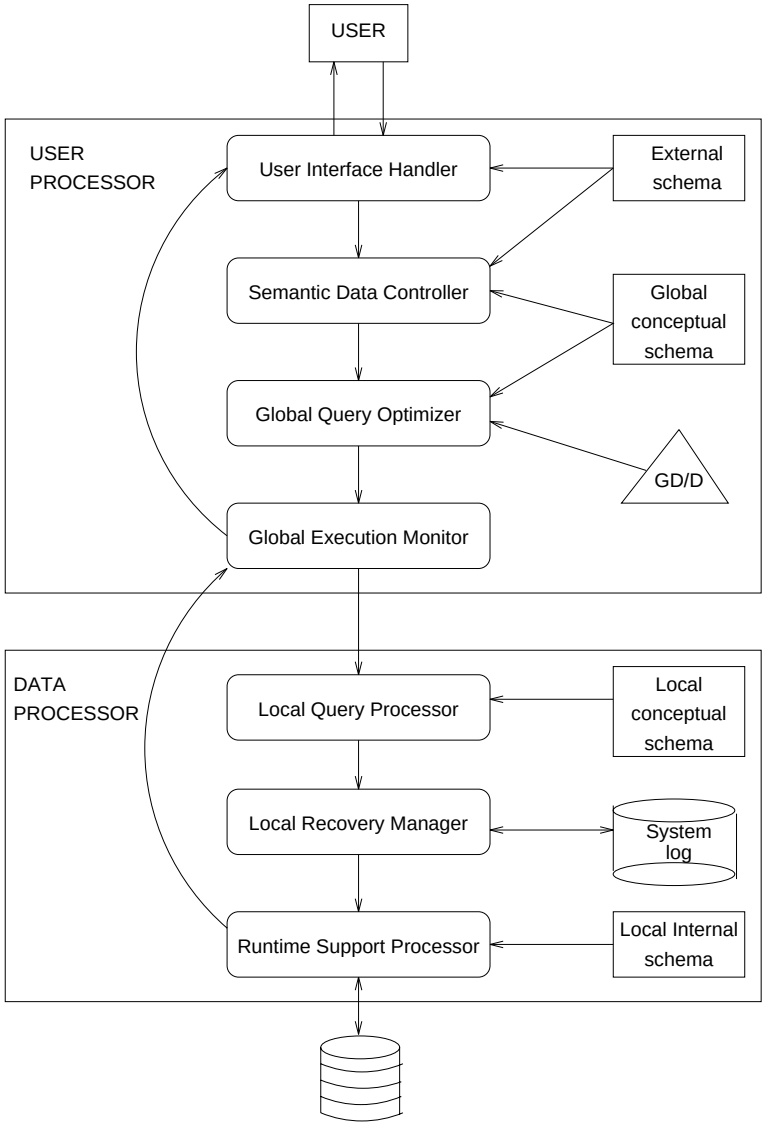


Figura 2.7. Componenti di un DDBMS

- *User interface handler.* E' l' interfaccia con l'utente: interpreta i comandi impartiti e formatta l' output.

- *Semantic data controller.* Applica i vincoli di integrità e controlla le autorizzazioni definite nel GCS per decidere se la query può essere eseguita.

- *Global query optimizer and decomposer.* Determina la strategia di esecuzione a minor costo e traduce le query globali in query locali utilizzando i GCS ed LCS eseguendo il mapping globale attraverso la GD/D.

- *Distributed execution monitor.* Coordina l' esecuzione distribuita dei comandi dell' utente.

2.2.2 Data processor

Il *data processor* si occupa della gestione e del controllo della memorizzazione e del recupero delle informazioni. E' composto da 3 componenti:

- *Local query optimizer* E' l'ottimizzatore locale della query, sceglie il miglior tipo di accesso (tramite indici) ai dati locali.

- *Local recovery manager* Si occupa di garantire che il database locale rimanga consistente anche in caso di guasto a qualche componente.

- *Run-time support processor* Accede fisicamente al database.

2.3 Frammentazione e replica dei dati

Vediamo ora i motivi che fanno preferire la decomposizione (partizionamento) delle tabelle in piccole viste nella progettazione di un database distribuito.

Nei database distribuiti la partizione dei dati è necessaria per migliorare il processo di replica, infatti scegliendo di replicare solo alcune viste il peso della replica sulla rete diminuisce considerevolmente, analogamente sui siti periferici possono essere duplicate solo le informazioni effettivamente utilizzate e non tutte. Inoltre la frammentazione consente di trattare ogni partizione come unità, quindi le transazioni possono avvenire parallelamente sulle varie viste (anche le singole query globali possono essere decomposte in sub-query da eseguire concorrentemente *intra-query concurrency*).

Gli svantaggi maggiori della frammentazione sono dati dal fatto che non tutte le applicazioni concordano sulle viste da utilizzare, quindi è possibile che le viste ideali per una applicazione non lo siano per un' altra, e questa debba accedere a più viste diverse degradando le prestazioni. La frammentazione deve essere fatta in modo tale da ridurre al minimo tali problemi.

2.3.1 Tipi di partizione

Vediamo ora i diversi tipi di partizione , per farlo introduciamo una tabella da utilizzare come esempio: la tabella ARTICOLI.

Cod_art	Descr.	Prezzo	Categ.	Q.tà
1	Articolo 1	10000	C1	23
2	Articolo 2	8500	C1	3
3	Articolo 3	14500	C3	32
4	Articolo 4	3600	C4	4
5	Articolo 5	12500	C3	20

La frammentazione può avvenire in tre diverse modalità:

- *Partizione orizzontale.* Le tabelle vengono scomposte per gruppi di righe secondo particolari caratteristiche degli attributi. Ad esempio supponiamo di frammentare la tabella ARTICOLI in modo tale da avere solo gli articoli con quantità minore di 5, il risultato è il seguente:

Cod_art	Descr.	Prezzo	Categ.	Q.tà
2	Articolo 2	8500	C1	3
4	Articolo 4	3600	C4	4

  Equivale quindi alla seguente relazione in algebra relazionale:

$$\delta_{\text{Q.tà}<5}(\text{Articoli})$$

  .

  In questo modo il destinatario della partizione (ad esempio il reparto acquisti) ha solo le righe che lo interessano con gli articoli che stanno per esaurirsi.

- *Partizione verticale.* Le tabelle vengono scomposte per gruppi di colonne (quindi solo alcuni attributi). Ad esempio partizionando verticalmente la tabella sugli attributi Cod_art, descr., Prezzo si ottiene la seguente:

Cod_art	Descr.	Prezzo
1	Articolo 1	10000
2	Articolo 2	8500
3	Articolo 3	14500
4	Articolo 4	3600
5	Articolo 5	12500

In algebra relazionale è:

$$\pi_{[\text{Cod\_art,Descr.,Prezzo}]}(\text{Articoli})$$

.

Anche in questo caso il destinatario (ad esempio il reparto vendite) ha solo le informazioni necessarie, senza quelle inutili.

- *Partizione mista.* E' l' unione dei due tipi di partizione precedenti, permette di restringere una tabella alle colonne e alle righe desiderate. Ad esempio la seguente tabella è ottenuta partizionando verticalmente rispetto alle colonne Cod_art, descr., Prezzo, e orizzontalmente alle righe con prezzo superiore a 10000:

Cod_art	Descr.	Prezzo
1	Articolo 1	10000
3	Articolo 3	14500
5	Articolo 5	12500

Il reparto vendite potrà avere un ufficio riservato alle vendite dei beni preziosi che avrà solo gli articoli di un certo valore.

2.3.2 Grado di frammentazione

Il *grado di frammentazione* è una scelta importante perchè da esso dipendono le prestazioni dell' esecuzione della query. Il minimo livello è ovviamente la tabella intera: nessuna partizione, quindi non si hanno vantaggi, si devono gestire tabelle intere che appesantiscono la distribuzione. Al contrario il livello massimo di frammentazione è la *tupla* (nel caso di partizione orrizzontale) o il singolo attributo (nel caso di partizione verticale), in questo caso aumenta l' accesso ai dati remoti creando sovraccarico inutile.

2.3.3 Alternative di allocazione

Dopo aver scelto come frammentare il database si deve decidere dove allocare i frammenti nei vari siti della rete. I dati allocati possono essere replicati oppure in singola copia. I motivi per la replica sono la robustezza e l' efficienza delle query read-only, infatti se ci sono copie multiple dei dati aumentano le possibilità di reperire le informazioni anche nel caso di guasti alla rete o al

sistema propietario dei dati. Inoltre è possibile eseguire query sugli stessi dati in parallelo, accedendo a copie su server distinti degli stessi dati. I problemi nascono con le query di update, il sistema deve assicurare che tutte le copie dei dati sono modificate correttamente. La scelta di replicare i dati dipende quindi dal rapporto tra la quantità di dati acceduta in modo read-only e le query di update.

Le alternative possono essere le seguenti:

- *database partizionato*: non ci sono dati replicati, ogni frammento del database risiede su un solo server della rete.
- *database replicati parzialmente*: i frammenti del database sono replicati su tutti i server che hanno necessità di accedervi.
- *database replicati interamente*: tutto il database viene replicato sui server della rete.

La tabella seguente riassume un confronto tra le alternative viste:

	Full replication	Partial replication	Partitioning
Query processing	Easy	Same difficulty	
Directory management	Easy or none	Same difficulty	
Concurrency control	Moderate	Difficult	Easy
Reliability	Very high	High	Low
Reality	Possible application	Realistic	Possible application

Problemi di allocazione

Consideriamo un insieme di frammenti $F = \{F_1, F_2, \ldots, F_n\}$ e un insieme di server $S = \{S_1, S_2, \ldots, S_m\}$ sui quali sono eseguiti un insieme di applicazioni (query) $Q = \{q_1, q_2, \ldots, q_q\}$.

L' allocazione dei dati implica il problema di trovare la distribuzione "ottima" dei frammenti F sui server S. L' ottimo è definito rispetto due criteri:

- Costo minimo: comprende il costo di memorizzazione di ogni F_i sul server S_j, il costo della query q_i sul sito S_j, il costo di modifica di F_i su tutti i siti che lo contengono e infine il costo di comunicazione. L' allocazione deve essere fatta in modo da minimizzare il costo globale di questi componenti.

- Performance: l' allocazione deve mantenere le prestazioni, misurate in termini di:
  - Tempi di risposta: devono essere minimizzati.
  - Throughput: deve essere massimo in ogni sito.

Purtroppo un modello che permette di ottenere questi requisiti è complesso, e non è stato ancora sviluppato. Sono stati proposti schemi di allocazione più semplici tutti NP-completi, quindi la realizzazione di uno schema ottimo nel caso di un numero elevato di frammenti non è computazionalmente fattibile. Inoltre gli schemi semplici sviluppati non si possono applicare con successo ai database distribuiti in quanto:

- I frammenti non possono essere trattati come singoli file che possono essere allocati uno alla volta, ma ogni frammento coinvolge nella decisione sull' allocazione anche gli altri che sono utilizzati insieme (ad esempio nel caso di join distribuiti). Quindi la relazione tra i frammenti deve essere tenuta in considerazione.
- L' accesso ai dati da parte delle applicazioni non può essere modellato semplicemente come: richesta - risposta (ad esempio il semplice accesso remoto ai file), nei database distribuiti l' utilizzo dei dati è più complicato.
- Soddisfare i vincoli di integrità nel caso di frammenti è molto costoso, e questi costi non sono introdotti nei modelli semplificati.
- In modo analogo i costi del controllo della concorrenza.

Una ulteriore difficoltà nella definizione dello schema di allocazione ottimo è dovuta alla centralità che occupa l' allocazione dei dati rispetto i vari componenti del DDBMS, quindi deve essere nota la sua relazione con questi componenti.

E' necessario quindi separare il problema dell' allocazione dei file (*file allocation problem*, FAP) dall' allocazione dei frammenti nel database distribuito (*database allocation problem*, DAP). Essendo il FAP un problema NP-completo, possiamo aspettarci che anche il DAP lo sia, e di solito è vero, gli algoritmi risolutivi sono di tipo euristico, ad esempio branch-and-bound (problemi tipo "knapsack problem").

E' tuttavia possibile semplificare il problema utilizzando diverse strategie:

1. Assumere che le partizioni siano determinate contemporaneamente con i loro costi e benefici in termini di query processing.

2. Ignorare inizialmente la replica, trovare una soluzione ottimale per il problema non replicato, poi introdurre la replica e applicare un algoritmo che estenda la soluzione precedente al nuovo problema.

2.3.4 Replica dei dati

I concetti di partizionamento e frammentazione dei dati rappresentano le basi per la replica dei dati.

La replica dei dati porta all' esistenza di copie multiple di una stessa porzione di dati su diversi siti. Il problema della loro gestione è quindi molto importante per garantire la consistenza del database.

Tutti le repliche di uno stesso frammento devono essere identiche (*mutual consistency*), questo può essere garantito utilizzando una tecnica denominata *one copy serializable* che indica la possibilità di serializzare le repliche da parte dello scheduler, deve rispettare le seguenti condizioni:

- ogni scheduler locale deve essere serializzabile;

- due operazioni in conflitto devono essere nello stesso ordine in tutti gli scheduler.

Un protocollo di controllo della replica che segue queste indicazioni è il *read-once/write-all* (ROWA). Consideriamo una partizione di dati x (definita *logical data*) e un insieme di copie x_1,x_2,...,x_n (definiti *physical data*). Le c' è la trasparenza della replica allora le transazioni dell' utente vanno a modificare x. Il processo di replica deve mappare ogni operazione di lettura "read(x)" in una operazione "read(x_j)", cioè la lettura avviene solo sulla replica del dato originale. Ogni scrittura "write(x)" però deve essere eseguita solo sulla partizione originale x poi replicata, questo significa che l' operazione di modifica di un dato viene trasformata in una operazione di modifica in ogni sito contenente dati replicati.

Il protocollo ROWA ha il difetto di diminuire la disponibilità dei dati in caso di guasti alla rete o ad un sito, infatti in questi casi l' operazione di scrittura in ogni sito contenente una replica può non completarsi. La soluzione di questo problema non è semplice, sono stati proposti molti algoritmi per consentire una gestione sempre consistente e sicura dei malfunzionamenti, tutti

partono dal presupposto di ignorare il guasto su un sito e modificare i dati di tutti gli altri, il disallineamento del database del sito non funzionante dovrà essere recuperato dopo la riparazione del guasto attraverso una operazione di sincronizzazione dei dati.

2.4 Sicurezza dei dati

Il DBMS deve garantire la sicurezza dei dati, distinguiamo quindi due aspetti:

- *Data protection*: impedire all' utente la visione fisica dei dati, questo è un compito svolto abitualmente dai file systems dei sistemi operativi distribuiti. La tecnica principale è la crittografia: i dati crittografati possono essere decrittografati solo dagli utenti che possiedono le chiavi corrette. I sistemi più utilizzati sono il DES *Data Encription Standard* e gli algoritmi a chiave pubblica (alcuni sono RSA, RC4 e IDEA).

- *Authorization control* deve garantire che solo gli utenti autorizzati possano eseguire determinate operazioni sul database. Questo controllo viene effettuato in parte dal sistema operativo (l' accesso fisico al sistema), e in parte dai DBMS distribuiti o centralizzati, i quali possono permettere o negare l' accesso anche a sottoinsiemi dei dati, permettendo un accesso differenziato agli utenti con previlegi diversi.

Il secondo aspetto è quello che ci interessa maggiormente, in particolare per le funzioni svolte dai DBMS. Infatti rispetto ai sistemi operativi che controllano le autorizzazioni, i DBMS consentono una maggiore differenziazione in modo tale che utenti diversi hanno diritti diversi sulla stessa porzione di dati, questo implica che un DBMS deve distinguere gli utenti non solo attraverso il nome e la password, ma anche con il ruolo, la funzione e il gruppo di appartenenza.

2.4.1 Controllo centralizzato delle autorizzazioni

Il controllo delle autorizzazioni viene svolto da tre entità:

- Utenti.

- Operazioni.

- Oggetti del database.

Il controllo dell' autorizzazione consiste nel verificare se una data tripla `(utenti,operazioni,oggetti)` può procedere o no, nel sistema di sicurezza sono definite delle altre triple del tipo: `(utente,tipo operazione,definizione)` chiamate *autorizzazioni*, a questo punto è sufficiente verificare che la tripla data verifica l'autorizzazione, in caso affermativo può proseguire.

L' accesso dell' utente viene verificato attraverso il username e password, l' username identifica univocamente un utente nel sistema, la password autentica l' utente. Gli oggetti da proteggere sono sottoinsiemi del database, nei sistemi relazionali sono viste, relazioni, tuple, attributi. Le relazioni tra gli utenti e gli oggetti possono essere stabilite utilizzando dei comandi SQL:

```
GRANT <operation type> ON <object> TO <user>

REVOKE <operation type> FROM <object> TO <user>
```

L' user specificato può essere "public", questo significa che l' operazione sull' oggetto può essere compiuta da tutti gli utenti del DBMS.

Utilizzando GRANT/REVOKE il controllo viene delegato ad un utente previlegiato (di solito l' amministratore) che può decidere sui diritti degli altri, questo è definito controllo centralizzato.

Un metodo più efficiente è il controllo decentralizzato: Il creatore di un oggetto ne diventa proprietario e possiede tutti i previlegi sull' oggetto. Utilizzando una operazione GRANT il proprietario può autorizzare altri utenti, i quali ne possono autorizzare altri ancora. La complessità si ha quando i diritti devono essere revocati: questa operazione infatti diventa ricorsiva e deve interessare tutti gli utenti che sono stati autorizzati.

Per memorizzare le autorizzazioni ci sono diversi metodi, il più efficiente è l' utilizzo della *matrice delle autorizzazioni*: nelle righe abbiamo gli utenti, nelle colonne gli oggetti:

	E	ENAME	G
Casey	UPDATE	UPDATE	UPDATE
Jones	SELECT	SELECT	SELECT WHERE RESP != "Manager"
Smith	NONE	SELECT	NONE

2.4.2 Controllo distribuito delle autorizzazioni

Nei sistemi distribuiti il problema della sicurezza si complica. Oltre agli accessi locali il sistema deve controllare anche gli accessi remoti, le autorizzazioni distribuite e la gestione di gruppi remoti.

Ogni sito deve accettare procedure iniziate e autorizzate nei siti remoti, quindi l' utente deve essere autenticato nel sito acceduto:

- Le informazioni per l' autenticazione degli utenti (username e password) sono replicate in tutti i siti, i programmi locali iniziati nei siti remoti devono indicare anche l' username e la password.
- Tutti i siti del DDBMS si identificano ed autorizzano usando un metodo simile a quello degli utenti. Le comunicazioni tra siti sono protette dall' uso di password associate al sito, dopo che un sito autorizza l' inizio di un programma, non servono ulteriori autenticazioni nei siti remoti.

La seconda soluzione è necessaria se non è installata la replica dei dati, in ogni caso facilita e rende efficiente l' autenticazione remota. Se non vengono replicati username e password, le tabelle che li contengono devono trovarsi sul sito in cui l' utente accede, questo però prevede che l' utente sia "statico", e questo è vero solo in casi particolari.

Le autorizzazioni distribuite sono espresse nello stesso modo di quelle centralizzate e devono essere definite nelle tabelle di sistema. Possono essere:

- replicate interamente verso ogni sito: in questo modo l' autorizzazione può essere verificata quando la query viene compilata, però ha maggior costo in quanto comporta dati duplicati.
- oppure memorizzate nei siti contenenti gli oggetti.

Dal punto di vista dell' amministrazione di DDBS la creazione di gruppi permette di semplificare la gestione delle autorizzazioni. In un sistema centralizzato tutti gli utenti possono essere definiti come "public", in un sistema distribuito è utile avere lo stesso concetto: il "public" denota tutti gli utenti del sistema, ed è un caso particolare di gruppo. La creazione dei gruppi è effettuata dall' amministratore tramite questo comando:

```
DEFINE GROUP <group_id> AS <list of subject ids>
```

La gestione dei gruppi porta problemi aggiuntivi: il subject può essere locato in diversi siti e l' accesso ad un oggetto deve essere garantito a diversi gruppi anche internamente distribuiti. Se le autorizzazioni e le informazioni sui gruppi sono replicate a tutti i siti, allora la gestione è simile a quella del database centralizzato. Altrimenti se l' autonomia deve essere mantenuta con un controllo decentralizzato, possiamo affrontare il problema autorizzando l' accesso dopo aver verificato i diritti mediante una query remota sul sito che mantiene la tabella con le autorizzazioni.

In conclusione, la replica delle informazioni legate all' autorizzazione ha due grandi vantaggi:

1. Il controllo delle autorizzazioni è più semplice e può essere fatto al compile-time.

2. L' overhead dovuto alla gestione delle tabelle relative all' autorizzazione può essere elevato se il numero di siti coinvolti è alto.

2.5 Esecuzione delle query

L' esecuzione delle query è svolta da un modulo denominato *query processor*, che consente di esprimere le interrogazioni in linguaggio ad alto livello (anche naturale) in modo tale da nascondere completamente i dettagli implementativi. Questo modulo deve anche occuparsi di ottimizzare le query introdotte dall' utente, il quale così non deve preoccuparsi della "qualità" delle interrogazioni introdotte.

Il query processor è un punto critico dell' intero DBMS, le performance globali del sistema dipendono in modo considerevole dal metodo scelto per eseguire le query, di conseguenza la scelta di un buon algoritmo è molto importante ma ed è difficile il confronto di più metodi diversi, sono stati introdotti quindi dei parametri per la valutazione:

Linguaggio. Il query processor deve eseguire un mapping tra il linguaggio di input (quello utilizzato dall' utente) e il linguaggio di output (di solito in algebra relazionale con primitive di comunicazione).

Tipo di ottimizzazione. La ricerca del miglior metodo di esecuzione implica un costo notevole per l' ottimizzatore, un overhead che deve essere

minimo. Non è sufficiente quindi esaminare il costo delle query ottimizzate (che in ogni caso deve essere minimo), ma si deve tenere in considerazione anche il costo di ottimizzazione del DBMS.

Tempo di ottimizzazione. Una query può essere ottimizzata staticamente: cio'e prima dell' esecuzione; oppure dinamicamente: durante l' esecuzione. I due metodi hanno vantaggi e svantaggi:

- Statico: il vantaggio è che l' ottimizzazione viene fatta una volta sola durante la compilazione, poi riutilizzata anche per le query successive, in modo tale che il costo di ottimizzazione viene in parte ammortizzato. Il metodo utilizzato è di tipo statistico, questo è il punto debole: infatti nel caso di dati errati tutta l' ottimizzazione è inutile.
- Dinamico: l' ottimizzazione viene effettuata durante l' esecuzione, in ogni momento è possibile scegliere il metodo migliore per la singola operazione da eseguire. Non sono richieste le statistiche sul database (ad eccezione della prima query), i dati vengono ricavati run-time, quindi la propbabilità di commettere errori è molto inferiore al metodo statico. Il più grosso svantaggio è dato dal costo che viene moltiplicato per ogni operazione da eseguire.

Sono possibili metodi misti: si segue l' approccio statico fino a quando i dati statistici si dimostrano errati, a questo punto si effettua l' ottimizzazione dinamica con la correzione delle statistiche (per le esecuzioni successive).

Statistiche. Le statistiche racchiudono informazioni sullo stato del database, il numero e la cardinalità dei frammenti, propietà degli attributi. L' utilizzo delle statistiche è fondamentale nel caso statico, marginale in quello dinamico (dove si usa solo per la prima query), è importante però che siano sempre aggiornate e corrette, per questo vengono fatti update periodici, che hanno un costo notevole ma sono indispensabili. Dopo ogni modifica alle statistiche alcune query ottimizzate in modo statico possono essere riottimizzate per migliorarne l' efficienza.

Sito decisionale. Nel caso di ottimizzazione statica, uno o più siti possono essere coinvolti nella decisione della strategia. Se il sistema decisionale è centralizzato allora un singolo sito si occupa dell' ottimizzazione, e

sarà l'unico contenente il database delle statistiche. Viceversa, se la decisione è distribuita tra tutti i server del sistema allora le query sono ottimizzate utilizzando dati locali. Sono possibili approcci ibridi, in cui un sito è prevalente e si occupa della maggior parte delle ottimizzazioni, gli altri siti effettuano decisioni locali.

Topologia di rete. E' importante la conoscenza della topologia della rete in quanto il peso del costo di comunicazione cambia. Nel caso delle reti geografiche il costo da minimizzare può essere ristretto al solo costo di comunicazione trascurando le altre componenti. Nel caso di reti locali il costo di comunicazione può essere considerato uguale al costo di I/O, quindi diventa ragionevole aumentare il parallelismo dell' esecuzione delle query.

Replica dei frammenti. E' utile avere frammenti del database replicati su diversi siti, l' ottimizzatore deve conoscere quindi il luogo in cui i dati vengono acceduti: se su una replica locale, oppure sul sito proprietario.

Utilizzo dei semijoins. Il semijoin ha la propietà di ridurre la dimensione delle relazioni, di consenguenza quando il costo di comunicazione è importante, conviene utilizzare dei semijoin che riducono le quantità di dati da scambiare.

L' esecuzione di una query in un ambiente distribuito viene scomposta in quattro diversi sotto-problemi, rappresentati a livelli nella figura 2.8. L' input è una query su dati distribuiti espressa mediante gli operatori relazionali, è costituita da relazioni globali (distribuite), questo significa che la distribuzione viene nascosta all' utente. L' obiettivo è trasformare la query globale in una sequenza di operazioni locali sul database locale.

I primi tre livelli:

- Query decomposition,
- Data localization,
- Global query optimization,

sono eseguiti da un sito centrale ed utilizzano informazioni globali, il quarto:

- Local query optimization

è eseguito localmente.

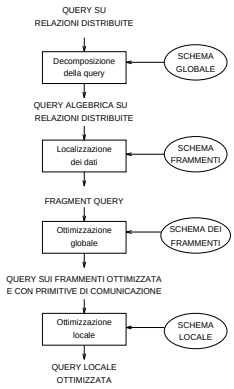


Figura 2.8. Schema di esecuzione di una query su DDBS

2.5.1 Query decomposition

Il primo livello decompone una query globale distribuita in una query sulle relazioni distribuite. Questo obiettivo viene raggiunto mediante una sequenza di passi successivi:

Normalizzazione

La query in input può essere arbitrariamente complessa, questo dipende dalle caratteristiche del linguaggio usato per scriverla. L' obiettivo della normalizzazione è quello di trasformare la query nella forma normale, in modo tale da facilitare le operazioni successive.

Nel caso di linguaggi relazionali come l' SQL la trasformazione più importante è eseguita sulla clausola `WHERE`. Ci sono due possibili forme per i predicati, in funzione alla precedenza dell' operatore AND o dell' OR.

Analisi semantica

Nel secondo passo viene eseguita l' analisi semantica della query, in modo tale da rilevare gli errori e in questo caso non procedere ulteriormente con l' esecuzione, ma rifiutare la query e visualizzare un messaggio di errore.

I casi possibili di errore sono due:

- Tipo errato: un attributo o un nome non sono definiti nello schema globale, oppure se l' operazione applicata sugli attributi non è del tipo corretto. Le tecniche utilizzate per cercare questo tipo di errore sono simili a quelle usate nel type checking dei linguaggi di programmazione

ad alto livello. I tipi fanno parte dello schema globale in quanto una query relazionale non produce nuovi tipi.

Esempio: data la tabella `EMP(ENUM,ENAME,TITLE` e la query:

```
SELECT E#
FROM   EMP
WHERE  ENAME > 200
```

Gli errori sono due: il campo `E#` non è un attributo dichiarato nello schema, inoltre il confronto avviene tra due tipi diversi: una stringa ed un intero.

- Query semanticamente errata: quando ha un componente che non contribuisce in nessun modo nella produzione del risultato. Dal punto di vista del calcolo relazionale non è possibile determinare la correttezza semantica di una generica query, tuttavia è possibile farlo per una ampia classe di query: tutte quelle che non contengono disgiunzioni e negazioni. Il metodo utilizzato per determinare questo tipo di errori si basa sull' utilizzo di grafi (*query graph*).

Eliminazione della ridondanza

Il terzo passo è la semplificazione della query in modo tale da eliminare ogni ridondanza. In particolare ad una query su una view può essere arricchita con dei predicati per la corrispondenza tra la vista e la relazione assicurando l' integrità semantica, questi predicati possono essere ridondanti rispetto quelli già presenti nella query. Occorre quindi eliminare i predicati ridondanti utilizzando le propietà di idempotenza degli operatori logici.

Esempio: la seguente query:

```
SELECT  TITLE
FROM EMP
WHERE  (NOT (TITLE = ‘‘Programmer’’)
AND (TITLE = ‘‘Programmer’’
OR TITLE = ‘‘Elect.Eng.’’)
AND NOT (TITLE = ‘‘Elect.Eng.’’)
OR ENAME = ‘‘J. Doe’’
```

Chiamiamo: p_1 = `<TITLE = ‘‘Programmer’’>`, p_2 = `<TITLE = ‘‘Elect.Eng’’>` e p_3 = `<ENAME = ‘‘J. Doe’’>`, utilizzando le propietà dell' AND ($\wedge$), OR ($\vee$) e NOT (!) abbiamo:

$$(!p_1 \wedge (p_1 \vee p_2) \wedge !p_2) \vee p_3$$

semplificando:

$$(!p_1 \wedge p_1 \wedge !p_2) \vee (!p_1 \wedge p_2 \wedge !p_2) \vee p_3,$$

$$(\text{false} \wedge !p_2) \vee (!p_1 \wedge \text{false}) \vee p_3,$$

$$\text{false} \vee \text{false} \vee p_3.$$

Rimane quindi solo p_3, la query semplificata è:

```
SELECT TITLE
FROM EMP
WHERE  ENAME = ‘‘J. Doe’’
```

Ricostruzione

Infine, ultimo passo è la ricostruzione della query: la sua riscrittura utilizzando le regole dell' algebra relazionale. Abbiamo due operazioni da compiere:

- trasformazione della query dal calcolo relazionale all' algebra relazionale.
- ristrutturare la query in algebra relazionale per ottimizzare le performance.

Questo può essere fatto utilizzando un metodo grafico: *relational algebra tree* consistente in un albero in cui ogni foglia è una relazione memorizzata nel database, e ogni nodo interno una relazione intermedia prodotta da una operazione relazionale.

L' SQL è un linguaggio che si presta bene alla creazione dell' albero, infatti è sufficiente esaminare le clausole FROM per avere le foglie, attraverso la WHERE possiamo ricavare le operazioni relazionali che ci portano verso la radice.

2.5.2 Localizzazione dei dati

Il secondo livello dello schema dell' esecuzione della query distribuita (figura 2.8) riguarda la localizzazione dei dati utilizzati dalla query (ottenuta dal livello precedente). Questo livello determina quali frammenti sono necessari, e trasforma la query distribuita in una query di frammento (*fragment query*) utilizzando lo schema di frammentazione. Questa trasformazione viene eseguita applicando delle regole di ricostruzione alla relazione globale, si ottiene un programma relazionale (*localization program*) che ha come operandi i frammenti.

Un primo approccio (inefficiente) consiste nel sostituire alle foglie dell' albero relazionale i programmi di localizzazione. In questo modo dalla ricostruzione otteniamo una query generica. L' inefficienza è nel fatto che la query ottenuta non è suffientemente ottimizzata, in quanto può essere ancora semplificata.

Un metodo migliore è la *reduction techniques* che genera query semplici ed ottimizzate. Ogni tipo di frammentazione ha un proprio metodo di riduzione:

- Partizione orizzontale: è basata su predicati SELECT, e la riduzione consiste nel determinare la query ricostruita che produce relazioni vuote ed eliminarla. Questo permette di ottimizzare selezione e join:
- Partizione verticale: è basata di operatori di proiezione, il programma di localizzazione consiste nel join dei frammenti con attributi comuni.
- Partizione mista: il programma di localizzazione utilizza unioni e join di frammenti.

2.5.3 Ottimizzazione globale

Questo livello riceve in ingresso le query sui frammenti generate al livello precedente, in uscita le fornisce ottimizzate e con le operazioni per le comunicazioni. I livelli precedenti hanno già ottimizzato in parte la query, però in modo indipendente dalle caratterisctiche del frammento.

L' ottimizzatore globale deve trovare la strategia migliore per l' esecuzione della query, cioè un insieme di operazioni relazionali e di primitive di comunicazione "send" e "receive" tra i siti. Permutando l' ordine delle operazioni su un frammento è possibile individuare delle query equivalenti, ciascuna di costo diverso, l' ottimizzatore deve scegliere le query a costo minimo. Per

semplificare, molti DBMS trascurano i diversi costi legati all' elaborazione locale per evidenziare solo i costi di comunicazione, questo può essere vero solo nel caso di reti metropolitane e geografiche con banda limitata.

In generale i costi totali sono determinati dalla seguente formula:

$$\text{Costi}_{\text{tot}} = C_{\text{CPU}} \cdot \#\text{instr} + C_{\text{I/O}} \cdot \#\text{I/O} + C_{\text{MSG}} \cdot \#\text{msg} + C_{\text{TR}} \cdot \#\text{bytes}$$

che è la somma dei tempi di elaborazione (locale) espressi dal costo della CPU: C_{CPU} e dal costo dell' I/O: $C_{\text{I/O}}$, poi il costo di inizializzazione dei messaggi C_{MSG} che è un valore fisso, infine il costro di trasmissione C_{TR} moltiplicato per il numero #bytes dei bytes trasmessi, questi ultimi due addendi rappresentano il costo di comunicazione. In generale il costo è espresso in secondi, poi può essere convertito in costo economico.

L' ottimizzazione viene fatta prima dell' esecuzione (*static optimization*), pertanto la strategia viene scelta utilizzando informazioni di tipo statistico sui frammenti e sulla cardinalità delle operazioni relazionali, da queste informazioni dipende la "bontà" dell' ottimizzazione.

Ottimizzazione centralizzata

I due algoritmi più diffusi di ottimizzazione centralizzata sono implementati in due DBMS:

- INGRES: utilizza un algoritmo *dynamic optimization* (denominato INGRES-QOA) che ricorsivamente spezza le query in pezzi di dimensione minore (utilizzando selezioni e proiezioni) fino a che tulle le query diventano monovariabile. Il risultato della query monovariabile è memorizzata in una struttura, per essere poi utilizzata nell' ottimizzazione successiva.

- System R: basato su un algoritmo *static optimization* (R-QOA), in input riceve il relational algebra tree, l' output è una strategia di esecuzione che implementa l' albero "ottimo". Per prima cosa si determina il metodo di accesso migliore per ogni relazione basata sul select, poi si calcola il miglior ordinamento per i join.

Ottimizzazione distribuita

Analizziamo ora gli algoritmi di ottimizzazione per le query distribuite, alcuni sono ricavati da estensioni dei due algoritmi descritti nel caso centralizzato:

Distributed INGRES query optimization algorithm (*D-INGRES-QOA*) e System R* algorithm (*R*-QOA*) altri sono nati con i DBMS distribuiti: SDD-1 algorithm e la famiglia di algoritmi chiamata AHY (dai nomi degli autori: Apers, Hevner e Yao). Vediamo brevemente un confronto tra i diversi algoritmi:

Algoritmo	Timing	Obiettivo	Fattore	Topologia	Semijoin	Stats [a]	Frag.
Distributed INGRES	Dynamic	Tempo di risposta Costo totale	Dim. MSG proc.cost	Tutte Broadcast	NO	1	Oriz.
R*	Static	Costo totale	Dim.MSG, I/O, CPU	Tutte Locale	NO	1,2	NO
SDD-1	Static	Costo totale	Dim.Msg	Tutte	Si	1,3, 4,5	No
AHY	Static	Tempo di risposta Costo totale	Dim.Msg, #Msg	Tutte	Si	1,3,5	NO

[a] Tipo di statistiche: 1 = Cardinalità, 2 = Valori unici per ogni attributo, 3 = Selettività del join, 4 = Proiezione degli attributi del join, 5 = Dimensione attributi e tuple

Vediamo le singole differenze rispetto le caratteristiche dei query processor descritte in precedenza (2.5):

1. L' ottimizzazione è dinamica per il D-INGRES-QOA, statica per gli altri.

2. L' obiettivo del SDD-1 e del R*-QOA è minimizzare il costo totale, mentre per D-INGRES-QOA e AHY minimizzano la combinazione dei costi totali e dei tempi di risposta.

3. I fattori di ottimizzazione sono:

   - le dimensioni dei messaggi per SDD-1,
   - le dimensioni e il numero dei messaggi per AHY,
   - le dimensioni e il numero dei messaggi, costi di CPU locali nel sito di esecuzione e i costi di I/O per R*-QOA,
   - la combinazione di tutti i fattori precedenti per D-INGRES-QOA.

4. La topologia è assunta come point-to-point WAN da SDD-1 e AHY, mentre D-INGRES-QOA e R*-QOA lavorano sia con LAN che WAN.

5. L' uso dei semijoins come tecnica di ottimizzazione è caratteristica di SDD-1 e AHY. D-INGRES-QOA e R*-QOA eseguono i join in modo simile alla versione non distribuita dei rispettivi algoritmi.

6. Tutti gli algoritmi prelevano le statistiche dai dati, SDD-1 e AHY che utilizzano i semijoin richiedono un numero maggiore di informazioni. Le informazioni utilizzate sono: la cardinalità delle relazioni, il numero di valori unici per attributo, il fattore di selettività dei join, la dimensione della proiezione su ogni attributo dei join e infine la dimensione delle tuple e degli attributi.

7. Solo INGRES può gestire la frammentazione, ma solo in partizioni orizzontali.

Distributed INGRES Algorithm

L' algoritmo D-INGRES-QOA deriva direttamente dalla versione non distribuita utilizzata in INGRES. Ottimizza dinamicamente le query, cercando di minimizzare i costi di comunicazione e il tempo di risposta, siccome questi due obiettivi sono in contrasto tra loro si deve dare un "peso" all' obiettivo che si ritiene prioritario. Sono considerate anche reti broadcast l' algoritmo deve distinguere la topologia della rete in cui lavora per effettuare il calcolo corretto dei costi di comunicazione, nel caso broadcast un sito invia i dati a tutti gli altri in un singolo trasferimento, questo modo viene usato per replicare i dati e quindi per massimizzare il grado di parallelismo.

Descriviamo in breve l' algoritmo:

1. Vengono eseguite subito tutte le query in una sola variabile, che sono eseguite localmente.

2. Viene applicato un algoritmo di riduzione alla query, in modo tale da isolare le query non riducibili e le query ridotte ad una variabile.

3. Le query ad una variabile non sono processate in quanto già eseguite nel primo passo, quindi viene scelta la prima query di almeno due variabili.

4. Si scelgono prima le query che non hanno predecessori e coinvolgono frammenti di piccola dimensione, in questo modo si minimizza la dimensione dei risultati intermedi. In base alla topologia della rete viene scelto il sito che deve processare i frammenti.

5. Determiniamo come eseguire la subquery e selezioniamo i frammenti da replicare al sito che deve processare la query (metodo denominato *fragment and replicate*).

6. Quindi trasferiamo fisicamente i frammenti verso i loro siti destinazione.

7. Infine su questi siti possiamo eseguire le query localmente e concludere l' esecuzione della query globale.

8. Per ogni query ancora da eseguire si torna al passo 4.

Il principale problema di questo algoritmo è che ricerca la migliore tra un campo limitato di soluzioni: effettua tutte le decisioni senza tenere in cosiderazione l' ottimizzazione globale, ogni passo sceglie la propria miglior strategia, ma non vi è legame tra le decisioni sequenziali, questo è necessario poiché il metodo è dinamico e di conseguenza deve essere eseguito velocemente. Un modo per risolvere questo inconveniente è la ricerca esaustiva tra tutte le possibili soluzioni, da studi effettuati si prova che la ricerca esaustiva fornisce prestazioni migliori non appena la query accede a più di tre relazioni.

L' approccio dinamico ha il vantaggio di poter conoscere la dimensione dei risultati intermedi in modo esatto.

R* algorithm

Questo algoritmo è statico, di conseguenza può effettuare la ricerca su tutte le possibili soluzioni, in quanto il tempo di ottimizzazione non è critico come il caso dinamico. L' overhead introdotto in questa fase viene rapidamente ammortizzato se la query è eseguita frequentemente. La compilazione delle query è coordinata da un master che è il sito in cui la query inizia. L' ottimizzatore sul master prende tutte le decisioni riguardanti la query, anche il sito di esecuzione, i frammenti e il metodo di trasferimento dei dati; i siti che hanno relazioni coinvolte nella query prendono unicamente le decisioni locali.

Vediamo brevemente l' algoritmo R*-QOA:

1. Per ogni relazione in ingresso si determina il metodo di accesso a costo minimo.

2. Quindi si esaminano tutti le possibili permutazioni dei join (per n relazioni ci sono $n!$ permutazioni), e si sceglie la miglior strategia di accesso della query, questo viene fatto costruendo dinamicamente un albero

contenente le permutazioni, poi si percorre fino a trovarne una di costo minimo.

3. Si sceglie il sito che dovrà contenere il risultato e il metodo per trasferirvi i dati.

4. Si inviano i dati al sito remoto, il quale utilizzando una strategia calcolata localmente.

I primi due passi vengono eseguiti utilizzando i dati statistici sul database.

Un algoritmo di questo tipo coinvolge ricerche su permutazioni, ed è un problema di ottimizzazione combinatoria che coinvolge un elevato numero di relazioni. Per ridurre il numero di alternative R* utilizza tecniche di programmazione dinamica che permettono di costruire l' albero delle alternative eliminando le scelte inefficienti (in modo simile agli algoritmi branch-and-bound).

SDD-1 algorithm

Questo algoritmo è nato esclusivamente per l' ottimizzazione delle query distribuite. E' basato su algoritmi approssimati (di tipo "greedy") che data una soluzione ammissibile tentano di migliorarla.

In questo caso la soluzione è uno schema di esecuzione globale, comprendente le comunicazioni tra siti. E' ottenuta calcolando i costi di tutte le strategie di esecuzione che trasferiscono tutte le relazioni richieste verso i siti candidati, e poi sceglie quella a costo minimo. Un difetto è quello di trovare soluzioni ottime locali che non lo sono globalmente.

Nel SDD-1 si usa ampiamente il semijoin, si vuole minimizzare i costi di comunicazione, quindi non si considerano i costi locali e i tempi di risposta. L' algoritmo è statico pertanto richiede le statistiche sul database (le chiameremo "*database profiles*"), ciascun profile è associato ad una relazione.

L' algoritmo procede in quattro fasi:

1. Inizializzazione: viene generato l' insieme BS dei semijoin più vantaggiosi (cioè con i costi minori dei benefici).

2. Selezione dei semijoin migliori:

   - Se l' insieme BS non è vuoto, si sceglie il semijoin con maggiori benefici e minor costo, modificando le statistiche e aggiungendolo alla strategia di esecuzione ES.

- Si ripete finché tutti i semijoin di BS sono stati aggiunti a ES.

L' ordine con cui i semijoin sono in ES determina l' ordine di esecuzione.

3. Selezione del sito di assemblaggio: tra i vari siti coinvolti sceglie quello che ha un costo di trasferimento dati minore (cioè quello che ha già la maggior parte dei dati da accedere locali).

4. Post-ottimizzazione: si rimuovono dalla strategia di esecuzione ES tutti i semijoin che accedono solo a tabelle locali sul sito destinazione. Questa fase è necessaria in quanto il sito viene scelto dopo l' ordinamento dei semijoin.

Nel caso di qualche semijoin scelto erroneamente l' ottimizzazione finale provvede ad eliminarlo, in questo modo si ottiene un duplice controllo sul risultato.

Questo algoritmo ignora i semijoin con costo più elevato, di conseguenza è possibile che non riesca a trovare la soluzione globale di costo minimo.

AHY algorithm

Apers, Hevner e Yao hanno proposto una famiglia di algoritmi che utilizzano i semijoins per minimizzare il tempo di risposta o il tempo totale. Il metodo è basato su un insieme di algoritmi applicabile solo a "query semplici", cioè quelle query che interessano (dopo una elaborazione locale) solo relazioni con un attributo di join in comune e che è l' output della query. Sono ignorati i costi di elaborazione locale, in quanto si assume la rete point-to-point WAN.

L' AHY processa una query applicando prima delle operazioni locali, quindi decompone la query in query semplici per le quali viene pianificata una strategia ottima utilizzando il semijoin, infine vengono integrate le diverse strategie in una comune.

Gli algoritmi proposti sono due: SERIAL per minimizzare il tempo totale, PARALLEL per minimizzare il tempo di risposta.

Il SERIAL basa il suo funzionamento sulla minimizzazione della quantità globale di dati trasmessa, cerca di spostare le piccole relazioni verso i siti in cui risiedono quelle di dimensione maggiore, e in questi siti effettuare il semijoin, questo viene fatto preparando una lista contenente le relazioni ordinate per dimensione crescente, quindi si trasferiscono le relazioni più

piccole nel sito contenente la relazione successiva (di dimensione maggiore) fino a quando è possibile (cioè finché ci sono relazioni piccole dirette verso un sito contenente una relazione maggiore).

Il PARALLEL cerca di minimizzare la quantità di dati da trasferire serialmente per favorire la trasmissione in parallelo, le relazioni sono ordinate per dimensione crescente, poi viene scelta come soluzione possibile (iniziale) quella che consente di portare il maggior numero di relazioni verso un unico sito. A questo punto l' ottimizzatore sceglie tra le alternative anche le relazioni da inviare attraverso siti intermedi.

Negli ottimizzatori di solito si implementa un algoritmo che racchiude entrambi i precedenti, poi è l' ottimizzatore può scegliere a quale obiettivo dare la priorità. L' algoritmo comune, denominato GENERAL procede nel seguente modo:

1. In ogni sito periferico viene processata localmente la query per ricavare la query semplice che corrisponde ad ogni attributo di join della query.

2. Ogni query semplice viene isolata e in funzione dell' obiettivo da minimizzare si esegue l' algoritmo SERIAL o il PARALLEL, si produce un insieme di strategie candidate (una per ogni query semplice).

3. Le strategie per le query semplici vengono integrate in una strategia globale, questo avviene introducendo per prime le strategie che minimizzano il parametro che si vuole ottimizzare.

4. Infine vengono eliminate le ridondanze, cioè le relazioni che sono già state trasmesse come parte di altre.

2.5.4 Ottimizzazione locale

L' ultimo livello dello schema di esecuzione delle query distribuite è l' ottimizzazione locale. Riceve in ingresso le query sui frammenti già ottimizzate dal livello precedente e fornisce in uscita query ottimizzate sul database locale.

L' ottimizzazione è fatta seguendo gli algoritmi tradizionali, utilizzati sui DBMS centralizzati.

2.6 Gestione delle transazioni distribuite

Le transazioni devono avere precise caratteristiche:

- Atomicità: se una transazione si interrompe (per qualunque motivo) i risultati parziali ottenuti fino a quel momento devono essere annullati. Questo equivale a dire che le transazioni devono essere viste come una unica istruzione, anche se racchiudono più operazioni, se anche solo una di queste fallisce, tutta la transazione deve fallire.

- Consistenza: una transazione è una trasformazione dello stato del sistema in un altro, ma deve mantenere tutte le invarianti predefinite.

- Isolamento: una transazione non completa non può fornire un risultato ad un' altra transazione concorrente prima del commit.

- Persistenza: quando una transazione termina (commit) il sistema deve garantire che il risultato non venga perso, indipendentemente dai guasti che possono accadere dopo il commit.

Una transazione è di solito parte di una applicazione che non ha queste caratteristiche, è possibile utilizzare la primitiva `begin transaction` per definire l' inizio di una sequenza di istruzioni che diverranno parte di una stessa transazione, e quindi con le caratteristiche descritte, che termina quando incontra un comando `commit` o *abort*.

Le transazioni distribuite eseguono processi che si trovano su differenti siti, uno di questi è denominato "root" e si trova sul sito di origine della transazione. In questo sito viene iniziata la transazione (con il `begin transaction`) e viene conclusa, inoltre si occupa di creare i processi da eseguire sui siti remoti.

Esempio: una applicazione deve trasferire una somma da un conto ad un altro. I dati immessi dall' utente sono nelle seguenti variabili:

```
$c_prelievo = conto da cui prelevo la somma
$c_deposito = conto su cui deposito la somma
$trasf = somma da trasferire
```

Il processo ROOT è il seguente:

```
...
begin transaction
select TOTALE from CONTO
where N_CONTO=$c_prelievo
```

```
if (TOTALE-$trasf) < 0 then abort
else begin
  update CONTO
     set TOTALE=TOTALE-$trasf
   where N_CONTO=$c_prelievo
  create PROC1
  send to PROC1($trasf,$c_deposito)
  commit
end
```

Il processo PROC1 creato dal ROOT è il seguente:

```
receive from ROOT($somma,$n_conto)
update CONTO
   set TOTALE=TOTALE+$somma
 where N_CONTO=$n_conto
```

Possiamo ora notare come il processo ROOT abbia il controllo su tutta l' esecuzione della transazione, dopo averla iniziata se si presenta un errore la termina, altrimenti crea il processo da eseguire sul sito remoto e gli passa i parametri necessari, quindi esegue il commit. Sul sito remoto abbiamo la procedura PROC1 che ha i propri parametri, esegue la parte di transizione che interessa quel sito, poi termina.

2.6.1 Transaction manager

L' esecuzione di una transazione distribuita avviene tramite un modulo del DDBMS denominato "distributed execution monitor" (figura 2.9), il quale è composto a sua volta da due componenti: *transaction manager* che coordina l' esecuzione delle operazioni sul database (le applicazioni) e uno *scheduler* che controlla la concorrenza dell' esecuzione delle transazioni per sincronizzare l' accesso al database. Può essere presente anche un modulo di recovery, che si occupa della gestione delle condizioni di errore e di ripristino.

Ogni transazione inizia in un sito che chiameremo "sito origine", il transaction manager di questo sito ne coordina l' esecuzione.

Trascuriamo per ora il caso di transazioni concorrenti, l' interfaccia tra il transaction manager e l' applicazione utente è composta da cinque comandi:

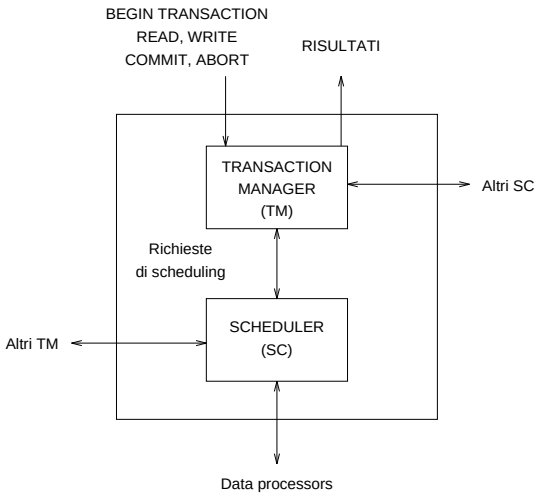


Figura 2.9. Schema del distributed execution monitor

1. `begin transaction`. Questo comando indica al transaction manager che sta per iniziare una nuova transazione, in questo istante vengono raccolte alcune informazioni come il nome dell' applicazione, il nome della transazione, etc ....

2. `read`. Se un dato è memorizzato localmente il suo valore è letto e ritornato all' applicazione. Altrimenti viene selezionata una copia del dato e ritornato il suo valore.

3. `write`. Il transaction manager si occupa di coordinare l' update del dato in tutti i siti in cui risiede (originale e copie replicate).

4. `commit`. Il transaction manager effettua la modifica fisica del database confermando tutti i dati alterati dalla transazione.

5. `abort`. Le modifiche ai dati vengono ignorate, il database fisico rimane inalterato.

Durante l' esecuzione di queste istruzioni il transaction manager può interagire con gli scheduler di diversi siti, il coordinamento rimane del sito di origine della transazione.

2.7 Controllo della concorrenza distribuita

In un sistema distribuito riveste fondamentale importanza la gestione della concorrenza delle transazioni, deve garantire che la consistenza del database sia mantenuta anche nell' ambiente distribuito e multiutente. Se le transazioni sono internamente consistenti, il metodo più semplice è quello di eseguirle una dopo l' altra sequenzialmente (*serializzabilità*), questo metodo assicura la consistenza di tutta l' applicazione, ma il troughput che si ottiene è il minimo. Il numero di transazioni concorrenti è uno dei parametri più importanti dei DDBMS, il meccanismo che controlla la concorrenza deve assicurare un buon equilibrio tra il numero di transazioni concorrenti e la consistenza del database. Gli algoritmi per il controllo della concorrenza possono essere di due tipi:

- *Pessimistici*: partono dal presupposto che molte transazioni andranno in conflitto, quindi sincronizzano l' esecuzione delle transazioni concorrenti subito dopo il loro inizio.
- *Ottimistici*: non molte transizioni andranno in conflitto, allora possono essere sincronizzate alla fine della loro esecuzione.

Il criterio di classificazione è la primitiva di sincronizzazione utilizzata: *locking* cioè le tabelle sono accedute in modo mutuamente esclusivo, oppure *timestamp ordering* in cui l' ordine di esecuzione delle transazioni viene riordinato secondo criteri specifici.

Una classificazione dei diversi algoritmi di controllo è in figura 2.10.

Con l' approccio locking based la sincronizzazione delle transazioni è fatta da meccanismi logici o fisici che bloccano alcune porzioni del database, in base alla posizione in cui l' algoritmo agisce possiamo avere un locking:

- Centralizzato: un sito della rete blocca le tabelle per l' intero database
- Copia principale: una delle copie (realizzate mediante replica) dei dati bloccati è la principale, tutte le transazioni che interessano quella porzione di dati devono essere autorizzate dal sito che contiene la principale. Se il database non è replicato il controllo delle tabelle viene distribuito tra i vari siti.
- Distribuito: il controllo del locking è condiviso tra tutti i siti della rete, se un sito deve utilizzare una porzione di dati deve ottenere il permesso da tutti i possessori delle tabelle.

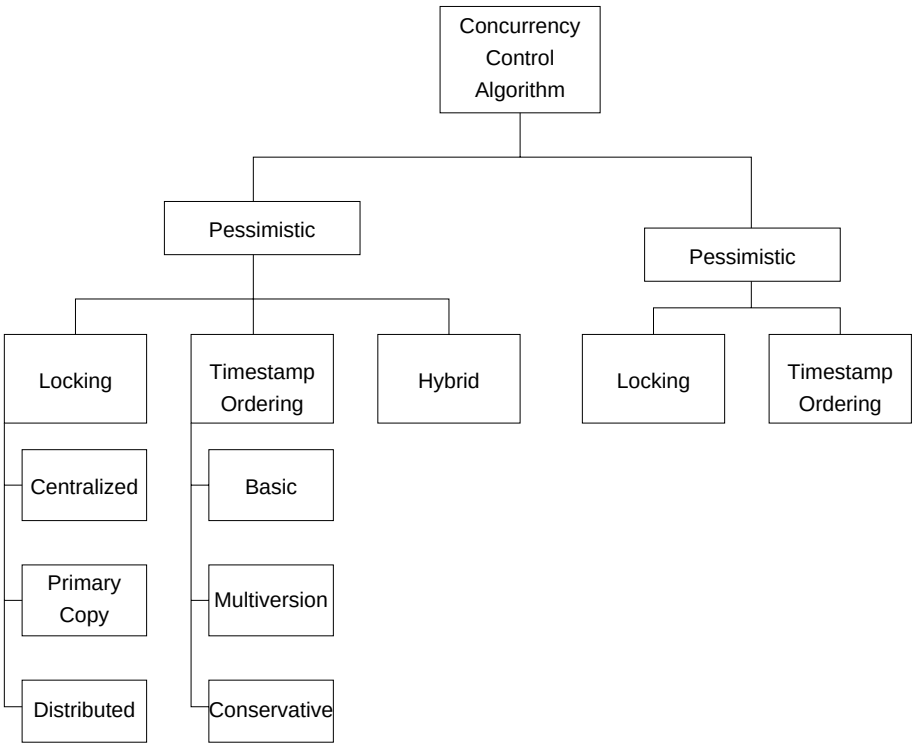


Figura 2.10. Classificazione degli algoritmi di controllo della concorrenza.

L' approccio timestamp-ordering coinvolge l' ordine di esecuzione delle transazioni, in modo da riorganizzarle per evitare conflitti. Gli algoritmi sono tre: Basic, Multiversion e Conservative.

2.7.1 Locking based concurrency control

Il principio su cui si basa questo tipo di controllo è assicurare che i dati condivisi da operazioni in conflitto tra loro siano acceduti da una operazione alla volta. Il lock viene settato utilizzando una transazione prima di accedervi, poi viene resettato al termine del suo utilizzo.

Esistono due tipi di lock utilizzabili: *read lock* (rl) e *write lock* (wl) applicabili ai dati, supponiamo ad esempio una transazione T_i che deve leggere dei dati contenuti in un blocco x, deve ottenere un read lock sul blocco x cioè: $\mathrm{rl}_i(x)$. I due tipi possono essere utilizzati anche da due transazioni concorrenti (T_i e T_j) secondo la seguente tabella di compatibilità:

	$\mathrm{rl}_i(x)$	$\mathrm{wl}_i(x)$
$\mathrm{rl}_j(x)$	compatibile	non compatibile
$\mathrm{wl}_j(x)$	non compatibile	non compatibile

Tabella 2.7.1: Tabella di compatibilità

Riassumendo: due transazioni concorrenti possono eseguire contemporaneamente un accesso ad una partizione di dati solo se è bloccato in lettura.

Esempio: consideriamo due transazioni:

T_1:	Read(x)	T_2:	Read(x)
	$x = x + 1$		$x = x \cdot 2$
	Write(x)		Write(x)
	Read(y)		Read(y)
	$y = y - 1$		$y = y \cdot 2$
	Write(y)		Write(y)
	Commit		Commit

Una possibile soluzione è la seguente:

$$S = \{wl_1(x), R_1(x), W_1(x), lr_1(x), wl_2(x), R_2(x), wl_2(x), lr_2(x),$$

$$wl_2(y), R_2(y), W_2(y), lr_2(y), C_2, wl_1(y), R_1(y), W_1(y), lr_1(y), C_1\}$$

Questa soluzione ha un difetto: quando l' algoritmo rilascia il lock (appena il comando associato è stato eseguito) la transazione ne richiede un' altro per il comando successivo, questo può aumentare la concorrenza, ma permette alle transazioni di interferire tra loro.

Per risolvere questo problema si utilizza un algoritmo denominato *two phase locking* (2PL).

Two phase locking algorithm

L' algoritmo 2PL non consente ad una transazione di richiedere un nuovo lock dopo aver rilasciato quello in uso. La transazione viene eseguita in due parti (figura 2.11):

1. Salita: tutti i lock vengono richiesti ed ottenuti.

2. Discesa: vengono rilasciati i lock.

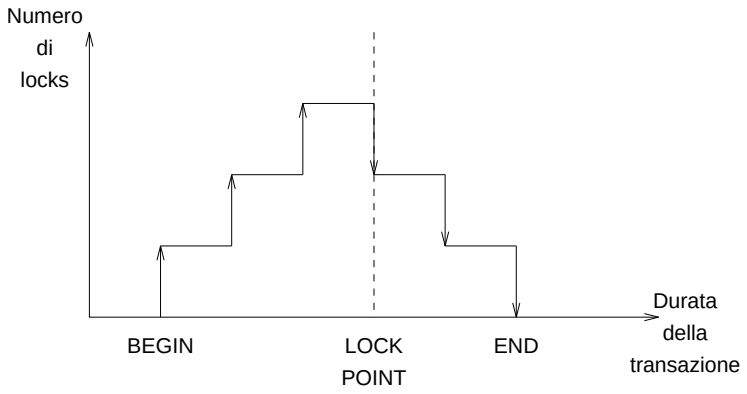


Figura 2.11. 2PL graph

Il punto di inversione tra salita e discesa viene chiamato *lock point*, da quel momento inizia il rilascio di tutti i lock acquisiti.

Il lock manager rilascia i lock appena l' accesso ai dati è terminato, questo permette alle altre transazioni in attesa dell' accesso di procedere ed acquisire a loro volta il lock.

E' difficile implementare questo algoritmo, infatti il lock manager deve sapere quando la transazione ha ottenuto tutti i lock che aveva richiesto e quando termina l' accesso ai dati e quindi la necessità del lock. Inoltre se una transazione termina con un abort subito dopo aver rilasciato il lock, questo deve portare all' abort anche delle transazioni che hanno acceduto a quei dati appena sbloccati (*cascading aborts*). A causa di queste difficoltà viene implementata una versione ridotta denominata *strict two phase locking* che rilascia tutti i lock nello stesso momento (figura 2.12), cioè al commit o all' abort della transazione.

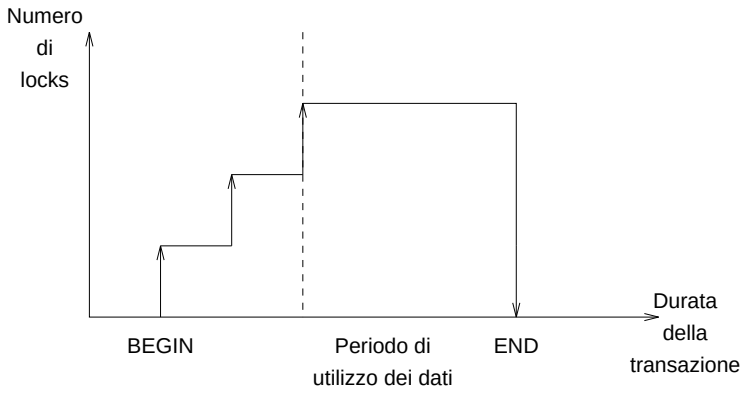


Figura 2.12. Strict 2PL graph

L' algoritmo 2PL può essere facilmente esteso ai DDBMS, sono possibili diverse alternative: la gestione centralizzata dei lock, la gestione "primary copy" e la gestione distribuita.

2PL centralizzato. La gestione centralizzata dei lock prevede che esista un sito della rete che si occupa della gestione dei lock di tutto il database distribuito. Il transaction manager degli altri siti deve comunicare con questo unico lock manager, per questo serve un protocollo di comunicazione che si occupi della gestione delle richieste dei lock.

Esaminiamo la struttura di comunicazione dell' algoritmo C2PL proposto in figura 2.13, i siti che eseguono le transazioni richiedono il lock dei dati al sito in cui risiede il lock manager, il quale in funzione dello stato attuale della porzione di dati rilascia il lock (compatibilmente con le situazioni presentate nella tabella 2.7.1) al sito periferico, il quale dopo averlo ricevuto esegue le operazioni e comunica il termine al lock manager il quale sblocca i dati.

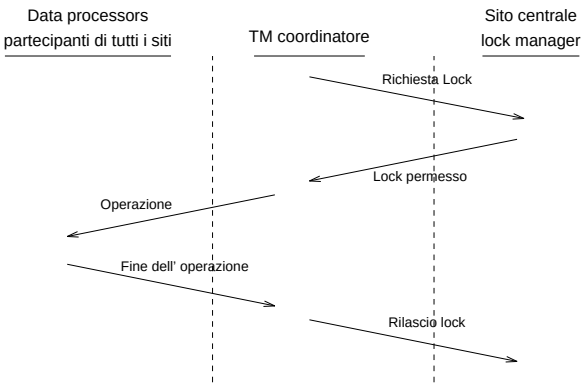


Figura 2.13. 2PL centralizzato

Il principale difetto di questo metodo è il collo di bottiglia che il sito con il lock manager rappresenta, al quale tutti i siti devono accedere, le prestazioni peggiorano soprattutto se il numero di transazioni è elevato. Inoltre un solo sito che si occupa della gestione rappresenta un punto debole nell' affidabilità dell' intero sistema, infatti in caso di guasti a questo sito o nei canali di comunicazione che lo collegano con gli altri, la gestione delle transazioni fallisce.

Primary copy 2PL. Il PC2PL è l' estensione del C2PL in modo tale da

evitare i problemi di prestazioni evidenziati. Il lock manager è implementato in più siti, ciascuno di questi è responsabile dei lock di una porzione di dati, i transaction manager inviano le richieste di lock e unlock al sito che si occupa del controllo di quei dati.

Il funzionamento è identico al 2PL quindi non lo analizzeremo nuovamente, l' unica differenza è che deve essere determinata la posizione di ogni copia primaria dei dati prima di inviare le richieste di lock al lock manager di quel sito.

Questo alogritmo è stato utilizzato nel DDBMS Distributed-INGRES.

Distributed 2PL. L' algoritmo D2PL prevede la presenza di un lock manager in ogni sito, pertanto se il database non è replicato l' algoritmo corrisponde al PC2PL. Se il database è replicato l' algoritmo è simile al C2PL, ma i messaggi che erano inviati al sito centrale il D2PL li invia a tutti i siti che hanno il lock manager.

2.7.2 Timestamp-based

Gli algoritmi timestamp-based non utilizzano la mutua esclusione come i precedenti, ma si basano sulla selezione a priori di un ordine di esecuzione favorevole. Il transaction manager assegna ad ogni transazione durante l' inizializzazione T_i un timestamp univoco $ts(T_i)$. Ordinando i timestamp possiamo serializzare le transazioni.

Il *timestamp* è un valore che serve ad identificare ogni transazione in modo univoco, possiede le seguenti propietà:

- *Unicità*: ogni timestamp deve essere univoco, in modo tale da permettere una identificazione delle transazioni associate.
- *Monotonia*: due timestap generati dallo stesso transaction manager devono essere monotoni crescenti.

I metodi per generare timestamp sono molteplici: il più diffuso nei sistemi centralizzati è l' utilizzo di un contatore globale che ad ogni assegnamento di un timestamp viene incrementato. Questo metodo nei sistemi distribuiti non è conveniente, è preferibile che ogni sito assegni autonomamente un timestamp basandosi su un contatore locale, per evitare conflitti globali ogni sito

appende al valore ottenuto dal contatore un proprio identificatore univoco nell' intero sistema distribuito.

Ora possiamo formalizzare il metodo di ordinamento dei timestamp: date due operazioni in conflitto tra loro O_{ij} e O_{kl} appartenenti rispettivamente alle transazioni T_i e T_k, O_{ij} è eseguita prima di O_{kl} se e solo se $ts(T_i) < ts(T_k)$. In questo caso possiamo dire che T_k è una transazione più recente di T_i.

Lo scheduler deve ripetere l' ordinamento dei timestamp ad ogni nuova transazione, in modo tale da evitare i conflitti, se la nuova operazione appartiene ad una transazione più recente di tutte quelle in conflitto è accettata, altrimenti viene rifiutata e tutta la transazione riparte con un nuovo timestamp. Il confronto tra i timestamps delle transazioni può essere fatto solo se lo scheduler ha ricevuto tutte le operazioni, se invece queste arrivano una per volta (come nelle applicazioni usuali) deve essere in grado di rilevare una operazione arrivata fuori sequenza.

Con riferimento alla figura 2.10 esaminiamo i tre tipi di algoritmi timestamp-based:

Basic Timestamp Ordering : è l' implementazione della regola di ordinamento dei timestamp esattamente come descritta dalla definizione. Il gestore delle transazioni assegna un timestamp ad ogni transazione, determina i siti in cui ogni dato è memorizzato ed invia le operazioni rilevanti a questi siti.

Le transazioni non aspettano i diritti di accesso, di conseguenza non sono possibili deadlocks, però una transazione può essere rifutata e quindi rieseguita diverse volte consecutive.

Conservative Timestamp Ordering : questo metodo cerca di ridurre l' overhead causato dal numero delle esecuzioni multiple delle transazioni. Per ridurre la probabilità di restart lo scheduler ritarda l' esecuzione di una operazione fincé non c'è la garanzia di non avere altre operazioni con timestamp minore. Utilizzando questo metodo è possibile evitare l' esecuzione multipla delle transazioni. Però il ritardo da introdurre può causare dei deadlock, per evitarlo utilizziamo un buffer delle operazioni di ciascuna transazione, quando possiamo stabilire un ordine tra queste operazioni in modo tale da non dover rifiutare la transazione le eseguiamo in questo ordine.

Multiversion Timestamp Ordering : è un altro tentativo di eliminare la

riesecuzione delle transazioni, ma è un metodo orientato soprattutto ai sistemi centralizzati. Il meccanismo di base è la creazione di "versioni" successive dei dati: in seguito ad un update il dato originario non viene modificato ma ne viene inserito uno completamente nuovo marcato con il timestamp della transazione che lo ha prodotto, così che le due versioni sono marcate con due timestamps diversi. L' esistenza delle versioni è trasparente all' utente, ma lo scheduler può eseguire read e write sui dati basandosi sul timestamp della transazione e di quello associato ai dati. La lettura dei dati è sempre eseguita, in quanto è possibile recuperare i dati con timestamp minore di quello dell' operazione di read. In scrittura la transazione viene rifiutata se lo scheduler ha già eseguito una lettura sui dati con timestamp maggiore del timestamp dell' operazione di write.

Occorre periodicamente ripulire il database dalle versioni, ma solo quando si è sicuri che nessuna transazione deve accedere ai dati prima di un determinato timestamp.

2.8 Reliability

L' attendibilità dei dati memorizzati è importante, e deve essere garantita anche se il sistema diventa inaffidabile, cioè anche se un componente del sistema distribuito si guasta un DDBMS affidabile deve poter continuare a soddisfare le richieste dell' utente senza violare la consistenza del database.

Ogni deviazione del sistema dal comportamento descritto nelle specifiche è da considerarsi come un guasto, può essere di due tipi: *permanente* quando è irreversibile, e richiede un intervento per ripararlo, *temporaneo* (o intermittente) se è dovuto a instabilità hardware o software o dipendente dalle condizioni ambientali (ad esempio l' aumento della temperatura), può essere riparato dopo aver individuato la causa del problema, nel caso del fattore ambientale invece può essere impossibile l' annullamento della condizione che provoca il malfunzionamento.

Il termine reliability si riferisce alla probabilità $R(t)$ che il sistema non presenti nessun problema durante un intervallo di tempo specificato $[0,t]$. Viene di solito utilizzato per descrivere sistemi che non possono essere riparati o che eseguono attività critiche cioè non è possibile interromperle per eseguire la riparazione. Per quantificare l' affidabilità di un sistema si utilizzano due

misure: *mean time between failures* MTBF e *mean time to repair* MTTR, indicano rispettivamente il tempo medio tra due guasti successivi e il tempo medio di riparazione, a volte è utile misurare anche il *mean time to fail* MTTF cioè il tempo medio entro cui avviene il primo guasto, tramite la relazione: $MTBF = MTTF + MTTR$ è possibile legarlo agli altri valori. Nei prodotti commerciali sono valori forniti dal produttore.

Per la progettazione di sistemi affidabili si seguono tre metodi:

- *fault tolerance* sono i sistemi progettati per riconoscere gli errori quando avvengono, in modo tale da eseguire determinate procedure per rimuovere o correggere l' errore prima che si trasformi in un guasto del sistema.

- *fault prevention* i sistemi sono progettati cercando di eliminare le cause di errore. Si possono utilizzare due tecniche:

  - *fault avoidance* la progettazione avviene in modo tale da evitare l' introduzione di errori nel progetto, questo si ha utilizzando metodologie di progetto particolari.

  - *fault removal* dopo aver applicato la tecnica precedente solo in parte (per non aumentare troppo i costi di progetto) si effettua una verifica (collaudo) sul prodotto per rimuovere eventuali cause di errore rimaste.

- *fault detection* dopo la progettazione si cerca ogni causa di errore e la si elimina. Questa di solito è utilizzata anche assieme ad una o più delle altre tecniche descritte.

Analizziamo ora i guasti specifici dei sistemi distribuiti di basi di dati, il sistema di recovery del DDBMS deve gestire quattro tipi di problemi:

1. Transaction failure: dipendono da diverse cause: dati errati in input, deadlock, controllo della concorrenza, lock, ..., e ogni ulteriore caso in cui la transazione non può terminare con un commit. L' approccio da utilizzare per risolvere questi problemi è terminare la transazione con un abort, quindi resettare il database ad uno stato consistente precedente all' inizio della transazione.

2. System failure: causati da problemi hardware (guasti fisici agli elaboratori, mancanza di energia elettrica) o software (bug nel programma o nel DBMS o nel sistema operativo). Il database si suppone integro, ad eccezione eventualmente del risultato delle transazioni in esecuzione al momento del guasto contenute in un buffer nella memoria centrale. Se il sistema è distribuito il sito in cui è avvenuto il problema diventa inacessibile dall' esterno, e in funzione della topologia della rete provocare una interruzione che determina due sottoreti isolate tra loro.

3. Media failure: la memoria di massa contentente di dati si danneggia, può avvenire sia per cause fisiche (danni alle testine di lettura/scrittura dei dispositivi, o guasti di natura magnetica) che per cause software (bug nel sistema operativo). Backup periodici e copie multiple dei dati (tecnologia RAID) permettono di recuperare in parte o completamente i dati. Di solito è possibile trascurare i guasti ai supporti, in quanto sono meno probabili degli altri tipi.

4. Communication failure: a differenza degli altri tre tipi che sono comuni anche ai sistemi centralizzati, i problemi relativi alle comunicazioni sono caratteristica dei sistemi distribuiti. La perdita di messaggi, la sequenza di arrivo errata o guasti alle linee di trasmissione sono i casi che si presentano più frequentemente. Alcuni problemi non sono di competenza del DDBMS, infatti i livelli inferiori delle architetture di rete di dovrebbero già occupare della verifica della correttezza della comunicazione. Per quanto riguarda i messaggi persi dovrà essere il DDBMS ad accorgersi delle incongruenze tra la situazione prima e dopo il guasto, e rilevare così la perdita di informazioni.

Ogni malfunzionamento in un sistema distribuito deve essere gestito sempre sotto due aspetti: sul sito locale (come un sistema centralizzato) e durante la comunicazione. Dovranno esserci quindi degli algoritmi che permettono di conservare l' integrità dei dati locale e dei protocolli di comunicazione che permettono di eseguire transazioni distribuite affidabili.

2.8.1 Protocolli locali

Su ogni sito esiste un componente del DDBMS denominato *local recovery manager* LRM, il quale deve mantenere l' atomicità e la persistenza delle

transazioni locali. L' LRM accede ad un buffer in memoria centrale (questo permette di aumentare le prestazioni) contenente delle copie dei dati acceduti più recentemente, questa memoria è utilizzata a pagine corrispondenti alle pagine del database (memorizzate stabilmente su supporto non volatile). Quando l' LRM vuole leggere una pagina di dati la richiede ad un gestore della memoria (chiamato anche *buffer manager*) il quale verifica se la pagina è già con contenuta nel buffer oppure no. Nel primo caso viene immediatamente resa disponibile, nel secondo deve essere recuperata dalla memoria di massa, copiata nel buffer e quindi messa a disposizione dell' LRM. Ovviamente ogni nuova pagina inserita nel buffer dovrà sostituire una già presente, la scelta della pagina da sovrascrivere è fondamentale per le prestazioni, occorre infatti eliminare la pagina che ha meno probabilità di essere acceduta nell' intervallo di tempo immediatamente seguente all' inserimento della nuova.

Nel caso di guasti che comportano la perdita del risultato di una o più transazioni si può utilizzare un *database log* che mantiene i cambiamenti allo stato del database, cioè ad ogni transazione si registrano nel database log tutte le informazioni necessarie per il ripristino di uno stato consistente. Quando occorre ricostruire una parte del database possiamo utilizzare un comando "redo" che partendo dai dati contenuti nel log ricostruisce l' effetto prodotto dalla corrispondente transazione sul database, al termine della ricostruzione abbiamo di nuovo il database in uno stato stabile. In modo simile il comando "undo" permette di eliminare dal database gli effetti prodotti da una errata transazione.

Per evitare che l' undo o il redo debbano scandire tutto il log alla ricerca dell' ultimo stato valido del database, possiamo utilizzare dei checkpoint che permettono di individuare subito uno stato consistente.

2.8.2 Protocolli distribuiti

Anche le transazioni distribuite devono mantenere le caratteristiche di atomicità e persistenza, opportuni protocolli devono controllare la transazione anche durante le comunicazioni tra i siti, in modo tale da poter sempre garantire che il database sia in uno stato consistente.

Le tecniche utilizzate sono principalmente tre:

1. *Commit protocols*: mantiengono l' atomicità della transazione distribuita anche quando coinvolge diversi siti, uno dei quali può fallire. Alcune

implementazioni di questo tipo di protocollo possono essere applicate anche ai DBMS centralizzati.

2. *Termination protocols*: utilizzati solo nei DDBMS assumono che se durante l' esecuzione di una transazione distribuita uno dei siti fallisce, anche gli altri devono terminare la transazione.

3. *Recovery protocols*: determinano come deve terminare la transazione in modo indipendente, cioè localmente. L' esistenza di questi protocolli può ridurre il numero di messaggi che percorre la rete durante un recovery.

Two-phase Commit Protocol

Il protocollo 2PC estende gli effetti del commit locale alle transazioni distribuite, viene richiesta ai siti coinvolti nell' esecuzione la disponibilità ad effettuare un commit prima che gli effetti della transazione diventino permanenti. La sincronizzazione tra i siti è necessaria in quanto alcuni scheduler possono non essere pronti a terminare la transazione col commit e di conseguenza per evitare il commit a cascata anche di transazioni valide occorre evitare che un dato modificato da una transazione che fallisce sia utilizzato da un' altra operazione prima del commit. Un' altra ragione per cui un sito partecipante può rifiutare la richiesta di commit è dovuta alla presenza di un deadlock che porta all' abort della transazione, in questo caso il sito può effettuare l' abort anche senza consultare gli altri siti, questo è detto *unilateral abort*.

Il 2PC viene eseguito con i seguenti passi (figura 2.14):

- Il coordinatore scrive il "begin commit" nel suo log.

- Quindi invia un messaggio "prepare" a tutti i partecipanti, informandoli dell' inizio del commit, quindi entra in uno stato di wait.

- Quando un partecipante riceve il messaggio controlla se può terminare la transazione con successo. Ci sono quindi i due casi:

    1. Si (commit), invia un messaggio "vote-commit" al coordinatore ed entra in uno stato di attesa.

2. No (abort), invia un messaggio "vote-abort" al coordinatore ed entra in uno stato di attesa. In questo caso il sito localmente può dimenticare la transazione, in quanto effettua un abort unilaterale.

- Se la decisione è di terminare con abort (è sufficiente un solo sito), allora il coordinatore invia un abort globale a tutti i partecipanti ed entra in uno stato "commit"

- I partecipanti possono eseguire un commit o un abort coerentemente con quanto inviato dal coordinatore, e ritornano un messaggio di conferma.

- Il coordinatore quando riceve tutte le risposte dei partecipanti termina la transazione.

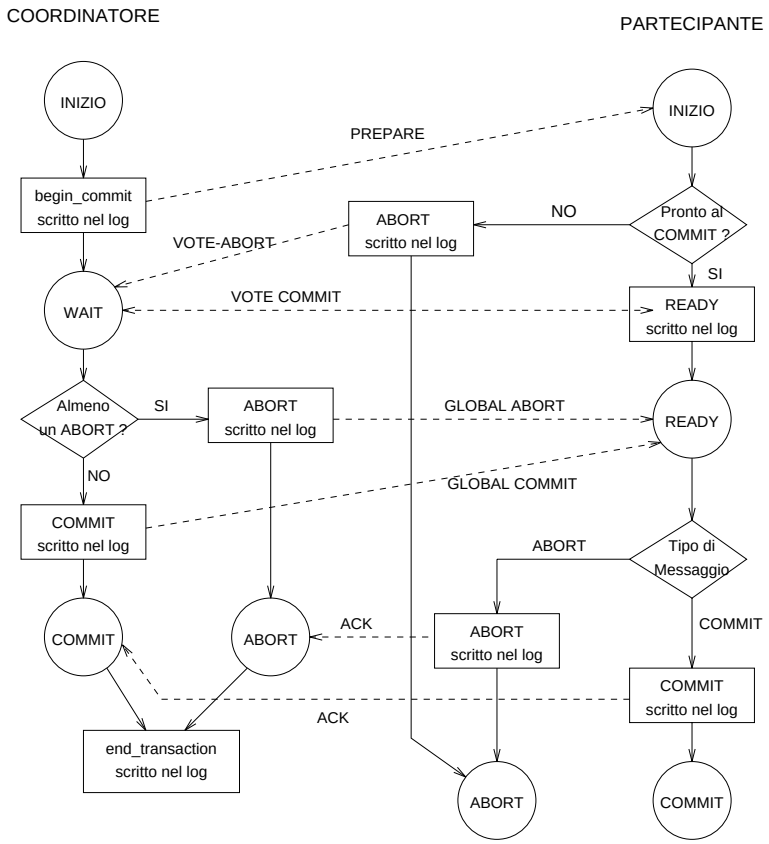


Figura 2.14. Funzionamento del two-phase commit protocol

La scelta sul modo di terminare la transazione è presa dal coordinatore seguendo due regole di base (che insieme vengono chiamate *global commit rule*):

1. Se anche un solo partecipante vota per l' abort, il coordinatore decide per l' abort globale.

2. Se tutti i partecipanti votano per il commit, il coordinatore decide per il commit globale.

Vediamo alcune osservazioni: la votazione è irrevocabile, non sono possibili cambiamenti nelle decisioni prese. Inoltre tutti gli stati di attesa sono controllati anche da dei timer, per evitare effetti indesiderati vengono imposti dei timeout da rispettare, entro cui la risposta deve arrivare.

E' possibile implementare il 2PC seguendo diverse topologie, quello descritto fino ad ora è il *centralized two-phase commit*, in quanto la comunicazione avviene solo tra un sito centrale che svolge la funzione di coordinatore e gli altri siti del sistema.

Una alternativa è il *linear 2PC* (figura 2.15) in cui la comunicazione avviene da un sito ad un altro, seguendo un ordine prestabilito.

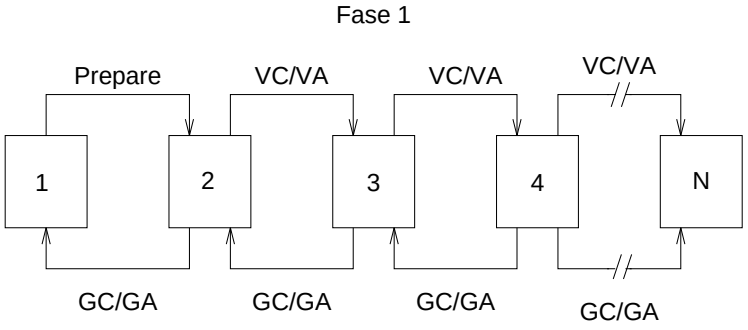


Figura 2.15. Linear two-phase commit protocol

Il coordinatore è il numero 1, le richieste passano dal primo all' ultimo sito, quindi ritornano indietro le votazioni locali, e ne viene preparata una globale. Il coordinatore invia il messaggio "prepare" al secondo sito, il quale se non è pronto per il commit invia un messaggio "vote-abort" (nel caso contrario un "vote-commit") al successivo, così fino all' ultimo sito in cui termina la prima fase. Se N decide per il commit invia al sito $N-1$ il "global commit"

(GC) (in caso contrario un "global abort" GA), il messaggio quindi viene propagato fino al primo sito.

Lo svantaggio della struttura lineare è che non è possibile alcun tipo di parallelismo, quindi è un metodo naturalmente lento. Può trovare tuttavia impiego nelle reti che non dispongono della possibilità del broadcast.

Un' altra struttura è il *distributed 2PC* (figura 2.16), che a differenza del precedente coinvolge tutti i siti nella prima fase, così che la decisione viene presa indipendentemente. Non servè più la seconda fase, in quanto i partecipanti hanno già preso una decisione. In dettaglio ecco la sequenza di operazioni che viene seguita:

- Il coordinatore invia il messaggio "prepare" a tutti i partecipanti contemporaneamente.
- Ogni partecipante prende autonomamente una decisione sul commit o sull' abort, e la invia a tutti gli altri partecipanti come "vote-abort" o "vote-commit".
- Ogni partecipante aspetta i messaggi dagli altri, ed esegue un commit o un abort seguendo la regola globale.

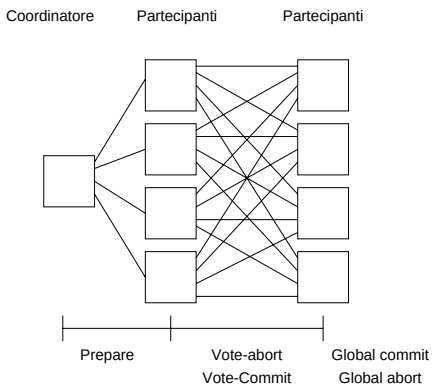


Figura 2.16. Struttura del distributed two-phase commit protocol

Si nota che non è necessaria la seconda fase, in quanto ogni sito prende in proprio sia la proposta temporanea che la decisione globale.

Osserviamo che in questa struttura (come nella precedente) è necessario che ogni sito componente il sistema sia a conoscenza dell' identità degli altri partecipanti (o di uno solo nel caso del linear 2PC). Questo problema può

essere risolto utilizzando una lista dei partecipanti alla transazione creata dal coordinatore e spedita assieme al messaggio "prepare". Questo problema non si pone nella versione centralizzata, dove comunque il controllore (che è il sito centrale) conosce i siti periferici.

Variazioni del 2PC

Sono state proposte diverse varianti ai protocolli base del 2PC per incrementare le prestazioni. Questo può essere fatto in due modi:

1. Riducendo il numero di messaggi trasmessi tra coordinatore e partecipanti (*presumed abort*).
2. Riducendo il numero di scritture sui log (*presumed commit*).

La prima variante parte dal presupposto che quando un partecipante interroga il coordinatore per conoscere i risultati della transazione e il coordinatore non ha questi dati in memoria virtuale, la risposta sarà sempre l' abort. Questo funziona fino a quando il coordinatore rimuove le informazioni della transazione in seguito ad un commit, in ogni caso questo non succede finché tutti i partecipanti non spediscono l' acknowledge. Questa tecnica permette di risparmiare sul numero di messaggi da scambiare quando una transazione deve terminare con abort.

Il secondo metodo parte dal presupposto che non ci sono informazioni sulla transazione e quindi deve essere considerata commit. Non è il duale della precedente metodologia, in quanto il commit non è forzato immediatamente, inoltre il commit non richiede l' acknowledge. Il coordinatore invia il messaggio "prepare" aggiungendo i nomi di tutti i partecipanti e si pone in attesa, quando questi ricevono il messaggio decidono sulla fine della transazione e la comunicano al coodinatore mediante i messaggi "vote-abort" e "vote-commit". Quando il coordinatore riceve le votazioni di tutti i partecipanti decide per il "global-abort" o per il "global-commit". Nel caso del commit dimentica la transazione e aggiorna il database. Se il risultato è abort termina la transazione e invia un acknowledge ai partecipanti i quali a loro volta terminano l' esecuzione.

2.8.3 Gestione dei guasti ai siti

Per poter controllare un guasto ad un sito partecipante ad una transazione occorre utilizzare un protocollo non bloccante, cioè che non comprometta la

conclusione della transazione. Mentre è dimostrato che un tale protocollo è realizzabile nel caso di un singolo sito guasto, non è ancora possibile affermare con certezza la fattibilità di un protocollo non bloccante quando più di un sito si guasta. Gli algoritmi che descriveremo sono tutti basati sul 2PC e consideriamo solo il caso di un singolo sito danneggiato.

Meccanismi di Timeout

La prima estensione al 2PC per renderlo non bloccante è già stata introdotta, e si basa sull' utilizzo di timeout sia nel sito coordinatore che sui partecipanti. In questo modo è possibile forzare l' uscita dagli stati di attesa anche nel caso in cui un sito, per qualunque motivo, non risponda entro tempi accettabili.

Consideriamo il diagramma a stati di figura 2.17 che rappresenta le possibili transizioni di stato del 2PC sia nel coordinatore che nei partecipanti.

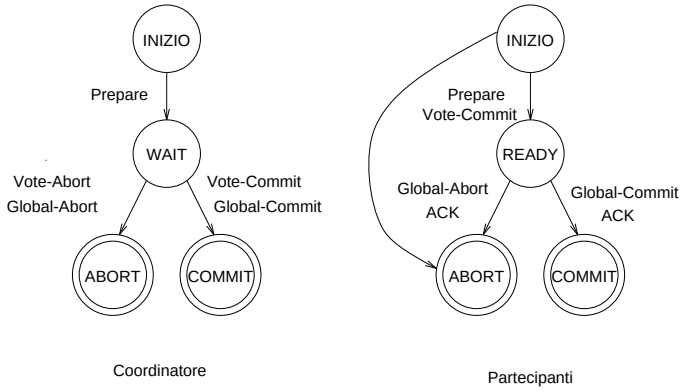


Figura 2.17. Transizioni di stato nel protocollo 2PC

Distinguiamo i timeout nei due tipi di siti della rete:

Timeout sul Coordinatore. Ci sono tre stati in cui il coordinatore si pone in attesa: WAIT, COMMIT e ABORT. Il timeout degli ultimi due stati è gestito nello stesso modo.

1. Timeout nello stato WAIT: il coordinatore sta aspettando la decisione locale dei partecipanti, e non può effettuare il commit unilaterale fino a quando la global commit rule non è soddisfatta. Può decidere solo per l' abort globale.
2. Timeout negli stati COMMIT o ABORT: in questo caso il coordinatore non può sapere se le procedure per il commit o l' abort

sono state terminate dai LRM nei siti partecipanti, quindi invia ripetutamente dei commit o abort globali ai siti che non rispondono.

Timeout sui partecipanti. Possono bloccarsi in due stati: iniziale (INIT) e READY:

1. Timeout nello stato INIT: il partecipante stà aspettando per un un messaggio "prepare", questo significa che il coordinatore non ha completato correttamente la fase di inizializzazione. Il partecipante può decidere unilateralmente di terminare con abort la transazione.
2. Timeout nello stato READY: il partecipante ha votato il commit, ma non conosce ancora la decisione globale del coordinatore. Il partecipante non può prendere la decisione autonomamente, quindi in questo caso rimane bloccato fino a quando qualcuno (il coordinatore o un altro partecipante) gli comunica il termine della transazione.

Protocolli di recovery

Vediamo ora che protocollo può seguire un sito per recuperare uno stato consistente dopo un errore. Questi protocolli devono essere indipendenti e mantenere l' atomicità della transazione distribuita, e questo non è sempre possibile.

Consideriamo ancora il diagramma di figura 2.17 e supponiamo che la scrittura di un record sia atomica e la transizione di stato avvenga dopo la trasmissione del messaggio di risposta.

Consideriamo quindi i possibili stati in cui è possibile avere una condizione di errore nei due diversi ruoli della transazione.

I casi possibili sul coordinatore:

1. Errore nello stato INIT: il guasto avviene quindi prima dell' inizializzazione della procedura di commit. Dopo il recovery è sufficiente far ripartire la procedura.
2. Errore nello stato WAIT: il coordinatore ha spedito il "prepare", dopo il recovery deve ricominciare il processo per la transazione dall' inizio inviando di nuovo il messaggio "prepare".

3. Errore nello stato COMMIT o ABORT: il coordinatore ha informato i partecipanti della decisione di terminare la transazione, dopo il recovery non occorre fare nulla se sono arrivati tutti gli acknowledgment, altrimenti si utilizza il protocollo di terminazione basato sui timeout.

I casi possibili sui partecipanti:

1. Errore nello stato INIT: dopo il recovery il partecipante dovrebbe effettuare un abort unilaterale, questo è accettabile in quanto il coordinatore è nello stato WAIT di conseguenza se non riceve nulla dal sito scatta il timeout e la transazione termina.

2. Errore nello stato READY: dopo il recovery il partecipante deve gestire la situazione come se fosse scattato un timeout nello stato di READY.

3. Errore negli stati COMMIT o ABORT: questi sono stati finali, di conseguenza non è necessaria alcuna operazione.

Sono possibili altre situazioni che possono essere interrotte da errore (scritture sui log, preparazione dei messaggi, ...), di solito sono gestite come un errore nel corrispondente stato del sito e gestite di conseguenza come descritto.

Capitolo 3

La replica nei DDBMS commerciali

I sistemi DDBMS commerciali, quali SQL Server 6.x, ORACLE7 Symmetric Replication e Sybase 10.0 Replication Server permettono una grande flessibilità sulla definizione dei dati di replica.

Ogni sito di replica può decidere sia riguardo i siti primari a cui richiedere i dati, sia riguardo alle specifiche tabelle da replicare. SQL Server utilizza la terminologia di articolo (partizione di una tabella) e di pubblicazione (collezione di articoli), garantendo la massima flessibilità riguardo alle pubblicazioni da ricevere e, all'interno di una pubblicazione, riguardo ai singoli articoli.

In particolare è possibile definire: partizione verticale della tabella, partizione orizzontale e partizione mista:

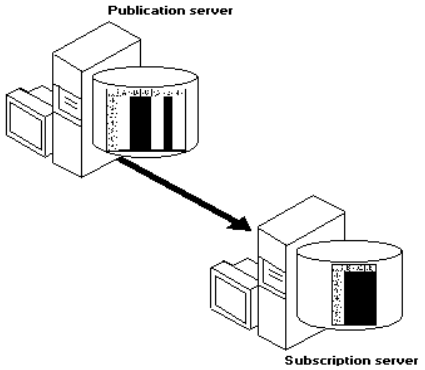


Figura 3.1. Partizione verticale

- *Partizione verticale della tabella.* Possiamo decidere di replicare soltanto alcune colonne di una tabella, cioè solo alcuni attributi (fig. 3.1). Ricordiamo che la partizione deve contenere la chiave primaria della tabella.

- *Partizione orizzontale della tabella.* Possiamo decidere di replicare soltanto alcune righe di una colonna, selezionando rispetto a particolari caratteristiche degli attributi (fig. 3.2).

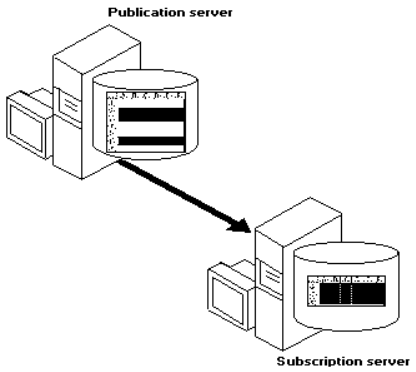


Figura 3.2. Partizione orrizzontale

- *Partizione mista.* Possiamo combinare la partizione orizzontale e verticale per ottenere l'insieme di dati desiderati per la replica (fig. 3.3).

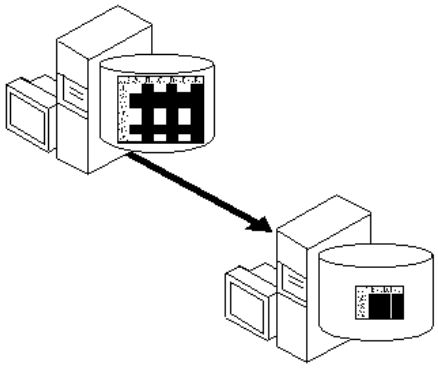


Figura 3.3. Partizione mista

In generale, la modalità di replica dei dati in applicazioni distribuite può essere suddivisa in due categorie:

- *Tight consistency*: In questo tipo di modello viene mantenuta, in ogni istante, la consistenza dei dati replicati rispetto all' originale. Per rispondere ad una esigenza così stringente, il sistema deve essere dotato di una rete ad alta velocità (tipicamente una LAN), prevedere l' indisponibilità di accesso ai dati durante tutti i periodi di aggiornamento (ridotta disponibilità) ed un protocollo *two phase commit* (2PC) per garantire l' integrità e la consistenza dei dati.

- *Loose consistency*: è un modello di replica che ammette un periodo di latenza tra il momento in cui i dati originali sono modificati ed il momento in cui tutte le copie vengono aggiornate. Naturalmente non viene garantita, in ogni istante, l' uguaglianza dei dati replicati rispetto all' originale.

Il vantaggio del modello loose consistency è quello di supportare anche reti LAN o WAN piuttosto lente ed eventualmente non collegate in modo permanente. Inoltre garantisce una maggiore disponibilità d' uso del database ed una maggiore scalabilità rispetto al tight consistency.

L' interesse delle applicazioni considerate ricade nel modello loose consistency, poichè si ipotizza una rete di DBMS debolmente connessi, tipocamente collegati da reti a velocità medio/bassa (quindi LAN, WAN o collegamenti non permanenti.

3.1 Il sistema di replica di SQL Server 6.5

SQL Server 6.5 Replication è basato sulla modalità Loose Consistency e sulla metafora Publishing / Subscribing. Ciò significa che il flusso dei dati è unidirezionale, dal publisher ai subscriber, ed in generale le applicazioni considerano i dati replicati ai subscriber come read-only (anche se in realtà sui dati replicati non abbiamo vincoli di update per permettere al subscription server di rendere persistenti le variazioni giunte dal publisher).

Il processo di replica in SQL Server 6.5 avviene tramite due operazioni distinte e sequenziali:

1. Processo di Sincronizzazione

2. Processo di Replica

Durante il processo di sincronizzazione, che viene eseguito una sola volta durante la configurazione dell'applicazione, l'obiettivo è di inviare a tutti i subscriber una copia completa dei dati (e dello schema) presenti nel publication server.

Terminata questa prima parte, che è obbligatoria, può iniziare il processo di replica dei dati che avviene ad intervalli di tempo definiti secondo le due diverse modalità:

- Transaction Based: vengono memorizzate tutte le transazioni inerenti la pubblicazione in esame che verranno spedite (al tempo opportuno) ai subscriber per la riesecuzione remota.
- Scheduled Refresh: al tempo opportuno (definito da utente) tutti i dati della pubblicazione (l'ultimo aggiornamento) vengono inviati ai server remoti.

3.1.1 La metafora publishing/subscribing

Una possibile terminologia dei modelli di replica è basata sulla metafora publishing/subscribing:

- Quando il dato è reso disponibile per la replica si dice che viene pubblicato (*published*). Il server che contiene il database sorgente e rende disponibile il dato è detto *publication server* o *publisher*.
- L' unità base di replica è chiamata articolo (*article*). L' articolo contiene i dati relativi ad una sola tabella e deve appartenere ad una publication.
- Una *publication* è una collezione di articoli (almeno uno). Una o più publication possono essere create da ciascun utente di un publisher.
- Quando un server richiede una pubblicazione diventa un *subscriber* (o *subscription server*). Un subscriber può richiedere la replica di alcune pubblicazioni (anche tutte) e di specifici articoli (anche tutti) all' interno di una pubblicazione.

3.1.2 Componenti del sistema di replica

I principali componenti sono:

- Synchronization Process Prepara i file di sincronizzazione iniziale contenenti sia lo schema (intensione) che i dati (estensione) delle tabelle da replicare; memorizza i file nella directory di lavoro di SQL Server sul distribution database e registra i job di sincronizzazione nel distribution database. Il processo di sincronizzazione ha effetto solo sui nuovi subscriber.

- Distribution Database E' uno "store-and-forward" database che mantiene traccia di tutte le transazioni che sono in attesa della distribuzione ai subscriber. Il Distribution Database riceve le transazioni speditegli dal publisher tramite il log reader process e le mantiene sino a quando il processo di distribuzione le porta ai subscriber. Il Distribution Database viene utilizzato solo per i processi di replica e non contiene dati e tabelle degli utenti.

- Log Reader Process Il Log Reader Process è il processo incaricato all'invio delle transazioni marcate dal Transaction Log del publisher come coinvolte al processo di replica, al Distribution Database, dove le transazioni sono poste in attesa per la distribuzione ai subscriber.

- Distribution Process E' il processo che porta le transazioni ed i job di sincronizzazione iniziale mantenute nelle tabelle del Distribution Database ai subscriber (in particolare nelle tabelle di destinazione dei Database ai siti di replica).

Tutti e tre i processi (Log Reader Process, Synchronization Process, Distribution Process) sono presenti nel distribution server (come sottosistema di SQL Executive) e possono essere utilizzati solamente dal system administrator.

3.1.3 Ruoli dei server nella replica

Il server può assumere tre ruoli differenti nel processo di replica di SQL Server.

- Publication Server (Publisher) E' il server che rende disponibili i dati per la replica. Il publication server gestisce il publication database, genera i dati da pubblicare estraendoli dalle tabelle opportune, invia copia di tutti i cambiamenti avvenuti al distribution server. Viene lasciata ampia libertà sulla configurazione dei dati pubblicati:

  - Secure publications: ciascuna pubblicazione ha uno stato di sicurezza marcato tipo unrestricted (default) oppure restricted. Una pubblicazione di tipo unrestricted è visibile e può essere sottoscritta da tutti, mentre una pubblicazione restricted limita la possibilità di sottoscrizione ad un elenco di server specificato.
  - Publication of vertically partitioned tables: possiamo creare articoli di replica costituiti da alcune colonne di una tabella (frammentazione verticale).
  - Publication of horizontally partitioned tables: possiamo creare articoli di replica costituiti da alcune righe di una tabella (frammentazione orizzontale).
  - Publication of mixed partitioned tables: possiamo partizionare una tabella in modo verticale e orizzontale contemporaneamente.

- Distribution Server (Distributor) E' il server che contiene il distribution database: riceve tutte le transazioni relative ai dati pubblicati, le memorizza nel suo distribution database e, al momento opportuno, dipendentemente dal tipo di replica definito, trasmette ai subscription server le transazioni o i dati relativi alla pubblicazione. Può risiedere sia nella stessa macchina del publisher che in una separata.

- Subscriber Server (Subscriber) E' il server che riceve e gestisce i dati replicati. Il progettista può configurare in modo flessibile i dati da sottoscrivere:

  - Selective subscription to publications: un subscription server può decidere di sottoscrivere nessuna, alcune o tutte le pubblicazioni di un publisher
  - Selective subscription to articles: un subscription server può decidere di sottoscrivere alcuni o tutti gli articoli di una pubblicazione.

Molto spesso i ruoli di publisher e di distributor coesistono nella stessa macchina. Inoltre il ruolo di publisher e subscriber non sono esclusivi ed un singolo server può eseguire entrambi (naturalmente riferendosi a pubblicazioni diverse).

3.1.4 Processo di sincronizzazione

Il processo di sincronizzazione iniziale viene eseguito in fase di configurazione dell'applicazione ed assicura che lo schema della tabella ed i dati del database sul sito di pubblicazione e sul sito di replica siano identici: terminato il processo di sincronizzazione, i subscriber sono pronti a ricevere gli aggiornamenti dei dati modificati tramite il processo di replica.

Infatti, quando viene creata una pubblicazione, una copia dello schema (compresi gli eventuali indici) e dei dati viene memorizzata su file nella directory di lavoro di SQL Server (nei file .SCH e .TMP rispettivamente). Questi file rappresentano l'insieme di sincronizzazione e vengono creati per ciascun articolo della pubblicazione. Dopo aver creato l'insieme dei file di sincronizzazione, il processo di sincronizzazione può portare una copia dei file a ciascun Subscriber che, in questo modo, dispone di una copia esatta dei dati e dello schema del Publisher.

Non appena il Publisher ha generato l'insieme di sincronizzazione, tutti gli aggiornamenti, inserimenti e cancellazioni dei dati pubblicati sono memorizzati dai processi di Log descritti, ma potranno essere ricevuti da ciascun Subscriber per aggiornare i dati di replica solamente quando sarà terminato il proprio processo di sincronizzazione iniziale. In questo modo SQL Server garantisce che le modifiche ai dati di replica siano portate al Subscriber quando quest'ultimo possiede una copia esatta dei dati attuali nel Publisher. Inoltre, quando durante la sincronizzazione l'insieme dei dati viene distribuito, solamente quei Subscriber che sono in attesa della sincronizzazione iniziale ricevono i dati; gli altri Subscriber, che hanno già terminato la sincronizzazione oppure hanno ricevuto le ultime modifiche dei dati pubblicati non compiono il processo di sincronizzazione. Poiché è SQL Server che gestisce automaticamente la coda dei Subscriber in attesa, il carico di lavoro dovuto alla sincronizzazione viene ridotto.

La sincronizzazione iniziale può essere compiuta sia in modo automatico che manuale: in particolare si può fissare l'intervallo di tempo tra due sincronizzazioni successive delle nuove pubblicazioni. Tutti gli elementi di una pubblicazione (gli articoli) vengono sincronizzati simultaneamente, in modo da preservare l'integrità referenziale dei dati.

Automatic Synchronization

La sincronizzazione automatica viene eseguita da SQL Server: per fare questo viene mantenuta su file una copia (snapshot) della tabella e dei dati degli articoli da pubblicare. Il processo di sincronizzazione crea un job di sincronizzazione che viene posto nel Distribution Database; quando il job viene attivato, invia i file di sincronizzazione a tutti i subscription database che sono in attesa di sincronizzazione. Quindi il job di sincronizzazione viene inviato dal Distribution Database come se fosse un qualsiasi altro job di aggiornamento.

Come detto, quando il processo di sincronizzazione iniziale viene inviato ai siti remoti, solo i subscriber che sono in attesa partecipano alla sincronizzazione, gli altri restano indifferenti. Questo permette di ridurre i carichi di lavoro.

Manual Synchronization

La sincronizzazione manuale è eseguita dall'utente. Come per la sincronizzazione automatica, il publisher genera i file di sincronizzazione contenenti gli snapshots dello schema e dei dati; nel processo manuale l' utente ottiene una copia su nastro dei file e si incarica di inserirli nei subscription server. Al termine del processo di sincronizzazione manuale l' utente comunica a SQL Server che il processo è terminato correttamente e si può procedere con la fase di replica (analogamente a quello che faceva direttamente il sistema nel processo automatico).

La sincronizzazione manuale è particolarmente utile quando la rete che collega publisher e subscriber è lenta, costosa oppure la quantità dei dati da copiare è molto elevata.

No Synchronization

Quando si dichiara una subscription, possiamo specificare di non volere la sincronizzazione di un particolare articolo. In questo caso SQL Server assume che la sincronizzazione sia sempre terminata, in modo che ogni variazione dei dati sul publisher viene comunicata ai subscriber durante il primo processo di replica successivo alla definizione della subscription (senza aspettare il completamento della sincronizzazione).

Nel caso di no synchronization deve essere il progettista della pubblicazione a farsi carico della consistenza dei dati replicati in fase iniziale. Il vantaggio di questa modalità risiede nella maggiore agilità del processo di copia dei dati replicati, in quanto non compare il sovraccarico dovuto alla sincronizzazione.

Snapshot Only

Quando si definisce una subscription, esiste l'opzione Snapshot Only per la quale SQL Server sincronizza gli articoli pubblicati con le tabelle di destinazione e ripeterà l'operazione ad intervalli definiti nel tempo (a scadenza giornaliera, mensile, ...). In questo modo le modifiche sui dati del publisher non vengono inviate alle repliche, che vengono aggiornate solo durante la successiva operazione di sincronizzazione. La metodologia è stata introdotta come Replica Scheduled Refresh.

3.1.5 Processo di replica delle pubblicazioni

Come detto, la replica consiste nell'allineamento delle copie dei dati presenti ai siti remoti con l'ultima versione aggiornata presente nel publication server. Le modalità previste sono due: Transaction Based e Scheduled Refresh.

Transaction Based

Per le pubblicazioni definite Transaction Based l'obiettivo è quello di trasmettere ai siti remoti la sequenza corretta delle transazioni avvenute sui dati di una pubblicazione. Quando giungono ai siti remoti, tali transazioni vengono eseguite e, poiché si parte da una situazione di dati sincronizzati, il risultato finale è quello di avere in ciascun sito una copia dei dati presenti nel publisher.

La determinazione dell'insieme delle transazioni spedite durante il processo di replica è basato sui Log file: esiste un processo, chiamato Log Reader Process, che compie il monitoraggio dei log delle transazioni dei database abilitati alla pubblicazione. Quando una transazione viene compiuta su di una tabella di tipo published, tale transazione viene marcata per la replica ed inviata (dal Log Reader Process) al distribution database. Le transazioni vengono qui mantenute in attesa di poter essere inviate ai subscriber per l' aggiornamento dei dati di replica.

Naturalmente solo le transazioni che hanno terminato con il commit sono spedite ai server di replica; inoltre, poiché l'invio delle transazioni è basato sul log, siamo sicuri che le transazioni sono spedite dal distribution database e ricevute dal subscriber nello stesso ordine in cui vengono eseguite dal publisher e quindi portano le varie copie dei dati nello stesso stato dell' originale.

Scheduled Refresh

Per le pubblicazioni definite Scheduled Refresh non interessa conoscere le transazioni che modificano i dati poiché vengono trasmessi ai subscriber tutti i dati che appartengono alla pubblicazione. L'operazione viene eseguita ad intervalli prefissati dall'utente ed equivale, sostanzialmente, ad una sincronizzazione in cui vengono inviati solo i dati e non lo schema delle tabelle.

3.1.6 Modalità di subscription: PUSH e PULL

Per poter ricevere la copia dei dati, un server deve definire una subscription verso le pubblicazioni di interesse: esistono due modalità di abbonamento (tipo push e pull) a seconda che l'attenzione sia rivolta al publication server o al subscription server.

Push Subscription

Una subscription di tipo push viene utilizzata quando l'interesse dell' applicazione è posto sul publication server. La copia dei dati di una applicazione avviene attraverso l'invio contemporaneo dei dati stessi da parte del publisher a tutti i subscriber presenti.

Il principale vantaggio dell' utilizzo del "push" è la semplificazione e la centralizzazione delle procedure di replica, non dovendo gestire separatamente ogni sito di replica.

Pull Subscription

Una subscription di tipo pull viene utilizzata quando l'interesse dell' applicazione è posto sul subscription server. La replica è basata sulla richiesta da parte di un subscription server ad un publisher per l'invio della replica di una pubblicazione.

Il vantaggio risiede nella maggiore autonomia di un subscriber nei confronti delle operazioni di replica: ogni sito di replica può decidere quando e quali pubblicazioni (e articoli) richiedere tra quelli in abbonamento, escludendo quelli di minor interesse.

3.1.7 Tipologie di replica

Central Publisher

Lo scenario è quello di un publisher centralizzato che replica i dati ad un numero N di subscriber. Il publisher è il proprietario (primary-owner) dei dati, mentre i subscriber utilizzano le informazioni in modalità read-only. Il Distributor Server (che invia fisicamente i dati) può risiedere sia nel publisher che in una stazione separata, quando i carichi di lavoro rendono pesante la gestione su piattaforma singola.

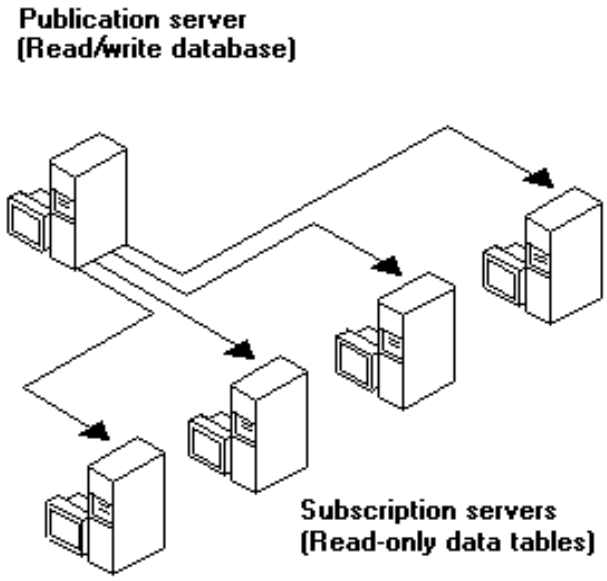


Figura 3.4. Tipologia central publisher

Publishing Subscriber

In questa situazione abbiamo due server che pubblicano gli stessi dati: il publisher invia i dati ad un solo subscriber il quale ripubblica i dati a tutti gli altri subscriber. Questo metodo risulta utile quando il publisher invia i dati

su di una rete lenta e costosa: utilizzando il subscriber come sender/receiver possiamo spostare il carico di lavoro in una parte della rete con caratteristiche superiori (di velocità, di costi, ...).

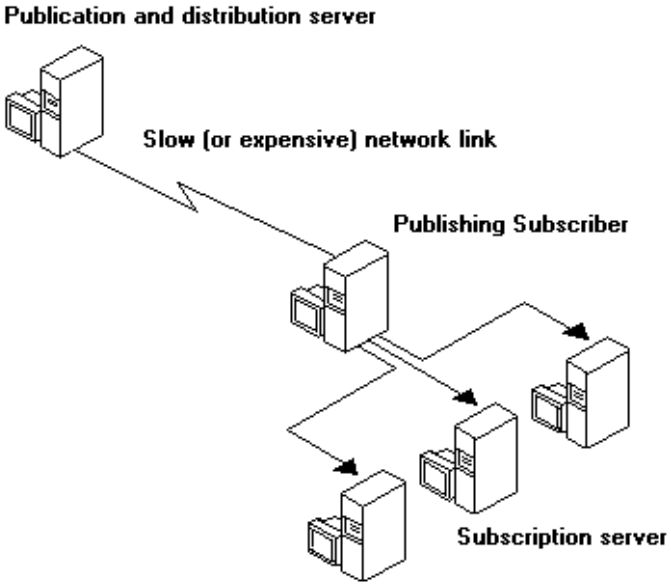


Figura 3.5. Tipologia Publishing Subscriber

Central Subscriber

Abbiamo diversi publisher che replicano i loro dati in una tabella di destinazione comune di uno stesso subscriber. Questa tabella di destinazione è ottenuta dall'unione delle varie tabelle dei publisher (che non sono tra loro sovrapposte) e ciascuna partizione della tabella di destinazione viene aggiornata da un publisher in tempi e modalità che possono essere diverse in funzione del publisher considerato.

Multiple Publisher of One Table

In questo scenario abbiamo una tabella che è mantenuta su vari server. Ciascun server è proprietario di una partizione orizzontale della tabella (per la quale è publisher) mentre è subscriber verso gli altri server per la rimanente parte dei dati. Ogni server può aggiornare i dati di cui è proprietario e vede (read/only) i rimanenti dati: le operazioni riguardanti i permessi sulle modifiche ai dati sono regolate da Stored Procedures.

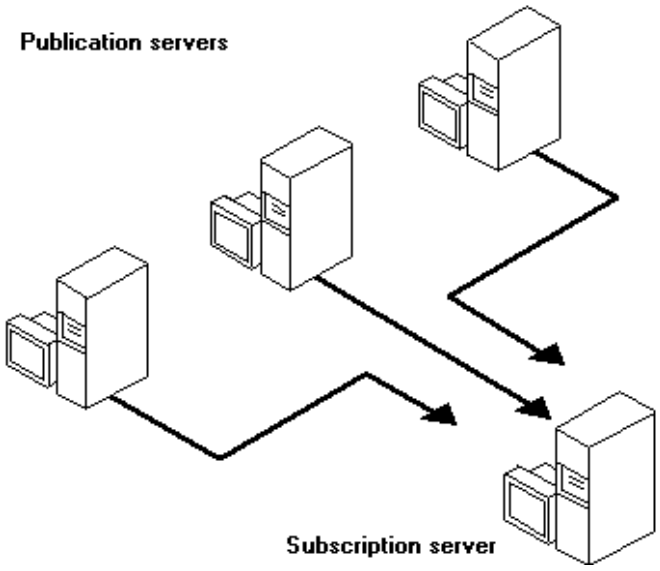


Figura 3.6. Tipologia central subscriber

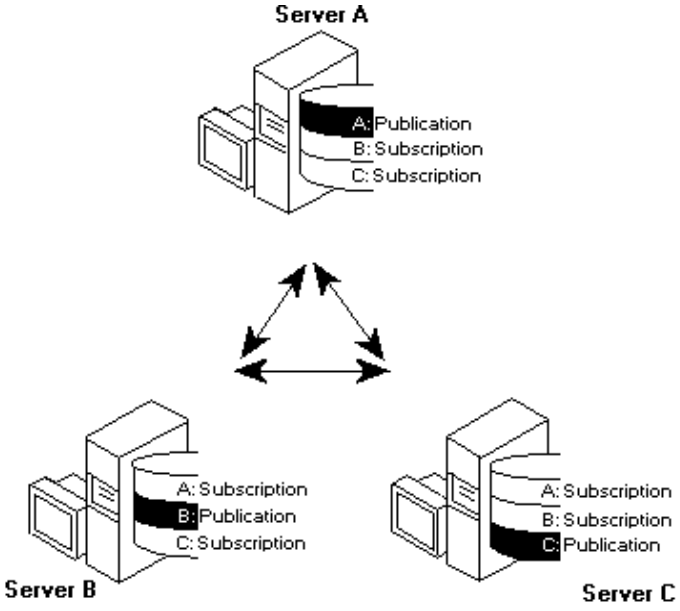


Figura 3.7. Tipologia multiple publisher of one table

3.1.8 Replica di dati di tipo text ed image

Il processo di Replica è limitato per quanto riguarda colonne di tabelle che contengono dati di tipo text ed image, poiché è possibile definire solamente

la modalità di replica Scheduled Refresh mentre quella Transaction Based non è permessa.

Possiamo rendere trasparente l'utente rispetto a questa limitazione attraverso una doppia pubblicazione della tabella che contiene campi di testo o immagine. Ad esempio possiamo pubblicare un articolo A che contiene i campi immagine e testo della tabella e definire per questo articolo una replica di tipo Scheduled Refresh (ad esempio ad intervalli di 1 ora) e poi definire un articolo B che contiene gli altri campi della stessa tabella con una modalità di replica Transaction Based. Avendo previsto di pubblicare in entrambi gli articoli l'attributo TIMESTAMP posso avere una visione globale della tabella determinando anche se le due parti fanno riferimento alla stessa versione.

3.1.9 Replica e ODBC

Un distribution server si collega a tutti i subscription server come un client ODBC (*Open Data Base Connectivity*). Per questo, la replica richiede che il driver ODBC 32-bit sia installato su tutti i distribution server. L' installazione dei driver necessari su Windows NT viene eseguita automaticamente dal programma di setup di SQL Server 6.5. Per quanto riguarda i subscriber, non è richiesta la configurazione degli ODBC Data Sources, poichè il processo di distribuzione utilizza direttamente il nome di rete del subscriber per stabilire la connessione.

3.2 Il sistema di replica ORACLE7 Symmetric Replication

ORACLE7 Symmetric Replication fornisce due meccanismi principali per la replica dei dati su DB remoti che supportano le categorie tight consistency e loose consistency:

- Synchronous Replication. Nella Synchronous Replication tutte le copie dei dati sono mantenute, in ogni istante, consistenti e sincronizzate (tight consistency). Infatti, quando una copia di un dato viene aggiornata, questa variazione viene immediatamente applicata a tutte le altre copie all'interno della stessa transazione. Di conseguenza il metodo di replica sincrono è opportuno quando la consistenza dei dati continua nel tempo è il requisito principale dell'applicazione.

- Asynchronous Replication. Nella Asynchronous Replication, le copie dei dati possono subire un temporaneo disallineamento le une dalle altre (loose consistency). Se un dato viene aggiornato, il cambiamento viene comunicato alle altre copie solo in un secondo momento tramite una transazione indipendente che, su decisione del system administrator, può avvenire a breve distanza di tempo (secondi o minuti) come pure a distanza di giorni. Assicurare la consistenza dei dati in un sistema basato sull'Asynchronous Replication è un operazione critica nella gran parte delle applicazioni. Pensiamo infatti a cosa può succedere se uno stesso dato, ad esempio una colonna di una data relazione, viene aggiornato in due siti diversi nello stesso momento (presumendo che ciascun sito possieda una replica locale dei dati interessati), o per meglio dire nello stesso intervallo di replica coincidente con due aggiornamenti consecutivi delle repliche: ovviamente abbiamo quello che viene chiamato conflitto sull'aggiornamento dei dati.

Per assicurare la consistenza dei dati occorre prevedere, in senso generale, metodi di rilevamento e risoluzione dei conflitti attraverso sistemi di decisione in grado di determinare automaticamente le opportune le contromisure necessarie. Oppure, soluzione più utilizzata, si evitano i conflitti limitando, a livello logico, i permessi di aggiornamento dei dati ad un solo "proprietario".

3.2.1 Modelli di replica

Nell'ambito della replica asincrona dei dati entrano in gioco, come detto sopra, tematiche relative all'aggiornamento di copie diverse di uno stesso dato da parte di siti diversi, che portano a conflitti nella gestione consistente dell'integrità dei dati.

Le due metodologie utilizzate dalle applicazioni per la gestione dei conflitti riguardano la:

- rilevazione e risoluzione dei conflitti (conflict detect and resolution)
- evitare i conflitti (conflict avoidance)

ORACLE7 Symmetric Replication è stato progettato per permettere di gestire applicazioni che usano indifferentemente una delle metodologie introdotte (conflict avoidance or conflict detection and resolution). Per quanto

riguarda la prevenzione dei conflitti i modelli utilizzati sono: *primary-site ownership* and *dynamic ownership*; mentre lo *shared-ownership* permette la rilevazione e risoluzione.

Primary-site ownership

Nel modello primary-site ownership, esiste un sito "proprietario" dei dati che è l'unico ad avere il permesso di aggiornamento sui dati in suo possesso. Gli altri siti coinvolti "sottoscrivono" i dati di interesse (subscriber), il che significa che i dati vengono replicati nei siti remoti dove le applicazioni hanno solo permessi di lettura sui dati replicati. In questo modo, ovviamente, si evita qualsiasi conflitto sugli aggiornamenti di copie diverse.

ORACLE7 Symmetric Replication permette sia la frammentazione orizzontale che verticale sui dati replicati.

Principali esempi di applicazioni del modello primary-site sono:

Distribution of Centralized Information. Informazioni sui prodotti (es. tipico il listino prezzi) di un'impresa sono mantenute ed aggiornate nella sede centrale, mentre ciascun ufficio vendite ne mantiene una copia read/only che, per un certo periodo di tempo, si ammette essere non completamente aggiornata.

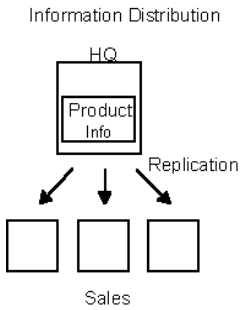


Figura 3.8. Distribution of centralized information

Consolidation of Remote Information. Dati relativi alle vendite locali in siti remoti possono venire integrati in una copia read/only mantenuta nella sede centrale.

Offloading of OLTP Data for DSS Analisys. Dati relativi ad una o più

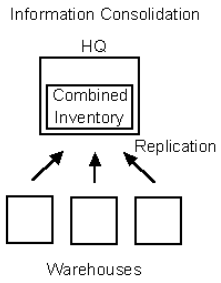


Figura 3.9. Consolidation of remote information

OLTP possono essere replicati, al limite anche nel sito locale, per essere utilizzati in analisi di tipo DSS, che sappiamo essere tipicamente read/only.

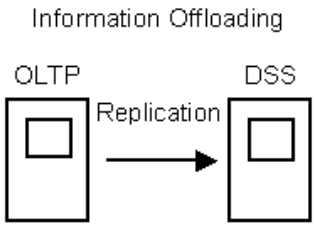


Figura 3.10. Information offloading

Dynamic ownership

Nella modalità dynamic ownership, il diritto di aggiornare in modo asincrono i dati passa da un sito all'altro garantendo la condizione che, ad ogni istante, un solo sito possieda i diritti di aggiornamento. Esempio tipico è il processo di approvazione di un ordine, che deve seguire un iter specifico ed ad ogni passo subisce un aggiornamento (eventualmente in siti diversi).

Shared ownership

Nei precedenti modelli l'idea comune è quella che, in ogni istante, un solo sito possiede i privilegi per l'aggiornamento dei dati, mentre gli altri possono accedere in modalità read/only. Ci possono essere applicazioni in cui è

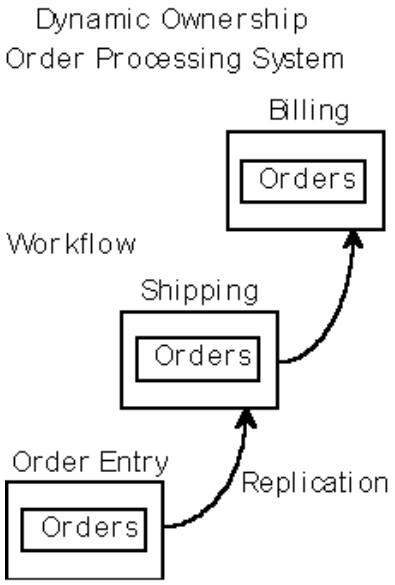


Figura 3.11. Dynamic ownership

conveniente permettere a più siti l'aggiornamento degli stessi dati, potenzialmente nello stesso istante. Ad esempio può essere comodo replicare i dati dei clienti su vari siti remoti piuttosto che nella sede centrale, permettendo a ciascun sito di poter aggiornare liberamente qualsiasi dato. In questo ambito diventa molto importante la garanzia di una corretta gestione dei conflitti sull' aggiornamento dei dati.

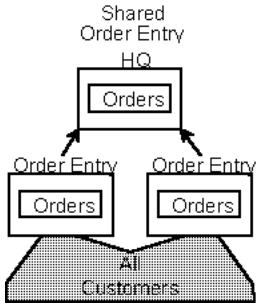


Figura 3.12. Shared ownership

Update Conflicts. Diciamo che abbiamo *Update Conflicts* quando dati replicati in modo asincrono sono divenuti inconsistenti poiché aggiornati contemporaneamente su più siti. Per alcune applicazioni, inconsistenze temporanee possono essere permesse se l'applicazione si fa carico

della rilevazione e della risoluzione di questi conflitti per assicurare la convergenza di tutti i dati replicati verso una situazione consistente.

Conflict Detection and Resolution ORACLE7 Symmetric Replication può rilevare automaticamente i conflitti di update ed invocare funzioni per la risoluzione dei conflitti specificate dall'utente; in questo modo può essere assicurata la convergenza dei dati verso una situazione consistente. I conflitti sono rilevati automaticamente attraverso il confronto tra l' immagine precedente del dato originale ed il valore corrente al sito di replica (confronto tra before image e after image): nel caso in cui i valori sono diversi abbiamo un conflitto.

Riguardo alla risoluzione dei conflitti, ORACLE7 mette a disposizione una serie di routine standard che possono essere selezionate in modo dichiarativo dall'utente. Queste routine sono basate su:

- Latest timestamp
- Earliest timestamp
- Priority Group
- Site Priority
- Additive
- Minimum
- Maximum
- Average
- Overwrite
- Discard

In più l'utente può definire proprie routine di risoluzione utilizzando procedure di linguaggio PL/SQL: queste routine possono contenere anche chiamate a quelle standard, inserire un history delle transazioni eseguite, inviare via e-mail all'utente le informazioni riguardo un conflitto avvenuto, ... In questo modo l'utente può definire una risoluzione di routine multipla, basata sulla definizione di priorità che specifica l'ordine di esecuzione delle routine.

3.2.2 Protezioni rispetto alle cadute di sistema (FAIL-OVER)

ORACLE7 Symmetric Replication può essere impiegato come supporto ad una configurazione di fail-over, con la quale si protegge l'applicazione rispetto alla caduta del sistema primario abilitando l'esecuzione dei processi su di un sistema secondario. Il mantenimento della copia dei dati su di un sistema secondario (detto fail-over system) può essere compiuto tramite il processo di replica asincrona, permettendo il mantenimento dell'esecuzione di un' applicazione in caso di caduta del sistema primario.

Sono possibili due soluzioni per garantire il fail-over:

1. Parallel Server
2. Stand-by Database Configuration

che forniscono diversi livelli nella capacità di troughput, semplicità d' uso, sicurezza rispetto alla perdita di transazioni od altre consistenze, limitazioni sull'uso del fail-over, tipi di failure dalle quali possiamo proteggerci.

Parallel Server

Oracle Parallel Server permette a sistemi multipli dislocati nello stesso sito la condivisione negli accessi ad uno stesso Database mantenuto in memoria secondaria su di un disco condiviso. Se un sistema fallisce, viene immediatamente istanziato uno tra i rimanenti sistemi attivi che automaticamente recupera le transazioni incomplete continuando cos l'esecuzione delle transazioni attive nel sistema interrotto.

Oracle Parallel Server fornisce elevato troughput e facilità d'uso, ma non fornisce alcuna protezione riguardo alle cadute del sito che possono rendere inutilizzabile l'intero ambiente locale (tipicamente failure dovute alla mancanza di alimentazione, incendio, sabotaggio, ...).

Stand-by Database Configuration

Questo approccio garantisce le applicazioni rispetto alla caduta di un sito offrendo anche elevata capacità di troughput. Utilizza una diversa alternativa della replica sincrona che impiega le procedure di sistema operativo per trasferire i file di log generati dal sistema primario al sito di fail-over. Giunti al sistema remoto, i file di log sono applicati ai dati utilizzando il meccanismo

di recovery di Oracle. Definendo il sito di fail-over in un area geograficamente distinta da quella del sito primario preveniamo i fallimenti relativi ad un intero sito. L'elevato troughput deriva dall'utilizzo delle direttive di sistema operativo per il trasferimento dei file di log (che non generano sovraccarico al sito primario) e dall'efficienza del meccanismo di recovery utilizzato sul sito di fail-over.

Il sistema di fail-over viene mantenuto continuamente in modalità recovery, che comporta l'impossibilità di utilizzarlo in altre attività che non sia il fail-over. Inoltre, se il sistema primario fallisce, non è possibile trasferire i file di log più recenti con la conseguente perdita nel sito di fail-over delle ultime transazioni eseguite.

Oracle Symmetric Replication e fail-over

Oracle7 Symmetric Replication può essere utilizzato per la protezione rispetto alle cadute di un sito come alternativa alla modalità stand-by. In questa configurazione ciascun sito può essere definito contemporaneamente sia attivo (corrispondente al sito primario della configurazione stand-by) che fail-over, con il vantaggio che ciascun sito fail-over può compiere tutte le operazioni sui dati e non solo quelle relative all'operazione di fail-over. Ad esempio, i dati che sono replicati sul sito di fail-over possono essere acceduti (in modalità read/only) per operazioni di DSS. Oppure, se utilizziamo la configurazione shared-ownership è possibile aggiornare contemporaneamente i dati in entrambi i siti di fail-over. Naturalmente questa libertà di esecuzione viene pagata in termini di prestazioni poichè non possiamo utilizzare il meccanismo di recovery di Oracle per compiere le operazioni di fail-over.

3.2.3 Replica dei dati mediante Remote Procedure Call (RPC)

Le RPC (*Remote Procedure Call*) sono direttive molto flessibili e general purpose utilizzate da Oracle come meccanismo di propagazione della replica. Inoltre possono essere utilizzate direttamente all'interno di applicazioni tipo PL/SQL ed eseguite in modo asincrono o store-and-forward. Vediamo come avviene il procedimento di replica dei dati.

Una transazione sul sito primario inizia l'esecuzione di una RPC differita per l'invio dei log comunicando la relativa richiesta ad una coda locale.

Ciascuna richiesta nella coda locale viene poi spedita al sito remoto opportuno dove verrà eseguita tramite una nuova transazione (separata dalla prima di invio). Se il sistema remoto non risulta disponibile durante l'invio della RPC, allora la richiesta rimane nella coda del sito primario per essere rispedita in seguito. La coda delle RPC garantisce la durability, grazie ai sistemi di recovery e backup di Oracle. Ciò garantisce che ciascuna richiesta non venga persa ma rimandata al tempo in cui il sistema locale sarà di nuovo disponibile. Oracle garantisce inoltre la possibilità di inviare una RPC a più siti contemporaneamente e che una lista di RPC sottomesse alla coda locale tramite una singola transazione siano eseguite nel sito remoto mediante un'unica transazione mantenendo l'integrità dei dati.

3.2.4 Configurazione

ORACLE7 Symmetric Replication consente la replica delle tabelle (sia completa di tutti i dati che di una partizione orizzontale e/o verticale) attraverso due meccanismi:

- Multiple Master
- Updatable Snapshot
- Hybrid configuration

I due meccanismi, inoltre, possono essere combinati tra loro dando origine ad una configurazione ibrida.

Multiple Master

Multiple Master Replication consente la replica di intere tabelle tra siti diversi detti master: la gerarchia tra i due siti è peer-to-peer. Ciascuna tabella di un sito master può essere aggiornata ed i cambiamenti registrati in una tabella master sono propagati direttamente a tutte le altre tabelle master. In questo contesto, la caduta di un sito master non blocca la propagazione delle variazioni agli altri siti. Multiple Master Replication utilizza le RPCs (Remote Procedure Calls) differite come meccanismo per la propagazione degli aggiornamenti ai siti remoti; se abbiamo la replica di più tabelle collegate tramite referenza, i cambiamenti avvenuti nel master sono inviati tramite una transazione ai vari siti, garantendo l'integrità referenziale dei dati.

Gli aggiornamenti possono essere propagati sia istantaneamente (a meno del tempo di comunicazione della rete) quando l'applicazione è event-based, oppure ad un tempo predefinito, durante il quale si presume che la comunicazione sia più veloce ed economica. Nel caso in cui il sistema remoto non sia disponibile, allora la RPC differita viene mantenuta nel sistema remoto in coda alle transazioni sospese, in modo da essere eseguita in seguito quando il sistema sarà ripristinato.

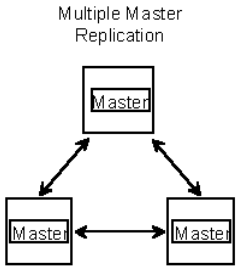


Figura 3.13. Updatable snapshot

Updatable Snapshot

ORACLE7 supporta un sistema di replica basato sulle repliche di tipo Snapshot, le quali fanno riferimento ad una master table e possono esse stesse essere aggiornate. Quando avviene un aggiornamento ad una Snapshot remota, questo viene comunicato al master tramite le RPCs. Le Snapshot possono contenere sia la copia completa della master table, sia partizioni orizzontali o verticali opportunamente selezionate. L'aggiornamento delle Snapshot può avvenire sia ad intervalli definiti di tempo che su richiesta da parte del sito di Snapshot. Come per il multiple masters, in presenza di siti di Snapshot multipli gli aggiornamenti sono ottenuti tramite transazioni in modo da assicurare l'integrità referenziale dei dati.

Hybrid Configuration

Multiple Master Replication e Updatable Snapshot possono essere combinate assieme per generare una configurazione ibrida. Ad esempio, nel caso in cui la rete dell'applicazione possa essere vista come l'insieme di due reti locali (LAN) collegate a grande distanza tramite WAN, risulta comodo definire due siti master, uno per ciascuna zona geografica, e poi replicare i dati ai

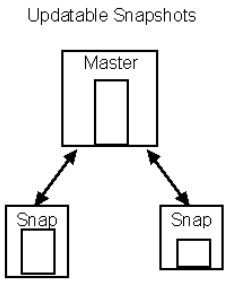


Figura 3.14. Updatable snapshot

restanti siti tramite Snapshot (sia updatable che read/only). In questo modo ciascun sito remoto "vede" il proprio master tramite la sola LAN (più veloce della WAN) ed inoltre i due master, che sono collegati tra loro, garantiscono il funzionamento dell'applicazione anche in caso di guasti in uno di essi (a questo punto l'altro funziona da master per ciascuno di essi).

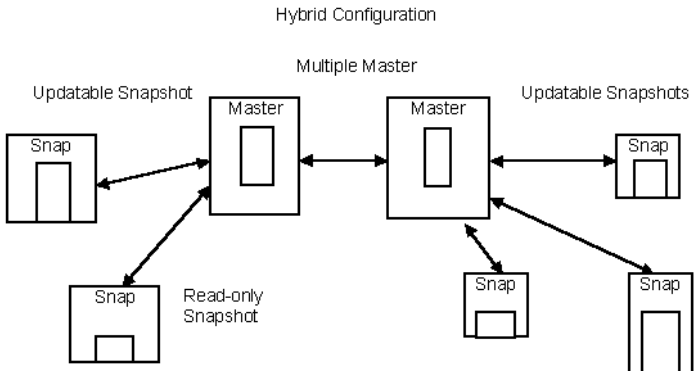


Figura 3.15. Hybrid configuration

3.3 Il sistema di Replica di Sybase Replication Server 10.0

SYBASE Replication Server fornisce meccanismi per la replica dei dati su DB remoti, basati su di una architettura aperta in grado di costruire copie di dati a partire da sistemi ed applicazioni esistenti ed eterogenee. La modalità di funzionamento del processo di replica è di tipo tight consistency,

prevedendo l'aggiornamento automatico delle copie dei dati dopo ogni update degli originali (a meno della latenza introdotta dai componenti software e dalla rete). In aggiunta a questo funzionamento "normale", è possibile forzare l' aggiornamento dei dati replicati ad intervalli regolari ottenendo così una modalità tipo loose consistency sovrapposta a quella di aggiornamento sincrono.

Le funzionalità previste sono:

- Descrizione dei dati e delle *stored procedure* che possono essere replicati in siti remoti.
- Creazione di subscription per le righe da replicare ai siti remoti.
- Inizializzazione dei dati delle tabelle replicate.
- Aggiornamento continuo delle copie per garantire la consistenza dei dati.
- Definizione di stored procedure ai siti remoti che aggiornano i dati primari anche da parte di utenti remoti.
- Gestione della protezione sugli accessi ai Data Server e Replication Server.
- Mantenimento dei log riguardo alle *failed transaction* per permettere al System Administrator di poter gestire gli errori.
- Definizione di un linguaggio RCL (*Replication Command Language*), per la gestione delle funzionalità.

3.3.1 I processi di replica

Alle operazioni partecipano diversi processi al fine di garantire un continuo aggiornamento dei dati replicati rendendoli omogenei a quelli detti primari, che rappresentano i valori originali e veri delle informazioni.

Replication Server Ad ogni sito, il Replication Server coordina le attività di replica tra il Data Server locale ed i Replication Server degli altri siti:

- Riceve i primary data (che sono i dati originali), che vengono aggiornati dal DB locale, tramite il Log Transfert Manager (LTM) e li invia ai Replication Server che hanno richiesto la replica dei dati (subscription).
- Riceve i dati aggiornati dagli altri Replication Server e li memorizza nel DB locale.

Le informazioni necessarie al Replication Server per compiere le operazioni sono memorizzate in tabelle che costituiscono il Replication Server System Database (RSSD) che viene gestito dal Data Server al pari degli altri DB.

Data Server Gestisce i DB contenti sia i primary che i replicated data, fornendo all'utente la possibilità di eseguire le comuni transazioni. Poiché il Replication Server è un sistema aperto (ad open interface), il Data Server può essere diverso dal SYBASE SQL Server, purché rispetti alcune caratteristiche generali riguardo ai dati ed alle transazioni.

Log Transfert Manager Specifica al Replication Server le azioni che sono state eseguite sul DB locale e che quindi debbono essere ripetute sui dati replicati. Un Log Transfert Manager (LTM) è richiesto su ciascun DB che possiede primary data o replicated stored procedure (che manipolano primary data remoti), mentre non è necessario per i DB che compiono solamente operazioni di lettura sui replicated data.

Client Application Client Application è il programma che accede al Data Server e rappresenta l' interfaccia con l'utente finale. L'utente aggiorna i primary data tramite la Client Application e queste modifiche si riflettono sull'aggiornamento delle repliche ai siti remoti da parte del Replication Server.

3.3.2 Architettura aperta

Il Replication Server ha una architettura aperta che permette l'utilizzo di data server diversi per lo sviluppo di sistemi di Replica. Il supporto per Sybase SQL Server è già presente all'interno del Replication Server; per gli altri data server è sufficiente costruire l'interfaccia per poter utilizzare propriamente le istruzioni del Replication Server. L'architettura "open" è definita dai seguenti componenti:

- Client/Server Interface (C/SI)
- Log Transaction Manager (LTM)
- Gestione e controllo degli errori
- Chiamate a funzione (RPC)

Client/Server Interface (C/SI)

Sybase Client/Server Interface (C/SI) definisce le procedure ed il protocollo per la comunicazione client/server.

Il Replication Server viene collegato ai server dei dati come un qualsiasi client di C/SI; se il data server non supporta C/SI, occorre creare un Open Server Gateway che permetta l'accesso del Replication Server al data server. Il Gateway deve essere in grado di accettare la connessione da sistemi C/SI per la ricezione dei processi da eseguire e, utilizzando l'interfaccia del data server, deve portare queste richieste ai data server.

Log Transaction Manager

Il LTM rileva le modifiche compiute sui dati primari e le porta al Replication Server in modo che possano essere distribuite ai siti remoti. Conseguentemente, ciascun sito che contiene primary data deve contenere un LTM. Il LTM fornito da Sybase legge le transaction log rilevando così i cambiamenti nei dati primari, garantendo la consistenza sugli aggiornamenti delle repliche.

Gestione e controllo degli errori

Il Replication Server elabora i codici e lo stato degli errori forniti dal Data Server seguendo le direttive impostate dal progettista dell'applicazione. Infatti vengono forniti i metodi per definire la gestione degli errori raggruppati in categorie:

- Possibilità di creare classi di errori sulla base dei valori definiti dal DBMS.
- Assegnazione di azioni specifiche (warning, retry, abort) a ciascun errore (o classe di errori) precodificato.
- Associazione di una classe di errori ad uno specifico Database.

Chiamate a funzione (RPC)

Tipicamente, il Replication Server utilizza chiamate a funzione (function) per distribuire le richieste di un data server agli altri siti. Vengono utilizzate le function string per inviare direttive di controllo sulle transazioni e sulla modifica dei dati. Una function string è un template (o metacomando) che il Replication Server trasforma in un comando nel formato compatibile a quello del data server.

Una function string può contenere variabili, che rappresentano valori di attributi, parametri di procedure, informazioni di sistema, ..., che vengono istanziate del Replication Server nel momento di invio ai siti remoti (quindi al tempo di esecuzione). Le function string possono essere espresse in due formati: language e RPC. Una language function string contiene comandi che sono interpretati dal Replication Server che sostituisce i parametri a run-time. La sintassi è libera ed in genere è di tipo SQL.

Una RPC function string contiene una richiesta remota ad una procedura rappresentata dal nome di procedura seguita dai parametri. Sono più efficienti delle language function, poichè il Replication Server riesce ad inviare la richiesta in remoto attraverso pacchetti più compatti.

3.3.3 Replica di dati di tipo text ed image

Il Replication Server non permette la replica diretta dei dati di tipo testo o immagine, anche se il vincolo si può superare utilizzando una tabella intermedia, le funzionalità dei trigger di SQL Server e un'applicazione Open Server.

Il metodo utilizzato non è esente da limitazioni, in quanto non viene mantenuta l'atomicità della transazione originale che ha modificato i dati testuali ed immagine: questi sono infatti replicati tramite una transazione separata che inizia al termine del commit della transazione originale. La strategia di lavoro prevede, al sito primario, la segmentazione del testo (o dell'immagine) in stringhe di caratteri o valori binari a dimensione fissa e la definizione di una tabella intermedia che contiene questi segmenti (la segmentazione può essere eseguita direttamente dall'applicazione oppure tramite trigger). Il sito di replica richiede una copia della tabella intermedia che contiene i vari segmenti dei dati: un trigger su questa tabella replicata fa partire, dopo ogni aggiornamento, l'esecuzione di una applicazione Open Server che riassembla i dati e li copia nella tabella di origine nel sito di replica.

3.3.4 Connessione tra i server

I tipi di connessione tra i server coinvolti nell'applicazione sono due:

Connection Una connessione è un messaggio che viene portato dal Replication Server al database. Il Replication Server (RS) manda i dati aggiornati, che ha ricevuto da sito primario, al(ai) database(s) che gestisce attraverso la connection. Naturalmente un RS può avere conessioni con più database, mentre ciascun DB può avere una sola connessione al RS.

Route E un messaggio che scorre da un RS ad un altro. Il RS utilizza la route per trasportare sia gli aggiornamenti dei dati che le informazioni di sistema.

In generale, quindi, si avranno conessioni che utilizzano le LAN (a volte database e RS sono nella stessa macchina), mentre le route utilizzano le WAN. Una rete può essere diretta (collegamento tra due RS tramite route) oppure indiretta (con l'ausilio di RS intermedi che ricevono i messaggi e li mandano ad altri RS).

I vantaggi di una route indiretta riguardano:

- Volume dei dati ridotto sulla WAN,
- Il Replication Server distribuisce una sola copia dei messaggi per tutti i siti raggiunti da un RS intermedio (il quale duplica i messaggi per tutti i siti per i quali funge da intermediario) con conseguente riduzione del carico di lavoro,
- Maggiore Fault Tolerance,
- Permette la continuità di lavoro sulle LAN a fronte di una WAN failure (migliora eventuale recovery dei siti più lontani)

Lo svantaggio maggiore è quello di aumentare i tempi di latenza necessari per eseguire l'aggiornamento dei dati a tutti i siti contenenti dati replicati.

In applicazioni in cui la struttura organizzativa dell'impresa è centralizzata allora una connessione a stella (senza route indiretti) può essere realizzata, ma in applicazioni in cui la disposizione delle sedi sul territorio risponde ad una logica gerarchica, allora è conveniente utilizzare il route intermedio e creare una rete di tipo gerarchico.

3.3.5 Progettare la replica

SYBASE Replication Server offre più modalità di progetto per la gestione dei dati replicati, sia per quanto riguarda la frammentazione dei primary data (sito singolo o multiplo), il numero e la composizione delle tabelle replicate e la possibilità di avere più copie delle primary data (tra le quali una viene definita attiva e le altre poste in stand-by). Per ciascuna di queste modalità SYBASE RS garantisce la consistenza attraverso l'aggiornamento delle repliche dei dati non appena un'applicazione esegue il commit di una transazione di update di un primary data.

Multiple Site Replicate

E una situazione tipica di applicazioni DSS (Decision Support System) in cui i dati vengono aggiornati in periferia ed in sede centrale occorre una visione aggregata (in sola lettura) delle informazioni che vengono quindi replicate.

Al sito primario abbiamo i seguenti componenti:

- Un Data Server per la gestione dei primary data (OLTP)
- Un Log Tansaction Manager che recupera gli aggiornamenti del primary database e li invia al RS
- Un Replication Server per gestire la distribuzione degli aggiornamenti

Ad ogni sito di replica abbiamo:

- Un Replication Server che riceve i dati dal primary RS e li porta al Data Server locale
- Un Data Server per la gestione dei dati replicati

Al sito primario:

- Le applicazioni (OLTP) aggiornano i primary data tramite il Data Server
- Il Log Transaction Manager legge il database log ed invia le informazioni al Replication Server
- Replication Server manda le informazioni ai RS dei siti remoti (al momento stesso in cui il primary Data Server esegue il commit dei dati)

Ovviamente per assicurare la consistenza dei dati, in ciascun sito remoto solamente il RS ha il permesso di modificare i replicated data, mentre le applicazioni si debbono collegare al sito primario e modificare direttamente i primary data.

Aggiornamento dei primary data con Asynchronous Procedure Calls

Quando un'applicazione remota (che lavora sui dati replicati) vuole aggiornare dati le possibilità offerte sono due:

- Collegarsi al sito primario attraverso la WAN come utente del Data Server
- Utilizzare le Asynchronous Procedure Calls (APCs) fornite dal RS

La prima possibilità è di fatto esclusa a causa della vulnerabilità riguardo ai guasti delle WAN ed alla congestione del traffico che introduce. Le APCs sono replicated stored procedure calls che vengono gestite dai Replication Servers come per l'aggiornamento dei dati; all'utente remoto appare come l'esecuzione di una normale SQL Server stored procedure, con l'eccezione che viene eseguita al sito primario.

Distribuzione dei dati primari su più sedi

Alcune applicazioni richiedono la distribuzione dei dati su più sedi (es. gestione bancaria su più sedi). L'insieme delle righe che sono primarie ad un sito sono dette primary fragment. Le modalità della distribuzione dei dati sono molteplici, in relazione al luogo in cui i dati replicati sono mantenuti. Naturalmente, per assicurare la consistenza dei dati replicati, ciascun sito deve avere il permesso di aggiornare solo i propri primary data. La gestione è lasciata al system administrator.

Consolidated Primary Fragment

Tipica applicazione in cui un'impresa ha varie sedi locali che gestiscono ciascuna un insieme di dati utilizzati dalla sede centrale in lettura. La struttura è la medesima di quella descritta in Multiple Site Replicate, con l'aggiunta del fatto che il sito che possiede i dati replicati li ottiene da diversi primary site. Questo comporta principalmente:

- Il primary fragment a ciascun sito è definito da una partizione della tabella

- Il sito di replica (server della sede centrale) sottoscrive (subscription) ciascuna di queste partizioni di replica dei primary server

Replicated Consolidated Primary Fragment

L'applicazione è una estensione della precedente in cui il sito di replica che consolida i primay data dei siti remoti diventa esso stesso un distributore di questi dati (in modo globale) verso altri siti che richiedono la replica. L'unico cambiamento è che ora anche al sito di replica è presente un LTM per l'invio dei dati ad altri siti. L'esecuzione di questo LTM è particolare (viene eseguito con l'opzione -A, cioè all) in quanto occorre spedire tutte le transazioni eseguite, non solo quelle che riguardano dati aggiornati.

Distributed Primary Fragments

Lo scenario di questa applicazione è un sistema nel quale ciascun sito mantiene sia dati primari che replicati.

Ciascun sito possiede i seguenti componenti:

- Un Data Server che gestisce la tabella contenente sia i dati primari che quelli replicati

- Un LTM che preleva le informazioni riguardo agli aggiornamenti dei dati dal database log

- Un Replication Server che:

  - Invia l'aggiornamento dati ricevuto dal LTM
  - Replica le informazioni che riceve dagli altri RS tramite la subscription

La tabella replicata a ciascun sito contiene sia primary che replicated data. Una replication definition descrive la tabella locale. Gli aggiornamenti alle porzioni delle tabelle sono ricevuti tramite le subscription nelle replication definition ai vari siti.

Redundant Primaries

Un sistema di replica ridondante mantiene DB primari multipli. Le applicazioni fanno riferimento ad un DB primario "attivo", mentre gli altri sono mantenuti in stand-by dal Replication server. Il modello permette il cambiamento del primary database durante l'esecuzione, con la tecnica detta "passing the book".

Per poter permettere la gestione ridondante, ad ogni sito le tabelle sono replicate:

- I siti di replica hanno una subscription per ciascun database primario
- I database primari hanno una subscription per le tabelle in ognuno degli altri DB primari

Questo permette primary site multipli per i dati, che possono portare a notevoli inconsistenze. Le inconsistenze possono essere evitate richiedendo a ciascuna applicazione di riferirsi allo stesso db primario.

3.3.6 Topologia della rete

Le topologie che andiamo a presentare riguardano la connessione logica dei vari sistemi che costituiscono l'applicazione ma non vincolano in alcun modo i collegamenti fisici tra i server. Chiaramente, in generale, più la connessione fisica segue le direttive di quella logica, maggiore sarà la resa dell'applicazione.

Rete a stella

Una rete a stella prevede l'esistenza di un sito protagonista, chiamato centro stella, al quale tutti gli altri siti sono direttamente collegati: la comunicazione avviene solamente dal centro stella ai siti remoti e viceversa, senza la possibilità di comunicazione diretta tra due siti remoti se non attraverso il centro stella.

Il caso tipico di applicazioni che si appoggiano su di una rete a stella lo possiamo pensare quando abbiamo una sede centrale che gestisce una grande mole di dati ed invia (in parte, o completamente) alle varie filiali dell'azienda le informazioni di interesse: ad esempio un'azienda di commercio nella quale la sede centrale aggiorna i cataloghi ed i listini prezzi, mentre le filiali sparse sul territorio consultano continuamente i cataloghi per il servizio ai clienti.

In questo tipo di scenario l'unico sito autorizzato ad aggiornare i dati è il centro (centro stella) mentre in periferia sono necessarie copie dei dati sulle quali compiere operazioni read/only (fig. 3.16).

Nei sistemi considerati possiamo gestire questo tipo di applicazioni tramite le configurazioni:

- Central Publisher in SQL Server 6.x
- Information Distribution in ORACLE7 Symmetric Replication
- Multiple Site Replication in Sybase 10.0 Replication Server

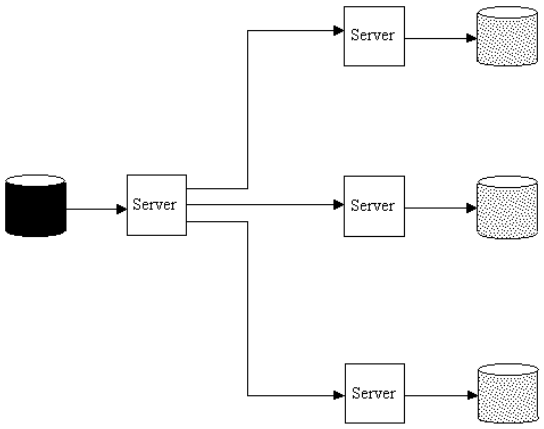


Figura 3.16. Information distribution

Sempre con topologia di rete a stella possiamo considerare applicazioni duali a quella vista in precedenza: pensiamo, cioè, di avere più siti periferici che gestiscono localmente informazioni dello stesso genere (assimilabili ad una o più tabelle), ed una sede centrale che richiede la totalità dei dati per avere informazioni di tipo globale (read/only). Un classico esempio di queste applicazioni sono le informazioni relative all'inventario delle merci di una azienda, che sono gestite a livello locale nelle sedi periferiche e poi spedite all'amministrazione centrale la quale compie operazioni di consolidamento dei dati che le permettono scelte strategiche supportate da informazioni globali (sistemi DSS di supporto alle decisioni).

Come nel caso precedente, per ciascun dato esiste un solo proprietario (i singoli siti remoti), mentre i dati replicati sono utilizzati per operazioni di tipo read/only (fig. 3.17).

Nei sistemi considerati possiamo gestire questo tipo di applicazioni tramite le configurazioni:

- Central Subscriber in SQL Server 6.x
- Information Consolidation in ORACLE7 Symmetric Replication
- Consolidated Primary Fragment in Sybase 10.0 Replication Server

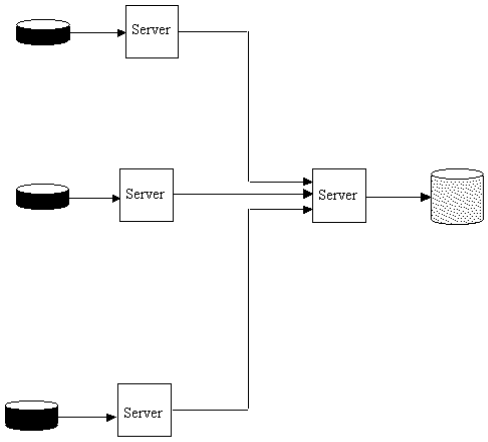


Figura 3.17. Information consolidation

Rete a bus

La rete a bus è costituita da una tratta principale a cui ciascun sito si collega: nessuna stazione è favorita rispetto alle altre e la comunicazione è broadcast. Quindi, la caratteristica rilevante è l'assoluta parità di ruolo di ciascuna stazione.

Esistono applicazioni in cui ogni protagonista (ogni sede periferica) accede ad un insieme di dati condivisi con gli atri, e ciascuno ha il diritto di modificare un sottinsieme di questi dati (mantenendo ciascun sottinsieme disgiunto dagli altri). Ad esempio si colloca in questa categoria la gestione di dati bancari, basata su più sedi che sono proprietarie di alcuni dati (relativi ai clienti locali) e che necessitano di accedere ad informazioni riguardanti i clienti di altre sedi per compiere operazioni di sportello (accrediti, prelievi, versamenti, ecc.). In questo caso non si ha la predominanza di un sito rispetto agli altri, quindi si può pensare ad un collegamento a bus delle varie

sedi, ciascuna delle quali possiede un frammento orizzontale (e/o verticale) dei dati della tabella ed utilizza i restanti dati per operazioni read/only (fig. 3.18).

Entrambi i sistemi supportano tali applicazioni che vanno sotto il nome di:

- Multiple Publisher in SQL Server 6.x
- Combinazione di Information Consolidation and Information Distribution in ORACLE7 Symmetric Replication
- Distributed Primary Fragment in Sybase 10.0 Replication Server

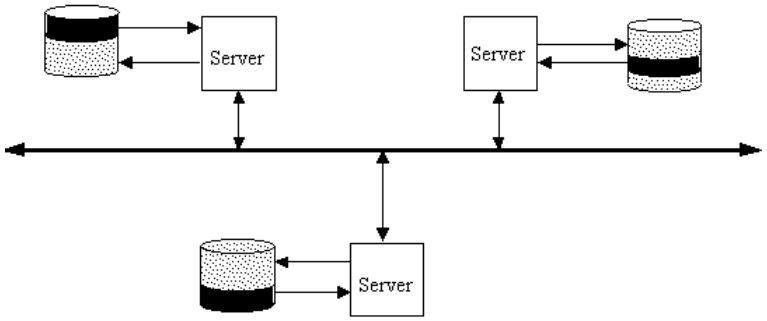


Figura 3.18. Multiple publisher

Rete a stella modificata

Si parla di topologia a stella modificata quando la rete è costituita da due o più reti a stella collegate tra loro tramite i rispettivi centri-stella. Le applicazioni che interessano questa topologia sono pressoché le stesse di quelle con topologia a stella, con la particolarità che possiamo localizzare due o più regioni nettamente separate tra loro (in termini di distanza, o di collegamenti in rete), per cui è conveniente avere per ciascuna regione un centro-stella locale.

Ad esempio se l'azienda in esame è costituita da alcune filiali situate in Italia e le altre negli USA, è conveniente scegliere due siti primari (uno in Italia e l'altro negli USA) che fungono da centro-stella per gli altri siti della loro regione (Italia ed USA rispettivamente).

ORACLE7 soddisfa questi requisiti attraverso un collegamento peer-to-peer tra due siti Master, uno situato in Italia e l'altro negli USA. All'interno di ciascuna regione le altre filiali sono delle Snapshot collegate al proprio Master da cui ricevono i dati replicati. Un ulteriore vantaggio di questa configurazione è un aumento dell'affidabilità ai guasti: ad esempio se il master di una regione non è funzionante le Snapshot della stessa regione possono fare riferimento al Master dell'altra regione.

SQL Server 6.x permette questa situazione attraverso lo scenario Publishing-Subscriber, in cui esiste un solo Publisher (posto indifferentemente in una regione) che pubblica i dati ai Subscriber della propria regione ed ad un particolare Subscriber posto nell'altra regione che ha il permesso di ripubblicare i dati ricevuti dal Publisher ai propri Subscriber "regionali" (fig. 3.19).

Sybase Replication Server gestisce questo scenario tramite la configurazione Redundant Primaries in cui abbiamo la presenza contemporanea di più siti primari, dei quali uno per volta viene mantenuto attivo per quanto riguarda gli aggiornamenti e gli altri sono posti in attesa.

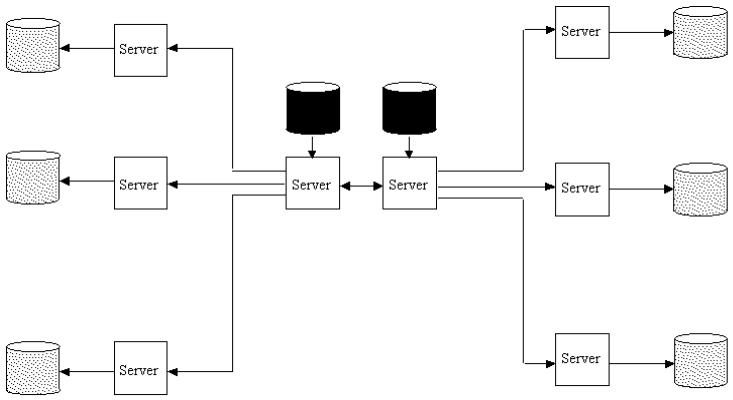


Figura 3.19. Multiple master

Capitolo 4

Nuovi modelli di replica per DDBMS

La gestione di sistemi distribuiti mediante replica dei dati è un argomento particolarmente complesso. Per facilitarne la comprensione verranno esaminati ed esemplificati diversi requisiti di uno stesso caso di studio in una progressione di funzionalità e complessità crescente. A questo livello di descrizione non verrà fornita la soluzione completa ma solo una descrizione del problema ed una idea di soluzione applicativa così come appare all'utente. La piattaforma DBMS considerata è Microsoft SQL Server.

Consideriamo il seguente esempio che verrà utilizzato in tutti gli scenari presentati: si vogliono gestire gli ordini e le fatture di una azienda dislocata sul territorio in varie filiali. Valgono i seguenti requisiti:

- Ciascuna filiale mantiene gli ordini relativi al proprio magazzino;
- Ciascun ordine è costituito da una intestazione e da una o più righe, ognuna delle quali contiene un articolo, la quantità e il prezzo applicato;
- La sede amministrativa (che è unica) gestisce le fatture che sono suddivise nella testata (che specifica il numero fattura, il cliente, la data, l'importo totale, ...) e un certo numero di righe che indicano gli articoli con i relativi prezzi;
- Ogni fattura, a seconda del cliente a cui è indirizzata, ha il proprio codice IVA.

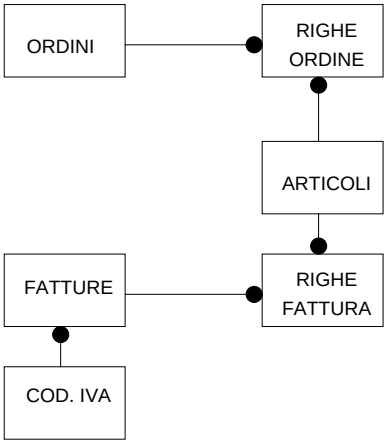


Figura 4.1. Tabelle dell' esempio ordini/fatture

La relazione che lega l'intestazione dell'ordine alle proprie righe è di tipo part-of, dando origine ad un'unica struttura dati complessa che rappresenta l'ordine. Nei sistemi relazionali queste tipi complessi sono gestiti attraverso la definizione di più tabelle correlate tra loro dall'introduzione di foreign key. Nell'esempio dell'entità ordine costituita da n righe, il modello E-R risolve la situazione attraverso un'associazione 1-N ed identificatore esterno:

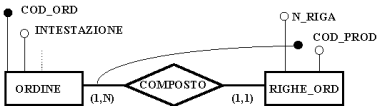


Figura 4.2. Schema E/R entità ORDINE

In casi come questi, è conveniente trasportare l'identificatore esterno all'interno dell'entità facendolo diventare foreign key ed eliminando così l'associazione:

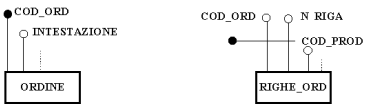


Figura 4.3. Tabelle dell' esempio ordini/fatture con foreign key

Lo stesso procedimento viene seguito per la descrizione della fattura, con l'unica variante dovuta alla presenza dell'informazione del codice IVA che viene risolto ancora tramite foreign key. Il database di esempio risulta quindi composto da:

Ordini: la parte riguardante l'intestazione dell'ordine

Righe ordine: associate come part-of ad un ordine

Codici IVA: referenziati dalla fattura

Fatture: l'intestazione che referenzia un codice IVA

Righe fattura: associate come part-of ad una fattura

Articoli: referenziato da ordini e fatture (tramite le righe)

4.1 Modello "Static Ownership" su tabelle complete

Questo modello prevede che un solo utente sia proprietario di una intera tabella per tutto il ciclo di vita della tabella, mentre gli altri utenti possono consultare i dati con operazioni read only.

4.1.1 Broadcasting di copie identiche in sola lettura

Dal punto di vista della distribuzione dei dati il modello permette l'invio broadcasting di copie identiche read only verso un insieme di server senza alcuna restrizione sui dati presenti nella tabella. Nel nostro database di esempio i requisiti sono:

- l'immissione e la gestione delle fatture è centralizzata in un unico server
- si vuole che tutti gli altri server vedano copia di tutte le fatture emesse.

Operazioni di configurazione:

- L'amministratore del sistema indica la volontà di replicare la tabella fatture.
- Si determina in modo automatico (o manuale) che occorre replicare anche la tabella righe di fattura, articoli e codici IVA.
- Viene creata l'opportuna pubblicazione per SQL Server.

Con questa soluzione gli altri server non possono modificare fatture (e righe fatture), articoli o codici IVA, ma solo consultarle.

4.1.2 Broadcasting di copie non identiche in sola lettura

L'utente proprietario dei record può decidere di replicare partizioni di dati diverse per ciascun server remoto. Nell'esempio, rispetto al caso precedente, abbiamo questo nuovo vincolo: invece di spedire tutte le fatture a tutti si vogliono spedire le fatture alle filiali (server) secondo aree di interesse. I criteri di interesse sono misti e variabili: può dipendere dal cliente, dal tipo di importo, ecc..

Operazioni di configurazione:

- L'amministratore del sistema indica la volontà di replicare la tabella fatture
- Indica inoltre i criteri di replica (che corrispondono a viste sul file della testata fatture)
- Si determina in modo automatico (o manuale) che occorre replicare anche la tabella righe di fattura, articoli e codici IVA. Per ciascuna di queste tabelle, inoltre, si determina la restrizione necessaria per descrivere opportunamente le fatture inviate ai singoli server. (Quindi non è necessario mandare tutti gli articoli ma solo quelli referenziati nelle fatture spedite)
- Vengono create tante pubblicazioni per SQL Server quante sono le diverse viste.

Anche in questa soluzione gli altri server non possono modificare fatture, articoli o codici IVA.

4.1.3 Situazioni miste di broadcasting

Vengono rappresentate situazioni ibride in cui un server è al tempo stesso il proprietario di alcune tabelle (che pubblica) e l'abbonato di tabelle pubblicate da altri. Per l'esempio considerato, supponiamo che:

- esista una coppia di tabelle testata ordine-riga di ordine distinta in ciascuna filiale

- solo la filiale proprietaria della tabella può inserire/modificare gli ordini relativi
- ogni filiale deve spedire copia dell'ordine alla sede centrale
- ogni filiale deve avere una copia della tabella articoli per poter inserire/modificare gli ordini in modo consistente.

Possiamo rappresentare schematicamente la situazione nella seguente tabella, dove con Master si intende il proprietario dei dati e con Slave gli abbonati con diritti read/only. Ipotizziamo, per semplicità, la presenza di sue sole filiali (A e B) e della sede centrale.

	Tabelle				
Sito	ORDINI_A	RIGHE_A	ORDINI_B	RIGHE_B	ARTICOLI
Filiale A	Master	Master	–	–	Slave
Filiale B	–	–	Master	Master	Slave
Sede centrale	Slave	Slave	Slave	Slave	Master

Operazioni di configurazione:

- In ciascuna filiale l'amministratore del sistema indica la volontà di replicare la propria tabella ordini nella sede centrale
- Si determina in modo automatico (o manuale) che occorre replicare anche la tabella righe ordine
- Per ciascuna filiale viene creata una pubblicazione per SQL Server
- Viene determinata la necessità di replicare la tabella articoli in modo completo in ciascuna filiale
- Nella sede centrale viene creata una pubblicazione per SQL Server contenente la tabella articoli.

4.2 Modello "Static Ownership" su viste

Questo modello descrive scenari nei quali più server dell'applicazione hanno diritti di inserimento ed aggiornamento sulla stessa tabella: in particolare si vuole gestire la situazione in cui ciascun server mantiene diritti di possesso sui dati di una partizione orizzontale (disgiunta dalle altre) della tabella.

L'esempio considerato è simile al caso precedente in cui gli ordini venivano inviati dalle sedi periferiche alla sede centrale, con l'unica variante di considerare globali le tabelle ORDINI e RIGHE_ORDINI. La principale difficoltà introdotta dal modello riguarda il controllo dell'unicità della chiave primaria dei record resasi necessaria dalla possibilità di inserire nuovi record in una stessa tabella da più server contemporaneamente. A seconda delle caratteristiche dell'applicazione si possono considerare differenti strategie per la soluzione del problema che andiamo ad elencare.

4.2.1 Aggiornamento dei dati centralizzato

Una prima strategia, non priva di vincoli, prevede la presenza di un unico sito, detto Master, nel quale è possibile inserire/modificare i dati di una tabella; i siti remoti inviano al Master le richieste di aggiornamento tramite chiamate a Remote Procedure Call (RPC). Dopo aver ricevuto i nuovi dati, il Master replica a tutti i server (compreso quello che inserisce/modifica) le modifiche ai dati ottenendo il riallineamento delle copie in tempi accettabili per l'applicazione. Questa modalità di lavoro è limitata ad applicazioni nelle quali gli aggiornamenti sono poco frequenti e la rete ha una elevata velocità e disponibilità per assicurare tempi di latenza ragionevoli (si possono ottenere anche tempi di pochi secondi).

4.2.2 Aggiornamento on-line da più server

Vogliamo permettere la modifica/inserimento in tutti i server che possiedono una partizione di una tabella.

Tabelle locali che non vengono mai pubblicate.

Non ci sono vincoli riguardo alle operazioni di inserimento/aggiornamento, ad esclusione dell'eventuale integrità referenziale, risolta tramite la replica dell'intera tabella referenziata. Nell'esempio considerato, volendo consentire l'inserimento/modifica della tabella ordini-righe di ordini occorre avere a disposizione una copia della tabella articoli per garantire la consistenza sugli ordini di articoli presenti nel catalogo.

Tabelle che vengono pubblicate.

Quando non sono più valide le ipotesi di inserimenti poco frequenti ed alta velocità della rete, nascono i problemi relativi al controllo di duplicazione della chiave, che possono essere risolti adottando una delle seguenti strategie.

Sistemi che risolvono i conflitti: La soluzione permette la presenza di più record aventi la stessa chiave posti in siti diversi che viene rilevata automaticamente dal sistema come situazione di conflitto e riportata in uno stato consistente tramite opportune procedure di gestione. La strategia prevede di istituire la figura di un Central Server che contiene una copia globale dei dati al quale tutti i server sottopongono le richieste di nuovi inserimenti o di modifica della chiave. Operazioni eseguite per la risoluzione dei conflitti:

- Il Central Server mantiene una copia globale dei dati aggiornata (copia Master)
- Ciascun server remoto, dopo aver inserito localmente un nuovo record in una tabella in suo possesso ne definisce lo stato di "record in proposta di inserimento"
- A scadenze predefinite il Central Server riceve, in sequenza, le richieste dei server remoti riguardo i "record in proposta di inserimento"
- Per ciascun record giunto, il Central Server:
  - Controlla che non esista nei propri dati un record con lo stesso valore di chiave
  - Nel caso di conflitto sul valore della chiave decide come risolverlo (modifica il valore di chiave in un valore non duplicato, rifiuta l' inserimento del record)
  - Modifica la copia Master dei dati coerentemente con le decisioni dei passi precedenti (in particolare, in caso di operazione positiva, viene modificato lo stato del record in "accettato")
  - Comunica ai vari server le decisioni sui record (inviando cioè l' aggiornamento dello stato in "record accettato" oppure inviando notizia dell'eliminazione del record).

Sistemi che prevengono i conflitti: Soluzione ortogonale alla precedente è quella di evitare (da sistema) la possibilità della duplicazione della chiave primaria. Vi sono diverse possibilità al riguardo:

- Utilizzare una partizione della tabella con criteri di restrizione sul valore della chiave primaria: in questo caso non esiste la possibilità di duplicazione della chiave, in quanto la semplice verifica del rispetto della partizione di ciascun nuovo record inserito esclude qualsiasi problema di duplicazione.
- Aggiunta nella chiave primaria di un attributo che determini univocamente il server: in questo modo vengono risolti i problemi relativi alla non duplicazione della chiave primaria fisica
- Rimangono i problemi relativi alla chiave logica della relazione (indicata come Alternate Key), presenti anche in applicazioni centralizzate, che, come detto, possono essere risolti ricorrendo alla figura del Central Server come arbitro dei conflitti.

4.3 Modello "Dynamic Ownership"

Questo modello permette la gestione di situazioni in cui un singolo record cambia il possessore durante il proprio ciclo di vita in base al contesto in cui è inserito e al valore che assume.

Esempio: gli ordini, che sono inseriti dalle filiali, seguendo specifici criteri (importo, sconti, clienti fuori fido) debbono essere approvati dalla funzione Direzione Commerciale che sta nella sede centrale. Questo comporta che un ordine inizialmente è in possesso della filiale che lo inserisce e che successivamente diviene di proprietà della Direzione Commerciale che decide riguardo l'approvazione.

Per rappresentare questa situazione si può introdurre un concetto di iter di approvazione, descritto da una macchina a stati, che rappresenta i possibili stati che il record può assumere e la dinamica con cui può modificare il proprio stato. Per prevenire la concorrenza di accessi (non gestibile) ad ogni stato è associato uno ed uno solo proprietario. Nel nostro caso di esempio, i possibili possessori sono il venditore di filiale o la Direzione Commerciale.

Tabella dei possessori di un ordine:

Ruolo	Stato			
	Provvisorio	Attesa di approvazione	Approvato	Annullato
Venditore	Propietario	Read/Only	Read/Only	Propietario
Direz. Commerciale	Read/Only	Propietario	Propietario	–

Transizioni di stato di un ordine permesse:

Ruolo	Stato			
	Provvisorio	Att. di appr.	Approvato	Annullato
Provvisorio	Venditore	Venditore	–	Venditore
Att. di appr.	Dir. Comm.	Dir. Comm.	Dir. Comm.	–
Approvato	Dir. Comm	–	–	–
Annullato	Venditore	Venditore	–	–

La soluzione, che verrà descritta in seguito, deve garantire che:

- I dati siano replicati nei server dove risiedono gli Utenti-Ruoli che hanno il diritto di accesso
- In ogni istante (salvo brevi intervalli di latenza in cui nessuno è proprietario) uno ed un solo Utente-Ruolo abbia il diritto di modifica.

Debbono inoltre esistere metodi di gestione delle eccezioni: cosa fare se un utente si dimette senza aver eseguito la sua transazione, cosa fare in generale se è trascorso un tempo superiore a quello massimo desiderato per lo svolgimento della transazione? Deve quindi esistere un metodo di revoca dei diritti di possesso dei dati. E inoltre importante avere la traccia dei cambiamenti di stato.

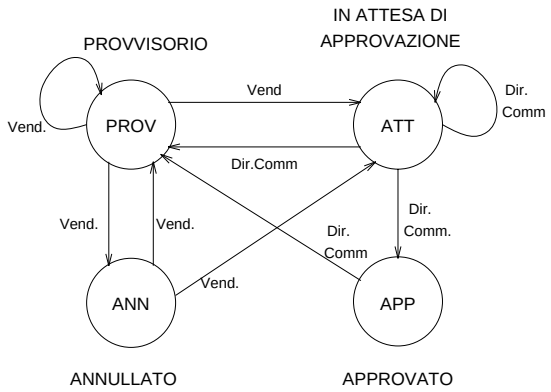


Figura 4.4. Diagramma a stati di un ordine

4.4 Definizione dei requisiti dei modelli

La panoramica di tecniche esposte in precedenza porta a due classi di soluzioni, il cui elemento discriminante è l'assunzione sulla disponibilità di collegamenti in rete sufficientemente veloci (20+ k bit/sec) e affidabili (cadute di linea non frequenti).

Nel caso in cui siano soddisfatti i requisiti di velocità e di affidabilità della rete, il modello più efficace è quello basato sull'aggiornamento centralizzato e sulla replica di dati read only. In termini generali la soluzione è la seguente:

- Per ogni server si assegnano i diritti di accesso alle funzioni. Da questi si determinano in via automatica le pubblicazioni da generare. Le pubblicazioni sono unidirezionali dal master verso tutti i server periferici.

- La gestione degli utenti è fatta in base al sistema di security: i ruoli e gli utenti sono noti a tutto il sistema e sono univoci, mentre le "registrazioni" sono definite in base alla terna: utente, ruolo, server e contengono le date di inizio e fine e i valori delle variabili per la parametrizzazione della protezione. I diritti di ciascuna registrazione debbono essere un sottoinsieme dei diritti del server.

- Gli aggiornamenti vengono eseguiti sempre in "doppio", sul master e sul server, mentre la consultazione delle informazioni agisce solo sul database locale.

A questo modello base si può aggiungere l'introduzione, per alcune tabelle, di un meccanismo di replica read/write che modella la Static Ownership. E' necessario che la tabella originale abbia partizioni mutualmente esclusive tra i server periferici, in modo che non esistano dati pubblicati da più di un server periferici.

Questo approccio offre come vantaggio principale la *solidità di gestione della concorrenza*: i dati sono presenti in un unico server, quindi la gestione della concorrenza può essere demandata completamente al DBMS (che può utilizzare o meno protocolli di Strict Two Phase Locking and Two Phase Commit) riportando il problema ad una gestione già consolidata.

D'altro canto questa opportunità si paga in termini di:

- Prestazioni legate fortemente alle linee di comunicazione Poiché le richieste sono demandate ai server remoti, la risposta del sistema dipende in modo determinante dalla velocità della rete di comunicazione.

- Gestione della caduta della linea o dei server remoti Quando la linea non è disponibile non possiamo accedere ai dati. Questo limite, che è molto stringente, può essere rilassato introducendo situazioni miste di accesso remoto/replica nelle quali temporaneamente, in caso di guasti, ammettiamo la presenza contemporanea di più copie dei dati su server remoti.

Con questi vincoli i problemi di conflitto sulla proprietà dei dati e sulla univocità delle chiavi sono evitati o risolti senza intervenire sul programma applicativo e/o sul disegno del database.

Qualora la rete non abbia invece queste caratteristiche è necessario ricorrere a sistemi di replica, scegliendo la soluzione fra la Dynamic Ownership e la Shared Ownership.

Il modello proposto per la Dynamic Ownership descrive un'architettura distribuita per la gestione di attività di workflow, relativa a più persone, residenti in luoghi diversi, con responsabilità sul Sistema Informativo. Il modello attribuisce, in ogni istante, ad una sola tra le varie copie dei dati il rango di "Originale". Questa particolare copia è l'unica che può essere modificata, quindi solo gli utenti del server contenente l'Originale dei dati possono aggiornare i dati, mentre gli altri li possono solo consultare. Dinamicamente possiamo cambiare il server che possiede l'Originale, consentendo (a turno) a tutti gli utenti di modificare i dati, con l'unico limite di un tempo di latenza dovuto all'esecuzione delle operazioni.

Questo approccio fornisce alcuni vantaggi rilevanti:

Resistenza alle cadute di rete. L'applicazione lavora, per la maggior parte del tempo, su dati presenti localmente sul server, quindi brevi periodi di interruzione della connessione di rete non sono critici per il funzionamento dell'applicazione. Nel caso in cui non siano presenti vincoli di tempo stringenti riguardo l'allineamento tra le varie copie dei dati si può operare anche con sistemi non connessi permanentemente.

Velocità negli accessi Data la possibilità di avere più copie dei dati è possibile mantenere su ciascun server una copia dei dati utilizzati, per cui

la velocità di accesso ai dati è elevata (non richiedendo, in generale, l'utilizzo della rete).

Questi vantaggi sono accompagnati da alcuni limiti introdotti dalla replica:

- Vincoli organizzativi sulla condivisione dei dati: Possibilità di modifica dei dati ristretta, a turno, al solo possessore dell'originale.
- Tempo di latenza fra l'aggiornamento e la disponibilità dello stesso sugli altri server: questo tempo è definibile dall'amministratore del sistema e può essere di pochi minuti o di diversi giorni.
- Complessità tecnica: il sistema si appoggia alle funzionalità di Replica offerte dai DBMS commerciali che debbono essere integrate con un appropriato strato di software superiore per l'effettiva gestione dell'applicazione.

4.5 Soluzione proposta

4.5.1 Utilizzo del Sistema di Sicurezza

La gestione dei permessi di accesso ai dati viene fatta attraverso il Sistema di Sicurezza preesistente, occorre avere a disposizione un insieme di tavole di database che consentono di definire le funzioni appartenenti all' applicazione. Una funzione è definita da un nome, da un titolo e da una finestra con cui la funzione viene implementata: a ciascuna funzione vengono associate le azioni permesse (visualizzazione, inserimento, modifica, cancellazioni più altre che si possono aggiungere) sulle tabelle che costituiscono la funzione.

I concetti presenti sono:

- Ruolo: descrive i diritti in relazione a Funzioni, in termini di Azioni permesse
- Utente: appartiene a più ruoli (uno per volta, quando accede deve specificare il ruolo). I diritti degli utenti appartenenti ad uno stesso ruolo possono essere differenziati con partizionamenti orizzontali (eventualmente sovrapposti) delle tabelle "principali" associate alla funzione.

Il Sistema di Sicurezza consente l'accesso all'applicazione ed alle funzioni solo agli utenti autorizzati; può essere usato a diversi livelli di complessità.

- Controllo sull'accesso: consiste nel verificare che la password d'accesso dell'utente sia corretta; tutte le funzioni dell'applicazione sono disponibili all'utente; durante la sessione di lavoro non vengono eseguiti altri controlli

- Controllo su funzione/ruolo: l'utente si connette all'applicazione con un certo ruolo; viene verificata la password dell'utente per quel ruolo; vengono disabilitate a menu tutte le funzioni non disponibili al ruolo; durante la sessione di lavoro non vengono eseguiti altri controlli

- Controllo su funzione/ruolo/azione: l'utente si connette all'applicazione con un certo ruolo; viene verificata la password dell'utente per quel ruolo; vengono disabilitate a menu tutte le funzioni non disponibili al ruolo; durante la sessione di lavoro viene verificato che l'azione richiesta dall'utente sia disponibile al ruolo

- Controllo su funzione/ruolo/azione parametrico: l'utente si connette all'applicazione con un certo ruolo; viene verificata la password dell'utente per quel ruolo e vengono valorizzate le variabili del ruolo per quell'utente; vengono disabilitate a menu tutte le funzioni non disponibili al ruolo; durante la sessione di lavoro viene verificato che l'azione richiesta dall'utente soddisfi una eventuale condizione espressa tramite le variabili.

E inoltre prevista la gestione del mapping tra la funzione ed entità (od oggetto) che la descrive (possono essere una o più in caso di part-of e di reference key) a partire dalle informazioni contenute nel dizionario dati riguardo le tabelle R/W e R/O.

Da queste premesse risulta semplice poter specificare i permessi di accesso ai dati di ciascun utente/ruolo: ad esempio, consideriamo al solito l'applicazione della gestione degli ordini e supponiamo che l'utente Mario Rossi sia un gestore del sistema per la parte relativa agli ordini, limitatamente alla propria filiale. La Filiale è, quindi, una variabile dell'applicazione rilevante ai fini della sicurezza; in particolare è una variabile significativa per il ruolo di gestore che per Mario Rossi vale, per esempio, T3000. Quando, come gestore, Mario Rossi accede all'applicazione deve selezionare, fra i suoi possibili

ruoli, il ruolo di gestore digitando correttamente la password che gli è stata assegnata in qualità di gestore. La variabile Filiale viene così valorizzata a T3000. Tutte le funzioni disponibili ai gestori sono a lui disponibili. Le azioni che può compiere su ogni singola funzione sono tutte quelle previste per il ruolo di gestore a patto che le eventuali condizioni sulle azioni siano soddisfatte. Ad esempio può inserire, modificare o cancellare nuove richieste d'ordine solo per la sua filiale (T3000), ma non per le altre filiali (sulle quali, ad esempio, può compiere solo azioni di lettura).

4.5.2 Modello centralizzato

Il modello di replica dei dati con aggiornamenti centralizzati prevede una rete di server con topologia a stella il cui centro, detto Master Server, ha il compito di mantenere una copia completa ed aggiornata dei dati che viene considerata l' "Originale". I server periferici mantengono una copia dei dati di loro interesse (specificati dalle funzioni eseguite dagli utenti presenti) che vengono utilizzati per le operazioni di consultazione. L'aggiornamento dei dati avviene sempre nel Master Server tramite RPC inviate dai server remoti: in questo modo la copia Master contiene sempre i dati aggiornati che possono essere mandati ai server periferici durante le operazioni di replica. L'unico sito di tipo publisher risulta quindi essere il Master server che definisce specifiche pubblicazioni per ciascun sito periferico secondo la seguente procedura:

- per ciascun server si assegnano i diritti di accesso alle funzioni
- dalle funzioni, tramite il sistema di sicurezza, vengono determinate le tabelle con le relative restrizioni che diventano gli articoli della pubblicazione
- il Master server genera una pubblicazione specifica per ciascun server periferico che si abbona a tale pubblicazione dando origine ad un flusso unidirezionale dei dati dal centro verso la periferia.

Per quanto riguarda le funzioni assegnate al server, esse sono ottenute come l'unione (o un sovrainsieme) di tutte quelle relative agli utenti del server. In questo modo, a livello locale e' possibile inserire un nuovo utente che svolge alcune (o tutte) le funzioni permesse al server.

L'aggiornamento dei dati da parte dell'utente di un sito periferico può essere gestita in due modalità:

- aggiornamento centralizzato
- aggiornamento locale e centralizzato (in "doppio")

Per aggiornamento centralizzato viene intesa la modalità secondo la quale l'utente di un server periferico che vuole aggiornare (inserire o cancellare) un record invia una opportuna RPC al Master server che provvede alla modifica. Tale aggiornamento viene dapprima eseguito nel Master che poi, previo un opportuno tempo di latenza, lo invia di nuovo alla periferia. Il vantaggio principale di questa tecnica e' insito nella centralità della operazione di aggiornamento che quindi può avvenire in concorrenza sotto la supervisione del DBMS che si comporta come in un caso centralizzato. Quindi, in questo caso possiamo rilasciare il vincolo di un unico proprietario per il singolo record, sicuri della gestione del DBMS nella concorrenza degli accessi. Lo svantaggio introdotto risiede nell'introduzione nel server periferico del tempo di latenza tra l'istante in cui si richiede la modifica e il momento in cui tale modifica viene eseguita sui dati locali.

L'altra tecnica, denominata aggiornamento in "doppio", prevede l'aggiornamento del record nel sito locale contestualmente all'invio della RPC al Master Server. In questo modo evitiamo di introdurre la latenza dovuta alla replica dei dati aggiornati. Occorre pero' tener conto del processo di replica di SQL Server, il quale fallisce nel caso si tenti di duplicare un record avente la stessa chiave primaria di uno già presente nel subscriber. La situazione di fallimento descritta e' quella in cui ci troviamo poiché il Master pubblica verso il server periferico lo stesso record inserito all'inizio dell'operazione di inserimento: per prevenire il fallimento occorre escludere, nella pubblicazione dal Master al server periferico, i record che possono essere inseriti dal particolare sito periferico. Questo vincolo forza la definizione di un protocollo di comunicazione dal Master server verso il server periferico per il trattamento delle operazioni seguenti al fallimento di transazioni al Master.

La gestione dell'operazione di aggiornamento e' analoga a quella di inserimento e valgono le stesse regole descritte in precedenza.

Per semplificare le operazioni, in entrambe le modalità la gestione dei permessi di accesso ai dati da parte degli utenti può essere demandata ai server periferici assumendo valide, al Master server, tutte le RPC che arrivano. In questo modo il Master server interagisce con le informazioni riguardanti la sicurezza solamente durante la fase di configurazione (o riconfigurazione) quando vengono definite le pubblicazioni.

Nel modello descritto non vi sono problemi derivanti dalla duplicazione della chiave primaria, poiché con la gestione centralizzata degli aggiornamenti è possibile utilizzare le funzionalità di controllo della concorrenza del DBMS presente nel Master server.

Riconfigurazione

La riconfigurazione è molto semplice nel caso in cui si voglia aggiungere un utente ad un server periferico con diritti di accesso limitati a quelli assegnati al server, poiché in questo caso è sufficiente definire l'utente nel sistema di sicurezza. Volendo invece aumentare le potenzialità, in termini di accesso ai dati, di un server periferico occorre ridefinire la pubblicazione del Master server relativa al server in esame. Ovviamente questa operazione deve avvenire "a freddo" e, in generale, comporta un sovraccarico di trasferimento dati dovuti ad una nuova sincronizzazione. L'eliminazione di un utente è indolore in quanto è sufficiente modificare il sistema di sicurezza (locale). Solo nel caso si voglia limitare la visibilità dell'intero server è necessaria la ridefinizione della pubblicazione tramite una procedura del tutto analoga a quella detta "a freddo" descritta precedentemente.

4.6 Caratteristiche del sistema di setup iniziale e di riconfigurazione

4.6.1 Set-up iniziale del modello Static Ownership

Lo scenario che vogliamo configurare è quello descritto nelle sezioni di Static Ownership, con la replica dei dati relativi ad una tabella (sia intera che singole viste) su più server, con partizioni diverse delle varie copie. Si ammette la possibilità di avere tabelle con diritti read/write su più server mantenendo il vincolo di partizioni disgiunte dei dati. Non si considera, per ora, la gestione della duplicazione delle chiavi che verrà trattata più avanti. Facciamo l' ipotesi di rete a stella.

L'idea è quella di mantenere, su ciascun server, tutti (e possibilmente soli) i dati che l'insieme degli utenti del server hanno diritto ad accedere (in lettura o in modifica), tramite la configurazione di una rete con topologia a stella nella quale abbiamo un Master Server per il controllo del meccanismo di replica. Tale meccanismo viene così ottenuto:

- Ciascun sito periferico pubblica verso il Master Server i dati sui quali ha diritti di possesso
- Il Master Server, che contiene una copia di tutti i dati, genera una pubblicazione di tipo restricted per ciascun sito periferico. Gli articoli di questa pubblicazione contengono la frazione di dati sui quali il sito periferico ha diritti di sola lettura. Le informazioni necessarie al Master Server sono contenute nel Dizionario dei Dati
- Ciascun sito periferico richiede di sottoscrivere tutte le pubblicazioni del Master che gli competono.

La determinazione del tipo di dati da pubblicare avviene, come detto, tramite il Sistema di Sicurezza ed è visualizzato in figura 4.5: l'utente si collega all'applicazione con un ruolo specifico al quale possiamo associare un insieme di funzioni applicative. Per ciascuna funzione otteniamo, sfogliando il Dizionario dei Dati, l'elenco delle tabelle coinvolte (e relative restrizioni parametrizzate) che definiscono gli articoli della pubblicazione. Avendo l'informazione del server (unico) a cui un utente si collega possiamo stabilire, per ciascuna pubblicazione, quali server sono i Publisher e quali i Subscriber. Vediamo ora i passi necessari per determinare il meccanismo di replica dei dati dai server periferici verso il Master Server (procedimento che chiameremo PUSH) e quello opposto dal Master Server alla periferia (chiamato PULL). Una rappresentazione topologia del processo è mostrata in figura 4.6.

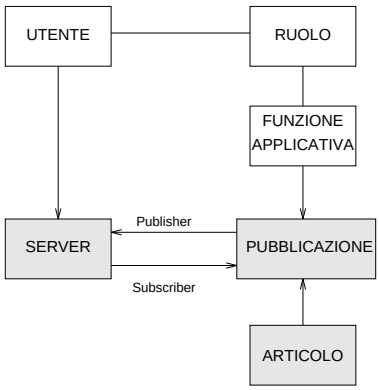


Figura 4.5. Definizione della pubblicazione

Procedimento di Pubblicazione nei server periferici (PUSH) Per ciascun server periferico occorre definire l'opportuna pubblicazione:

- Si parte da tutti gli Utenti di un Server.
- Si determinano le funzioni in OR su cui gli utenti hanno diritti di possesso (inserimento/modifica).
- Da queste funzioni determiniamo gli articoli attraverso il mapping Funzione-entità (con gli eventuali collegamenti tramite strutture part-of e foreign keys descritte nel dizionario dati).
- Sempre dal dizionario dei dati possiamo determinare, per ciascun articolo, la restrizione relativa ai dati da pubblicare (che è l'OR tra quelle degli utenti considerati).

Ogni pubblicazione è descritta da una quaterna:

1. Publisher: server in esame
2. Pubblicazione: una per server
3. Articoli: dal procedimento tratteggiato sopra
4. Subscriber: Master Server

Procedimento di Pubblicazione nel Master Server (PULL)

Al Master Server definiamo una pubblicazione relativa alle copie che interessano i server periferici:

- Per ciascun server consideriamo tutti gli Utenti
- Si determinano le funzioni in OR su cui gli utenti hanno diritti di visualizzazione.
- Da queste funzioni determiniamo gli articoli attraverso il mapping Funzione-entità (strutture part-of e foreign keys descritte nel dizionario dati).
- Sempre dal dizionario dei dati possiamo determinare, per ciascun articolo, la restrizione relativa ai dati da pubblicare (che è l'OR tra quelle degli utenti considerati).

- Definiamo la pubblicazione restricted al solo server considerato. Se esistono altri server con le stesse caratteristiche riguardo alle funzioni, possiamo aumentare la visibilità della pubblicazione anziché definirne una identica.

Quaterna che descrive la pubblicazione:

1. Publisher: Master Server
2. Pubblicazione: una per ciascun server periferico (a meno di pubblicazioni identiche)
3. Articoli: dal procedimento tratteggiato sopra
4. Subscriber: server in esame

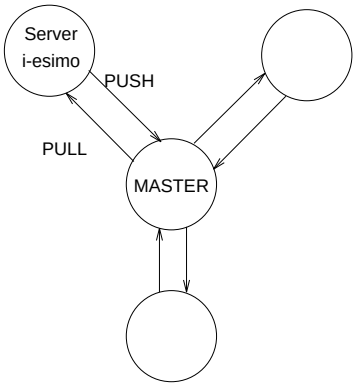


Figura 4.6. Topologia di replica a stella

Da notare come le pubblicazioni PUSH e PULL relative allo stesso server periferico i-esimo debbano avere partizioni sui dati mutualmente esclusive per non incorrere in fallimenti del processo di replica dovuto al tentativo ciclico di aggiornamento dello stesso record.

La soluzione proposta è ottimale in caso di flussi di dati che seguono una gerarchia predefinita o quando è richiesto un sito centrale master che deve mantenere una copia di tutti i dati. Risulta invece pessima nel caso di flussi a "collana" in cui ogni server spedisce ad uno solo e riceve da uno solo: in questo caso il traffico di rete raddoppia rispetto ad una soluzione senza master poiché il costo di replica è dato dall'invio dei dati dal server

periferico a quello centrale sommato al costo di invio dal sito centrale alla periferia rispetto ad un unica trasmissione tra i due siti periferici.

Questo modello funziona anche nel caso di Static Ownership su viste, qualora si risolva (in modo indipendente) il problema della chiave univoca.

Gestione della duplicazione della chiave primaria

Il modello considerato permette l'inserimento dei dati di una tabella da parte di più siti, introducendo problemi relativi alla duplicazione della chiave primaria durante gli inserimenti. Le strategie adottate per risolvere il problema sono due: rilevazione e risoluzione dei conflitti da una parte e prevenzione dei conflitti dall'altra.

Rilevazione e prevenzione dei conflitti

In questa soluzione ipotizziamo la presenza di un sito centrale (detto Master) contenente tutti i dati della tabella che ha il compito di validare gli inserimenti dei dati effettuati ai siti remoti. L'inserimento di un nuovo record avviene perciò in due fasi: la prima durante la quale il sito remoto propone l'inserimento inoltrandolo al Master e la seconda nella quale il Master valida o meno l'operazione.

Vediamo, in dettaglio, i passi necessari per l'inserimento:

- La tabella considerata contiene un campo aggiuntivo chiamato INS che assume il valore:
  - WAIT durante l'intervallo di tempo in cui e' già stato inserito localmente ma non ancora validato dal Master
  - OK quando il master accetta l'inserimento
- Le pubblicazioni dei server locali verso il Master hanno una clausola aggiuntiva sul campo INS sul valore 'OK', in modo da pubblicare solo i record validati dal Master
- Quando un sito locale inserisce un nuovo record marca il campo INS in "attesa di approvazione" ('WAIT') e, contestualmente, invia al Master la richiesta del nuovo inserimento
- Il Master confronta il record in richiesta di inserimento con i propri dati: se non c'è conflitto sul valore della chiave primaria comunica al

server locale la decisione positiva riguardo l'inserimento. Se invece viene rilevato un conflitto sulla chiave primaria il Master invia l'opportuno messaggio al server locale

- Il server locale agisce in funzione della risposta del Master:
  - se la risposta e' affermativa modifica il campo INS in 'OK' cosi' da replicare il record verso il Master
  - se la risposta e' negativa elimina il record precedentemente inserito
- Come variante, il Master può inviare, in caso di conflitto, un nuovo valore di chiave per il record: le operazioni sono analoghe alle precedenti con la (ovvia) modifica della chiave del record.

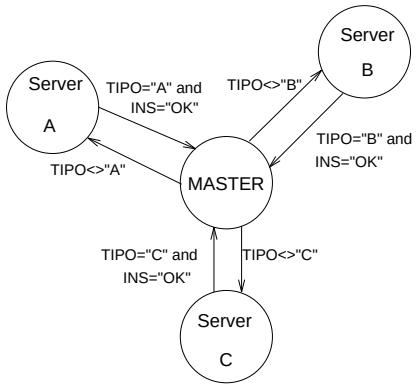


Figura 4.7. Esempio di applicazione con replica

In figura 4.7 viene presentato un esempio che descrive la topologia della replica per la rilevazione e la risoluzione dei conflitti: in figura, i nodi rappresentano i server, mentre gli archi orientati indicano la direzione dei dati replicati specificando le restrizioni dei campi interessati (viene ipotizzato il campo TIPO).

Prevenzione dei conflitti

Viene eliminata la possibilità di inserimenti di chiavi duplicate da parte di due siti diversi.

Le tecniche disponibili sono:

- Se la chiave è un "protocollo" è sufficiente un vincolo organizzativo (fasce di numerazione);

- Se la chiave è un identificatore si può partizionare;
- Se la chiave è significativa bisogna aggiungere alla chiave stessa un attributo che identifichi univocamente il server origine.

Riconfigurazione del modello static ownership

Consideriamo le operazioni necessarie per riportare la situazione in uno stato consistente a fronte di modifiche delle specifiche dell'applicazione.

Modifica degli abbonati (Subscriber)

Distinguiamo diverse tipologie di variazione dello scenario: utente A che si sposta dal server1 al server2.

- Prima soluzione:
  - Al Master Server la pubblicazione relativa al server1 viene resa visibile anche al server2
  - Al server2 viene sottoscritta la pubblicazione modificata dal Master che ora è visibile al server.

  La riconfigurazione è molto semplice, può avvenire senza l'interruzione dell'applicazione, ma ha lo svantaggio di duplicare un insieme di dati sovrabbondante poiché il server2 riceve i dati di interesse di tutti gli utenti del server1 e non solo dell'utente A che si è spostato; inoltre il server1 continua a ricevere le copie richieste dall'utente A anche se non sono più necessarie.
- Seconda soluzione:
  - Al Master Server viene creata una pubblicazione visibile al solo server2 che contiene i dati di cui ha bisogno l'utente A
  - Al server2 viene sottoscritta la nuova pubblicazione creata al Master Server.

Questo processo richiede un carico di lavoro aggiuntivo dovuto alla creazione della nuova pubblicazione, ma garantisce che il traffico di rete sia limitato a dati utili. In realtà per eliminare tutte le ridondanze occorre togliere nelle pubblicazioni verso il server1 gli articoli riguardanti il solo utente A. La

procedura aggiuntiva, non priva di costo, è la stessa di quella descritta più avanti per l'eliminazione di un utente da un server.

Nuovo abbonato

Viene utilizzata la seconda soluzione del caso precedente.

Eliminazione di un utente A abbonato al server1

- Prima soluzione: Non facciamo nulla. In questa ipotesi il Master continua ad inviare al server1 i dati di interesse dell'utente A con evidente sovraccarico della rete per informazioni prive di interesse. Il vantaggio è quello di non dover interrompere l'applicazione.
- Seconda soluzione:
  - Al server1 interrompiamo ed eliminiamo le sottoscrizioni verso il Master Server
  - Al Master Server viene modificata la pubblicazione relativa al server1 tenendo conto dell'eliminazione dell'utente A
  - Al server1 viene nuovamente generata la sottoscrizione alla pubblicazione modificata.

La soluzione garantisce il minimo flusso di dati (solo quelli necessari) in rete ma necessita dell'interruzione dell'applicazione (in particolare la parte relativa al server1) durante la riconfigurazione.

Modifica dei ruoli (riguardanti diritti di sola lettura) di un utente

Aggiunta di un ruolo ad un utente A al server1 nel Dizionario dei Dati

- Al Master Server viene creata una nuova pubblicazione contenente i nuovi dati visibili all'utente A
- Al server1 viene creata una nuova sottoscrizione alla pubblicazione creata da Master Server

La soluzione non comporta la sospensione dell'applicazione ma, in caso di modifiche frequenti, genera un'esplosione delle pubblicazioni che riducono l'efficienza di SQL Server. La soluzione che non genera nuove pubblicazioni è off-line e consiste nel modificare la pubblicazione PULL relativa al server1.

Eliminazione di un ruolo ad un utente A nel server1

La soluzione è simile al caso di eliminazione di un abbonato con l'unica variante che, nel caso di modifica della pubblicazione al Master Server si tiene conto dell'eliminazione di un solo ruolo e non dell'utente.

Modifica dei diritti di accesso (in sola lettura) di un ruolo

E un'operazione che avviene off-line poiché, potenzialmente, coinvolge tutti i siti.

- Nel Dizionario dei Dati vengono modificati i diritti di accesso di un particolare ruolo
- Eliminiamo tutte le sottoscrizioni di tutti i siti periferici
- Al Master Server vengono ridefinite tutte le pubblicazioni che coinvolgono il ruolo modificato
- In ciascun sito periferico vengono ridefinite le opportune sottoscrizioni.

Il sovraccarico di lavoro è notevole ed è dovuto soprattutto alla sincronizzazione iniziale, paragonabile a quello necessario per il setup dell'applicazione.

Modifica degli editori (Publisher)

Distinguiamo diverse tipologie di variazione dello scenario:

Utente A si sposta dal server1 al server2. L' utente si sposta permanentemente, analizziamo la sequenza:

- Occorre portare al server2 i dati in possesso dell'utente A Diverse alternative:
  - Sincronizzazione manuale al server2 da parte dell'utente
  - Al server2, sottoscrizione della pubblicazione del Master relativa al server1
  - Al Master, creazione di una pubblicazione dei soli dati di proprietà dell'utente A che viene sottoscritta al server2

- Al server2 viene generata una nuova pubblicazione contenente i dati in possesso dell'utente A
- Al Master Server viene creata una sottoscrizione alla pubblicazione creata al server2
- Non è necessario eliminare la parte di pubblicazione del server1 relativa ai dati in possesso dell'utente A poiché nessuno li può modificare e quindi, supponendo una replica Transaction Based, dal server1 non viene inviata nessuna nuova transazione. Se invece la replica è di tipo Scheduled Refresh occorre modificare la pubblicazione al server1 eliminando gli articoli riferiti all'utente A in modo da evitare conflitti derivanti dalla pubblicazione degli stessi dati da due siti diversi.

Utente A che si sposta (temporaneamente) dal server1 al server2. Questo caso differisce dal precedente poiché si vuole mantenere aggiornati i dati del server1 durante il periodo in cui l'utente A si trova al server2. Supponiamo, per semplicità che il solo utente A sia presente al server1.

- Al Master Server viene eliminata la sottoscrizione alla pubblicazione del server1
- Viene sincronizzato il server2 con i dati in possesso dell'utente A (vedi caso precedente)
- Al server1 viene sottoscritta la pubblicazione del Master Server relativa ai dati in possesso dell'utente A
- Al server2 viene creata una pubblicazione contenente i dati in possesso dell'utente A
- Al Master Server viene creata una sottoscrizione alla pubblicazione creata al server2

Nel caso in cui sia noto a priori l'insieme dei server ai quali l'utente A può collegarsi risulta conveniente mantenere una copia dei dati su ciascuno di questi server per semplificare le operazioni di riconfigurazione.

Inserimento di un nuovo utente A al server1. Il server1 è possessore di un insieme di dati.

- Al server1 viene creata una nuova pubblicazione contenente i dati in possesso dell'utente A
- Al Master Server viene creata una sottoscrizione alla pubblicazione creata al server1
- Al Master Server viene creata una pubblicazione unrestricted degli stessi dati in modo che ciascun server periferico può, se vuole, abbonarsi ai dati in possesso del nuovo utente A.

Eliminazione di un utente A al server1. Sono possibili due soluzioni alternative:

- Prima soluzione: Non faccio nulla. Soluzione a costo zero, praticabile in caso di replica transaction based, che pone problemi di tipo semantico poiché i vecchi abbonati continuano ad accedere (in lettura) ai dati di proprietà di un utente che non esiste più.
- Seconda soluzione:
  - Al Master Server viene interrotta la sottoscrizione della pubblicazione del server1
  - Al server1, nella pubblicazione esistente, vengono eliminati gli articoli contenenti i dati di proprietà dell'utente A
  - Al Master Server viene ridefinita la sottoscrizione precedentemente interrotta.

A seconda delle opzioni scelte nella definizione degli articoli è possibile:

- mantenere nei subscriber i dati già presenti all'atto dell'eliminazione di A
- eliminare dai subscriber la copia dei dati di proprietà di A.

Modifica dei diritti di possesso dell'utente A sul server1. Aggiunta di un ruolo (con diritti di possesso) all'utente A del server1

- Al server1 viene creata una pubblicazione contenente i nuovi dati in possesso all'utente A
- Al Master Server viene creata una sottoscrizione della nuova pubblicazione creata dal server1

- Al Master Server viene creata una pubblicazione dei dati visibile ai server periferici che mantengono utenti con il nuovo ruolo
- Ciascun server periferico autorizzato si abbona alla pubblicazione del Master Server.

Eliminazione di un ruolo dell'utente A del server1. Supponiamo che l' utente A abbia diritti di possesso.

- Prima soluzione: Non faccio nulla. Soluzione a costo zero, praticabile in caso di replica transaction based, che pone problemi di tipo semantico poiché i vecchi abbonati continuano ad accedere (in lettura) ai dati di un utente non ha più il diritto di proprietà su tali dati.
- Seconda soluzione:
  - Al Master Server viene interrotta la sottoscrizione relativa al server1
  - Al server1 viene ricreata la pubblicazione PUSH opportuna
  - Al Master Server viene creata una sottoscrizione alla pubblicazione del server1.

Modifica dei diritti sui dati di un ruolo. E un'operazione che avviene off-line poiché, potenzialmente, coinvolge tutti i siti.

- Nel Dizionario dei Dati vengono modificati i diritti di modifica di un particolare ruolo
- Al Master Server eliminiamo tutte le sottoscrizioni
- In ciascun sito periferico vengono ridefinite tutte le pubblicazioni che coinvolgono il ruolo modificato
- Al Master Server ridefiniamo tutte le sottoscrizioni ai siti periferici.

Il sovraccarico di lavoro è notevole ed è dovuto soprattutto alla sincronizzazione iniziale, paragonabile a quello necessario per il setup dell'applicazione.

4.7 Funzionamento del Sistema di Replica

4.7.1 Modello Static Ownership

Con questo modello la gestione della replica viene demandata al sistema di replica del DBMS (SQL Server) che viene integrato con funzioni "ad hoc" per semplificare le funzionalità di setup iniziale e di riconfigurazione. Le operazioni da implementare sono le seguenti:

Pubblicazioni PUSH definite nei siti periferici

- A partire dagli Utenti-Ruoli presenti nel sito determiniamo le funzioni svolte (con gli eventuali parametri)
- Per ciascuna funzione eseguiamo il mapping Funzione-Entità che permette di determinare le tabelle coinvolte. Il mapping deve specificare, per una data funzione, le tabelle in proprietà degli utenti del sito (con diritti di modifica) e quali sono solamente referenziate con operazioni read-only. Ad esempio, considerando una funzione abilitata alla modifica delle fatture nel Database di riferimento, il mapping rileva le tabelle Fatture e Righe fattura in proprietà e le tabelle Codice IVA e Articoli referenziate read-only.
- Ciascuna tabella selezionata con diritti di proprietà diviene un articolo della pubblicazione, aggiundendo eventuali predicati di selezione specificati dai parametri del Sistema di Sicurezza.

Pubblicazioni PULL definite al Master Server

Al Master Server viene creata una pubblicazione restricted per ciascun sito periferico che contiene i soli dati necessari alle funzioni associate al sito.

- Per ciascun sito periferico, a partire dagli Utenti-Ruoli presenti determiniamo le funzioni svolte (con gli eventuali parametri)
- Di nuovo, per ciascuna funzione eseguiamo il mapping Funzione-Entità che permette di determinare le tabelle coinvolte. Il mapping deve specificare, per una data funzione, le tabelle in proprietà degli utenti del sito (con diritti di modifica) e quali sono solamente referenziate con operazioni read-only.

- Ciascuna tabella selezionata con diritti di sola lettura diviene un articolo della pubblicazione restricted al sito periferico considerato, aggiundendo eventuali predicati di selezioni specificati dai parametri del Sistema di Sicurezza. Nel caso in cui una tabella sia selezionata nello stesso sito con la duplice caratterizzazione in proprietà ed in sola lettura, l'articolo ottenuto è dato dalla differenza tra l'insieme dei dati richiesti in sola lettura e quelli in possesso.

ESEMPIO

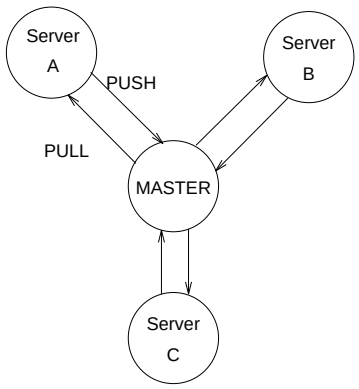


Figura 4.8. Esempio

Supponiamo che al Server A sia compiuta la funzione Amministrativa riguardante la fatturazione delle merci, al Server B e C sia compiuta la funzione Magazzino riguardante la gestione degli Articoli mantenuti nel proprio magazzino (supponiamo merci di tipo diverso tra i due siti). Inoltre al server C viene compiuta la funzione di Verifica Fatture che svolge la verifica contabile di tutte le fatture emesse. Vediamo come viene eseguito il setup dell'applicazione.

Pubblicazioni PUSH:

SERVER A Il mapping Funzione-Entità per la funzione Amministrativa determina le tabelle Fatture e Righe fattura in proprietà e la tabella Articoli referenziata in sola lettura. Per quanto riguarda la tabella Codici IVA supponiamo che la funzione si occupi anche della gestione dell'IVA e quindi viene rilevata in proprietà. Sotto l'ipotesi che la

funzione Amministrativa gestisce tutte le fatture non abbiamo parametri di restrizione per le tabelle selezionate. La pubblicazione è quindi costituita dalle tabelle Fatture, Righe fattura e Codici IVA senza partizionamenti.

SERVER B Dalla funzione Magazzino otteniamo la tabella Articoli parametrizzata sull'attributo TIPO (ad esempio "B") con diritti di possesso. Ne consegue che la pubblicazione è data dalla sola tabella Articoli con il vincolo sull'attributo TIPO (= "B").

SERVER C Il caso della funzione Magazzino è analogo al precedente con la sola differenza che il parametro è diverso (es. TIPO = "C"). In più al Server C viene compiuta la funzione Verifica Contabile che coinvolge, in sola lettura, le tabelle Fatture, Righe fattura, Codici IVA e Articoli. La pubblicazione, considerando le sole tabelle in proprietà, sarà costituita dalla tabella Articoli ristretta alla clausola TIPO = "C".

Pubblicazioni PULL:

Al Master Server, per ciascun sito periferico viene definita una pubblicazione.

SERVER A Di nuovo analizziamo le funzioni compiute al Server A, interessandoci delle tabelle con diritti di sola lettura: in questo caso si tratta della tabella Articoli (senza parametri). La pubblicazione è data dalla sola tabella Articoli.

SERVER B Dal mapping Funzione-Entità risulta che non occorre alcuna pubblicazione poiché non vi sono tabelle con diritti di sola lettura.

SERVER C Occorrono, in lettura, le tabelle Fatture, Righe fattura, Codici IVA ed Articoli. La pubblicazione contiene quindi le tabelle Fatture, Righe fattura, Codici IVA non partizionate ed Articoli ristretta alla condizione TIPO <> "C" poiché escludiamo i dati già presenti (in possesso) al Server C.

4.7.2 Modello Dynamic Ownership

Il modello Dynamic Ownership prevede, al solito, l'unicità del proprietario di un record con la possibilità del passaggio di proprietà da un utente ad un

altro durante il ciclo di vita dei dati. Il possesso di un record da parte di un utente è definito dal valore assunto dai dati(a cui è associato uno stato) ed è descritto da un diagramma a stati che individua il proprietario de dati per ciascuno stato permesso e da un diagramma di transizioni di stato che descrive l'evoluzione di stato permessa ai dati. Questi due diagrammi, che possiamo rappresentare sia in forma tabellare che tramite grafo, vengono memorizzati nel Dizionario dei Dati dell'applicazione sotto forma di tabelle che determinano:

- lo stato: per ciascuna relazione (tabella) individuano gli stati permessi con i relativi proprietari
- le transizioni permesse: per ciascuna relazione (tabella) individuano le transizioni ammesse da uno stato di partenza ad uno di arrivo.

Queste informazioni sono utilizzate dalle varie componenti dell'applicazione per distinguere le operazioni permesse da quelle vietate. La logica che è associata ai controlli sulle operazioni di inserimento, aggiornamento e cancellazione dati è la seguente:

- ogni utente/ruolo ha il permesso di inserire e cancellare solo i record aventi lo stato compatibile con quelli di cui risulta proprietario
- ogni utente/ruolo può modificare solamente i record di cui risulta proprietario all'atto dell'aggiornamento portandoli in uno stato finale compatibile con le transizioni permesse
- una transizione può portare il record in uno stato associato ad un utente/ruolo diverso da quello dello stato di partenza permettendo in questo modo il passaggio di proprietà dei singoli record
- sono previsti metodi di avocazione/delega dei dati da un utente proprietario ad un altro diverse da quello "normale" della transizione di stato che vengono descritti separatamente.

Il sistema di sicurezza dell' applicazione dovrà garantire il controllo dei permessi di inserimento e di cancellazione dei dati. Ciascun utente/ruolo collegato all'applicazione è abilitato ad eseguire specifiche funzioni. A ciascuna funzione, tramite il Sistema di Sicurezza, associamo un insieme di azioni permesse. Tra queste, quelle che inseriscono od eliminano dati vengono parametrizzate sullo stato (o gli stati) in possesso dell'utente/ruolo specificato. In

questo modo viene garantito che solo l'utente/ruolo che detiene i diritti può inserire o cancellare opportunamente i record di una particolare relazione.

Inoltre dovrà assicurare il controllo dell'aggiornamento dei dati, cioè le azioni di modifica (update) dei dati sono parametrizzate sullo stato (o sugli stati) che l'utente/ruolo collegato ha in possesso. Inoltre l'aggiornamento viene effettivamente eseguito solo se lo stato finale del record è identico a quello iniziale oppure è tra quelli previsti dallo schema delle transizioni. Questo controllo aggiuntivo sul valore finale di un record non è attualmente presente nel Sistema di Sicurezza.

Descrizione dello scenario di replica

Anche per il modello Dynamic Ownership pensiamo ad una topologia di replica di tipo a stella dotata di un sito centrale (Master Server) che funge da filtro per tutti i dati di interesse dell'applicazione. Il procedimento di definizione delle pubblicazioni è lo stesso tratteggiato per il modello static-ownership in cui ogni server periferico invia verso il Master i dati di cui risulta possessore e, viceversa, il Master Server pubblica verso ciascun server periferico i dati sui quali quest'ultimo ha solo diritti di lettura (e non di scrittura): in questo modello risulta semplice la definizione delle restrizioni sui dati in quanto sono basate sullo stato del record. Per permettere la variazione di possesso dei dati integrata con il processo di replica di SQL Server, definiamo la seguente procedura di operazioni che viene eseguita dai server:

- Tutte le tabelle utilizzate nel Dynamic Ownership possiedono un campo aggiuntivo che indica la posizione di "attivo" o "congelato" di un record. Il campo di validità fa parte della chiave primaria della relazione e si assume che solo i record marcati "validi" siano la versione corrente del dato, mentre gli altri rappresentano versioni precedenti non aggiornate

- Quando un utente/ruolo desidera modificare un record vengono attivate le procedure di controllo sull'azione che verificano che lo stato finale sia tra quelli permessi dal diagramma delle transizioni. Se lo stato finale dell'aggiornamento è riferito ad un qualsiasi utente/ruolo del server su cui avviene l'operazione non è necessario compiere alcuna procedura aggiuntiva. In caso contrario, cioè quando l'aggiornamento porta il

record in uno stato di proprietà di un utente di un altro server occorre porre il record nella condizione di "congelato"

- L'operazione di aggiornamento (update) viene trasformata in una equivalente di inserimento (cioè l'insert di un record identico a quello finale ottenuto con l'aggiornamento) forzando il campo di validità "attivo". Questa operazione di insert viene inviata al Master via RPC

- Il Master, ricevuta la richiesta di inserimento, in base al nuovo stato del record determina il server nel quale si trova il nuovo proprietario inviandogli la RPC

- Nel server periferico che ha ricevuto la RPC viene eseguito l'inserimento del record che, previo il tempo di latenza della replica, verrà pubblicato prima verso il Master Server e, di qui, portato agli altri siti periferici

- Periodicamente, ai siti periferici eliminiamo tutti i record non validi. Questa operazione è utile per eliminare al ridondanza dei dati ma non preclude il regolare funzionamento delle operazioni.

Tramite questa semplice procedura possiamo garantire che il passaggio di proprietà di un record avvenga con tempo di latenza massimo pari a:

$$T_{RPC} + 2 \cdot T_{REPLICA}$$

T_{RPC}: tempo di trasferimento dell'operazione di inserimento tra i due server coinvolti;

$T_{REPLICA}$: tempo di latenza del processo di replica (doppio perchè si passa sempre dal Master).

Inoltre il modello garantisce che, durante il periodo di latenza, nessun utente/ruolo abbia il permesso di modifica dei dati validi, in quanto durante l'intervallo di latenza il dato viene congelato e solo il nuovo proprietario ha il diritto di validare il record.

Possibilità di Versioning

Variando leggermente la sequenza delle operazioni descritte precedentemente è possibile definire versioni diverse del DataBase, costruendo in questo modo la "storia" di un record.

- Definiamo il campo di validità come intero che assume valore 0 per indicare il dato "attivo" e > 0 per indicare che si tratta di una versione precedente
- Quando il server periferico vuole modificare il record lo "congela" ponendo il campo di validità superiore di una unità al massimo valore di validità già presente per quel record.
- Non viene eseguita la cancellazione periodica dei record "congelati"

In questo modo vengono mantenute tutte le versioni di un record che si sono susseguite durante l'esecuzione dell'applicazione: la versione valida ha il campo relativo a 0, mentre le altre assumono valori crescenti del campo di validità rappresentando versioni cronologicamente successive.

Esempio completo di Dynamic Ownership

Vediamo come viene configurata l'applicazione descritta precedentemente e le cui tabelle di stato e delle transizioni sono:
Tabella dei possessori di un ordine:

	Provvisorio	In attesa	Approvato	Annullato
Venditore	Prop	Read/Only	Read/Only	Prop
Dir.Comm.	Read/Only	Prop	Prop	

Transizioni di stato di un ordine permesse:

	Provvisorio	In attesa	Approvato	Annullato
Provvisorio	Dir.Comm	Venditore		Venditore
In attesa	Dir.Comm	Dir.Comm	Dir.Comm	
Approvato	Dir.Comm			
Annullato	Venditore	Venditore		

L'applicazione prevede la presenza di tre server distinti (possono essere anche due soli in cui più ruoli coesistono):

- Master Server
- Server a cui è collegato il Venditore
- Server a cui è collegata la Direzione Commerciale

Vediamo la definizione delle pubblicazioni:

- Pubblicazione al server del Venditore:
  - Publisher: Server Venditore
  - Subscriber: Master Server
  - Articoli: tabella ordini (e righe ordini) ristretta agli stati Provvisorio e Annullato
- Pubblicazione al server della Direzione Commerciale:
  - Publisher: Server Direzione Commerciale
  - Subscriber: Master Server
  - Articoli: tabella ordini (e righe ordini) ristretta agli stati In Attesa di Approvazione e Approvato
- Pubblicazione 1 al Master server:
  - Publisher: Master Server
  - Subscriber: Server Venditore
  - Articoli: tabella ordini (e righe ordini) ristretta agli stati In Attesa di Approvazione e Approvato
- Pubblicazione 2 al Master server:
  - Publisher: Master Server
  - Subscriber: Server Direzione Commerciale
  - Articoli: tabella ordini (e righe ordini) ristretta allo stato Provvisorio

Azioni permesse al Venditore:

- Aggiornamento da Provvisorio a Provvisorio.
- Aggiornamento da Provvisorio a In Attesa di Approvazione.
- Aggiornamento da Provvisorio ad Annullato.

- Aggiornamento da Annullato a Provvisorio.
- Aggiornamento da Annullato a In Attesa di Approvazione.

Azioni permesse alla Direzione Commerciale:

- Aggiornamento da In Attesa di Appr. a In Attesa di Appr.
- Aggiornamento da In Attesa di Approvazione a Provvisorio.
- Aggiornamento da In Attesa di Approvazione a Approvato.
- Aggiornamento da Approvato a Provvisorio.

Supponiamo di avere la seguente situazione riguardo i record della relazione Ordini (in tutti i server):

Codice	Validità	Stato	Descr
1	0	"Provvisorio"	"AAAAAA"
2	0	"In Attesa"	"BBBBBB"
3	0	"Approvato"	"CCCCCC"

Supponiamo che il Venditore voglia sottoporre alla Direzione Commerciale l'ordine di codice 1 per l'approvazione.

1. Il Venditore "congela" l'ordine di codice 1 ponendo il campo valido = 1.

   Dopo il tempo di latenza di replica avremo in tutti i server:

Codice	Validità	Stato	Descr
1	1	"Provvisorio"	"AAAAAA"
2	0	"In Attesa"	"BBBBBB"
3	0	"Approvato"	"CCCCCC"

2. L'operazione di inserimento viene eseguita dapprima alla Direzione Commerciale che poi la replica al Master e di qui a tutti gli altri server:

Codice	Validità	Stato	Descr
1	1	"Provvisorio"	"AAAAAA"
1	0	"In Attesa"	"AAAAAA"
2	0	"In Attesa"	"BBBBBB"
3	0	"Approvato"	"CCCCCC"

Quindi adesso l'ordine di codice 1 valido è quello nello stato "In Attesa" di cui risulta proprietaria la Direzione Commerciale che è l'unica ad avere il permesso di aggiornamento. Inoltre è presente anche una copia congelata del record che rappresenta il valore precedente all'aggiornamento (versioning).

Capitolo 5

SQL Server: l' ambiente software di implementazione

Prima di analizzare l' implementazione dei modelli esaminati, in particolare, i modelli static ownership e dynamic ownership con topologia a stella, occorre introdurre diversi aspetti legati all' ambiente software utilizzato per lo sviluppo e il test.

L' applicazione realizzata permette di generare gli script (ovvero un file contenente uno o più programmi T-SQL[1] da eseguire in batch) di inizializzazione e gestione della replica nei sistemi distribuiti in modo completamente automatico, facilitando il compito dell' amministratore del sistema (che nel resto del capitolo verrà chiamato anche "utente"), con pochi dati sarà possibile configurare completamente le repliche in tutti i server della rete ed utilizzare automaticamente il modello di replica prescelto.

Gli script generati sono stati sviluppati sulla seguente piattaforma software: DBMS Microsoft SQL Server 6.5, sistema operativo Microsoft Windows NT 3.51/4.0 Server. Questi software devono essere installati su tutti i server appartenenti al sistema distribuito. Uno di questi deve essere configurato utilizzando Windows NT 4.0 Server con l' opzione "primary server" che lo identifica come responsabile del controllo delle operazioni di replica.

Nei prossimi paragrafi introdurremo alcuni strumenti software che permettono di scrivere ed eseguire transazioni distribuite e chiamate a procedure remote, che sono stati necessari per le implementazioni presentate nel

[1]Il Transact-SQL (*T-SQL*) è una estensione Microsoft del linguaggio ANSI SQL, consente controllo di flusso, stored procedures, triggers e comandi specifici per l' amministrazione dei database.

capitolo successivo.

5.1 Remote Procedure Call

Le procedure remote *Remote Procedure Call* (RPC) rappresentano il passaggio dall' usuale programmazione centralizzata ad un tipo di programmazione distribuita, in cui non solo si hanno i dati situati su diversi siti, ma anche porzioni di codice (procedure e funzioni) che possono essere eseguite in remoto mediante una "chiamata a procedura remota".

Lo stile di programmazione centralizzata, utilizzato attualmente per la realizzazione di programmi in ambienti non distribuiti, è basato sulla chiamata di procedure: ciascuna svolge un determinato e ristretto compito. Un programma può essere rappresentato schematicamente come in figura 5.1.

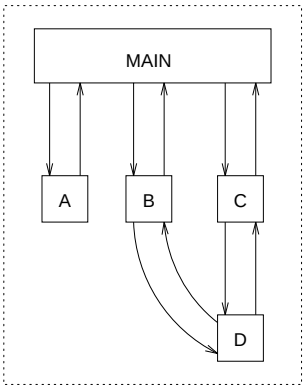


Figura 5.1. Programmazione procedurale centralizzata

In un sistema distribuito è utile associare ai dati le procedure per la loro gestione, così che l' elaborazione è piú veloce in quando i dati sono sempre locali all' applicazione. Sulla rete passano solo i parametri di input ed eventualmente ritornano i parametri di output. La figura 5.2 rappresenta schematicamente un' esempio di chiamata a RPC.

Le RPC nate per gli ambienti client/server, in cui si sfruttava la maggior potenza di calcolo del server anche per eseguire le procedure (*computational server*), sono poi state estese ai sistemi distribuiti: ogni server possiede le procedure necessarie per la gestione dei propri dati.

Analizziamo il funzionamento delle RPC nei sistemi distribuiti, a tal fine consideriamo due siti: uno lo definiamo "server" in quanto possiede le RPC ed è il sito in cui avviene l' elaborazione dei parametri della procedura, l'

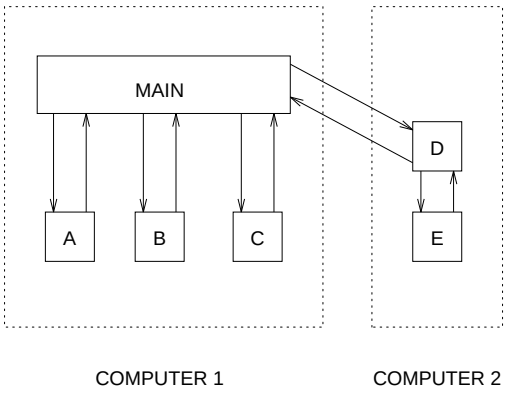


Figura 5.2. Utilizzo di Remote Procedure Call

altro "client" poiché richiede i servizi al server. Occorre notare che questa terminologia è utilizzata unicamente per stabilire chi offre e chi richiede un servizio in un determinato intervallo temporale, infatti i ruoli non sono fissi e dipendono dalla procedura che si deve utilizzare (il sito che l' ha definita è il server).

Lo spazio di indirizzamento rimane sempre distinto, le chiamate avvengono unicamente attraverso lo scambio di pacchetti attraverso la rete. La figura 5.3 mostra come avviene una chiamata.

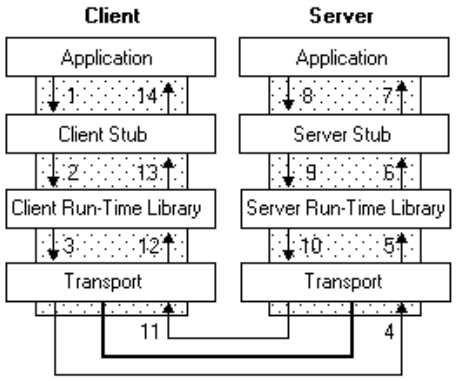


Figura 5.3. Schema di una chiamata a procedura remota

Il client effettua la chiamata localmente ad una propria interfaccia (*client stub*), da questo livello inizia, attraverso un mapping che permette di localizzare la procedura, l' esecuzione vera e propria che comporta il trasferimento dei parametri verso il server.

Il client effettua le seguenti operazioni:

- recupera i parametri necessari dal proprio spazio di indirizzamento;
- trasforma i parametri seguendo lo standard della rete di comunicazione (il *Network Data Representation* NDR);
- chiama la funzione nella libreria run-time locale che mappa la RPC remota e passa i parametri richiesti.

A questo punto è il sistema operativo ad occuparsi dei dettagli implementativi per l' esecuzione della richiesta.

Il server risponde ad una RPC eseguendo i seguenti passi:

- la libreria run-time del server accetta l' RPC e preleva i parametri dalla rete;
- i parametri vengono passati allo stub che effettua la conversione dal formato NDR al formato dei dati richieso dal server;
- viene eseguita la procedura.

Quando la procedura sul server è terminata, il risultato prodotto viene ritornato al client eseguendo una sequenza di operazioni simili alle precedenti, dal server verso il client.

5.1.1 OSF standard per le RPC

L' Open Software Foundation[2] (OSF) ha proposto uno standard per la realizzazione di ambienti distribuiti (*Distributed Computing Environment* DCE) rappresentato in figura 5.4. Le RPC sono alla base della struttura, e devono interagire con tutti gli altri moduli.

Le caratteristiche fondamentali delle RPC che lo standard DCE richiede sono le seguenti:

- Le RPC devono soddisfare le tre proprietà seguenti:
  1. Semplicità
  2. Trasparenza

[2]L' OSF è un consorzio di compagnie produttrici di software, che ha lo scopo di determinare standard industriali per la realizzazione dei prodotti software commerciali.

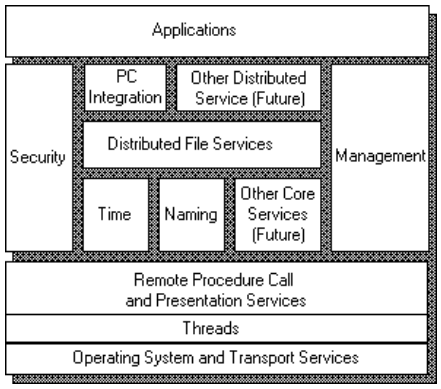


Figura 5.4. OSF distributed computing environment

3. Prestazioni

- Il modello di implementazione delle RPC deve aderire il più possibile al modello delle procedure locali (questo è per garantire la trasparenza).
- Il modello deve permettere l' indipendenza dal protocollo fisico di comunicazione, e deve supportare diversi protocolli di trasporto.
- Il modello di RPC deve interagire facilmente con gli altri moduli del DCE.
- Il modello deve permettere le RPC anche attraverso sistemi eterogenei. Questo è realizzato mediante il NDR.

Il modello OSF-DCE è stato adottato (non completamente, ma le differenze sono trascurabili) dalla Microsoft per realizzare i propri prodotti per i sistemi distribuiti.

Windows NT 4.0 mette a disposizione tutte le primitive per l' utilizzo delle RPC, ampiamente sfruttate dall' SQL Server attraverso il modulo MS-DTC (vedi paragrafo 5.3).

5.2 Transact-SQL

In questo paragrafo descriviamo brevemente le strutture che mette a disposizione il linguaggio Microsoft T-SQL e che non sono disponibili nell' ANSI SQL. Le variazioni sono soprattutto legate alla gestione delle procedure che permettono di mantenere la consistenza dei dati. Questo viene ottenuto mediante la scrittura di "trigger".

5.2.1 Stored procedures

Il T-SQL, come molti RDBMS commerciali, consente la definizione di stored procedure, cioè insiemi di comandi SQL definiti nel seguente modo:

```
create procedure <nome> [<parametri>]
as
<statements>
go
```

La prima volta che si esegue una stored procedure viene compilata e memorizzata dall' SQL Server in una apposita area associata al database nel quale è stata definita. In questo modo l' esecuzione successiva è più veloce e riduce l' overhead imposto dal caricamento e dall' interpretazione della procedura ad ogni esecuzione.

5.2.2 Triggers

Un trigger è un tipo particolare di procedura che è eseguita automaticamente quando un utente esegue un comando di modifica (insert, update o delete) su una tabella. I trigger sono di solito utilizzati per effettuare controlli sui dati immessi e per verificare l' integrità dei dati e mantenere uno stato consistente dell' intera base di dati.

La definizione di un trigger avviene tramite le seguenti istruzioni:

```
create trigger <name>
on table_name
for INSERT,UPDATE,DELETE
as sql_statements
```

Quando un utente immette uno statement SQL che altera il contenuto della tabella associata al trigger, il trigger viene automaticamente attivato dopo l' esecuzione dello statement. Prima dell' esecuzione delle istruzioni interne al trigger esiste una piccola parte di inizializzazione svolta automaticamente dal DDBMS, vengono infatti definite una o due tabelle temporanee che contengono le righe della tabella modificate. Tali tabelle vengono chiamate *inserted* e *deleted*. Avremo una sola tabella nel caso di un trigger associato ad istruzioni di INSERT o DELETE. Nel caso di INSERT viene creata solo la tabella *inserted*, che contiene le tuple immesse dalla istruzione utente. Nel

caso di DELETE è creata solo la tabella *deleted* con le tuple da eliminare prelevate dalla tabella del database. L' UPDATE possiede entrambe le tabelle, la inserted contiene le tuple dopo la modifica, la deleted le tuple originali prima della modifica.

Vediamo un piccolo esempio di un UPDATE:

Cod_art	Descr.	Prezzo	Categ.	Qta
1	Art 1	10000	C1	23
2	Art 2	8500	C1	3
3	Art 3	14500	C3	32
4	Art 4	3600	C4	4
5	Art 5	12500	C3	20

Con il seguente trigger:

```
create trigger articoli_trig for update as
/* Dichiarazione variabili globali */
declare @Qta int
declare @Qta1 int
/* Creazione del cursore sulla tabella inserted */
declare c_ins cursor for select Qta from inserted
open c_ins
/* Creazione del cursore sulla tabella deleted */
declare c_del cursor for select Qta from deleted
open c_del

declare @status int
select @status = 0

/* Preleva il primo dato puntato dal cursore su inserted */
fetch next from c_ins into @Qta
select @status = @@FETCH_STATUS
while @status = 0
begin
  /* Se modifichiamo la colonna Qta */
  if update(Qta)
  begin
    /* Preleva il dato in deleted */
    fetch next from c_del into @Qta1
    if @Qta > @Qta1+100 begin
```

```
      raiserror("Impossibile aggiungere piu' di 100 art.",ERR_100,1)
      rollback transaction
      break
    end
  end
  /* Preleva i valori successivi */
  fetch next from c_ins into @Qta
  select @status = @@FETCH_STATUS
end
/* Chiude i cursori e dealloca la memoria */
close c_ins, c_del
deallocate c_ins, c_del
```

Supponiamo l' utente effettui il seguente comando:

```
UPDATE articoli
   SET Qta = 100
 WHERE Categ = "C3"
```

La tabella viene modificata: in tutte le righe di categoria C3 viene portata la quantità a 100. A questo punto il trigger sull' UPDATE scatta: vengono create le seguenti due tabelle: inserted e deleted:

Cod_art	Descr.	Prezzo	Categ.	Q.tà
3	Art 3	14500	C3	100
5	Art 5	12500	C3	100

Tabella inserted

Cod_art	Descr.	Prezzo	Categ.	Q.tà
3	Art 3	14500	C3	32
5	Art 5	12500	C3	20

Tabella deleted

Ora il trigger può verificare se abbiamo eseguito un' UPDATE sulla colonna della quantità utilizzando lo statement `IF UPDATE()`. In caso affermativo effettua una comparazione tra i dati prima e dopo la modifica, e nel caso di errore (cioè se quantità immessa è maggiore di 100 unità di quella esistente) dopo aver visualizzato un messaggio e settato le variabili interne di errore di SQL Server (mediante l' istruzione `RAISERROR`) effettua un `ROLLBACK TRANSACTION` che termina la transazione annullando le modifiche fatte. Quindi le righe modificate tornano al loro valore originale. Altrimenti in caso di UPDATE corretto (come nell' esempio) il trigger si conclude senza rollback e la transazione termina col commit rendendo permanenti le modifiche.

Occorre evidenziare alcune cose: il trigger non sostituisce le normali procedure di controllo dell' integrità dell' SQL Server, in caso ad esempio di immissione di un duplicato di una chiave primaria sarà il DDBMS a informarci dell' errore e il trigger non viene eseguito.

Il trigger descritto come esempio prevede la gestione di modifiche a più righe contemporaneamente: per gestire ogni riga in modo separato vengono utilizzati i *cursori*, cioè opportune strutture che referenziano una riga alla volta dalla query a cui sono associati (nel nostro caso una semplice SELECT che proietta la colonna delle quantità). Con l' istruzione `FETCH NEXT` possiamo prelevare una riga indirizzata dal cursore e forzarla nella variabile specificata. L' utilizzo dei cursori è abbastanza costoso in termini di prestazioni, il suo utilizzo all' interno dei trigger è necessario per la gestione singola di query su righe multiple, ma comporta un notevole peggioramento delle prestazioni, occorre pertanto cercare di ottimizzare la sua esecuzione, eventualmente trasformando esternamente al trigger una modifica su più righe in diverse modifiche ad una sola riga anche utilizzando dei cursori ma esternamente al trigger. In questo modo internamente al trigger non è più necessario l' uso dei cursori ma sono sufficienti le normali istruzioni SQL e le prestazioni migliorano. A volte questo non è possibile (come nel progetto della tesi) e di conseguenza è necessario ottimizzarne l' utilizzo all' interno del trigger.

5.2.3 Replication Stored Procedures

Introduciamo ora alcune stored procedure fornite dal sistema dell' SQL Server per la gestione della replica. Queste procedure sono create dal DBMS all' inizializzazione del sistema di replica e sono associate al database master. Sono utilizzate intenamente dall' SQL Server quando l' utente decide di configurare ed utilizzare la replica sfruttando l' SQL Server Enterprise che è un programma che consente di amministrare i siti locali e remoti in modo visuale. Tuttavia, non sempre la gestione standard della replica soddisfa i requisiti richiesti dagli utenti, di conseguenza l' amministratore del sistema può decidere di configurare autonomamente un sistema di replica ad-hoc per l' applicazione, in modo tale da avere una gestione personalizzata.

Le procedure di gestione della replica in generale possono essere eseguite solo dagli utenti con i previlegi di amministratore, e in alcuni casi solo dai proprietari del database.

In totale le procedure per la replica fornite dall' SQL Server sono 45 suddivise in 6 gruppi funzionali:

1. Server configuration
2. Publication administration tasks
3. Subscription administration tasks
4. Replication operations
5. Replicated transaction management
6. Scheduling

Presentiamo ora per ogni gruppo funzionale le procedure principali, tralasciando quelle che hanno un utilizzo marginale o servono solo in fase di debug. Vedremo brevemente una descrizione del loro funzionamento.

Server configuration

Le procedure per la configurazione del server consentono di inizializzare il processo di replica sul proprietario delle informazioni da distribuire. Infatti è chi possiede inizialmente le tabelle che decide chi può abbonarsi e chi no.

- `sp_addpublisher` permette di aggiungere un nuovo publisher server all' elenco dei server collegati al sito in cui ci troviamo. E' possibile specificare un parametro opzionale che consente di identificare il sito remoto come publisher server per il sito di distribuzione in cui viene eseguita.
- `sp_addsubscriber` permette l' aggiunta di un subscription server e aggiunge un login remoto "repl_subscriber" che viene mappato in remoto come "sa" cioè come utente "system administrator".
- `sp_dboption` consente di modificare gli attributi del database specificato, infatti per pubblicare o sottoscrivere un database occorre settare degli opportuni flag booleani: "published" e "subscribed".

Sono presenti inoltre le corrispondenti funzioni per eliminare dei server dalla lista di server connessi: `sp_dropsubscriber` e `sp_droppublisher` e funzioni di debug: `sp_helpdistributor`, `sp_helpserver`.

Publication administration

Sono le stored procedure utilizzate dal sistema di replica per l' amministrazione delle pubblicazioni, cioè per la definizione dei dati da pubblicare.

- `sp_addpublication` crea una pubblicazione e ne definisce il tipo. Una pubblicazione può essere definita "restricted" per permettere la pubblicazione solo verso i siti autorizzati.
- `sp_addarticle` crea un articolo (cioè una tabella intera o una partizione) e lo aggiunge alla pubblicazione.
- `sp_articlecolumn` seleziona una colonna della tabella e crea una partizione verticale che può poi essere associata ad un articolo nella pubblicazione.

Le altre funzioni di questo gruppo (oltre alle corrispondenti funzioni "drop" per rimuovere pubblicazioni e articoli e alle funzioni "help" di debug) permettono di modificare gli attributi di un articolo già definito in una pubblicazione, in modo tale da riconfigurare la replica mantenendo definite tutte le pubblicazioni e i server. Infine la `sp_replstatus` che consente di modificare lo stato di una tabella (da replicata a non replicata e viceversa) permettendo di sospendere temporaneamente la pubblicazione di una tabella.

Subscription administration

Queste sono le procedure per la gestione delle sottoscrizioni (subscriptions) degli articoli pubblicati da un publisher.

- `sp_addsubscription` consente di aggiungere una subscription all' articolo e settare le opzioni relative al sincronismo: automatico, manuale o nessuno. Viene eseguita sul publisher per definire quale sito è abbonato alla pubblicazione.
- `sp_subscribe` è la duale della precedente, viene eseguita sul subscriber server per aggiungerer un articolo alla subscription.

Anche in questo caso sono presenti le funzioni "drop" per rimuovere le subscription dai server e le "help" per visualizzare lo stato attuale per il debug.

Replication operations

Sono le stored procedures utilizzate per le operazioni base della replica, la sola realmente importante è:

- `sp_replica` consente di definire una tabella abilitata alla replica, deve essere aggiunta nel creation script della tabella in modo tale che sia accessibile anche dai siti subscribers.

Le altre gestiscono parte delle operazioni per la creazione delle statistiche utilizzate dall' SQL Server Performance Monitor, cioè il programma che consente il profiling delle operazioni sul database.

Replicated Transaction Management

Queste procedure sono definite *extended stored procedure*, sono stored procedure caricate dinamicamente quando devono essere eseguite. Sono memorizzate in file .DLL e la loro esecuzione è gestita e controllata direttamente dal SQL Server. Queste procedure sono di utilizzo meno frequente, pertanto anche se non sono precaricate in memoria le prestazioni globali non ne risentono.

L' unica procedura utilizzata è la:

- `sp_MSkill_job` che permette di eliminare dei job (cioè delle transazioni da replicare) dalla coda dei comandi pronti per la replica, questo permette di recuperare uno stato corretto dopo un errore nella replica.

Questa procedura è importante in quanto il meccanismo di replica di SQL Server è particolarmente debole e sensibile ai malfunzionamenti. Quando la replica di una transazione fallisce (ad esempio perchè la chiave è duplicata o a causa di un errore interno al modulo ODBC del SQL Server) l' amministratore deve riportare manualmente il sistema in uno stato corretto esaminando il contenuto di alcune tabelle di sistema contenenti i job non eseguiti (e quindi quelli che tengono bloccato il sistema) e rimuoverli mediante la procedura.

Replication scheduling

Queste procedure sono di fondamentale importanza in quanto si occupano della definizione dei tempi e dei modi in cui devono essere schedulati i task, in particolare quelli di sincronizzazione e distribuzione della replica.

- `sp_addtask` consente di aggiungere un nuovo task associato ad una pubblicazione allo scheduler. Questo task deve essere di tipo "Sync" e si occupa della sincronizzazione periodica delle repliche.
- `sp_droptask` rimuove un task dallo scheduler.
- `sp_helptask` visualizza tutti i task presenti nello scheduler del sistema.

5.3 Microsoft Distributed Transaction Coordinator

Le applicazioni in un sistema distribuito sono costituite da componenti che risiedono su siti differenti e comunicano unicamente attraverso messaggi sulla rete. I componenti danno contemporaneamente modularità e distribuzione.

Strutturando una applicazione come un insieme di componenti indipendenti si crea il problema della loro gestione. Se un componente fallisce, questo non deve interferire con il funzionamento degli altri, deve esistere un metodo per isolare e limitare la propagazione degli errori. Questo si realizza mediante le transazioni e attraverso i protocolli che abbiamo introdotto nel capitolo 2.8.

La gestione e il coordinamento delle transazioni distribuite in SQL Server è svolta da un modulo denominato *Microsoft Distributed Transaction Coordinator* MS-DTC che riveste entrambi i ruoli di coordinatore (transaction manager) e di partecipante (resource manager), tutti i server del sistema distribuito devono avere il MS-DTC attivo e configurato, in modo tale che possano comunicare durante l' esecuzione delle transazioni distribuite. E' possibile utilizzare il MS-DTC anche per configurazioni client/server, questo permette di utilizzare lo stesso componente per collegare un server appartenente ad un sistema distribuito con dei client locali come rappresentato nella figura 5.5.

Per iniziare una transazione distribuita è possibile utilizzare l' istruzione T-SQL `BEGIN DISTRIBUTED TRANSACTION`.

Chi inizia la transazione con il `BEGIN` è il coordinatore che decide sul commit della transazione, lo chiameremo *commit coordinator*. Il coordinatore comunica con i MS-DTC subordinati (i partecipanti) per portare a termine la transazione (utilizzando il protocollo 2PC) e decidere sul abort o commit globale. In figura 5.6 è rappresentato il protocollo gestito dall' SQL Server.

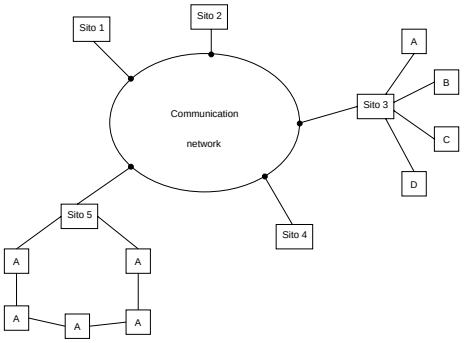


Figura 5.5. Esempio di sistema misto: distribuito e client/server

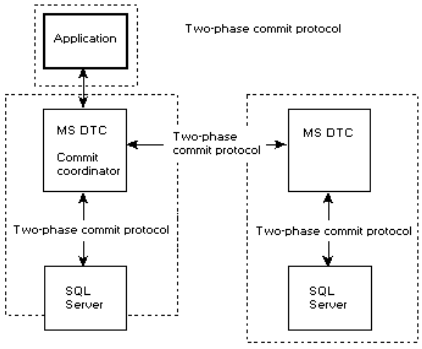


Figura 5.6. Implementazione del two-phase commit protocol nel MS-DTC

Il MS-DTC può essere installato parzialmente sui client, dove è sufficiente avere un substrato che non esegue compiti di coordinamento.

Nel caso di problemi tali da interrompere l' esecuzione della transazione è necessario intervenire manualmente. Consideriamo una situazione composta da quattro server: A (il coordinatore), B, C e D collegati tramite una rete a bus, e supponiamo che sia accaduto un errore sulla rete tra due siti (ad esempio tra B e C) dopo che la fase 1 del 2PC è terminata e il coordinatore ha scritto il pre-commit sul log. La transazione è rimasta in uno stato indeterminato: B ha ricevuto il commit da A, ma C e D non sono raggiungibili quindi la fase 2 non termina e i siti non raggiunti rimangono in uno stato "dubbio".

L' amministratore deve manualmente forzare il server C ad effettuare il commit, essendo la linea di comunicazione tra C e D integra anche D registra il commit.

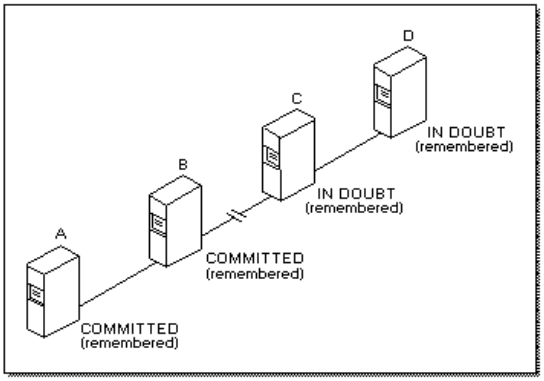


Figura 5.7. Recovery nel MS-DTC

Inizia la seconda fase, D elimina la transazione e manda la conferma dell' esecuzione verso C, quest' ultimo a sua volta vuole comunicare verso B ma trova la linea ancora interrotta. A questo punto tutti i server hanno effettuato il commit, ma la seconda fase è bloccata sul server C. L' amministratore deve forzare B ad eliminare la transazione come effettuata, quindi B invia ad A la conferma e la transazione distribuita termina.

5.3.1 Applicazioni distribuite in T-SQL

Le applicazioni che richiedono modifiche a dati distribuiti sono molto vulnerabili e risentono immediatamente di guasti hardware o software su uno qualunque dei siti coinvolti o nelle infrastrutture che li collegano. Le applicazioni devono essere in grado di rilevare e (per quanto possibile) riparare gli errori provocati da questi malfunzionamenti.

Una applicazione distribuita per SQL Server può essere realizzata in due modi: utilizzando il T-SQL e sfruttando l' interfaccia di SQL Server o in C/C++ tramite la libreria DB-Library oppure i driver ODBC.

Consideriamo solo il caso del T-SQL che ha i seguenti vantaggi:

- Lo sviluppo è semplice in quanto il T-SQL ha una semplice interfaccia per iniziare e terminare una transazione.

- Il costo del commit può essere minore se è l' SQL Server a iniziare la transazione, questo perchè SQL Server e MS-DTC transaction manager sono sulla stessa macchina. Questo riduce il costo del commit in quanto il DTC del client non deve intervenire nel 2PC.

- Il setup è semplice perchè il DTC non deve essere installato sui client.

Il T-SQL può iniziare una transazione distribuita in due modi:

1. `BEGIN DISTRIBUTED TRANSACTION` inizia in modo esplicito la transazione.

2. In modo implicito settando il parametro `REMOTE_PROC_TRANSACTIONS` in fase di installazione dell' SQL Server oppure all' inizio dell' applicazione utente.

Quando il DTC è abilitato per eseguire una transazione distribuita è possibile eseguire RPC su tutti i siti coinvolti, questa particolarità verrà utilizzata nel programma sviluppato.

Vediamo una istruzione `BEGIN DISTRIBUTED TRANSACTION` cosa comporta per il DTC:

- Quando l' applicazione inizia esplicitamente la transazione distribuita SQL Server chiama il metodo `ITransactionDispenser::BeginTransaction` e ottiene un oggetto che rappresenta la transazione. Quindi l' SQL Server inizia un dialogo con il DTC che gli consente di partecipare al 2PC.

- Tutti gli insert, update e delete nei database eseguiti nell' applicazione vengono eseguiti all' interno della transazione DTC.

- Se l' applicazione esegue una procedura remota in un altro SQL Server, il server d' origine invia la transazione del DTC con la richesta di chiamata.

- Quando una transazione è completa l' applicazione esegue un comando `COMMIT TRANSACTION`, l' SQL Server chiama il metodo `ITransactionDispenser::Commit` e il DTC attraverso il 2PC conclude la transazione. Se l' applicazione esegue un `ROLLBACK TRANSACTION` SQL Server esegue il metodo `ITransaction::Abort` che annulla gli effetti della transazione su tutti i server.

Vediamo un esempio di chiamata a procedura remota all' interno di una transazione distribuita.

```
create procedure upd_ord(@cod int,@descr varchar(20),@toServer varchar(12))
as
declare @execstr varchar(200)
/* Inizia la transazione */
begin distributed transaction upd_ord

/* Modifica localmente */
update ordini
   set descr = @descr
 where cod_ord = @cod

/* Prepara la stringa per la chiamata remota */
select @execstr = @toServer + '.database..update_ord'

/* Effettua la chiamata */
exec @execstr @cod,@descr

/* Termina con commit */
commit transaction
```

Sul server remoto definiamo la seguente procedura di update:

```
/* Update di ordini */
create procedure update_ord(@cod int, @descr varchar(20))
as
update ordini
   set descr = @descr
 where cod_ord = @cod
```

Capitolo 6

Implementazione dei modelli di replica proposti

Esaminiamo ora l' implementazione dei modelli di replica proposti, in particolare affrontiamo prima l' analisi del modello dynamic ownership, che è il più completo.

In seguito infatti vedremo come è possibile implementare lo static ownership utilizzando lo stesso software che abbiamo sviluppato per realizzare il dynamic ownership.

6.1 Un database di esempio

Introduciamo ora la situazione che seguiremo nel resto del capitolo per realizzare una applicazione di esempio: consideriamo una rete di tre server chiamati MASTER, SERVER1, SERVER2 e rispettivamente sarà master server il primo e server periferici i restanti due. I nomi assegnati ai server sono i nomi con cui vengono riconosciuti all' interno della rete, nell' esempio abbiamo assegnato nomi particolarmente significativi per riconoscerne immediatamente la funzione svolta. Ipotizziamo che ciascun server sia dislocato in un' area geografica distinta, e sia il server locale di un sistema client/server dipartimentale. Questo significa che possiamo associare ad ogni server un ruolo che è quello svolto dal dipartimento (o ufficio) in cui risiede. Supporremo i seguenti ruoli: SERVER1 con ruolo "Venditore" (abbreviato VEND) e SERVER2 con ruolo "Direzione Commerciale" (abbreviato DC). Al MASTER

non associamo nessun ruolo in modo tale da non complicarne la gestione avendo anche tabelle di proprietà.

Consideriamo un database chiamato "appldb" contenente le tabelle:

- ORDINI(cod_ord,descr)

```
create table ordini (
  cod_ord int not null,
  descr varchar(20) not null

  constraint pk_ord primary key ( cod_ord )
)
```

- ARTICOLI(cod_art,nome)

```
create table articoli (
  cod_art int not null,
  nome varchar(20) not null

  constraint pk_ord primary key ( cod_art )
)
```

Dovremo aggiungere altri campi alle tabelle, quindi le scegliamo semplici (con due soli attributi) per non complicare eccessivamente la trattazione.

Consideriamo infine il seguente diagramma a stati del cambiamento di propietà di un ordine:

Questi dati che abbiamo elencato sono gli stessi che serviranno al programma per generare automaticamente gli script per questa applicazione.

6.2 Inizializzazione della replica in SQL Server

Vediamo ora come inizializzare manualmente la replica utilizzando le stored procedures fornite da SQL Server. Consideriamo un sistema distribuito formato da tre server su cui è installato Windows NT 4.0, appartenenti allo stesso workgroup e con nomi: MASTER, SERVER1, SERVER2. Questi nomi sono quelli definiti al momento dell' installazione del sistema operativo e

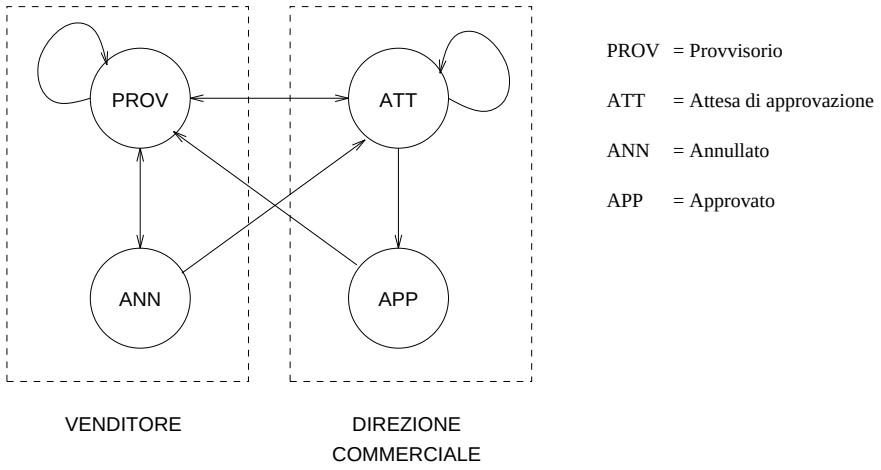


Figura 6.1. Diagramma a stati del cambiamento di propietà di un ordine

con i quali i server sono riconosciuti nella rete, la quale ovviamente deve già essere installata e funzionante così come l' SQL Server.

Supponiamo che MASTER sia il publication server, gli altri siano subscription server. Per ogni operazione si deve specificare quale server la deve eseguire, se il publisher server o i subscription servers.
I passi da eseguire per il setup della replica sono i seguenti:

1. **Sul publication server: definizione del distribution database.**

```
use master
disk init
name='distdata',
physname='c:\mssql\data\distrib.dat',
vdevno=5,
size=15360,
go
disk init
name='distlog',
physname='c:\mssql\data\distlog.dat',
vdevno=6,
size=7680
go

create database distribution on distdata log on distlog
go
```

In questa fase abbiamo creato i device che conterranno i dati e il log sul disco, quindi abbiamo creato un database chiamato "distribution" che utilizza questi device. Il database `master` è il database di sistema di SQL Server, contiene tutte le tabelle e le stored procedure necessarie per l' inizializzazione e l' amministrazione del DDBMS.

2. **Sul publication server: installazione del distribution database nel sistema.**

```
use distribution

run INSTDIST.SQL

xp_regwrite 'HKEY_LOCAL_MACHINE',
   'SOFTWARE/Microsoft/MSSQLServer/Replication',
   'DistributionDB','REG_SZ','distribution'

exec("xp_regwrite 'HKEY_LOCAL_MACHINE',
      'SOFTWARE/Microsoft/MSSQLServer/Replication',
      'WorkingDirectory','REG_SZ',
      '\\'+@servername+'\D$\MSSQL\REPLDATA'")
go
```

Installiamo il database di distribuzione eseguendo lo script INSTDIST.SQL che crea l' elenco delle stored procedures di gestione della replica, le tabelle che contengono i settaggi e altre funzioni di servizio. Con le due istruzioni che seguono installiamo il distribution database nel sistema modificando due registri del sistema operativo relativi all' SQL Server, identificando il database creato per la distribuzione.

3. **Sul publisher server: definizione del server come distribution e publisher server.**

```
sp_serveroption MASTER,'dist','true'
go
sp_addpublisher MASTER,'pub','true'
go
```

Il server MASTER viene definito distribution server e poi publisher server.

4. **Sul publisher server: Definizione dei subscription server.**

```
sp_addsubscriber SERVER1
go
sp_addsubscriber SERVER2
go
```

Queste due istruzioni definiscono i nomi dei server che sono abilitati a ricevere pubblicazioni dal publisher (il server su cui eseguiamo i comandi).

5. **Sui subscriber server: aggiungiamo il publisher server.**

```
sp_addpublisher MASTER
```

6. **Sui subscriber abilitiamo il database a ricevere dati replicati** (cioè a diventare un subscription database).

```
sp_dboption 'appldb','subscribed','true'
go
```

7. **Sul publisher: abilitazione del database a pubblicare i dati** (diventa il publication database).

```
sp_dboption 'appldb','published','true'
go
```

8. **Sul publisher: definizione della pubblicazione e del task di sincronismo e della pubblicazione associata.**

```
use msdb
go
declare @taskid int
sp_addtask 'ordini_sync','Sync','MASTER','sa','appldb',
    1,4,1,8,1,0,0,19970201,19991231,090000,230000,000000,160000,0,
    '',2,1,'',2,2,@newid = @taskid output
use dbname
sp_addpublication ordini_pub,@taskid,@status='active',
    @description='Pubblicazione degli ordini'
go
```

Con la prima istruzione creiamo un task di sincronismo che si occupa di passare allo scheduler il job 'ordini_sync' del server MASTER, al task è associato l' utente "sa" (il system administrator) e deve agire sul database "appldb". I parametri che seguono sono relativi ai parametri temporali dello scheduling e indicano di eseguirlo ogni ora tra le 9 e le 23 dal 1/2/1997 al 31/12/1999, con la prima esecuzione alle ore 16 del giorno di installazione, la stored procedure ritorna l' id del task . Ora possiamo definire la pubblicazione associandovi il task appena creato.

9. **Sul publisher server: creazione degli articoli e definizione della subscription su di essi.**

```
exec sp_addarticle ordini_pub,ordini_art,ordini,
     @creation_script='\\MASTER\C$\MSSQL\REPLDATA\ORDINI.SCH',
     @description='Articolo contenente la tabella ordini intera.'
go
sp_addsubscription ordini_pub,ordini_art,SERVER1
go
```

La prima istruzione definisce l' articolo definito dalla tabella "ordini" completa (senza partizioni), occorre passare il pathname di un creation script che è lo script utilizzato per la creazione della tabella, con l' aggiunta della istruzione `sp_replica ordini,true` che consente la replica della tabella, questo creation script può essere unico su tutta la rete e trovarsi sul server nel pathname specificato.

Quindi aggiungiamo la subscription della pubblicazione al SERVER1.

Seguiti questi passi abbiamo definito completamente la replica di una tabella intera verso un server. Per avere delle partizioni è sufficiente dopo aver definito l' articolo utilizzare la seguente procedura, ad esempio per una partizione orizzontale:

```
exec sp_addarticle ordini_pub,ordini,ordini,
     @creation_script='\\SERVER1\C$\MSSQL\REPLDATA\ORDINI.SCH',
     @description='Articolo contenente la tabella ordini'

exec sp_articlefilter ordini_pub,ordini,'ordini_filter',
     'stato = "STATO1" or stato = "STATO2"'
```

```
exec sp_articleview ordini_pub,ordini,'ordini_view',
     'stato = "STATO1" or stato = "STATO2"'
go
```

Dopo la definizione dell' articolo definiamo un filtro che permette di replicare solo le righe con determinate condizioni, quindi una vista che verrà utilizzata dal sincronismo. Automaticamente questi due oggetti vengono associati all' articolo ed utilizzati nella replica.

6.3 Implementazione del modello Dynamic ownership

Il dynamic ownership presenta la difficoltà di dover consentire il passaggio della proprietà di un dato (una singola riga di una tabella) da un server ad un altro, e di conseguenza il controllo del diritto di accesso alla riga deve tenere in considerazione il cambiamento di stato e le transizioni lecite.

Gli script da generare sono uno per ogni server del sistema distribuito, e sono di due tipi:

Sito Master: le operazioni che devono essere definite sul master server sono le seguenti:

- Inizializzazione della replica e del distribution server.
- Definizione delle stored procedures di redistribuzione delle RPC provenienti dai server.
- Definizione delle pubblicazioni (una per ogni subscriber) contenenti gli articoli richiesti in lettura e agli stati non di propietà di ciascun server.
- Definizione delle tabelle di servizio e pubblicazioni per la loro replica verso tutti i siti remoti.
- Inserimento nelle tabelle di servizio dei valori provenienti dalla topologia della replica, dal grafo e dalla configurazione dei server.

Siti periferici: le operazioni che devono compiere gli script nei server periferici sono le seguenti:

- Inizializzazione della replica come publisher/subscriber.
- Definizione delle pubblicazioni partizionate agli stati di propietà del server.
- Definizione delle RPC per l' inserimento locale di righe di propietà.
- Definizione del trigger sull' insert e l' update.
- Definizione del trigger sul delete.

6.3.1 Definizione della proprietà di una tupla

Per individuare il proprietario di una determinata riga occorre memorizzare lo stato in cui si trova ogni riga. Per fare questo dobbiamo aggiungere a tutte le tabelle dell' applicazione un attributo aggiuntivo: "STATO" che può assumere i valori corrispondenti agli stati dei diagrammi di stato definiti, nel nostro caso è di tipo varchar(10).

Dalla definizione dei server e degli stati possiamo ricavare una tabella di servizio che chiameremo STATO(nome,prop,iniz), la prima colonna contiene i nomi degli stati, è una chiave primaria unica, la seconda il ruolo del proprietario dello stato, la terza un flag booleano che sarà 1 se lo stato è iniziale, 0 altrimenti. Seguendo l' applicazione d' esempio la tabella diventa:

nome	prop	iniz
PROV	VEND	0
ATT	DC	1
APP	DC	0
ANN	VEND	1

Ad ogni ruolo dobbiamo associare il server corrispondente, occorre una altra tabella di servizio che chiameremo FUNZSRV(funz,nomesrv). Il primo campo contiene il nome del ruolo (chiave primaria) e il secondo il nome del server (come definito nella rete). Il master ha ruolo predefinito "MASTER", e solo il ruolo è chiave unica, di conseguenza possiamo avere server che possiedono più ruoli.

funz	nomesrv
MASTER	MASTER
VEND	SERVER1
DC	SERVER2
FUNZ	SERVERn

Infine terza ed ultima tabella di servizio è la TRANS(da,a) che identifica in modo univoco tutte le transizioni di stato, "da" uno stato origine "a" uno stato destinazione.

da	a
PROV	PROV
PROV	ATT
PROV	ANN
ATT	PROV
ATT	ATT
ATT	APP
APP	PROV
ANN	PROV
ANN	ATT

Utilizzando queste tabelle possiamo conoscere tutto sugli stati e le proprietà di tutti i server, ad esempio dato uno stato possiamo sapere se è iniziale o no (e quindi consentire l' insert) eseguendo la seguente query su un server:

```
select s.inizi
  from stato s, funzsrv f
 where s.prop=f.funz
   and s.nome=@stato
   and f.nomesrv=@@servername
```

Osserviamo che non è necessario fornire esplicitamente il nome del server in quanto viene prelevato dalla variabile globale `@@servername` definita automaticamente dal SQL Server con il nome definito del server.

Le tre tabelle di servizio dopo essere state create sul master vengono aggiunte come articoli di una particolare pubblicazione in broadcast verso tutti i server collegati alla rete, il sincronismo di queste repliche deve essere immediato, in modo tale che gli script eseguiti sui server trovano tutte le tabelle e le informazioni richieste per il setup.

Nuova struttura delle tabelle

Per l' implemetazione del passaggio di proprietà di una riga della tabella seguendo il metodo illustrato occorre aggiungere un' altra colonna alle tabelle: l' attributo "validità" (o brevemente "val"). Sarà un intero che avrà valore 0 quando la riga corrispondente si trova in uno stato valido, altrimenti sarà non nulla. E' compito dell' applicazione recuperare solo i dati validi (val=0),

tuttavia sono disponibili anche i dati provvisori (val=1), in modo tale da avere comunque i dati in attesa della conferma dell' inserimento.

Utilizzando il campo "val" è possibile implementare il versioning, cioè la storia di una determinata riga durante la sua esistenza. E' sufficiente incrementare il campo validità ad ogni passaggio di proprietà, in modo tale che la versione valida ha val=0, la versione meno recente ha val=1, e così via la versione più recenta ha il campo val pari al massimo valore del campo per quello stato.

Il campo validità ovviamente deve far parte della chiave primaria della tabella, in modo tale da poter distinguere le varie versioni della riga, l' applicazione dell' utente deve ovviamente essere a conoscenza di questa nuova forma della chiave, anche se tipicamente verrà utilizzata solamente la chiave definita dall' applicazione tenendo fissa la chiave val=0 per recuperare solamente la riga valida.

La nuova struttura delle tabelle dell' esempio quindi diventa:

- ORDINI(cod_ord,descr,stato,val)

```
create table ordini (
  cod_ord int not null,
  descr varchar(20) not null,
  stato varchar(10) not null,
  val int not null

  constraint pk_ord primary key ( cod_ord , val )
)
```

- ARTICOLI(cod_art,nome,stato,val)

```
create table articoli (
  cod_art int not null,
  nome varchar(20) not null,
  stato varchar(10) not null,
  val int not null

  constraint pk_ord primary key ( cod_art , val )
)
```

Il campo stato non deve entrare nella chiave (anche se per la soluzione di vari problemi implementativi sarebbe preferibile) in quanto la gestione del passaggio di proprietà deve essere completamente trasparente all' utente.

6.3.2 Gestione dell' INSERT e dell' UPDATE

Le operazioni di modifica dei dati devono essere gestite in modo tale da proteggere i dati da accessi non autorizzati. I motivi per negare l' accesso sono molteplici, alcune situazioni sono gestite direttamente dal DBMS quindi non verranno prese in considerazione (prima di tutto la gestione delle chiavi primarie), l' unico compito che rimane da svolgere è il controllo della compatibilità della transazione da effettuare con lo stato in cui si trova il dato.

Gestione dell' INSERT

L' insert di una nuova riga nel database può avvenire solo se sono rispettate le seguenti condizioni:

- Lo stato della riga immessa deve essere di proprietà del server in cui avviene l' INSERT.
- Lo stato della riga immessa deve essere definito iniziale.

Per verificare queste condizioni utilizziamo la seguente query all' interno del trigger:

```
/* Preleva i campi della riga aggiunta */
select @cod_ord=cod_ord, @descr=descr, @stato=stato, @val=val
  from inserted

select @in= (select s.iniz
               from stato s, funzsrv f
              where s.prop=f.funz
                and s.nome=@stato
                and f.nomesrv=@@servername )
if @in != 1
begin
  raiserror("Stato di partenza non iniziale.",1,2)
  rollback transaction
end
```

Se lo stato non è iniziale la transazione viene annullata con il rollback, altrimenti termina correttamente e viene eseguito l' insert locale del dato di propietà del sito.

L' INSERT locale verrà in seguito replicato verso il master, il quale a sua volta lo replica verso i server subscribed. Esaminando la tabella di sistema *MSjob_commands* associata al distribution database che contiene tutte le transazioni interessate alla replica si osserva che le transazioni replicate sono gestite localmente in modo tutto simile a quelle immesse direttamente sul sito locale. Questo comporta un problema fondamentale: lo stato della riga che il meccanismo di replica tenta di inserire nella tabella sul sito locale (che la riceve read/only) non è uno stato compatibile con il server in cui deve essere replicata, di conseguenza il trigger sull' insert nel sito destinazione scatta e annulla la transazione locale annullando quindi la replica.

Quello che serve è un meccanismo che consenta di inserire una riga proveniente dalla replica ignorando lo stato, questo significa che dobbiamo distinguere gli insert locali dagli insert replicati e su questi ultimi consentire il commit anche se lo stato non è di proprietà. Il problema nasce nella replica della transazione dal master verso i server, sfruttiamo allora il fatto che tutte le transazioni da replicare devono passare attraverso il master per modificare solo su questo SQL Server i comandi utilizzati dalla replica per inviare le transizioni. Questo è possibile settando opportuni parametri della `sp_addarticle` in modo tale che SQL Server utilizzi una nostra stored procedure per inviare la transazione da replicare e non quelle interne al DDBMS.

Utilizziamo il campo validità per distinguere un inserimento da replica o locale: mediante l' aggiunta di un offset negativo al campo val della transazione da replicare il trigger riuscirà a distinguerla da quella locale.

Sul master server modifichiamo la definizione degli articoli nel modo seguente (esempio con la tabella Ordini):

```
exec sp_addarticle server1_pub,ordini_a,ordini,
     @destination_table = ordini,
     @creation_script='\\MASTER\C$\MSSQL\REPLDATA\ORDINI.SCH',
     @ins_cmd='nuovoins'
```

Il parametro `@ins_cmd` specifica quale comando remoto eseguire nella replica di un insert, quindi sui server dovremo definire la seguente stored procedure di replica:

```
create procedure nuovoins @cod int, @descr varchar(20),
                          @stato varhcar(10), @val int
as
insert ordini values(@cod,@descr,@stato,-(@val+OFFSET_INS))
go
```

Dove `OFFSET_INS` è un valore costante pari ad esempio a 15000 (valore puramente indicativo). Con questo espediente un insert proveniente dalla replica viene trasformato in un insert locale con campo val minore del valore negativo di offset, quindi se l' offset è sufficientemente elevato sarà un valore certamente negativo. Da notare che sono possibili sovrapposizioni solo quando il numero di versioni di una stessa riga supera il valore assegnato all' offset, un valore dell' ordine delle migliaia consente sufficiente sicurezza. Nel trigger ora possiamo distinguere i due casi:

```
select @cod_ord=cod_ord, @descr=descr, @stato=stato, @val=val
  from inserted

if -(@val+OFFSET_INS) > 0 begin
  /* Campo validita minore di zero -> insert da replica */
  insert ordini values(@cod_ord,@descr,@stato,-(@val+OFFSET_INS)
end else begin
  /* insert locale */
  select @in= (select s.iniz
                 from stato s, funzsrv f
                where s.prop=f.funz
                  and s.nome=@stato
                  and f.nomesrv=@@servername )
  if @in != 1
  begin
    raiserror("Stato di partenza non iniziale.",1,2)
    rollback transaction
  end
end
```

Quando il campo val è minore dell' offset si tratta in un insert da replica, lo trasformiamo in un insert con validità positiva (quella determinata dal sito di origine) e lo inseriamo. Questo statement non innesca nuovamente il trigger in quanto è possibile disabilitare i nested triggers nella fase di inizializzazione,

pertanto l' operazione va a termine correttamente (anche se stato non è di proprietà del server corrente). Se il campo val è maggiore o uguale a zero proviene da un insert locale, quindi viene eseguito il controllo sulla proprietà del dato.

Gestione dell' UPDATE

L' update di una riga già esistente deve avvenire rispettando le seguenti condizioni:

- Lo stato iniziale deve essere di proprietà del server che esegue l' update.
- Lo stato finale deve essere raggiungibile dallo stato inizale mediante una transizione valida (definita nel grafo).

Occorre notare che le transizioni di stato vincolano enormemente l' applicazione utente, infatti anche un semplice update di un campo di una riga locale (di proprietà) pur senza modificare lo stato può essere non lecito se nel grafo non è definito un autoanello per quello stato. Lo sviluppatore deve quindi considerare tutti gli stati in cui non solo è possibile un passaggio di proprietà, ma anche quelli in cui è possibile modificare localmente una riga. Questo favorisce la gestione distribuita, in quanto i campi possono essere modificati solo in determinati stati, mentre negli stati di transito devono rimanere immutati (ad esempio un ordine non può essere modificato dopo essere stato confermato dalla direzione commerciale e mentre torna dal venditore per essere eseguito).

La transizione di stato è definita dallo stato della riga da modificare (stato di partenza) e lo stato della riga modificata (stato di arrivo), all' interno del trigger possiamo utilizzare le due tabelle inserted e deleted per ottenere queste informazioni in modo automatico: dalla tabella deleted otteniamo lo stato di partenza, dalla tabella inserted lo stato di arrivo:

```
/* Preleva i dati dalle tabelle */
select @cod_ord=cod_ord, @descr=descr, @stato=stato, @val=val
  from inserted
select @cod_ord1=cod_ord, @descr1=descr, @stato1=stato, @val1=val
  from deleted

select @s_da=@stato1
select @s_a =@stato
```

Ora possiamo verificare che gli stati siano corretti e la transizione lecita:

```
/* Verifica che lo stato di partenza sia di proprieta */
if not exists(select *
                from Stato s, funzsrv f
       where s.prop = f.funz
         and s.nome = @s_da
 and f.nomesrv = @@servername)
begin
  rollback transaction
  raiserror("Stato di partenza non di proprieta'",1,2) with seterror
  return
end

/* Verifica che la transizione sia consentita */
if not exists(select *
from funzsrv
       where da = @s_da
         and  a = @s_a)
begin
  rollback transaction
  raiserror("Transizione non lecita.",1,2) with seterror
  return
end
```

Se una di queste verifiche ha esito positivo il trigger deve terminare la transazione con rollback. Altrimenti possiamo aggiungere a questo punto le istruzioni per gestire l' update.

Il primo compito da eseguire è determinare il server proprietario dello stato di destinazione dell' update, possiamo determinarlo semplicemente attraverso la seguente query:

```
declare @funzdest varchar(50)
declare @proprietario varchar(50)

/* Prelevo il ruolo del server destinazione */
select @funzdest=(select s.prop
                    from Stato s, trans t
   where s.nome = t.a
     and t.da=@s_da and t.a=@s_a)
```

```
/* Prelevo il nome del server destinazione */
select @proprietario=(select nomesrv
                        from funzsrv
       where funz=@funzdest)
```

Ora nella variabile `@proprietario` abbiamo il nome del server che deve ricevere la proprietà della riga, possono verificarsi due casi:

1. Il proprietario è il server che ha effettuato l' update. In questo caso non dobbiamo fare nulla, il trigger può terminare e la transazione terminerà con un commit. Le modifiche eseguire verranno replicate agli altri siti che le hanno richieste.

2. Il proprietario è un server remoto, in questo caso dobbiamo utilizzare una procedura remota per eseguire l' update sul sito destinazione. Prima però la riga modificata localmente deve essere invalidata, cioè dobbiamo modificare il campo val portandolo diverso da 0.

Per eseguire la procedura remota si deve iniziare una transazione remota che inizializza il MS-DTC sui computer coinvolti, questo passo è necessario per consentire ad SQL Server di mantenere il controllo della transazione durante l' esecuzione della procedura.

Utilizzeremo in realtà due livelli di procedure remote: una prima procedura sul master server che riceve i parametri dell' update dal publisher server assieme all' indicazione del nome del nuovo proprietario sul quale si trova una seconda procedura remota (chiamata direttamente dal master su indicazione del publisher) che si occupa dell' update della riga nel nuovo stato.

Vediamo prima il segmento di codice dei trigger dei publisher necessario per la chiamata della prima procedura che si trova sul master:

```
if @proprietario != @@servername
begin
  declare @mastersrv varchar(50)

  /* Preleva il nome fisico del master server */
  select @mastersrv=(select nomesrv
                       from funzsrv
      where funz = 'MASTER')
```

```
  /* Annulla la transazione fatta fino a questo momento */
  rollback transaction

  declare @execstr varchar(200)
  declare @max int

  select @max=(select max(val)
                 from ordini
                where cod_ord = @cod_ord)

  begin distributed transaction

  update ordini
     set val = @max + 1
   where cod_ord = @cod_ord and val = 0

  select @execstr = @mastersrv+'.appdb.dbo.ordini_ins'
  exec @execstr @proprietario,@cod_ord,@descr,@stato,@val

  commit transaction
end
```

Vediamo che la procedura è chiamata specificando completamente il suo nome (nel nostro caso sarà: `MASTER.appdb.dbo.ordini_ins`), tale procedura da definire sul master ha unicamente il compito di indirizzare al server corretto la chiamata alla procedura remota che effetta realmente l' update, quindi dovrà aprire esplicitamente la comunicazione remota ed effettuare la chiamata con i parametri corretti, vediamo come è fatta:

```
create procedure ordini_ins @proprietario varchar(50), @cod_ord int,
                   @descr varchar(20), @stato varchar(10), @val int
as
declare @execstr varchar(200)
begin distributed transaction
select @execstr = @proprietario+'.appdb.dbo.ordini_ins'
exec @execstr @cod_ord,@descr,@stato,@val
commit transaction
go
```

Infine la procedura sul server nuovo proprietario della riga, è molto semplice in quando deve solamente verificare l' eventuale presenza di una riga con la stessa chiave di quella da inserire (in questo caso si cancella quella presente, in modo tale da evitare l' errore di chiavi duplicate), e infine viene eseguito un insert con la nuova riga, nel modo seguente:

```
create procedure ordini_ins @cod_ord @cod_ord int, @descr varchar(20),
                            @stato varchar(10), @val int
as
if exists(select * from ordini where cod_ord=@cod_ord and val=@val)
  delete ordini where cod_ord=@cod_ord and val=@val

insert ordini values (@cod_ord,@descr,@stato,@val)
go
```

Con quest' ultima procedura la transazione che effettua il passaggio di proprietà si conclude. Quando lo scheduler dei task di distribuzione della replica si attiva, questo insert verrà replicato verso il master e quindi verso tutti gli altri siti, compreso il proprietario originale, il quale a sua volta replica l' update che modifica il campo val. Dopo il sincronismo avremo in tutti i siti entrambe le versioni del record: quella valida con lo stato modificato e quella precedente resa inattiva da val > 0.

In figura 6.2 è riassunto il funzionamento del trigger per l' update.

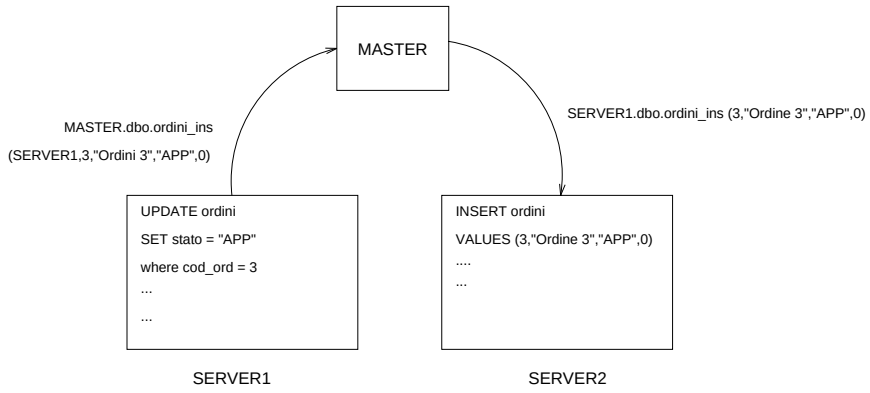


Figura 6.2. Funzionamento del trigger sull' update.

Come nel caso dell' insert anche con l' update possiamo avere dei problemi nella replica delle transazioni remote con stato non compatibile con quello del server, e quindi può scattare il trigger anche durante la replica e pertanto

fallire. Questa situazione si presenta nella replica della riga con validità diversa da zero che raggiunge il sito del nuovo proprietario.

Per risolvere questo problema possiamo utilizzare un metodo simile al precedente utilizzato per l'insert, sommando però un offset diverso. Se sul master definiamo un `sp_addarticle` con un parametro `@upd_cmd` che installa un nuovo comando per la replica dell' update verso i subscribers.

Ad esempio se definiamo l' articolo nel modo seguente:

```
exec sp_addarticle server1_pub,ordini_a,ordini,
     @destination_table = ordini,
     @creation_script='\\MASTER\C$\MSSQL\REPLDATA\ORDINI.SCH',
     @upd_cmd='nuovoupd'
```

Sui subscriber dobbiamo avere la procedura `nuovoupd` che si occupa della gestione particolare dell' update, la replica invia la transazione di update attraverso i valori della riga modificati seguiti dalle chiavi primarie (precedenti alla modifica). Questi rappresentano i parametri della nostra procedura che viene definita nel seguente modo:

```
create procedure nuovoins @cod int, @descr varchar(20), @stato varhcar(10),
                          @val int, @pkcod int, @pkval
as
update ordini
   set cod_ord = @cod, descr=@descr, stato=@stato, val=-(@val+OFFSET_UPD)
 where cod_ord = @pkcod and val = @pkval
go
```

Dove `OFFSET_UPD` è un valore costante pari ad esempio a 10000 (valore fittizio ma deve essere diverso dall' offset utilizzato nell' insert). Con questo espediente siamo in grado di effettuare update locali di stati non di proprietà del server.

Naturalmente all' interno del trigger dobbiamo distinguere l' update proveniente dalla replica (che non deve subire restrizioni) dall' update eseguito dall' applicazione utente (che deve essere controllato). Per fare questo procediamo nel modo seguito per l' insert rilevando quando il campo val è negativo e di valore inferiore all' offset.

6.3.3 Gestione del DELETE

Il delete ha caratteristiche diverse da quelle delle altre operazioni, infatti ha solo una restrizione sulla sua esecuzione: un record può essere cancellato solo dai suoi proprietari, quindi l' operazione avviene tutta localmente e limitata ai record di proprietà. Quando la replica provvede ad estendere l' eliminazione dei record in tutte le tabelle nei subscribers i record saranno completamente cancellati da tutto il database distribuito.

L' unico controllo da effettuare è quindi la verifica della proprietà o meno di un determinato record da cancellare da parte del server che vuole eseguire l' operazione. Questo viene fatto attraverso la seguente query:

```
if not exists(select *
                from Stato s, funzsrv f
       where s.prop = f.funz
         and s.nome = @s_da
 and f.nomesrv = @@servername)
begin
  rollback transaction
  raiserror("Stato di partenza non di proprieta'",1,2) with seterror
  return
end
```

Il trigger dopo aver eseguito questa verifica esegue un rollback se lo stato non è di proprietà, altrimenti termina e la transazione si conclude con un commit che elimina la riga dalla tabella.

Il trigger sul delete non ha altri compiti da svolgere.

Il problema del delete proveniente da replica però non è banale da risolvere. Il trigger viene innescato all' invocazione di un delete, e fallisce se proviene dalla replica, quindi con uno stato non di proprietà, una possibile soluzione è basata sull' utilizzo dell' offset come nei casi precedenti.

Nel delete questo metodo non può funzionare, vediamone il motivo. Supponiamo di eseguire dal master una chiamata ad una procedura sul sito periferico che cancella un dato localmente, installiamo la chiamata alla procedura nell' articolo della tabella.

La procedura chiamata dal master sarà come la seguente:

```
create procedure nuovodel @pkcod int, @pkval int
as
delete ordini where cod_ord = @pkcod and val=-(@pkval+OFFSET_DEL)
```

Questo però non funziona, in quanto non esiste nel database destinazione una riga con campo validità pari al valore `@pkval-OFFSET_DEL` pertanto il delete fallisce prima di richiamare il trigger associato.

La soluzione adottata è quella di trasformare sul master il delete in una chiamata a procedura remota che esegue un insert nella seguente forma:

```
insert ordini values(@pkcod,"","",-(@pkval+OFFSET_DEL))
```

Il trigger sull' insert ora scatta, la tabella inserted contiene la riga inserita, quindi con il campo validità inferiore all' offset (scelto in modo tale da evitare conflitti con quello dell' insert e dell' update), ora il trigger riesce a distinguere il delete remoto da quello locale e l' operazione delete dalle altre, questo controllo viene svolto nella parte iniziale del trigger nel seguente modo:

```
declare @upd int

if exists(select * from deleted)
  select @upd = 1
else
  select @upd = 0

if @val < 0
  if -(@val+OFFSET_DEL) > 0
  begin
    select @val = -(@val+OFFSET_DEL)
    /* Gestione del DELETE proveniente da replica */
    delete ordini where cod_ord=@cod_ord and val=@val
  end else
    select @val = -(@val+OFFSET_UPD)
    if @upd = 0
    begin
      /* Gestione dell' INSERT proveniente da replica */
    end else
      /* Gestione dell' UPDATE proveniente da replica */

/* Seguito del trigger */
```

Osserviamo che l' update e il delete hanno lo stesso offset, la differenza è data dalla presenza o meno della tabella deleted. Il requisito per il funzionamento della scelta dell' operazione è che `OFFSET_DEL` sia superiore in valore

assoluto a `OFFSET_UPD`, infatti è verificato per primo. I valori suggeriti sono i seguenti:

```
OFFSET_DEL = 15000
OFFSET_UPD = 10000
```

Questa scelta è assolutamente arbitraria ma devono essere numeri tali da non sovrapporsi con i valori raggiunti dal campo validità. Ben difficilmente si potranno avere versioni di uno stesso record superiori a qualche decina, pertanto i valori scelti sono ampiamente sufficienti, tuttavia il valore deve essere di tipo integer, che nell' SQL Server è di 4 byte, pertanto sono possibili valori di offset anche molto elevati (fino a -2^{31}).

6.3.4 Gli script di configurazione

I triggers, le stored procedures e in generale i diversi meccanismi introdotti per l' implementazione del dynamic ownership devono essere raccolti in un unico script in modo tale da facilitarne l' installazione sui server. Dovranno essere generati uno script per ogni server periferico, uno script per il master server e una serie di files di supporto (i creation script) che devono essere accessibili da tutti i server, quindi possiamo duplicarli manualmente su tutti i server o più semplicemente lasciarli sul master e accedervi attraverso la rete indicando al momento della loro generazione anche il percorso di rete.

La struttura generale degli script generati con l' applicazione realizzata è la seguente:

1. inizalizzazione della replica;
2. creazione delle stored procedure remote;
3. definizione delle pubblicazioni;
4. creazione del trigger.

6.4 Implementazione del modello Static ownership

Il modello statico prevede che un solo utente sia proprietario di una tabella per tutta la durata dell' applicazione. Di conseguenza è sufficiente implementare la replica nel modo standard descritto (paragrafo 6.2), una tabella

viene pubblicata (interamente o partizionata) dal sito proprietario verso tutti i subscribers che la utilizzano in modo read/only.

Dopo aver definito la replica occorre garantire che solo il proprietario possa accedervi in scrittura, lo faremo attraverso i triggers.

Questo modello lo abbiamo gestito come caso particolare del dynamic ownership, infatti se consideriamo i seguenti requisiti:

- Ogni tabella ha le righe di un solo stato .
- Ogni stato appartiene a un solo server.
- Su ogni stato definiamo una unica transizione: l' autoanello.

Realizzando uno script dynamic ownership con questi requisiti si realizza il modello static ownership. Ovviamente vi saranno parti degli script superflue che con una implementazione specifica potevano essere eliminate, ma durante l' esecuzione dello script le parti non eseguite non verranno installate in memoria.

Il vantaggio di questa scelta consiste nella possibilità di avere un database gestito in parte con il dynamic ownership e in parte con lo static ownership. Infatti se realizziamo un grafo con alcuni stati isolati dagli altri e con un autoanello per ciascuno di questi, ed associando a ciascuno di questi stati un server distinto è possibile avere alcuni server che utilizzano tabelle scambiandosi la proprietà (dynamic ownership), e altri che utilizzano tabelle mantenendone il possesso per tutta la vita dell' applicazione (static ownership), consentendo la gestione di applicazioni miste.

Di conseguenza nel modello static ownership le uniche transazioni ammissibili sono gli insert e i delete nello stato proprietario, e gli update che non modificano il campo stato.

I trigger quindi devono solo controllare l' appartenenza dello stato della transazione all' insieme degli stati definiti per il server:

```
if not exists(select *
                from Stato s, funzsrv f
       where s.prop = f.funz
         and s.nome = @s_da
 and f.nomesrv = @@servername)
begin
  rollback transaction
```

```
  raiserror("Stato di partenza non di proprieta'",1,2) with seterror
  return
end
```

Non devono compiere nessuna altra operazione. In particolare se il modello è static ownership "puro" la tabella delle transizioni di stato è composta da righe con campi `da` e `a` identici (tutti autoanelli).

Lo stesso trigger utilizzato per il dynamic ownership utilizzando una tabella delle transizioni come quella descritta, effettua il controllo dell' ammissibilità dell' operazione, controllando indirettamente anche la proprietà dello stato. La transazione verrà terminata se l' autoanello è su uno stato di proprietà del server, verrà effettuato un `ROLLBACK` in caso contrario.

6.5 C.A.S.E. per la generazione degli script

Per facilitare la realizzazione di tutto il processo di installazione e di gestione della replica è stato realizzato un tool C.A.S.E. che con pochi dati immessi dall' amministratore del database genera in modo automatico tutti i file necessari per la configurazione.

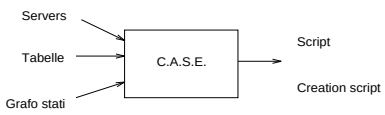


Figura 6.3. Funzionamento del C.A.S.E.

Il programma è stato realizzato in Visual C++ 4.0 per Windows NT. E' composto da due moduli:

1. Implementazione delle classi di generazione degli script
2. Implementazione dell' interfaccia utente mediante la libreria di classi: MFC (*Microsoft Foundation Class*).

Complessivamente composto da circa 6500 linee e da 22 classi (equamente divise tra i due moduli), esaminiamo solo la prima parte, quella che interessa direttamente la generazione automatica degli script.

La classe principale è la `Server` che contiene la descrizione comune di tutti i server della rete.

Questa classe viene utilizzata solamente come contenitore per tutti i dati relativi all' inizializzazione dei device per i dati e per il log da utilizzare per la creazione del database di distribuzione e della directory da utilizzare per i dati replicati. Questa classe è l' unica che si occupa della gestione del file in cui scriviamo lo script relativo al server, questa scelta consente di semplificare le altre classi che non dovranno preoccuparsi di gestire i file in modo corretto poiché è già stata svolta tutta la fase di controllo. Infine la classe contiene il nome del server (nome fisico della macchina nella rete), che viene ampiamente utilizzato negli script.

Descriviamo brevemente i metodi implementati: oltre al costruttore `Server::Server` che inizializza il nome del server e dello script e al distruttore `Server:: Server` che dealloca la memoria, sono stati realizzati i metodi per l' inizializzazione dei device `Server::SetDiskInit` e `Server::SetRepDB_Workdir`.

Infine il metodo più importante di questa classe è `Server::GenScript` che si occupa dell' inizializzazione del file su disco e genera la prima parte dello script relativa all' inizializzazione dei device e all' installazione del distribution database.

```
class Server
{
  private:
    DISKDEV *data,*log;      // Device per il distr database
    char *distrdb;           // nome distr database
    char *WorkingDirectory;  // Working directory sul server @@servername

  protected:
    FILE *f;                 // Stream dello scriptfile
    char *nomeFile;          // Nome del file script di questo server

    char *nomeSrv;           // Nome server
    BOOL distServer;         // TRUE se e' un publisher

    Server(char *nomeSrv,char *nFile);
    ~Server();
    void GenScript(void);

  public:
    char *GetName(void);
```

```
    void SetDiskInit(int tipo,char *nome,char *path,int dev,UINT size);
    void SetRepDB_WorkDir(char *nomeDistDB,char *workingDir);
};
```

Da questa classe deriviamo le due classi per i due tipi di server che dobbiamo gestire: `PushServer` e `MasterServer`.

Iniziamo a descrivere la classe `PushServer` che descrive il server periferico, vediamo l' interfaccia:

```
class PushServer : public Server
{
  private:
    char *nomeMasterDb;   // Nome del master db
    char *ruolo;          // nome del ruolo del server
    Art *art[MAXART];     // Articoli per le tabelle accedute
    int n_art;            // # tabelle accedute = # articoli (se RW)
    struct _st_ {
      char *nome;         // Nome stati associati al ruolo del server
      BOOL iniz;          // flag stato iniziale
    } stati[MAXTAB];   // MAXTAB = numero massimo tab nel db
    int n_stati;          // # stati
    char *nomeTabTrans,   // Nome tabella trans(da,a)
         *nomeTabStato,   // Nome tabella stato(nome,prop,iniz)
 *nomeTabFS;      // Nome tabella funzsrv(funz,nomesrv)
    TASK task;            // Task per le pubbl da e per il server

    // Costruisce la pubblicazione generale
    void BuildPub(FILE *f,char *nomePub,char *pubSrv,
                  char *desc,BOOL restricted=FALSE);
  public:
    // Costruttore
    PushServer(char *nomeSrv,char *script,char *ruolo,
               char *nDB,TASK tsk=defaultTask);
    ~PushServer();
    // Aggiunge una tabella cui accede il pushsrv
    void AddTab(Tab &t,PROPTAB prop);
    // Aggiunge uno stato a quelli del pushsrv
    void AddStati(char *st,BOOL iniz);
    // Setta le tabelle globali
    void SetGlobDB(char *nTTrans,char *nTStato,char *nTFS);
```

```
    // Costruisce le pubblicazioni
    void GenScript(char *nomeMaster);
    // Costruisce l' art del master
    void BuildMasterArt(FILE *f,char *nomeMaster);

    // Ritorna il nome della tabella trans
    inline char* GetTabTrans(void) const { return nomeTabTrans; }
    // Ritorna il nome della tabella stato
    inline char* GetTabStato(void) const { return nomeTabStato; }
    // Ritorna il nome della tabella funzsrv
    inline char* GetTabFS(void) const { return nomeTabFS; }

    // **** Le usa solo il master
    // Dati sulle tabelle utilizzate dal push server
    //
    // Ritorna # tabelle del pushsrv
    inline int GetNumArt(void) const { return n_art; }
     // Ritorna articoli tabelle del pushsrv
    inline Art *GetSrvArt(int i) const { return art[i]; }
    // **** Ritornano dati sugli stati del push server
    // Ritorna il nome di uno stato
    inline char *GetStato(int i) const { return stati[i].nome; }
    // Ritorna se stato iniziale
    inline BOOL GetStIniz(int i) const { return stati[i].iniz; }
    // Ritorna # stati del push
    inline int GetNumStati(void) const { return n_stati; }
    // Ritorna il ruolo del push
    inline char *GetRuolo(void) const { return ruolo; }
};
```

In questa classe sono definiti i metodi per il setup di una pubblicazione di tipo "PUSH" cioè dal server periferico al master server. Occorre a tal fine definire tutte le tabelle utilizzate nel server mediante un metodo `PushServer::AddTab` che consente di aggiungere una classe `Tab` al server specificando la modalità di accesso : read/only o read/write. Non appena si aggiunge una tabella inizia automaticamente la definizione dell' articolo per la pubblicazione (che è una sola, in quanto il master è l' unico subscriber dei Push Server). Sono implementati inoltre i metodi per settare gli stati appartenenti al server e i nomi del database e delle tabelle di servizio. I metodi per la preparazione

delle pubblicazioni e degli articoli per la successiva generazione della porzione di script per la gestione del push server sono `PushServer::BuildPub` e `PushServer::BuildMasterArt`. Quest' ultima genera gli articoli del master, vedremo in seguito infatti il modo particolare di gestire le pubblicazioni nel programma. Infine la classe `PushServer::GenScript` si occupa di scrivere le istruzioni T-SQL per il server.

La terza classe principale è `MasterServer` che definisce il master server. Ovviamente al master dobbiamo associare tutti i push server che sono fisicamente collegati ad esso, questo avviene mediante un metodo `MasterServer::AddSubServer` che permette alla classe del master di accedere a quelle dei subscription server. Il consueto metodo `MasterServer::GenScript` si occupa della realizzazione dello script.

```
class MasterServer : public Server
{
  private:
    dB *db;               // database applicazione costruito in base ad
    char *nomeDB;         // Nome database dell' applicazione
    PushServer *subSrv[MAXSERVER]; // Server periferici
    int n_subSrv;         // # server periferici
    RPC *rpc[MAXTAB*3];   // RPC speciali per il master(redistribuzione)
    int n_rpc;            // # rpc

    char *nomeTabTrans;   // Nome tabella transizioni
    struct _tr_str_ {     // tabella transizioni
      char *da,*a;
    } *tr[100];
    int n_tr;             // # righe trans
    char *nomeTabFS;      // Nome  tabella funz. server
    struct _fs_str_ {     // Tabella FUNZSRV
      char *funz,*nomeSrv;// Ruolo,nomeserver
    } *fs[100];
    int n_fs;             // Numero righe funzsrv

    char *nomeTabStato;   // Nome tabella stato
    struct _st_str_ {
      char *nome,*prop;   // tabella STATO
      BOOL iniz;          // stato iniziale
    } *st[100];
```

```
    int n_st;              // # righe stato

    // Aggiunge uno stato
    void AddStato(char *nome,char *prop,BOOL iniz);
    // Aggiunge una riga funzsrv
    void AddFunzSrv(char *ruolo,char *nomeSrv);
    // Costruisce le tabelle di servizio
    void BuildTabServiz(void);
    // Scrive creation script per le tab serv.
    void BuildCrScrServiz(char *nometab,char* nomefile,char *s);
  public:
    MasterServer(char *nome,char *path,char *nomedb);
    ~MasterServer();
    // Aggiunge un server periferico
    void AddSubServer(PushServer *sub);
    // Aggiunge una riga trans
    void AddTrans(char *da,char *a);
    // Crea lo script
    void GenScript(void);
};
```

Come si può notare la classe del master è più semplice di quella scritta per il push server, questo a causa della scelta di far eseguire al push server anche la parte di definizione delle pubblicazioni relativa al master ma indirizzata a quel server.

Il metodo `MasterServer::BuildTabServiz` si occupa della generazione delle tabelle di servizio utilizzate per la memorizzazione di stati, server e transizioni del grafo. Queste tabelle sono create mediante un comando `create procedure` e quindi vengono generati gli INSERT con i valori corretti. In seguito vengono generati i creation script (mediante il metodo `MasterServer::BuildCrScrServiz`) e infine la pubblicazione broadcast delle tre tabelle.

Le altre classi sono relative alla implementazione di tabelle (classe `Tab`), remote procedures (classe `RPC`) e triggers (classe `Trigger`) classi utilizzate internamente per la definizione di dati (ad esempio la `_el_stati`). Non le descriveremo esplicitamente nel dettaglio ma spiegheremo il loro funzionamento durante la definizione di una applicazione.

Come si può osservare l' interfaccia delle classi è stata mantenuta estremamente semplice e con un numero estremamente contenuto di metodi pubblici,

sono necessari pochi parametri da parte dell' utente (il programmatore che le utilizza), questo ovviamente è a scapito della complessità dell' implementazione delle classi. Infatti sono state rese completamente automatiche tutte le operazioni in cui è possibile ricavare i dati necessari da altri già in possesso del programma.

Esaminiamo ora il funzionamento del programma mostrando la logica utilizzata per realizzare uno script.

Mostriamo ora i passi da compiere per la definizione di una applicazione T-SQL per l' implementazione del modello dynamic ownership mediante i trigger e le stored procedure analizzate nel paragrafo precedente. Ovviamente l' utente finale non deve essere a conoscenza di queste successioni di operazioni da eseguire, questo avviene grazie all' interfaccia utente particolarmente intuitiva realizzata in Windows NT.

I passi in sequenza per la definizione di una applicazione in T-SQL completa sono i seguenti:

1. Definizione di due strutture dati contenenti i parametri per la creazione dei due device (dati e log), opzionalmente è possibile definirne una sola da condividere tra i due device. Questo passo è richiesto solo se non è mai stata installata la replica sul server in cui sarà eseguito lo script.

2. Definizione delle classi contenenti le tabelle del database. Per ogni campo occorre specificare il nome, il tipo di dato e il tipo del campo (primary key, stato o validità). E' necessaria una classe per ogni tabella, nel costruttore si specifica oltre al nome della tabella anche il pathname del creation script e una breve descrizione della tabella.

3. Definizione del master, con lo scopo di inizializzare tutte le variabili utilizzate internamente dalla classe, sarà necessario specificare il nome fisico di rete del master server oltre al pathname dello script da generare e il nome del database utilizzato.

4. Definizione del grafo delle transizioni di stato.

5. E' possibile ora iniziare la definizione dei push server mediante la sequenza di operazioni:

   - Eventuale definizione di una struttura dati contenente i parametri per la creazione del task di sincronismo.

- Mediante il costruttore si definisce il push server con il nome di rete del server, il pathname dello script da generare, il ruolo svolto dal server (ad esempio "Venditore") e il nome del database e opzionalmente il task di sincronismo da usare per la pubblicazione.
- Attraverso il metodo `PushServer::AddStati` si aggiungono tutti gli stati di proprietà del server.
- Attraverso il metodo `PushServer::AddTab` si aggiungono tutte le tabelle utilizzate dal server specificando se read/only o read/write mediante un flag.

6. Aggiungere il push server al master server utilizzando il metodo `MasterServer::AddSubServer`.

7. Ripetere i due passi precedenti per tutti i push server da associare al master.

8. Generare tutti gli script mediante il metodo `MasterServer::GenScript`.

Quando si aggiungono le tabelle ai push server inizia la costruzione vera e propria della pubblicazione, attraverso il flag che indica la modalità di accesso (read/only o read/write) si decide la proprietà della tabella: se read/write allora il server è proprietario di tutte le righe che hanno uno stato appartenente all' insieme definito per il server, quindi dovrà utilizzare la `PushServer::BuildPub` per costruire la pubblicazione ristretta agli stati di proprietà. Parallelamente dovrà essere invocata la `PushServer::BuildMasterArt` che prepara l' articolo per la pubblicazione che dal master invierà al push server la partizione della tabella con gli stati non di proprietà.

Se la tabella è read/only non è necessario costruire la pubblicazione sul push server, ma solo sul master server, la `PushServer::BuildMasterArt` allora definirà sul master un articolo contenente la tabella intera. E' interessante notare che è il push server a decidere le pubblicazioni del master che lo riguardano, attraverso questa soluzione è possibile realizzare un controllo completo su tutti gli articoli composti da tabelle intere e soprattutto partizionate, nelle quali occorre prestare molta attenzione nelle clausole di restrizione.

Tutte le immissioni delle istruzioni T-SQL negli script avvengono in due fasi: la prima di acquisizione dei dati necessari, elaborati e memorizzati dalle classi opportune. La seconda inizia quando si chiede al master server la generazione degli script, in questa fase vengono eseguiti a cascata tutti i

metodi delle classi istanziate relativi alla generazione della loro porzione di codice.

6.5.1 Costruzione di un articolo

Vediamo nel dettaglio la realizzazione di un articolo su entrambi i tipi di server.

La classe `Art` contiene tutte le informazioni relative all' articolo, che sono rappresentate da:

- La tabella cui è riferito.
- Il trigger da installare sul push server.
- Le RPC da installare sul push server per le operazioni locali eseguite da remoto.
- Le RPC da installare sul master che reindirizzano l' operazione locale verso il push server giusto.

La prima fase della preparazione dello script per l' articolo avviene quando aggiungiamo una tabella in un push server: viene istanziata una classe per l' articolo e associata al server. Il costruttore inizializza l' articolo e definisce le RPC e i trigger, le rispettive classi costruiscono tutti i dati necessari basandosi sulla tabella e sugli stati del server.

Tutta la generazione di codice relativo ad un articolo è svolta in `Art::GenArtScript` in cui avviene la scrittura su file del comando T-SQL sp_addarticle con tutti i parametri richiesti, in particolare se l' articolo riguarda un push server vengono scritte anche le procedure per la partizione della tabella agli stati del server.

Se il server non è il master vengono anche scritte le istruzioni per le RPC, per le stored procedures che eseguono le operazioni da remoto sulle tabelle e i trigger che controllano la modifica dei dati, utilizzando i metodi "GenScript" delle rispettive classi.

Le clausole di restrizione da utilizzare per la partizione orizzontale sono generate automaticamente in base agli stati appartenenti al server, e corrispondono alle condizioni "WHERE" del SELECT che realizza la procedura "filtro" delle righe da replicare.

Sul master deve essere gestito un articolo "complementare" a quello definito sul corrispondente server, la restrizione degli stati deve consentire la replica solo agli stati non di proprietà del subscription, anche questo articolo viene creato nel push server, poi viene generato su ordine del master e inserito nel file di script.

I task di sincronizzazione

Nella replica rivestono un ruolo fondamentale i meccanismi di sincronizzazione delle pubblicazioni, nella costruzione automatica dei task abbiamo stabilito due principi:

- Tutte le pubblicazioni push e pull di tabelle partizionate su un server periferico hanno il task di sincronismo identico.
- Le tabelle di servizio: stato, transizioni e funzioni hanno task di sincronismo che parte immediatamente dopo la sua definizione (supponendo la rete connessa).

Le tre tabelle di servizio create dal master devono essere replicate immediatamene, in quanto devono essere già disponibili sui server quando verranno eseguiti gli script, sono state gestite quindi utilizzando una pubblicazione apposita che viene inviata verso tutti i server con sincronismo immediato.

Se le pubblicazioni di uno stesso server hanno lo stesso sincronismo si semplifica la gestione della rete, infatti non è necessario che il server sia collegato continuamente, ma è sufficiente la connessione anche solo per un intervallo di tempo durante il quale avviene lo scambio delle transazioni sia dal master che dal server.

Questo ovviamente semplifica anche il software di creazione della pubblicazione, infatti lo stesso task immesso dall' utente per la pubblicazione push verso il master viene utilizzato anche su quest' ultimo per la pubblicazione pull verso il server.

Gestione del grafo

Il grafo delle transizioni di stato è memorizzato utilizzando una matrice di adiacenza realizzata nel modo seguente:

$$a_{ij} = \begin{cases} 1 & \text{se}(v_i,v_j) \in A \\ 0 & \text{altrimenti} \end{cases} \tag{6.1}$$

Dove la transizione (v_i,v_j) è diretta dallo stato v_i allo stato v_j, e A è l' insieme delle transizioni. In un vettore memorizziamo i nomi degli stati nello stesso ordine utilizzato per indicizzare la matrice.

Quindi viene passata a `MasterServer::AddTrans` che la memorizza in una lista di puntatori ad una struttura che contiene i nomi dello stato di partenza e di quello d' arrivo della transizione. Essendo strettamente legato alla sua visualizzazione, il grafo è implementato nella classe `CWGenView` che si occupa della gestione delle risorse visive dell' applicazione (cioè dialog box, MDI, menu, ...), in questo modulo raccogliamo i dati: nome dello stato, server proprietario e transizione. Quando istanziamo una classe `PushServer` passiamo gli stati di proprietà attraverso l' apposita funzione, questo serve unicamente per realizzare il trigger. Le transizioni le passiamo al master (utilizzando il metodo `MasterServer::AddTab`), il quale produrrà la tabella transizioni utilizzando una successione di INSERT dopo aver creato le tabelle di servizio nel metodo `MasterServer::BuildCrScr`. In questo modo le righe aggiunte vengono replicate.

Conclusioni

Questa tesi ha dimostrato la realizzabilità pratica dei modelli di replica static ownership e dynamic ownership. In particolare la possibilità di implementare quest'ultimo in un ambiente DDBMS commerciale che utilizza metodi di replica standard.

Il progetto presentato è un prototipo, di conseguenza l' ottimizzazione delle prestazioni e l' interattività con l' utente sono ad un livello piuttosto primitivo. Tuttavia, dove possibile, sono state eseguite scelte anche da questo punto di vista, in quando l' applicazione finale non dovrà essere appesantita troppo dall' overhead imposto dalla gestione della replica.

Sviluppi futuri

Per completare le applicazioni realizzate è necessario aumentare l' integrazione con l' SQL Server e con l' applicazione dell' utente, con riferimento in particolare alla fase iniziale di acquisizione delle informazioni per la generazione del codice.

Attualmente tali attività devono essere svolte manualmente anche se il database dell' applicazione è preesistente, di conseguenza è necessaria l' introduzione delle stesse informazioni due volte: una prima volta nella fase di costruzione del database dell' applicazione e la seconda per la generazione degli script. E' preferibile (solo nel caso di database esistente) l' analisi della sua struttura per l' acquisizione diretta delle tabelle e dei campi.

La proprietà delle tabelle per ora è descritta manualmente, ma anche questa operazione può essere automatizzata eseguendo un analisi dell' applicazione dell' utente: rilevando le tabelle utilizzate dalle query e identificando

il tipo di operazione compiuta.

Appendice A

Esempio di script completo

Riportiamo ora un esempio di script generato dall' applicazione realizzata, abbiamo utilizzato i dati dell' esempio introdotto nel sesto capitolo (vedi 6.1). Il modello utilizzato è il dynamic ownership, non riportiamo le tabelle generate che sono già state descritte nell' esempio del sesto capitolo (nel paragrafo 6.3.1).

Gli script generati sono tre:

1. Server MASTER: `master.sql`
2. Server SERVER1: `server1.sql`
3. Server SERVER2: `server2.sql`

Per eseguire questi file sono possibili due alternative:

1. Esecuzione da linea di comando: si utilizza il programma di SQL Server `ISQL` nel seguente modo:

   ```
   net start MSSQLServer

   ISQL /Usa /P < nomefile.sql
   ```

2. Esecuzione da ISQL/w: dopo aver avviato il server MSSQLServer all' interno dell' utility ISQL/w si carica lo script e lo si esegue.

Iniziamo riportando il file contenente lo script relativo al server MASTER.

A.1 Script relativo al master

```
/* Init PULL: Master Server: MASTER -> server periferici */
/* Publisher : MASTER */
/* Subscribers: SERVER1,SERVER2 */

/* Sun Mar 09 20:18:22 1997 */

use master
go
disk init
  name='distdata',
  physname='C:\Mssql\Data\distrib.dat',
  vdevno=5,
  size=15000
go
use master
go
disk init
  name='distlog',
  physname='C:\Mssql\Data\distlog.dat',
  vdevno=6,
  size=7000
go
create database distribution on distdata log on distlog
GO

use distribution
go

\MSSQL\INSTALL\INSTDIST.SQL

use master
exec sp_serveroption MASTER,dist,true
go
use master
exec sp_serveroption MASTER,dpub,true
go
use master
```

```
exec sp_addpublisher SERVER1,dist
go
use master
exec sp_addsubscriber SERVER1
go
use master
go
exec sp_addserver SERVER1
go
exec sp_addremotelogin SERVER1,'sa'
go
use master
exec sp_addpublisher SERVER2,dist
go
use master
exec sp_addsubscriber SERVER2
go
use master
go
exec sp_addserver SERVER2
go
exec sp_addremotelogin SERVER2,'sa'
go
use master
go
exec sp_dboption appldb,'published',true
go
use master
go
exec sp_dboption appldb,'subscribed',true
go
use appldb
go

/* RPC redistribuite per tutte le tabelle utilizzate dai push server */

/****** Object:  Stored Procedure ******/
if exists (select * from sysobjects
            where id = object_id('ordini_ins')
```

```
              and sysstat & 0xf = 4)
  drop procedure ordini_ins
go
create procedure ordini_ins @proprietario varchar(20),@cod_ord int,
                            @descr varchar(20),@stato varchar(10),@val int
as
declare @execstr varchar(200)
begin distributed transaction
select @execstr = @proprietario+'.appldb..ordini_ins'
commit transaction
exec @execstr @cod_ord,@descr,@stato,@val
go

/****** Object:  Stored Procedure ******/
if exists (select * from sysobjects
            where id = object_id('articoli_ins')
              and sysstat & 0xf = 4)
  drop procedure articoli_ins
go
create procedure articoli_ins @proprietario varchar(20),@cod_art int,
                              @nome varchar(20),@stato varchar(10),@val int
as
declare @execstr varchar(200)
begin distributed transaction
select @execstr = @proprietario+'.appldb..articoli_ins'
commit transaction
exec @execstr @cod_art,@nome,@stato,@val
go

/* Inizio definizione pubblicazione server1_pub */

use appldb
go
if exists(select * from syspublications where name='server1_pub')
  exec sp_droppublication server1_pub
go

/****** Object:  Sync task ******/
use msdb
```

```
go
if exists(select * from systasks where name='server1_pub_Sync')
  exec sp_droptask server1_pub_Sync
go
declare @taskid int
exec sp_addtask 'server1_pub_Sync','Sync','MASTER','sa','appldb',
  1,4,1,8,1,0,0,19970309,19991231,000000,235959,000000,000000,0,
  '',0,1,'',2,2,'',@newid = @taskid output
use appldb
exec sp_addpublication server1_pub,@taskid,
  @status=active,
  @description='Pubblicazione server1_pub dal master verso SERVER1',
  @restricted='true'
go

/* Inizio definizione articoli, pubblicazione: server1_pub */

use appldb
go
exec sp_addarticle server1_pub,ordini_a,ordini,
  @destination_table = ordini,
  @creation_script = 'ordini.sch',
  @description = "Tabella ordini part. hrz",
  @upd_cmd = 'call m_ordini_upd'
go
exec sp_articlefilter 'server1_pub','ordini_a','server1ordini_filter',
  'stato != "PROV" and stato != "ANN"'
go
exec sp_articleview 'server1_pub','ordini_a','server1ordini_view',
  'stato != "PROV" and stato != "ANN"'
go
exec sp_addsubscription server1_pub,'ordini_a',SERVER1,
  @sync_type='none',@status='subscribed'
go
exec sp_addarticle server1_pub,articoli_a,articoli,
  @destination_table = articoli,
  @creation_script = 'articoli.sch',
  @description = "Tabella articoli intera",
  @upd_cmd = 'call m_articoli_upd'
```

```
go
exec sp_addsubscription server1_pub,'articoli_a',SERVER1,
  @sync_type='none',@status='subscribed'
go

/* Inizio definizione pubblicazione server2_pub */

use appldb
go
if exists(select * from syspublications where name='server2_pub')
  exec sp_droppublication server2_pub
go

/****** Object:  Sync task ******/
use msdb
go
if exists(select * from systasks where name='server2_pub_Sync')
  exec sp_droptask server2_pub_Sync
go
declare @taskid int
exec sp_addtask 'server2_pub_Sync','Sync','MASTER','sa','appldb',
  1,4,1,8,1,0,0,19970309,19991231,000000,235959,000000,000000,0,
  '',0,1,'',2,2,'',@newid = @taskid output
use appldb
exec sp_addpublication server2_pub,@taskid,
  @status=active,
  @description='Pubblicazione server2_pub dal master verso SERVER2',
  @restricted='true'
go

/* Inizio definizione articoli, pubblicazione: server2_pub */

use appldb
go
exec sp_addarticle server2_pub,ordini_a,ordini,
  @destination_table = ordini,
  @creation_script = 'ordini.sch',
  @description = "Tabella ordini part. hrz",
  @upd_cmd = 'call m_ordini_upd'
```

```
go
exec sp_articlefilter 'server2_pub','ordini_a','server2ordini_filter',
  'stato != "ATT" and stato != "APP"'
go
exec sp_articleview 'server2_pub','ordini_a','server2ordini_view',
  'stato != "ATT" and stato != "APP"'
go
exec sp_addsubscription server2_pub,'ordini_a',SERVER2,
  @sync_type='none',@status='subscribed'
go
exec sp_addarticle server2_pub,articoli_a,articoli,
  @destination_table = articoli,
  @creation_script = 'articoli.sch',
  @description = "Tabella articoli part. hrz",
  @upd_cmd = 'call m_articoli_upd'
go
exec sp_articlefilter 'server2_pub','articoli_a','server2articoli_filter',
  'stato != "ATT" and stato != "APP"'
go
exec sp_articleview 'server2_pub','articoli_a','server2articoli_view',
  'stato != "ATT" and stato != "APP"'
go
exec sp_addsubscription server2_pub,'articoli_a',SERVER2,
  @sync_type='none',@status='subscribed'
go

/* Creo le tabelle di servizio: STATO,FUNZ,FUNZSRV */

/****** Object:  Table FunzSrv ******/
if exists (select * from sysobjects
            where id = object_id('FunzSrv')
              and sysstat & 0xf = 3)
  drop table FunzSrv
go

create table FunzSrv (
  funz varchar(20) not null,
  nomesrv varchar(20) not null
```

```
  constraint pk_FunzSrv primary key
  (
    funz
  )
)
go

insert FunzSrv values('MASTER','MASTER')
insert FunzSrv values('VEND','SERVER1')
insert FunzSrv values('DC','SERVER2')
go

/****** Object:  Table Stato ******/
if exists (select * from sysobjects
            where id = object_id('Stato')
              and sysstat & 0xf = 3)
  drop table Stato
go

create table Stato (
  nome varchar(20) not null,
  prop varchar(20) not null,
  iniz int not null

  constraint pk_Stato primary key
  (
    nome
  )
)
go

insert Stato values('PROV','VEND',1)
insert Stato values('ANN','VEND',0)
insert Stato values('ATT','DC',1)
insert Stato values('APP','DC',0)
go

/****** Object:  Table Trans ******/
if exists (select * from sysobjects
```

```
            where id = object_id('Trans')
              and sysstat & 0xf = 3)
  drop table Trans
go

create table Trans (
  da varchar(20) not null,
  a varchar(20) not null

  constraint pk_Trans primary key
  (
    da,a
  )
)
go

insert Trans values('PROV','PROV')
insert Trans values('PROV','ATT')
insert Trans values('PROV','ANN')
insert Trans values('ATT','PROV')
insert Trans values('ATT','ATT')
insert Trans values('ATT','APP')
insert Trans values('ANN','PROV')
insert Trans values('ANN','ATT')
insert Trans values('APP','PROV')
go


/* Definizione della pubblicazione che replica le tabelle */

use appldb
go
if exists(select * from syspublications where name='serv_replica_pub')
  exec sp_droppublication serv_replica_pub
go

/****** Object:  Sync task ******/
use msdb
go
```

```
if exists(select * from systasks where name='stato_replica_Sync')
  exec sp_droptask stato_replica_Sync
go
declare @taskid int
exec sp_addtask 'stato_replica_Sync','Sync','MASTER','sa','appldb',
  1,4,1,8,1,0,0,19970309,19991231,000000,235959,000000,000000,0,
  '',0,1,'',2,2,'',@newid = @taskid output
use appldb
exec sp_addpublication serv_replica_pub,@taskid,
  @status=active,
  @description='Pubblicazione delle tabelle verso tutti i subscriber',
  @restricted='false'
go
use appldb
go
exec sp_addarticle serv_replica_pub,stato_a,Stato,
  @creation_script = 'Stato.sch',
  @description = "Tabella 'stato'"
go
exec sp_addarticle serv_replica_pub,trans_a,Trans,
  @creation_script = 'Trans.sch',
  @description = "Tabella 'trans'"
go
exec sp_addarticle serv_replica_pub,funzsrv_a,FunzSrv,
  @creation_script = 'FunzSrv.sch',
  @description = "Tabella 'FunzSrv'"
go
exec sp_addsubscription serv_replica_pub,'all',SERVER1
go
exec sp_addsubscription serv_replica_pub,'all',SERVER2
go
use appldb
go
```

A.2 Script relativo al server SERVER1

Vediamo ora lo script relativo al server periferico SERVER1:

```
/* Init PUSH - Server periferico: SERVER1 */
```

```
/* Publisher : SERVER1 */
/* Subscriber: MASTER */

/* Sun Mar 09 20:18:22 1997 */

use master
go
disk init
  name='distdata',
  physname='C:\Mssql\Data\distrib.dat',
  vdevno=5,
  size=15000
go
use master
go
disk init
  name='distlog',
  physname='C:\Mssql\Data\distlog.dat',
  vdevno=6,
  size=7000
go
create database distribution on distdata log on distlog
GO

use distribution
go

\MSSQL\INSTALL\INSTDIST.SQL

use master
go
xp_regwrite 'HKEY_LOCAL_MACHINE',
  'SOFTWARE\Microsoft\MSSQLServer\Replication','DistributionDB',
  'REG_SZ','distribution'
go
exec ("xp_regwrite 'HKEY_LOCAL_MACHINE',
  'SOFTWARE\Microsoft\MSSQLServer\Replication','WorkingDirectory',
  'REG_SZ','\\"+@@servername+"\C$\MSSQL\REPLDATA'")
go
```

```
use master
exec sp_serveroption SERVER1,dpub,true
go
use master
exec sp_addpublisher MASTER
go
use master
exec sp_addsubscriber MASTER
go
use master
go
exec sp_addserver MASTER
go
exec sp_addremotelogin MASTER,'sa'
go
use master
go
exec sp_dboption appldb,'published',true
go
use master
go
exec sp_dboption appldb,'subscribed',true
go

/* Inizio definizione pubblicazione appldb_pub */

use appldb
go
if exists(select * from syspublications where name='appldb_pub')
  exec sp_droppublication appldb_pub
go

/****** Object:  Sync task ******/
use msdb
go
if exists(select * from systasks where name='appldb_pub_Sync')
  exec sp_droptask appldb_pub_Sync
go
declare @taskid int
```

```
exec sp_addtask 'appldb_pub_Sync','Sync','SERVER1','sa','appldb',
  1,4,1,8,1,0,0,19970309,19991231,000000,235959,000000,000000,0,
  '',0,1,'',2,2,'',@newid = @taskid output
use appldb
exec sp_addpublication appldb_pub,@taskid,
  @status=active,
  @description='Pubblicazione appldb_pub dal SERVER1 verso il master',
  @restricted='false'
go

/* Inizio definizione articoli, pubblicazione: appldb_pub */

use appldb
go
exec sp_addarticle appldb_pub,ordini_a,ordini,
  @destination_table = ordini,
  @creation_script = 'ordini.sch',
  @description = "Tabella ordini part. hrz"
go
exec sp_articlefilter 'appldb_pub','ordini_a','ordini_filter',
  'stato = "PROV" or stato = "ANN"'
go
exec sp_articleview 'appldb_pub','ordini_a','ordini_view',
  'stato = "PROV" or stato = "ANN"'
go
exec sp_addsubscription appldb_pub,'ordini_a',MASTER,
  @sync_type='none',@status='subscribed'
go

/****** Object:  Stored Procedure ******/
if exists (select * from sysobjects
            where id = object_id('m_ordini_upd')
              and sysstat & 0xf = 4)
  drop procedure m_ordini_upd
go
create procedure m_ordini_upd @cod_ord int, @descr varchar(20),
    @stato varchar(10), @val int, @pkcod_ord int, @pkval int
as
update ordini
```

```
  set cod_ord = @cod_ord,descr = @descr,stato = @stato,val = @val-10000
  where cod_ord = @pkcod_ord and val = @pkval
go

/* RPC per la tabella ordini */


/****** Object:  Stored Procedure ******/
if exists (select * from sysobjects
            where id = object_id('ordini_ins')
              and sysstat & 0xf = 4)
  drop procedure ordini_ins
go
create procedure ordini_ins @cod_ord int,@descr varchar(20),
                            @stato varchar(10),@val int
as
if exists(select * from ordini where cod_ord=@cod_ord and val=@val)
  delete ordini
  where cod_ord=@cod_ord and val=@val
insert ordini values(@cod_ord,@descr,@stato,@val)
go

/****** Object:  Trigger ******/
if exists (select * from sysobjects
            where id = object_id('ordini_trig')
              and sysstat & 0xf = 8)
  drop trigger ordini_trig
go
create trigger ordini_trig on ordini for insert,update as
/* Variabili locali */
declare @proprietario varchar(80)
declare @funzdest varchar(80)
declare @msg varchar(200)
declare @upd int
declare @nomefunzsrv varchar(30)
declare @status1 int
select @status1 = 0
declare @status2 int
select @status2 = 0
```

```
/* Queste dipendono dalla tabella */
declare @cod_ord int, @cod_ord1 int
declare @descr varchar(20), @descr1 varchar(20)
declare @stato varchar(10), @stato1 varchar(10)
declare @val int, @val1 int

print 'Siamo dentro al trigger'
/* Stato di partenza */
declare @s_da varchar(10)
/* Stato di arrivo */
declare @s_a varchar(10)

/* select @s_da */

declare c1 cursor for select * from inserted
open c1

if exists(select * from deleted) begin
  select @upd = 1
  declare c2 cursor for select * from deleted
  open c2
end else select @upd = 0


while @status1 = 0
begin
  /* Se e' update cerco lo stato di partenza da deleted */
  if (@upd = 1) and (@status2 = 0) begin
    fetch next from c2 into @cod_ord1,@descr1,@stato1,@val1,
    select @status2=@@FETCH_STATUS
    if @status2 = 0 begin
      select @s_da=@stato1
      print @s_da
    end
  end

  /* Preleva da inserted */
  fetch next from c1 into @cod_ord,@descr,@stato,@val
  select @status1=@@FETCH_STATUS
```

```
if @status1 = 0 begin
  /* Caso particolare se val<0: e' proveniente dalla replica */
  if @val < 0 begin
    rollback transaction
    select @val = @val + 10000
    if @upd = 0 begin
      insert ordini values(@cod_ord,@descr,@stato,@val)
      continue
    end else begin
      update ordini
         set cod_ord = @cod_ord, descr = @descr,
        stato = @stato, val = @val
       where cod_ord = @cod_ord1 and descr = @descr1
         and stato = @stato1 and val = @val1
       continue
    end
  end
  select @s_a=@stato
  if @upd = 0 begin
    /* INSERT */
    select @s_da=@stato
    declare @x int
    select @x=(select s.iniz
           from Stato s,FunzSrv f
          where s.prop=f.funz and s.nome=@s_da
            and f.nomesrv=@@servername)
    if @x!=1
    begin
      rollback transaction
      raiserror('Stato di partenza non lecito!',1,2) with seterror
      continue
    end
  end else begin
    /* Verifica se la transizione e' corretta */
    /* oppure no (quindi esiste in TRANS) */
    /* UPDATE */
    if not exists(select * from Stato s,FunzSrv f
                  where s.prop=f.funz and s.nome=@s_da
                    and f.nomesrv=@@servername)
```

```
begin
  rollback transaction
  raiserror("Stato di partenza non di proprieta'",1,2) with seterror
  continue
end
if not exists(select * from Trans where da=@s_da and a=@s_a)
begin
  rollback transaction
  raiserror('Transazione non lecita!',1,2) with seterror
  continue
end
/* Cerca il proprietario del nuovo stato destinazione (trans.a) */
/* Preleva la funzione richiesta al server */
select @funzdest = (select s.prop from Stato s,Trans t
                     where s.nome=t.a and t.da=@s_da
                       and t.a=@s_a)
/* Preleva il server associato alla funzione */
select @proprietario = (select nomesrv from FunzSrv
                         where funz=@funzdest)
if @proprietario != @@servername begin
  declare @mastersrv varchar(20)
  /* Preleva il server MASTER */
  select @mastersrv = (select nomesrv from FunzSrv
                        where funz='MASTER')
  /* Lo stato porta in un altro server: annullo l'insert fatto */
  rollback transaction
  print 'Dentro'
  declare @execstr varchar(200)
  begin distributed transaction update_remoto
    if @upd=1 begin
      /* Se UPDATE: Inserimento locale con stato congelato */
      /* Update val=0 in val=maxval+1 */
      declare @max int
      select @max=(select max(val) from ordini
                    where cod_ord=@cod_ord)
      update ordini
         set val=@max+1
       where cod_ord=@cod_ord and val=0
      print 'Update: inserimento locale'
```

```
          end /* upd=1 */
        commit transaction
        select "Masterserver : " + @mastersrv
        /* Operazione remota */
        select @execstr = @mastersrv+'.appldb..ordini_ins '
        exec @execstr @proprietario,@cod_ord,@descr,@stato,@val
        print @execstr
      end else begin
        print 'UGUALI'
        /* Operazione su questo server. Non si deve fare nulla */
        /* Nella versione finale togliere l' else */
      end /* if @proprietario */
    end /* if upd=0 */
  end /* if @status1 */

end /* while */

close c1
deallocate c1

if @upd = 1 begin
  close c2
  deallocate c2
end
go
```

A.3 Script relativo al server SERVER2

Il terzo script è relativo al server periferico SERVER2:

```
/* Init PUSH - Server periferico: SERVER2 */
/* Publisher : SERVER2 */
/* Subscriber: MASTER */

/* Sun Mar 09 20:18:22 1997 */
use master
go
disk init
  name='distdata',
```

```
  physname='C:\Mssql\Data\distrib.dat',
  vdevno=5,
  size=15000
go
use master
go
disk init
  name='distlog',
  physname='C:\Mssql\Data\distlog.dat',
  vdevno=6,
  size=7000
go
create database distribution on distdata log on distlog
GO

use distribution
go

\MSSQL\INSTALL\INSTDIST.SQL

use master
go
xp_regwrite 'HKEY_LOCAL_MACHINE',
  'SOFTWARE\Microsoft\MSSQLServer\Replication','DistributionDB',
  'REG_SZ','distribution'
go
exec ("xp_regwrite 'HKEY_LOCAL_MACHINE',
  'SOFTWARE\Microsoft\MSSQLServer\Replication','WorkingDirectory',
  'REG_SZ','\\"+@@servername+"\C$\MSSQL\REPLDATA'")
go
use master
exec sp_serveroption SERVER2,dpub,true
go
use master
exec sp_addpublisher MASTER
go
use master
exec sp_addsubscriber MASTER
go
```

```
use master
go
exec sp_addserver MASTER
go
exec sp_addremotelogin MASTER,'sa'
go
use master
go
exec sp_dboption appldb,'published',true
go
use master
go
exec sp_dboption appldb,'subscribed',true
go

/* Inizio definizione pubblicazione appldb_pub */

use appldb
go
if exists(select * from syspublications where name='appldb_pub')
  exec sp_droppublication appldb_pub
go

/****** Object:  Sync task ******/
use msdb
go
if exists(select * from systasks where name='appldb_pub_Sync')
  exec sp_droptask appldb_pub_Sync
go
declare @taskid int
exec sp_addtask 'appldb_pub_Sync','Sync','SERVER2','sa','appldb',
  1,4,1,8,1,0,0,19970309,19991231,000000,235959,000000,000000,0,
  '',0,1,'',2,2,'',@newid = @taskid output
use appldb
exec sp_addpublication appldb_pub,@taskid,
  @status=active,
  @description='Pubblicazione appldb_pub dal SERVER2 verso il master',
  @restricted='false'
go
```

```
/* Inizio definizione articoli, pubblicazione: appldb_pub */

use appldb
go
exec sp_addarticle appldb_pub,ordini_a,ordini,@destination_table = ordini,
  @creation_script = 'ordini.sch',
  @description = "Tabella ordini part. hrz"
go
exec sp_articlefilter 'appldb_pub','ordini_a','ordini_filter',
  'stato = "ATT" or stato = "APP"'
go
exec sp_articleview 'appldb_pub','ordini_a','ordini_view',
  'stato = "ATT" or stato = "APP"'
go
exec sp_addsubscription appldb_pub,'ordini_a',MASTER,
  @sync_type='none',@status='subscribed'
go

/****** Object:  Stored Procedure ******/
if exists (select * from sysobjects
            where id = object_id('m_ordini_upd')
              and sysstat & 0xf = 4)
  drop procedure m_ordini_upd
go
create procedure m_ordini_upd @cod_ord int, @descr varchar(20),
  @stato varchar(10), @val int, @pkcod_ord int, @pkval int
as
update ordini
  set cod_ord = @cod_ord,descr = @descr,stato = @stato,val = @val-10000
  where cod_ord = @pkcod_ord and val = @pkval
go

/* RPC per la tabella ordini */


/****** Object:  Stored Procedure ******/
if exists (select * from sysobjects
            where id = object_id('ordini_ins')
```

```
              and sysstat & 0xf = 4)
  drop procedure ordini_ins
go
create procedure ordini_ins @cod_ord int,@descr varchar(20),
  @stato varchar(10),@val int
as
if exists(select * from ordini where cod_ord=@cod_ord and val=@val)
  delete ordini
  where cod_ord=@cod_ord and val=@val
insert ordini values(@cod_ord,@descr,@stato,@val)
go

/****** Object:  Trigger ******/
if exists (select * from sysobjects
            where id = object_id('ordini_trig')
              and sysstat & 0xf = 8)
  drop trigger ordini_trig
go
create trigger ordini_trig on ordini for insert,update as
/* Variabili locali */
declare @proprietario varchar(80)
declare @funzdest varchar(80)
declare @msg varchar(200)
declare @upd int
declare @nomefunzsrv varchar(30)
declare @status1 int
select @status1 = 0
declare @status2 int
select @status2 = 0
/* Queste dipendono dalla tabella */
declare @cod_ord int, @cod_ord1 int
declare @descr varchar(20), @descr1 varchar(20)
declare @stato varchar(10), @stato1 varchar(10)
declare @val int, @val1 int

print 'Siamo dentro al trigger'
/* Stato di partenza */
declare @s_da varchar(10)
/* Stato di arrivo */
```

```
declare @s_a varchar(10)

/* select @s_da */

declare c1 cursor for select * from inserted
open c1

if exists(select * from deleted) begin
  select @upd = 1
  declare c2 cursor for select * from deleted
  open c2
end else select @upd = 0


while @status1 = 0
begin
  /* Se e' update cerco lo stato di partenza da deleted */
  if (@upd = 1) and (@status2 = 0) begin
    fetch next from c2 into @cod_ord1,@descr1,@stato1,@val1,
    select @status2=@@FETCH_STATUS
    if @status2 = 0 begin
      select @s_da=@stato1
      print @s_da
    end
  end

  /* Preleva da inserted */
  fetch next from c1 into @cod_ord,@descr,@stato,@val
  select @status1=@@FETCH_STATUS
  if @status1 = 0 begin
    /********/
    /* Caso particolare se val<0: e' proveniente dalla replica */
    if @val < 0 begin
      rollback transaction
      select @val = @val + 10000
      if @upd = 0 begin
        insert ordini values(@cod_ord,@descr,@stato,@val)
        continue
      end else begin
```

```
    update ordini
       set cod_ord = @cod_ord, descr = @descr,
           stato = @stato, val = @val
     where cod_ord = @cod_ord1 and descr = @descr1
       and stato = @stato1 and val = @val1
     continue
  end
end
select @s_a=@stato
if @upd = 0 begin
  /* INSERT */
  select @s_da=@stato
  declare @x int
  select @x=(select s.iniz
         from Stato s,FunzSrv f
        where s.prop=f.funz and s.nome=@s_da
          and f.nomesrv=@@servername)
  if @x!=1
  begin
    rollback transaction
    raiserror('Stato di partenza non lecito!',1,2) with seterror
    continue
  end
end else begin
  /* Verifica se la transizione e' corretta */
  /* oppure no (quindi esiste in TRANS) */
  /* UPDATE */
  if not exists(select * from Stato s,FunzSrv f
                where s.prop=f.funz and s.nome=@s_da
                  and f.nomesrv=@@servername)
  begin
    rollback transaction
    raiserror("Stato di partenza non di proprieta'",1,2) with seterror
    continue
  end
  if not exists(select * from Trans where da=@s_da and a=@s_a)
  begin
    rollback transaction
    raiserror('Transazione non lecita!',1,2) with seterror
```

```
  continue
end
/* Cerca il proprietario del nuovo stato destinazione (trans.a) */
/* Preleva la funzione richiesta al server */
select @funzdest = (select s.prop from Stato s,Trans t
                      where s.nome=t.a and t.da=@s_da
                        and t.a=@s_a)
/* Preleva il server associato alla funzione */
select @proprietario = (select nomesrv from FunzSrv
                         where funz=@funzdest)
if @proprietario != @@servername begin
  declare @mastersrv varchar(20)
  /* Preleva il server MASTER */
  select @mastersrv = (select nomesrv from FunzSrv
                        where funz='MASTER')
  /* Lo stato porta in un altro server: annullo l'insert fatto */
  rollback transaction
  print 'Dentro'
  declare @execstr varchar(200)
  begin distributed transaction update_remoto
    if @upd=1 begin
      /* Se UPDATE: Inserimento locale con stato congelato */
      /* Update val=0 in val=maxval+1 */
      declare @max int
      select @max=(select max(val) from ordini
       where cod_ord=@cod_ord)
      update ordini
         set val=@max+1
       where cod_ord=@cod_ord and val=0
      print 'Update: inserimento locale'
    end /* upd=1 */
  commit transaction
  select "Masterserver : " + @mastersrv
  /* Operazione remota */
  select @execstr = @mastersrv+'.appldb..ordini_ins '
  exec @execstr @proprietario,@cod_ord,@descr,@stato,@val
  print @execstr
end else begin
  print 'UGUALI'
```

```
        /* Operazione su questo server. Non si deve fare nulla */
        /* Nella versione finale togliere l' else */
      end /* if @proprietario */
    end /* if upd=0 */
  end /* if @status1 */

end /* while */

close c1
deallocate c1

if @upd = 1 begin
  close c2
  deallocate c2
end

go
exec sp_addarticle appldb_pub,articoli_a,articoli,
  @destination_table = articoli,
  @creation_script = 'articoli.sch',
  @description = "Tabella articoli part. hrz"
go
exec sp_articlefilter 'appldb_pub','articoli_a','articoli_filter',
  'stato = "ATT" or stato = "APP"'
go
exec sp_articleview 'appldb_pub','articoli_a','articoli_view',
  'stato = "ATT" or stato = "APP"'
go
exec sp_addsubscription appldb_pub,'articoli_a',MASTER,
  @sync_type='none',@status='subscribed'
go

/****** Object:  Stored Procedure ******/
if exists (select * from sysobjects
            where id = object_id('m_articoli_upd')
              and sysstat & 0xf = 4)
  drop procedure m_articoli_upd
go
create procedure m_articoli_upd @cod_art int, @nome varchar(20),
```

```
  @stato varchar(10), @val int, @pkcod_art int, @pkval int
as
update articoli
  set cod_art = @cod_art,nome = @nome,stato = @stato,val = @val-10000
  where cod_art = @pkcod_art and val = @pkval
go

/* RPC per la tabella articoli */


/****** Object:  Stored Procedure ******/
if exists (select * from sysobjects
            where id = object_id('articoli_ins')
              and sysstat & 0xf = 4)
  drop procedure articoli_ins
go
create procedure articoli_ins @cod_art int,@nome varchar(20),
  @stato varchar(10),@val int
as
if exists(select * from articoli where cod_art=@cod_art and val=@val)
  delete articoli
  where cod_art=@cod_art and val=@val
insert articoli values(@cod_art,@nome,@stato,@val)
go

/****** Object:  Trigger ******/
if exists (select * from sysobjects
            where id = object_id('articoli_trig')
              and sysstat & 0xf = 8)
  drop trigger articoli_trig
go
create trigger articoli_trig on articoli for insert,update as
/* Variabili locali */
declare @proprietario varchar(80)
declare @funzdest varchar(80)
declare @msg varchar(200)
declare @upd int
declare @nomefunzsrv varchar(30)
declare @status1 int
```

```
select @status1 = 0
declare @status2 int
select @status2 = 0
/* Queste dipendono dalla tabella */
declare @cod_art int, @cod_art1 int
declare @nome varchar(20), @nome1 varchar(20)
declare @stato varchar(10), @stato1 varchar(10)
declare @val int, @val1 int

print 'Siamo dentro al trigger'
/* Stato di partenza */
declare @s_da varchar(10)
/* Stato di arrivo */
declare @s_a varchar(10)

/* select @s_da */

declare c1 cursor for select * from inserted
open c1

if exists(select * from deleted) begin
  select @upd = 1
  declare c2 cursor for select * from deleted
  open c2
end else select @upd = 0


while @status1 = 0
begin
  /* Se e' update cerco lo stato di partenza da deleted */
  if (@upd = 1) and (@status2 = 0) begin
    fetch next from c2 into @cod_art1,@nome1,@stato1,@val1,
    select @status2=@@FETCH_STATUS
    if @status2 = 0 begin
      select @s_da=@stato1
      print @s_da
    end
  end
```

```
/* Preleva da inserted */
fetch next from c1 into @cod_art,@nome,@stato,@val
select @status1=@@FETCH_STATUS
if @status1 = 0 begin
  /* Caso particolare se val<0: e' proveniente dalla replica */
  if @val < 0 begin
    rollback transaction
    select @val = @val + 10000
    if @upd = 0 begin
      insert articoli values(@cod_art,@nome,@stato,@val)
      continue
    end else begin
      update articoli
         set cod_art = @cod_art, nome = @nome,
       stato = @stato, val = @val
       where cod_art = @cod_art1 and nome = @nome1
         and stato = @stato1 and val = @val1
       continue
    end
  end
  select @s_a=@stato
  if @upd = 0 begin
    /* INSERT */
    select @s_da=@stato
    declare @x int
    select @x=(select s.iniz
           from Stato s,FunzSrv f
          where s.prop=f.funz and s.nome=@s_da
            and f.nomesrv=@@servername)
    if @x!=1
    begin
      rollback transaction
      raiserror('Stato di partenza non lecito!',1,2) with seterror
      continue
    end
  end else begin
    /* Verifica se la transizione e' corretta */
    /* oppure no (quindi esiste in TRANS) */
    /* UPDATE */
```

```
if not exists(select * from Stato s,FunzSrv f
               where s.prop=f.funz and s.nome=@s_da
                 and f.nomesrv=@@servername)
begin
  rollback transaction
  raiserror("Stato di partenza non di proprieta'",1,2) with seterror
  continue
end
if not exists(select * from Trans where da=@s_da and a=@s_a)
begin
  rollback transaction
  raiserror('Transazione non lecita!',1,2) with seterror
  continue
end
/* Cerca il proprietario del nuovo stato destinazione (trans.a) */
/* Preleva la funzione richiesta al server */
select @funzdest = (select s.prop from Stato s,Trans t
                     where s.nome=t.a and t.da=@s_da
                       and t.a=@s_a)
/* Preleva il server associato alla funzione */
select @proprietario = (select nomesrv from FunzSrv
                         where funz=@funzdest)
if @proprietario != @@servername begin
  declare @mastersrv varchar(20)
  /* Preleva il server MASTER */
  select @mastersrv = (select nomesrv from FunzSrv
                        where funz='MASTER')
  /* Lo stato porta in un altro server: annullo l'insert fatto */
  rollback transaction
  print 'Dentro'
  declare @execstr varchar(200)
  begin distributed transaction update_remoto
    if @upd=1 begin
      /* Se UPDATE: Inserimento locale con stato congelato */
      /* Update val=0 in val=maxval+1 */
      declare @max int
      select @max=(select max(val) from articoli
       where cod_art=@cod_art)
      update articoli
```

```
              set val=@max+1
             where cod_art=@cod_art and val=0
            print 'Update: inserimento locale'
          end /* upd=1 */
        commit transaction
        select "Masterserver : " + @mastersrv
        /* Operazione remota */
        select @execstr = @mastersrv+'.appldb..articoli_ins '
        exec @execstr @proprietario,@cod_art,@nome,@stato,@val
        print @execstr
      end else begin
        print 'UGUALI'
        /* Operazione su questo server. Non si deve fare nulla */
        /* Nella versione finale togliere l' else */
      end /* if @proprietario */
    end /* if upd=0 */
  end /* if @status1 */

end /* while */

close c1
deallocate c1

if @upd = 1 begin
  close c2
  deallocate c2
end

go
```

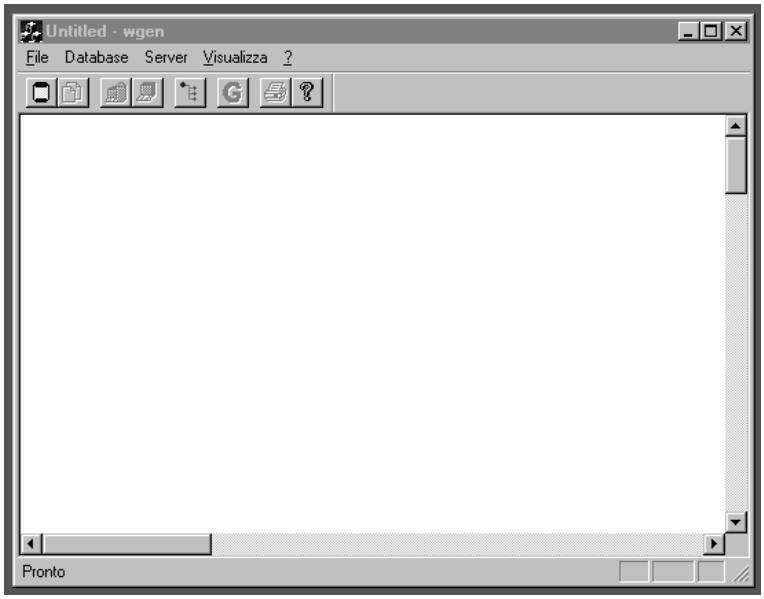
Negli script non abbiamo riportato il file INSTDIST.SQL poiché fa parte di SQL Server e non viene generato dal programma realizzato.

Appendice B

Utilizzo del programma

Vediamo ora i passi compiuti per realizzare gli script per l' applicazione di esempio introdotta nel sesto capitolo.

Il programma si presenta all' utente nel seguente modo:



I comandi possono essere introdotti mediante i menu o la toolbar:

Vediamo il significato di ogni singolo pulsante della toolbar (da sinistra a destra):

- *Set database name* consente di definire il nome del database dell' applicazione;
- *Add Table* permette di introdurre manualmente una tabella del database, specificando i campi e i tipi;
- *Add Master Server* consente l' immissione del master server, nome e pathname dello script;
- *Add Push Server* permette l' introduzione di un push server, sarà possibile aggiungere i task e le tabelle per la pubblicazione;
- *Server tree* visualizza la gerarchia dei server connessi;
- *Gen Script* inizia il processo (completamente automatico) di creazione degli script per tutti i server introdotti;
- *Print* stampa i dati raccolti (solo per il debug);
- *About ...* visualizza alcune informazioni sul programma.

Esaminiamo ora i passi da compiere per introdurre tutti i dati relativi alla applicazione utente e quindi realizzare gli script.

1. Definire il database dell' applicazione: premendo il pulsante sinistro del mouse sull' apposito pulsante (o scegliendo l' apposita voce del menù). Il dialog box che si presenta all' utente è riportato in figura B.1. Occorre immettere il nome del database utilizzato, è unico per tutto il sistema distribuito.

Figura B.1. Nuovo database

2. Introdurre le tabelle del database, l' utente dovrà introdurre tutti i dati richiesti: nome della tabella, pathname del creation script, nome e tipo dei campi e flag per primary key, stato o val. Il dialog box è in figura B.2.

Figura B.2. Inserimento di una nuova tabella

3. Definizione del master server, premendo l' apposito pulsante un dialog box consente di inserire nome fisico e pathname dello script relativo al master server.

4. Iniziamo quindi la definizione del grafo, abbiamo due scelte: introdurre contemporaneamente gli stati e i push server proprietari, oppure definire prima tutti gli stati (disegnando il grafo) poi i push server. Nel secondo caso un doppio click nella client window del programma apre un menu per l' introduzione dello stato (figura B.3).

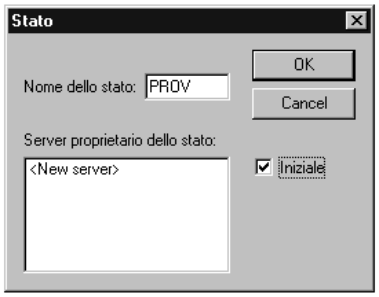


Figura B.3. Nuovo stato.

Possiamo immettere il nome dello stato e il flag che indica se stato iniziale o no. Se decidiamo di immettere solo gli stati usciamo utilizzando il pulsante OK.

Altrimenti se abbiamo già dei server definiti possiamo associarvi lo stato selezionandoli dalla list box. Se non esistono o ne vogliamo creare uno

nuovo un doppio click su `<new server>` apre il dialog box di immissione dei nuovi server (figura B.4).

Figura B.4. Immissione nuovo server.

5. Introduciamo i dati relativi al server e specifichiamo eventuali altri stati di proprietà. Mediante i pulsanti "Set task" e "Set tables" possiamo associare rispettivamente task e tabelle al server.

   Immettiamo un task utilizzando il dialog box di figura B.5.

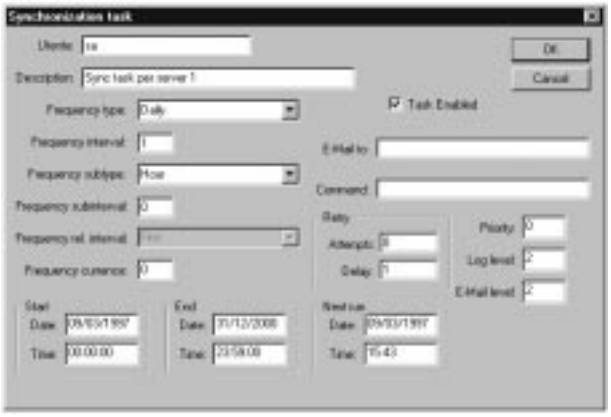


Figura B.5. Immissione nuovo task.

Questo task verrà associato alla pubblicazione utilizzata per gli articoli relativi alle tabelle specificate utilizzando il pulsante "Set tables", il dialog box è in figura B.6.

Il dialog presenta tutte le tabelle, l' utente sceglie quali introdurre con modalità "read-only" (quindi solo una pubblicazione dal master verso questo server) oppure "read/write" (cioè di proprietà del server ristretta agli stati).

Figura B.6. Inclusione di una tabella ad un server.

6. Dopo aver introdotto tutti i server e i relativi stati possiamo definire le transizioni del grafo: per realizzare una transizione occorre premere il pulsante sinistro del mouse prima sullo stato di partenza, poi sullo stato di arrivo, automaticamente viene disegnata la transizione (figura B.7), nel disegno abbiamo utilizzato un cerchio più spesso per indicare l' autoanello.

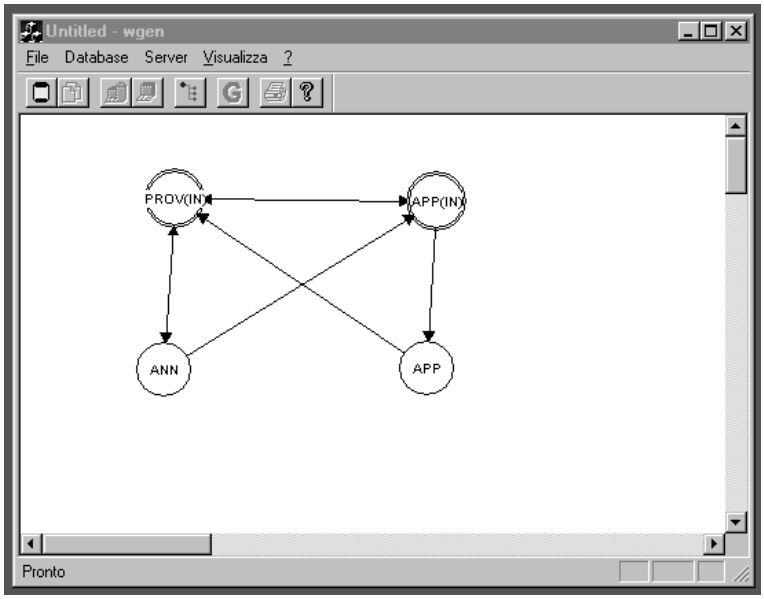


Figura B.7. Grafo delle transazioni.

Ora la definizione dell' applicazione è terminata, possiamo generare gli script utilizzando l' apposito pulsante.

Bibliografia

[1] M. Tamer Özsu, P. Valduriez: *Principles of distributed database systems*, Prentice-Hall, 1991.

[2] S. Ceri, G. Pelagatti: *Distributed databases principles & systems*, McGRAW-HILL, 1985.

[3] M. Vincini: *Distributed Server Architecture*, rapporto tecnico.

[4] A. Tanenbaum: *Computer networks (2nd edition)*, Prentice-Hall, 1991.

[5] B. Stroustrup: *Il linguaggio C++ (Seconda edizione)*, Addison-Wesley, 1991.

[6] D. J. Kruglinski: *Inside Visual C++*, Microsoft Press, 1996.

[7] MSDN Library: *SQL Server Books Online*, Microsoft, 1996.

[8] MSDN Library: *Windows NT Server RPC SDK*, Microsoft, 1996.

Indice analitico