UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Informatica

PARIDBMS: ACCESSO REMOTO

Relatore : Prof. Enoch Peserico Stecchini Negri De Salvi

Correlatori : Prof. Luca Pretto, Ing. Paolo Bertasi

Laureando : Deniz Tartaro Dizmen

Anno Accademico 2009-2010

Sommario

PariDBMS è uno dei primi gestori di basi di dati distribuiti creati su una rete peer-to-peer. Un'elevata affidabilità, efficienza ed efficacia nel soddisfare le esigenze degli utenti intenzionati a gestire i propri database attraverso il nostro gestore sono alcuni degli obiettivi principali.

Il plugin è giunta alla sua seconda versione; questa tesi ha lo scopo di descrivere alcuni limiti che si sono eliminati e i miglioramenti che sono stati apportati con il lavoro svolto negli ultimi mesi sul progetto PariDBMS.

Dopo una breve introduzione generale del mondo PariPari nel quale sono impegnati ormai un centinaio di studenti del Dipartimento di Ingegneria dell'Informazione dell'Università di Padova, si descriverà la struttura del plugin PariDBMS e i suoi processi di funzionamento basilari, ponendo poi, nel finale dell'elaborato, una particolare attenzione alla parte che ho seguito personalmente, cioè il meccanismo con cui si permette agli utenti di accedere ai dati memorizzati nel supporto fisico di calcolatori remoti.

Indice

1 INTRODUZIONE

L'intenzione di questo elaborato è descrivere i miglioramenti che sono stati apportati a PariDBMS. Quando sono entrato in questo progetto esisteva già una prima versione del plugin, realizzata da tre colleghi che si sono laureati nell'anno accademico 2007 – 2008 : A. Cecchinato (attuale team leader), A. Costa e J. Buriollo. Assieme a me, ha cominciato a lavorare sul progetto anche il collega Alberto Rizzi; il compito che ci è stato assegnato era quello di aggiornare il plugin, effettuando di fatto un *refactoring*, testare il funzionamento di ciò che era già esistente dalla prima versione e aggiungere alcune nuove funzionalità non presenti nel primo rilascio del plugin. Poiché PariDBMS si è sviluppato in momenti successivi per opera di persone diverse, per avere la visione dettagliata di ogni sua parte si consiglia la lettura delle tesi di tutti gli studenti fin qui citati.

Ci si potrebbe a questo punto chiedere cosa sia PariDBMS. Come si può notare, la prima parte deriva da ***Pari****Pari*, un progetto ambizioso nell'ambito delle reti di calcolatori nato nel Dipartimento di Ingegneria dell'Informazione, mentre la seconda è ***DBMS***. Questo acronimo sta per DataBase Management System, software specializzato nel gestire grandi collezioni di dati. PariDBMS è quindi un DBMS distribuito sui nodi della rete PariPari, una rete peer-to-peer che eroga molti servizi differenti. Come vedremo via via nella tesi, la sua natura distribuita pone problematiche di cui i DBMS tradizionali non devono tener conto, come l'elevata frequenza con cui può cambiare la topologia della rete, con la conseguente possibile indisponibilità dei dati presenti sui nodi che si scollegano senza preavviso. Si deve in tutti i casi garantire una buona continuità di servizio.

Nei prossimi paragrafi si presenterà brevemente la rete PariPari mostrandone le caratteristiche principali e i servizi da essa forniti, prima di iniziare a descrivere il plugin PariDBMS.

1.1 Le reti peer-to-peer e le reti client-server

Una rete è formata da un insieme di calcolatori interconnessi tra loro in maniera più o meno complicata. Il mezzo fisico di connessione prende il nome di *linea di connessione*, mentre i computer ad esso connessi vengono chiamati *nodi*. Esistono due architetture di reti di calcolatori fortemente in antitesi l'una con l'altra.

La prima è formata da due tipologie di nodi, secondo il ruolo che essi assumono nella rete. Esistono i *server* (servienti) i quali hanno il compito di erogare certi tipi di servizi e i *client* (clienti) che necessitano di risorse. Una rete che distingua nettamente e permanentemente i due tipi di nodi viene chiama *client-server*. In esse, il ruolo di server viene assunto da nodi con capacità superiori e con hardware molto potente, in grado di rimanere sempre connessi alla rete per garantire che possano stabilmente essere soddisfatte le richieste dei clienti. I client, invece, sono dei comuni calcolatori che di solito necessitano di richiedere servizi e risorse ai server per poter svolgere le proprie attività.

La seconda architettura, contrariamente alla prima, è costituita da nodi considerati equivalenti tra loro. "Rete peer-to-peer" potrebbe infatti essere tradotto come "rete tra pari". In un'architettura di questo tipo, non esistono nodi che siano perennemente server oppure client ma ogni nodo assume il ruolo di server quando soddisfa le richieste di un suo pari e funge da client quando è lui stesso a necessitare di altri.

Eseguendo un confronto, emergono pregi e difetti in entrambe le architetture. Nel client-server, per esempio, il fatto che i calcolatori con il ruolo di server siano macchine con elevate prestazioni permette di soddisfare velocemente le richieste avanzate dai client, se sono tra loro in corretta proporzione numerica. Dall'altra parte, il guasto di un server o l'improvviso aumento di nodi client costituisce una serie minaccia per il funzionamento di un sistema basato su questa architettura. In una rete peer-to-peer, questa problematica non esiste. Le risorse sono totalmente condivise tra i nodi, nessuno è indispensabile ma tutti sono importanti. Poiché in una rete peer-to-peer i nodi potrebbero disconnettersi a loro piacimento, la condivisione delle risorse garantisce l'affidabilità della rete, in quanto esistono sicuramente altri nodi che sono in grado di rimpiazzare il vuoto creatosi dopo che alcuni calcolatori si sono scollegati.

1.2 La rete PariPari

La rete PariPari è dunque una rete di calcolatori peer-to-peer, completamente serverless e multifunzionale. La differenza principale tra PariPari e altre reti peer-to-peer sta nella sua natura strutturata; essendo una rete completamente serverless, nessun nodo è indispensabile per il corretto funzionamento del sistema. Quando un host necessita di reperire delle risorse può sfruttare un modulo che realizza una tabella chiamata *tabella hash distribuita* (DHT, Distribuited Hash Table). Tale modulo rappresenta le risorse come degli indirizzi di rete, in modo che quando le si cercano si riesca ad associare per ciascuna il suo nodo detentore.

La struttura di PariPari è modulare. Ogni componente (denominato plugin) si occupa solamente di certi compiti, delegando ad altri il lavoro che non gli compete. In questo modo, ognuno di essi può considerare gli altri come entità di cui non importa conoscere la struttura interna ma è sufficiente sapere come possano venir utilizzati. Si tratta del concetto, molto comune in ingegneria, di *black box*.

All'interno di PariPari esistono due gruppi di plugin. I primi, considerati basilari per il funzionamento di tutti gli altri, compongono la cosiddetta *cerchia interna*. I plugin che vi rientrano sono:

- Connectivity: gestisce la connettività dei nodi assegnando di volta in volta ad ogni plugin una porzione della banda disponibile affinché uno solo non monopolizzi l'utilizzo della rete.
- Local Storage: gestisce l'accesso in scrittura alla memoria locale del nodo. Ad ogni plugin viene assegnata una cartella sul disco che rappresenta l'unica parte a lui accessibile per scrivere dati.
- DHT: realizza la tabella hash distribuita di cui si è accennato prima. Permette ai plugin di ricercare delle risorse e di pubblicizzare quelle proprie.
- Crediti: realizza un meccanismo con cui ad ogni plugin vengono assegnati dei crediti virtuali necessari perché possa "acquistare" delle risorse.

Esiste poi una *cerchia esterna* di plugin, i quali, basandosi su quelli della cerchia interna, offrono una vasta gamma di servizi, alcuni dei quali sono *IRC, Web Server, Instant Messagge, File Sharing, Voip, DBMS...*

Si è finora tralasciato un aspetto cruciale, ovvero come interagiscano tra loro i diversi plugin. Esiste un nucleo, denominato *Core*, che non può essere considerato né un modulo né un plugin, che

rappresenta la parte fondamentale di tutto il sistema PariPari. Ha il compito di permettere il caricamento e la comunicazione di tutti i plugin. Paragonando PariPari ad un elaboratore, si potrebbe affermare che la cerchia interna sia il suo sistema operativo ed il Core il suo kernel. Una risorsa non può quindi essere richiesta direttamente al plugin corrispondente della cerchia interna, ma deve prima essere domandata al Core, il quale controllerà se è possibile soddisfare la richiesta e poi eventualmente concede la risorsa desiderata.

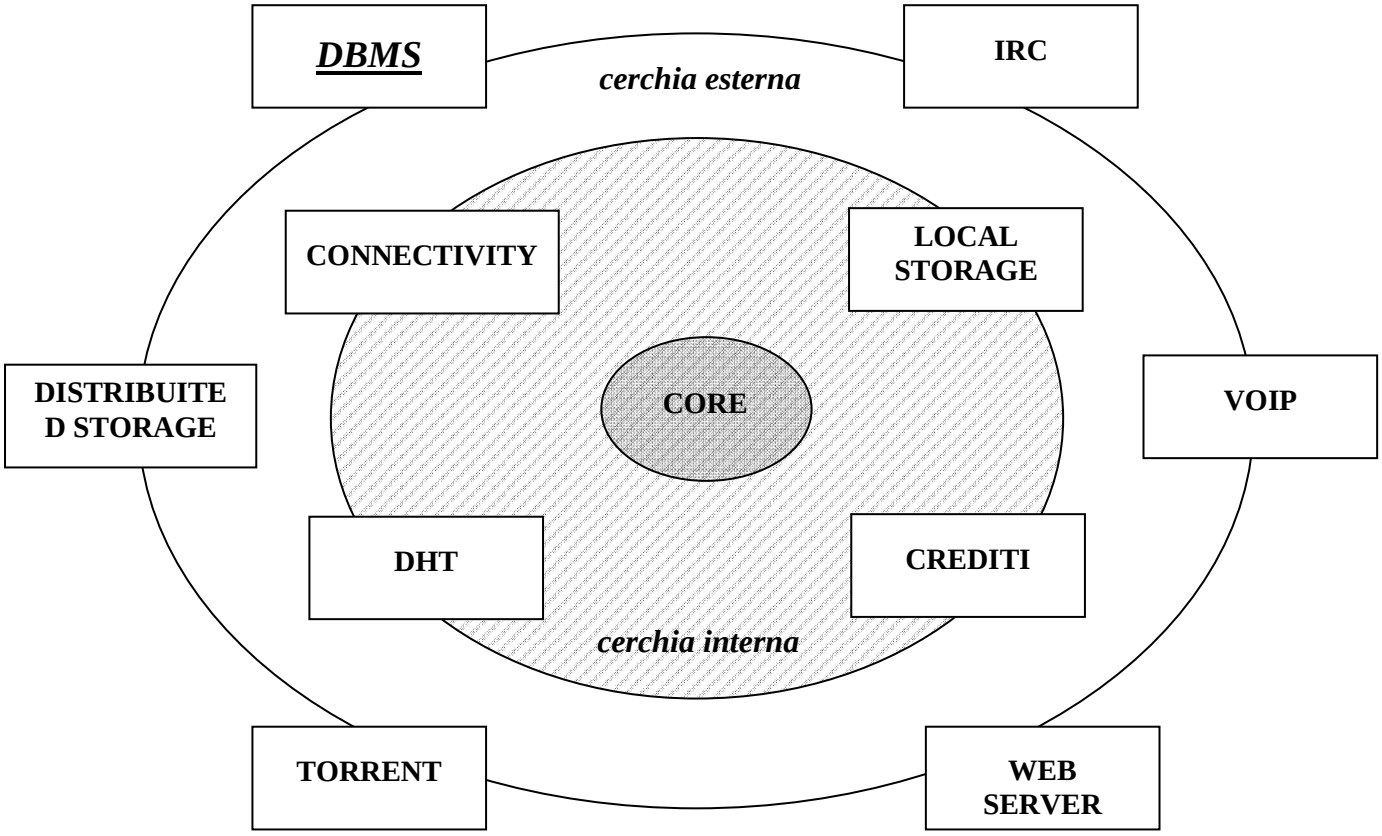

Figura 1.1 : Struttura di PariPari

Dopo questa breve introduzione, si può procedere alla descrizione del nostro plugin. Vediamo ora le principali caratteristiche di PariDBMS.

2 IL PLUGIN PARIDBMS

Dopo l'introduzione, utile per avere un'idea di quale sia il contesto in cui si colloca il plugin PariDBMS, si esporrà in questo capitolo la sua architettura e il suo funzionamento senza entrare troppo nei dettagli realizzativi, tenendo conto anche di alcune importanti considerazioni sulla teoria delle basi di dati. Verranno esposti i principali meccanismi e processi di funzionamento, gli strati dell'architettura e varie problematiche cercando fornire un quadro generale del plugin. Per maggiori dettagli implementativi si rimanda alla lettura degli elaborati di J. Buriollo, A. Cecchinato e A. Costa, colleghi che hanno progettato e realizzato la prima versione di PariDBMS: nelle loro tesi gli argomenti tra poco esposti sono sviluppati a livello molto superiore che in questo elaborato.

PariDBMS è dunque un plugin facente parte della cerchia esterna di PariPari. Come gli altri di questa cerchia, ha il compito di erogare uno specifico servizio, servendosi dei plugin della cerchia interna per richiedere tutte le risorse di cui necessita per il suo funzionamento e che svolgono dei compiti che a lui non competono. Il servizio che si offre con questo plugin è la gestione di collezioni di dati, come può essere intuito dall'acronimo DBMS presente nel suo nome.

Il linguaggio di programmazione utilizzato è Java; la scelta è dettata dal fatto che un'applicazione scritta con questo linguaggio può essere eseguito liberamente su qualsiasi sistema operativo, senza doverlo adattare ad uno particolare.

2.1 Considerazioni preliminari

Prima di cominciare a descrivere in dettaglio il plugin vero e proprio è importante soffermarsi su delle considerazioni che serviranno per comprendere meglio il resto dell'elaborato; sono un semplice richiamo di alcuni concetti e si da per scontato una conoscenza quanto meno generica del modello e della teoria delle basi di dati.

Un DBMS è un sistema software in grado di gestire collezioni di dati che siano *grandi*, *condivise* e *persistenti*, assicurando la loro *affidabilità* e *privatezza*. Essendo un prodotto informatico, deve essere *efficiente* ed *efficace*. Una *base di dati*, invece, è una collezione di dati gestita da un DBMS. I significati dei termini precedenti saranno spiegati con maggior dettaglio all'interno dell'elaborato, mostrando anche come PariDBMS abbia tentato di rispondere a queste esigenze.

Le basi di dati possono essere rappresentate attraverso quattro tipi di modelli logici:

- il modello gerarchico, basato sull'uso delle strutture ad albero;
- il modello reticolare, basato sull'uso dei grafi, sviluppato successivamente al modello gerarchico;
- il modello relazionale, basato sul concetto di relazione (o tabella), che consente di organizzare i dati in insiemi di record a struttura fissa;
- il modello a oggetti, sviluppato come evoluzione del modello relazionale, che estende alle basi di dati il paradigma di programmazione ad oggetti.

I precedenti modelli vengono detti logici in quanto le strutture utilizzate da questi modelli riflettono una particolare organizzazione, cioè ad albero, a grafo, a tabella o a oggetti. Ora, si porrà maggior attenzione al modello relazionale poiché è quello adottato da PariDBMS. Un DBMS che presenti un modello relazionale prende il nome di RDBMS (Relational DBMS).

Il modello relazionale, attualmente il più diffuso, è strutturato attorno al concetto di relazione, detta anche tabella. Trattandosi di un modello logico, esso fornisce dei costrutti per la definizione della struttura dei dati, dei costrutti per la definizione dei vincoli e operazioni da poter eseguire sui dati. L'unico costrutto per la definizione della struttura è la relazione, costituita da un nome di relazione e da una serie di attributi che possiamo definire nel seguente modo:

$$\mathbf{R(A_1, A_2, ..., A_N)}$$

dove **R** è il nome della relazione e $\mathbf{(A_1, A_2, ... , A_N)}$ sono gli attributi della relazione; ogni attributo ha uno specifico dominio (intero, stringa, ecc...) e il nome dell'attributo descrive, in un certo senso, il "ruolo" del dominio nella relazione.

Per quanto riguarda la definizione dei vincoli, il modello relazionale permette la dichiarazione di quattro tipi di vincoli di base: il vincolo di dominio (all'interno di ciascuna tupla il valore di ogni attributo $\mathbf{A_i}$ deve essere un valore atomico del dominio $\mathbf{dom(A_i)}$), il vincolo di chiave (stabilisce l'univocità chiave primaria), il vincolo di integrità dell'entità (stabilisce l'impossibilità che gli attributi della chiave primaria possano assumere valori nulli) e il vincolo di integrità referenziale (stabilisce come possano sussistere dei riferimenti tra due relazioni).

Le operazioni sui dati possono essere classificati in operazioni di reperimento (retrival) e di aggiornamento (update); la differenza tra le due tipologie sta nel fatto che la prima produce nuove relazioni applicando operatori algebrici a un insieme di relazioni già esistenti senza però modificare lo stato della base di dati e viene usata per effettuare interrogazioni sulla base di dati stessa, mentre la seconda, modifica lo stato della base di dati attraverso tre modalità, cioè l'inserimento di nuovi dati, la cancellazione di dati già esistenti o la modifica degli stessi (che corrisponde ad una cascata cancellazione-inserimento). Le operazioni di aggiornamento, cambiando lo stato della base di dati, sono critici in quanto potrebbero portare lo stato da valido a uno stato che viola certi vincoli di integrità specificati sullo schema iniziale; bisogna quindi porre particolare attenzione a questa categoria di operazioni.

Infine, un'ultima breve considerazione sul linguaggio standard per basi di dati relazionali: SQL (structured query language) è un linguaggio completo, come tale comprende sia istruzioni per la definizione dei dati, l'interrogazione che l'aggiornamento; è quindi sia un DDL (data definition language) che un DML (data manipulation language).

2.2 Problematiche di un DBMS distribuito

Nelle considerazioni preliminari si è accennato alle caratteristiche che ogni DBMS deve presentare per poter essere ritenuto un buon prodotto; un DBMS distribuito come PariDBMS non solo deve affrontare le difficoltà di poter proporre le stesse proprietà ma deve risolvere una serie di problematiche legate alla sua natura distribuita.

Un primo importante problema riguarda la replicazione. Quando viene creata una base di dati in un nodo, questa deve essere replicata in altri nodi della rete per aggirare il problema dell'indisponibilità dei dati. Bisogna decidere quante copie verranno replicate; tale valore di soglia

non deve essere casuale ma deve tener conto del fatto che se il numero totale di repliche di una base di dati è troppo piccolo esiste il rischio che nessuna di esse sia accessibile in un certo istante perché i nodi in cui è salvata potrebbero essere indisponibili o inaccessibili, mentre se il numero è troppo alto si ha uno spreco dello spazio di memorizzazione per altre basi di dati. La dinamicità topologica della rete PariPari obbliga ad un monitoraggio continuo del numero di copie attive nella rete per ogni base di dati: se le repliche diminuiscono fino al valore di soglia prefissato deve avviarsi un meccanismo di replicazione a caldo dei dati senza però rendere indisponibile la base di dati stessa per eventuali accessi.

Il problema della replicazione porta ad affrontare quello della sincronizzazione e dell'aggiornamento. Si è appena affermato che l'unico modo per ovviare al problema dell'indisponibilità è la replicazione dei dati; più copie della stessa base di dati in nodi diversi porta immediatamente ad affrontare la complicazione della sincronizzazione dei dati. Per garantire l'allineamento tra le diverse copie, si deve procedere in modo tale che le operazioni di aggiornamento su quella base di dati vengano eseguite nello stesso ordine anche su tutte le altre repliche salvate nella rete. Inoltre, sempre per la dinamicità della rete PariPari, alcuni dei suoi nodi potrebbero disconnettersi improvvisamente e senza preavviso e rimare inattivi per un certo periodo di tempo. Durante la fase di inattività del nodo potrebbero avvenire nelle repliche delle modifiche ai dati; ovviamente il nodo inattivo non ha modo di aggiornarli. Nel momento in cui ritorna attivo, presenterà dati disallineati dalle altre copie; si dice che i dati sono inconsistenti. Quindi, per risolvere questo problema, quando un nodo si riconnette alla rete deve prima di tutto aggiornare i propri database richiedendo la lista delle modifiche alle repliche attive ed eseguirle nello stesso ordine per allinearsi con loro; solo dopo questa fase di aggiornamento, può diventare anch'essa una copia attiva e utilizzabile per gli utenti. Per garantire il corretto funzionamento dei processi di sincronizzazione e di aggiornamento dei dati, in ogni istante, deve esistere almeno una copia della base di dati aggiornata e attiva. Per questo motivo, la creazione del database deve avvenire in locale dove i dati vengono salvati nella memoria del nodo e diventano immediatamente disponibili (e sono anche aggiornati) per qualsiasi richiesta, mentre altre X copie vengono replicate nella rete; se la creazione non avvenisse in locale ma potesse essere una operazione eseguita in remoto, si correrebbe il rischio di, nel caso di un fallimento dovuto alla caduta del nodo creatore, non poter accedere ai dati perché non sono stati creati o perché non ho copie aggiornate della base di dati stessa.

Il fatto di possedere dati ridondanti permette di ragionare su un aspetto interessante, quello di accedere in lettura/scrittura a dati che non sono direttamente salvati nella memoria locale del nodo in uso. Tale caratteristica, chiamata *accesso remoto,* pone problemi soprattutto per quanto riguarda

le autorizzazioni che ogni utente possiede o meno per poter leggere o scrivere su certe basi di dati rispetto ad altri; questa seconda parte è chiamata *controllo dell'autorizzazione* (oppure *login*). L'accesso remoto e il login sono rispettivamente oggetto della mia tesi e di quella del mio collega Alberto Rizzi.

2.3 Processi di funzionamento di PariDBMS

La versione attuale del plugin, pur essendo una versione con funzionalità ridotte, fornisce delle metodologie efficienti ed efficaci per risolvere le principali problematiche esposte nel paragrafo 2.2. Nei prossimi paragrafi verranno brevemente spiegati i principali processi di funzionamento del plugin. I primi quattro erano già presenti quando mi sono messo a lavorare sul progetto; non sono stati né pensati né scritti da me ma ho ritenuto fosse indispensabile la loro esposizione perché rappresentano le fondamenta di PariDBMS. Non essendo un lavoro svolto da me verranno descritti in modo poco approfondito.

2.3.1 Condivisione e gestione delle risorse

Come ampiamente affermato, PariDBMS è un sistema software distribuito nei nodi di una rete peer-to-peer, che per definizione è un insieme di nodi equivalenti. Ogni calcolatore può mettere a disposizione degli altri certe risorse come capacità di calcolo, spazio di memorizzazione, database, ecc... Dall'esterno si avrà così l'impressione di lavorare con un DBMS tradizionale che possiede un enorme spazio di memorizzazione, una elevata capacità di calcolo, ecc... La memoria di massa disponibile per PariDBMS è dunque un insieme di frammenti di memoria separati, ognuno dei quali appartenente ad un diverso nodo della rete; tuttavia tale separazione fisica sarà inosservabile per un utilizzatore (un utente o un'applicazione) esterno. Infatti, ciascun nodo può salvare e leggere dal supporto fisico di qualsiasi altro calcolatore della rete, inviando agli host remoti le query SQL da far eseguire al loro DBMS locale: tale processo, chiamato *accesso remoto*, è l'oggetto principale della mia tesi e verrà ampiamente descritto più avanti nel capitolo ad esso dedicato. Per poter scrivere o leggere da un supporto diverso da quello locale, bisogna innanzitutto conoscere dove salvare e recuperare i dati; per poterlo fare la rete PariPari dispone di un meccanismo chiamato *Distribuited Hash Table* (DHT) che permette di pubblicizzare delle risorse e

ricercare i nodi che le posseggono; nel momento in cui si dichiara il possesso di una risorsa deve essere associata ad essa una chiave e la ricerca deve avvenire in base ad essa. Come altri plugin della cerchia esterna di PariPari, anche PariDBMS si serve del modulo DHT; infatti il plugin periodicamente avvia un processo di pubblicizzazione e uno di ricerca. Quando un nodo si accorge di avere ancora spazio per poter memorizzare nuovi dati allora dichiara su DHT una risorsa identificata come *disponibilità di memoria* che andrà a far parte della memoria distribuita di PariDBMS di cui si è parlato precedentemente, ma anche i dati già memorizzati sul nodo devono essere reperibili agli altri host e quindi un'altra risorsa sono i *database* locali e vengono reclamizzati con una chiave che li renda unici in tutta la rete, in modo che una ricerca non produca risultati ambigui o sbagliati; in realtà ogni base di dati viene pubblicizzata con tre chiavi differenti: la prima è una stringa contenente il nome del database al quale vengono concatenati duecentocinquantacinque caratteri casuali, la seconda è una stringa prodotta dalla concatenazione del nome della base di dati e dal nome utente del suo creatore e la terza è semplicemente il nome utente del creatore della base di dati.

Una ricerca effettuata su DHT con una delle precedenti chiavi restituisce una lista di indirizzi IP di nodi che posseggono la risorsa associata a tale chiave; a seconda che si scelga l'unione o l'intersezione dei risultati delle ricerche con le tre chiavi si possono ottenere diversi esiti. In questo caso, volendo trovare le risorse identificate come *database* si chiederà a DHT di cercare quali nodi sono possesso di risorse alle quali sono associate tutte e tre le chiavi di ricerca e si farà quindi l'intersezione dei tre risultati. Alla fine di ciò, il risultato finale della ricerca conterrà una lista di nodi che hanno le replicazioni delle basi di dati locali. L'utilità di tale processo di ricerca viene compresa se ci si ricorda della problematica dell'indisponibilità dei dati e della sua conseguente soluzione che consiste nella replicazione dei dati nella rete. Si era detto che quando un nodo rimane inattivo per un certo periodo di tempo e successivamente ritorna operativo è molto probabile che i suoi dati si siano disallineati dalle altre replicazioni e quindi prima di rendersi disponibile deve eseguire tutte le operazioni di aggiornamento che gli altri nodi in possesso delle stesse copie hanno già eseguito. Deve chiedere a questi host la lista delle transazioni eseguite durante la sua fase di inattività. A questo punto si concepisce l'importanza di questo processo di ricerca sfruttando il modulo DHT.

2.3.2 Replicazione a caldo

Il processo di replicazione a caldo è stato curato da A. Cecchinato e lo si esporrà a livello concettuale, senza entrare nei dettagli implementativi. Per i particolari del protocollo di replicazione si consiglia di leggere il capitolo ad esso dedicato nella tesi del mio collega.

Per quanto riguarda il processo in esame, rispetto alla versione precedente esistono delle piccole differenze che verranno illustrate nell'ultima parte dell'elaborato nel capitolo *innovazioni di PariDBMS*; tutto ciò che riguarda la logica del protocollo è comunque invariato.

La replicazione a caldo è la giusta risposta alla problematica dell'indisponibilità dei dati. Il plugin ha modo di sapere in ogni istante quanti nodi contenenti copie dei propri database sono attivi; tuttavia questi, pur essendo attivi in un certo istante, potrebbero scollegarsi a loro piacimento in qualsiasi istante successivo. Deve pertanto essere stabilito un valore di soglia: finché il numero di nodi attivi è al di sopra di essa non ci sono problemi, ma quando scende troppo e diventa inferiore al valore prestabilito deve partire nei nodi di PariDBMS il processo di replicazione, perché si rischia di perdere ogni copia di quei dati. L'avvio della replicazione ha luogo con una certa probabilità p: essendo tale probabilità $p<1$ non tutte le copie prenderanno parte alla replicazione. Il motivo per cui si assegna una probabilità di avvio di replicazione è che durante questo processo non è possibile eseguire su quella base di dati nessuna operazione. E' giusto quindi che alcuni database siano coinvolti nella replicazione a caldo mentre altri rimangano disponibili a fronte di richieste di accesso. Durante questo periodo, le basi di dati impegnate nel processo memorizzano in una coda tutte le richieste di modifica che le copie attive hanno nel frattempo eseguito, in modo da riallinearsi immediatamente prima ancora di dichiararsi disponibili estraendo dalla coda le operazioni ed eseguendole nello stesso ordine delle repliche attive. Per quanto riguarda le nuove copie che nascono per replicazione in nuovi nodi bisogna spendere qualche parola in più.

Quando una nuova base di dati viene creata in nodo, il plugin salva una rappresentazione della sua struttura logico-relazione nella memoria di massa[1]: vengono salvati gli schemi delle tabelle, i nomi degli attributi e i suoi vincoli. Il sistema impone due limitazioni per quanto riguarda la creazione[2]: esiste un limite del numero di database che possono essere ospitati ed attualmente tale limite è fissato a cinque. Inoltre non possono nascere due basi di dati uguali nello stesso supporto di memoria. Il processo di replicazione può essere suddiviso in due sotto processi da eseguire consecutivamente: il primo è la ricerca di un nodo disponibile. Per disponibile si intende in questo caso che abbia ancora spazio nella memoria e che non violi le precedenti limitazioni. La ricerca avviene sfruttando il modulo DHT di PariPari come spiegato nel paragrafo precedente. Solo dopo

[1] vedi *database di supporto* , par. 2.4.1

[2] In realtà esistono anche altre limitazioni, i quali saranno spiegati nel paragrafo *login*

aver trovato un nodo con le caratteristiche ricercate si può proseguire alla fase successiva che consiste nell'invio di istruzioni per poter creare una copia della base di dati. In realtà in questa fase non vengono inviati i dati veri e propri ma si utilizza un altro metodo. Dal momento che ogni nodo possiede lo schema logico-relazione delle proprie basi di dati, si generano a partire da questi le istruzioni SQL che, una volta eseguiti, permettano di creare un database che sarà perfettamente identico a quello da cui si sono generati i comandi SQL. Sfruttando questa proprietà del linguaggio SQL (è sia un DDL che un DML, vedi paragrafo 2.1) invece che inviare dei dati al nodo ospitante la replicazione si inviano i codici SQL generati nel modo appena descritto. Una volta che il destinatario riceverà le istruzioni, sarà sufficiente la loro esecuzione usando il DBMS locale perché su quel nodo venga creata una replica della base di dati originale.

Rimane un ultimo aspetto da chiarire che riguarda la scelta dei valori di soglia minimi di allerta e la probabilità *p* che definisce la probabilità di avviare la replicazione. Nel protocollo elaborato da A. Cecchinato i due parametri valgono rispettivamente 12 e 0,7. Ciò significa che, quando il numero di copie di una base di dati diventa meno di 12, ogni copia fa partire un processo di replicazione con probabilità del 70%. Nella scelta di tali valori, il mio collega ha dovuto tener conto di diversi fattori. Quando si arriva alla condizione di allerta (meno di 12 copie attive del database) attraverso il parametro *p* si può manipolare quante nuove copie nasceranno (senza considerare che non tutti i processi di replicazione potrebbe avvenire con successo). Se la probabilità di replicazione fosse troppo elevata si rischierebbe di rendere indisponibili tutte le basi di dati senza che l'utente abbia la possibilità nel frattempo di eseguire query su di esse; in tal modo, il sistema diventerebbe altamente inefficiente, sia perché l'utente non riuscirebbe ad interagire sia perché il carico di lavoro per gestire molti processi di replicazione e la successiva sincronizzazione sarebbe troppo elevato e rallenterebbe tutte le altre funzionalità di PariDBMS. Al contrario, se la probabilità assegnata fosse troppo bassa il rischio sarebbe quello che, una volta finita la fase di replicazione, il numero totale di copie rimaste si ancora troppo basso e quindi un solo ciclo sarebbe stato sufficiente e servirebbe una seconda iterazione. Si capisce immediatamente l'inefficienza di una tale situazione. Con i parametri attuali, il sistema inizia a replicare una base di dati quando ci sono meno di 12 copie attive e in ogni ciclo vengano create in media 7 nuove replicazioni.

2.3.3 Sincronizzazione

Il protocollo di sincronizzazione di PariDBMS è l'oggetto della tesi di A. Costa, un altro collega che prima di me ha lavorato su questo progetto. Vediamo brevemente qual è la soluzione adottata dal plugin a tale proposito.

La sincronizzazione è una problematica che per certi aspetti emerge come conseguenza sia della replicazione dei dati studiata nel punto precedente che dell'allineamento (si vedrà tale aspetto nel punto successivo). Ogni volta che una base di dati diventa indisponibile, perché il nodo si disconnette o perché il database viene coinvolto in un processo di replicazione, potrebbe trascorre del tempo durante il quale nelle altre copie avvengono delle modifiche ai dati. Abbiamo detto che se succede questo evento quella base di dati deve eseguire le stesse operazioni delle repliche nello stesso ordine. Ci si accorge a questo punto della difficoltà di ordinare le query, ovvero di come contrassegnare l'ordine di esecuzione delle stesse. Si è elaborato pertanto un meccanismo che permetta a tutti i nodi di poter elaborare le query salvate nello stesso ordine. Il problema è che i nodi non eseguono tali operazioni nello stesso istante perché i messaggi inviati in rete giungono ai destinatari dopo un lasso di tempo nel quale è compresa anche la latenza introdotta dalla rete.

Attualmente la sincronizzazione avviene in questo modo: ogni nodo controlla una variabile che rappresenta il proprio clock temporale. Esistono innanzitutto una serie di regole. Ogni evento è contrassegnato con il valore del clock temporale di quel momento (il valore del clock letto in un determinato istante prende il nome di *timestamp*[3]); quindi gli eventi sono etichettati con il timestamp. Non possono esistere due eventi dello stesso processo con lo stesso timestamp, tra due eventi deve sempre avvenire uno scatto del clock. Ogni volta che un nodo invia un messaggio allega anche il timestamp, mentre ad ogni messaggio ricevuto calcola il nuovo valore del clock prendendo il massimo tra quello interno e quello ricevuto e poi incrementa il valore scelto di uno. Queste sono le regole di base. Esistono due tipi di operazioni: interrogazioni e modifiche. Per la prima tipologia non avrebbe senso sincronizzare le operazioni in quanto non avviene nessuna modifica ai dati della base di dati. Per quanto riguarda le operazioni di modifica ciò è indispensabile. La sincronizzazione delle operazioni per quanto riguarda questa seconda tipologia di operazioni deve essere realizzata in modo tale che se anche una sola replicazione non può compiere l'aggiornamento ai dati, tutte le altre ne vietino l'esecuzione. Le operazioni di modifica approvate da tutti i nodi contenenti una replicazione di un database vengono inserite in una coda a priorità e aspettano qui di essere eseguite. Successivamente verranno estratte in ordine di timestamp crescente. PariDBMS riesce pertanto, utilizzando questo processo di sincronizzazione, ad eseguire le operazioni nell'ordine cronologico in cui esse sono state create.

Tutti i dettagli del protocollo, compresa la spiegazione dello scambio di messaggi di sincronizzazione, sono descritti nell'ultimo capitolo della tesi di A. Costa.

---

[3] Termine preso dal nome di un algoritmo studiato da un ricercatore statunitense: "timestamp distribuiti", di Lamport. La sincronizzazione di PariDBMS sfrutta una sua variante.

2.3.4 Aggiornamento

Come la replicazione a caldo e la sincronizzazione, anche questo processo è stato scritto da un altro collega. L'aggiornamento è stato curato da J. Buriollo; come nei casi precedenti, il grado di dettaglio sarà il minimo per comprendere questo processo a livello concettuale.

Quando un nodo si ricollega dopo un periodo in cui è rimasto inattivo i dati salvati nella memoria locale potrebbe non essere più aggiornati; non è dato sapere se nelle altre copie siano state eseguite operazioni di aggiornamento o meno. Per non correre rischi la decisione migliore è considerarle non aggiornate, senza nemmeno controllare se effettivamente è così o meno. Immediatamente dopo al risveglio del nodo, tutti i database contenuti in esso non saranno disponibili per un certo periodo durante il quale si eseguirà il processo di aggiornamento.

PariDBMS memorizza per ogni base di dati locale una lista dove vengono di volta in volta aggiunte le operazioni di modifica eseguite su di essi[4]. Il salvataggio prevede che oltre al codice SQL della transazione si memorizzi anche il timestamp ad esso associato. La lista delle transazioni è indispensabile per quanto riguarda l'aggiornamento dei dati. Un nodo che si risvegli da un periodo di inattività deve aver modo di sapere quali operazioni nel frattempo si è perso. Una volta reperita la lista può eseguire le operazioni che hanno timestamp maggiore rispetto a quelle che ha già eseguito prima della disconnessione. Dalla lista delle transazioni periodicamente vengono cancellate quelle che hanno timestamp inferiore a una certa soglia; questo perché non è possibile, per problemi di memoria, aggiungere voci all'infinito. Può quindi succedere che un nodo rimasto inattivo per molto tempo non riesca a recuperare tutte le operazioni necessarie perché un suo database ritorni allineato rispetto alle altre copie; in tal caso sarà costretto a cancellarlo.

Nella lista delle operazioni vengono aggiunte solo quelle operazioni di aggiornamento che il DBMS locale ha già eseguito fisicamente. Potrebbero però esserci nella coda delle transazioni da eseguire operazioni approvate ma non ancora processate. Quando un nodo si riconnette, non solo deve recuperare tale lista delle operazioni eseguite ma, prima ancora, deve ricopiare nella propria coda le operazioni che il nodo remoto ha consentito e ha intenzione di eseguire negli istanti a venire. Vengono prima effettuate le operazioni salvate nella lista e successivamente quelle estratte dalla coda. In questo modo non viene persa nessuna operazione di aggiornamento e il database riallinea i propri dati.

Mentre l'aggiornamento di una replicazione sta procedendo, essa partecipa nel frattempo anche alla sincronizzazione delle nuove query. Queste operazioni, se approvate, vengono anch'esse inserite nella coda di operazioni e verranno eseguite dopo quelle contenute nella lista delle operazioni effettuate.

---

[4] vedi *database di supporto* , par. 2.4.1

2.4 Struttura di PariDBMS

PariDBMS presenta un'architettura composta da tre strati sovrapposti, ognuno dei quali utilizza le funzionalità del livello inferiore per rispondere alle esigenze del livello superiore. La seguente figura illustra tale situazione:

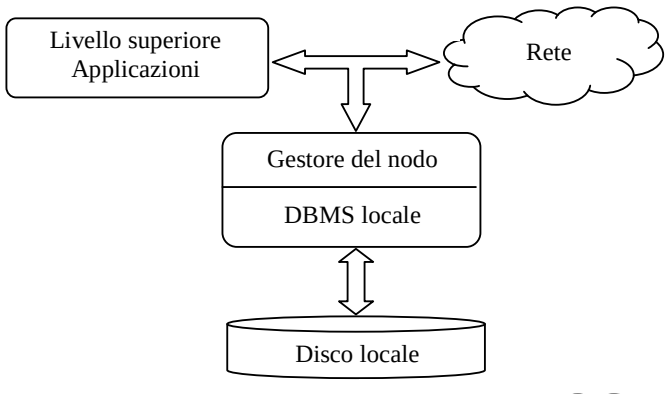

Figura 2.1 : Architettura a tre strati di PariDBMS

Il livello più alto prende il nome di *gestore del nodo* ed è il fulcro del progetto in quanto svolge tutte le funzionalità descritte nel paragrafi precedenti. Il livello intermedio è un DBMS tradizionale che si occupa di gestire localmente le basi di dati; il livello inferiore è la memoria di massa locale, dove il plugin ha modo di leggere e scrivere i dati necessari al suo funzionamento.

Per quanto riguarda il DBMS tradizionale dello strato intermedio è stato scelto HSQLDB, un software open source scaricabile direttamente dal sito web http://www.hsqldb.org. Questa decisione è figlia del fatto che il software in questione è completamente scritto in Java e quindi si adatta perfettamente all'integrazione con PariPari; inoltre è dettata dall'impossibilità oggettiva di scriverne uno ex-novo, che richiederebbe forse anni per essere progettato e realizzato. HSQLDB permette due tipi di memorizzazione delle basi di dati: in maniera temporanea nella memoria RAM del calcolatore oppure permanentemente nella sua memoria di massa. Per i nostri scopi è richiesta la memorizzazione persistente e quindi usufruiamo della seconda modalità. Il DBMS crea una serie di file nella memoria di massa alla prima richiesta di connessione ad un database; se esiste allora la connessione viene stabilita, in caso contrario viene generato il database desiderato. In Java, la connessione ad un database è rappresentato attraverso un oggetto di tipo *Connection*. Per tentare di stabile una connessione ad una base di dati il codice Java è il seguente:

```
Connection conn = DriverManager.getConnection ( "jdbc:hsqldb:file:" + dbpath +
                                                dbname, properties );
```

Il primo argomento della precedente chiamata è l'URL della base di dati, la seconda invece un'istanza della classe *Properties* mediante la quale si possono definire delle proprietà aggiuntive. Anche se la precedente chiamata funziona correttamente e riesce ad accedere in lettura/scrittura nella memoria locale del nodo, dalle prossime versioni del plugin sarà probabilmente necessaria una modifica in quanto per gestire lo spazio di memorizzazione non ci si affida ad un modulo di PariPari dedicato a questo scopo come Local Storage ma si forza l'utilizzo di HSQLDB passandogli l'URL di una cartella privata del nostro plugin. Tuttavia, per ora può essere tralasciata.

Il livello superiore, cioè il gestore del nodo, è quello più importante in quanto è proprio il software sul quale lavoriamo noi programmatori di PariDBMS. Ora cerchiamo di addentrarci all'interno di questo strato, che finora è stato considerato come una sorta "scatola nera" (*black box)* con certe funzionalità senza però spiegare quasi nulla della sua struttura interna.

2.4.1 Il database di supporto

Finora si è volutamente tralasciato come PariDBMS riesca a reperire tra una sessione e l'altra tutte le informazioni di cui ha bisogno per eseguire le operazioni di aggiornamento, di sincronizzazione e di replicazione a caldo descritti nel paragrato 2.3. Esiste un database di supporto (*SupportDB*), invisibile agli utenti che permette di memorizzare al suo interno i dati fondamentali del plugin, come le query eseguite, la lista dei database creati sul nodo, i nomi delle relazioni per ogni base di dati, ecc.

Il database di supporto è costituito da cinque tabelle: DATABASES, TABLES, FIELDS, LOG e USERS[5]. Le prime tre tabelle permettono di salvare la struttura logica-relazionale di tutte le basi di dati locali del nodo.

DATABASES contiene le tuple con i riferimenti alle basi i dati locali e ha due attributi; NAME e HASH. Il primo attributo rappresenta il nome che l'utente ha scelto per una base di dati, il secondo è il suo codice identificativo. Tale codice viene generato concatenando il valore dell'istante di clock del calcolatore nel momento di creazione del database (sfruttando il metodo `System.currentTimeMillis()`) e una stringa casuale di sessantaquattro caratteri (utilizzando i metodi `Math.round()` e `Math.random()`). Non si può essere certi che due database diversi abbiano lo stesso codice hash, ma viste le modalità con cui viene creato lo si considera come evento impossibile. La chiave primaria di questa tabella è data da entrambi gli attributi; sarebbe un errore

[5] La tabella USERS non era prevista nella prima versione di PariDBMS. Essa serve da supporto per il controllo delle autorizzazioni degli utenti. Vedi il paragrafo *login*.

considerare solamente NAME come chiave perché database differenti ma omonimi possono coesistere sul nodo a patto che siano creati da utenti diversi.

TABLES contiene tuple che rappresentano le tabelle delle basi di dati specificati in DATABASES. Possiede tre attributi, DBNAME, DBHASH e NAME. I primi due rappresentano rispettivamente il nome e il codice identificativo della base di dati al quale la tabella appartiene, mentre il terzo è proprio il nome scelto per la tabella. Come per DATABASES, anche per TABLES la chiave primaria è composta da tutti i suoi attributi; infatti tabelle con lo stesso nome sono valide solo se appartenenti a basi di dati diverse. Esiste inoltre un vincolo di chiave esterna nei campi DBNAME e DBHASH che si riferiscono alla chiave primaria di DATABASES (NAME , HASH).

FIELDS contiene tuple che rappresentano gli attributi delle varie tabelle dei database locali; ha otto attributi. DBNAME e DBHASH rappresentano rispettivamente il nome e il codice identificativo della base di dati di appartenenza; TABLE contiene il nome della tabella nella quale è inserito l'attributo; NAME è il nome scelto dall'utente per questo attributo; TYPE rappresenta il tipo SQL; ISNULL deve essere un valore booleano che indica se l'attributo può o meno avere valore nullo; PK sta per *primary key* e se ha valore "true" allora indica che questo attributo fa parte della chiave primaria; REF è un campo che deve essere riempito solo se questo attributo fa parte di una chiave esterna e in tal caso bisogna scrivere il nome della tabella a cui si riferisce; UNIQUEID contiene un valore intero che indica se l'attributo fa parte di un insieme di campi con la proprietà relazionale *unique* e tutti gli attributi di un certo insieme hanno valore identico in questo campo. La chiave primaria di FIELDS è formata dai primi quattro attributi descritti, cioè da DBNAME, DBHASH, TABLE e NAME; ciò perché all'interno della stessa tabella di un certo database non si possono avere due campi con gli stessi nomi. Esiste anche un vincolo di chiave esterna formata dai campi DBNAME, DBHASH e TABLE, riferiti alla chiave primaria di TABLES.

LOG contiene tuple che rappresentano le query di aggiornamento eseguite su una base di dati locale e che nel loro insieme rappresentano il log delle transazioni del nodo. La sua utilità è emersa quando si è esposto il processo di aggiornamento. Questa tabella contiene tre attributi. TIMESTAMP è un intero che rappresenta il timestamp associato alla transazione; DBNAME e DBHASH, come al solito, sono il nome e il codice identificativo della base di dati sulla quale la transazione è stata eseguita; QUERY è una stringa formata da tutte le istruzioni SQL della transazione. La chiave primaria di questa tabella è formata da TIMESTAMP, DBNAME e DBHASH; questo perché è impossibile eseguire due operazioni con lo stesso timestamp sulla stessa base di dati.

USERS è una delle novità di questa seconda versione di PariDBMS. E' una tabella che ci permette di memorizzare il nome utente (*username*) e la password degli utenti che posseggono un

database creato da loro. I suoi attributi sono: DBNAME, USERNAME e PASSWORD. Il primo è il nome della base di dati creata, mentre gli altri due sono, come si può intuire, username e password associati a un utente. La coppia di attributi DBNAME e USERNAME costituisce la chiave primaria visto che lo stesso utente non può creare due database omonimi.

2.4.2 Il gestore del nodo

In precedenza si è mostrata con la figura 2.1 l'architettura del plugin e si è evidenziato come il nostro lavoro si svolga nel livello superiore della pila. Questo strato, denominato *gestore del nodo*, può essere suddiviso in tre blocchi principali, ad ognuno dei quali corrisponde uno dei processi di base di PariDBMS.

Il blocco ***gestione database*** si occupa di tutto ciò che riguarda i database locali se si stanno usando risorse possedute dal nodo, altrimenti deve essere in grado di gestire correttamente l'accesso remoto[6]; gestisce l'esecuzione di query sulle basi di dati e governa il processo di sincronizzazione delle operazioni nell'ordine prestabilito su tutte le replicazioni.

Il blocco ***gestione nodi*** si occupa principalmente di pubblicizzare e ricercare delle risorse in rete, monitorare il numero di copie attive di ciascun database locale e replicare i dati quando corrono il rischio di indisponibilità. In tal caso gestisce anche il processo di replicazione.

Il blocco ***gestione aggiornamento*** è responsabile di riallineare le basi di dati locali dopo un periodo di inattività del nodo. Si occupa quindi di eseguire il processo di aggiornamento.

I tre blocchi comunicano tra loro attraverso una serie di classi che costituiscono la cosiddetta ***struttura principale***. Essa serve per il corretto funzionamento e l'integrazione delle varie parti. Inoltre gestisce lo scambio di messaggi tra la rete (quindi altri nodi di PariDBMS) e il nodo locale e smista i messaggi ricevuti alle parti corrette del plugin; deve anche interpretare i comandi che l'utente dalla console vuole eseguire.

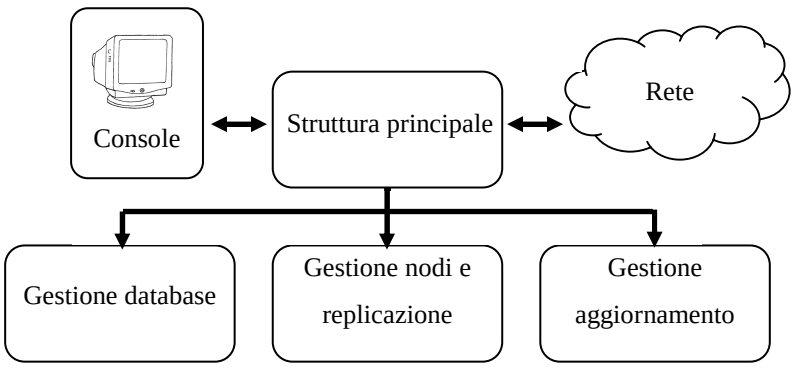

Figura 2.2 : Struttura interna di PariDBMS

[6] Si descriverà ampiamente il funzionamento dell'*accesso remoto* nel capitolo 3.

Possiamo a questo punto analizzare la struttura principale, spiegando velocemente da quali classi è formata. I tre blocchi di base invece li considereremo come delle black box senza, per ora, addentrarci in nessuna di esse.

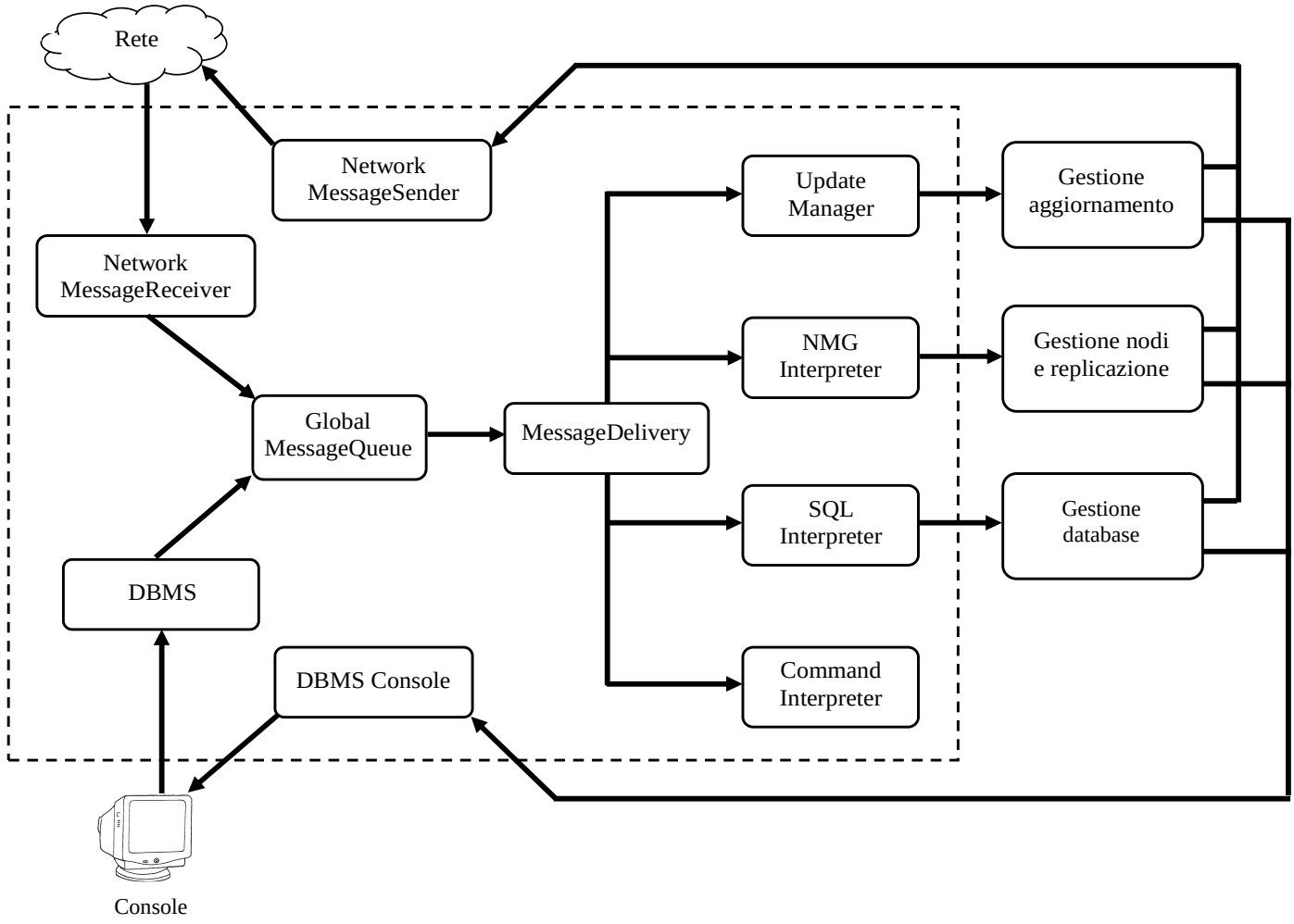

Figura 2.3 : Struttura di PariDBMS.
Nel rettangolo tratteggiato le classi della struttura principale

Descriviamo brevemente le principali classi della struttura principale.

CommandInterpreter : è un thread che ha il compito di interpretare i comandi che l'utente scrive dalla console di PariPari. Tutti i messaggi di questo tipo devono iniziare con il carattere "\" . Riesce ad interpretare ed eseguire i seguenti comandi:

- **\use DBNAME:USERNAME:PASSWORD**[7] : permette di appropriarsi dell'uso di un database, che non deve per forza essere memorizzato nella memoria locale. L'uso

[7] Comando modificato rispetto alla prima versione del plugin. Vedi *accesso remoto* e *login* del capitolo 3.

viene concesso solo se le credenziali sono corrette. Nel caso non esista una copia locale del database in questione, attiva la parte della *gestione accesso remoto*.

- **disconnect** : permette di rilasciare l'uso della base di dati precedentemente richiesta.
- **list.db** : stampa a video la lista delle basi di dati memorizzate nel nodo.
- **list.nodeset** : stampa a video la lista dei nodi che posseggono una replicazione di un database locale.
- **list.schema** : mostra lo schema logico relazione del database attualmente in uso.
- **send TOHOST MESSAGE** : invia all'host specificato il messaggio desiderato.
- **status** : visualizza lo stato attuale, se si è connessi o meno ad un database.
- **verbosity x** : imposta a x il livello di verbosity di PariDBMS.

DBMS : è la classe principale del plugin. Viene lanciato dal Core di PariPari quando si richiede il servizio DBMS e ha il compito di attivare tutte le parti richieste per il funzionamento del plugin. Deve inoltre saper memorizzare i messaggi ricevuti da Console.

DBMSConsole : classe che rappresenta la Console di PariDBMS. Per stampare a video sfrutta i metodi del Core.

DBMSPluginFirstRun : è il thread che viene lanciato da DBMS al primo avvio sul nodo del plugin. Deve inizializzare il database di supporto creando la struttura logica delle sue cinque tabelle attraverso HSQLDB.

MessageDelivery : è un thread che ha il compito smistare i messaggi che legge dalla coda globale di PariDBMS (*GlobalMessageQueue*) al blocco corretto del plugin. Per farlo legge il prefisso del messaggio e in base a quello sa dove deve aggiungerlo. I messaggi possono essere provenienti dalla console o dalla rete.

MessageQueue : è una classe che offre le funzionalità della classica coda, con la differenza che permette di distinguere la tipologia del messaggio (da Console o da Network). Esiste una sua istanza principale creata da DBMS che prende il nome di *GlobalMessageQueue* in cui vengono scritti tutti i messaggi provenienti dalla Console o da rete indifferentemente. Ogni parte del plugin possiede una sua MessageQueue privata.

NetworkMessageReceiver : è un thread. Usando le server socket, legge i messaggi provenienti dalla rete e li memorizza nella GlobalMessageQueue.

NetworkMessageSender : è un thread che permette l'invio di un messaggio ad altri nodi della rete PariPari. Deve ricevere, oltre al messaggio da spedire, anche l'indirizzo IP del destinatario.

NMGInterpreter : è il thread che crea il collegamento tra il blocco *gestione nodi* e la struttura principale del plugin. Una volta ricevuto un messaggio da MessageDelivery lo interpreta e lo smista nella parte corretta della sua struttura interna.

SQLInterpreter : è il thread che crea il collegamento tra il blocco *gestione database locali* e la struttura principale. Ha il compito di ricevere da MessageDelivery le stringhe SQL da eseguire sulle basi di dati e interpretarli nel modo corretto, smistando poi la query alla parte corretta della sua struttura interna.

SupportDB : è la classe che rappresenta il database di supporto descritto nel paragrado 2.4.1. Permette di mantenere permanentemente le informazioni necessarie al corretto funzionamento del plugin.

Update Manager : è il thread che crea il collegamento tra il blocco *gestione aggiornamento* e la struttura principale. Quando riceve un messaggio da MessageDelivery deve interpretarlo e smistarlo nella giusta maniera all'interno della sua struttura.

Nonostante non si siano forniti i dettagli a livello di codice, si è mostrata la struttura interna di PariDBMS. Per ciascuno dei blocchi e delle classi sarebbero molte le spiegazioni da fornire per una comprensione approfondita. Tuttavia non era questa l'intenzione; si voleva semplicemente, anche attraverso qualche immagine, mostrare a livello di flussi di comunicazione la configurazione di PariDBMS. Nella figura 2.3 si è scomposta la struttura principale della quale si è fatta poi una veloce carrellata delle classi che la compongono. Dei tre blocchi di base ho finora spiegato solo che funzioni svolgono senza esporre, come per la struttura principale, le loro componenti interne. Questa scelta è dettata dal fatto che nella parte dell'elaborato fin qui non è stato necessario e nell'ultima parte rimasta mi occuperò solo del lavoro che personalmente ho seguito. Come già più

volte scritto, il blocco *gestione aggiornamento* è il lavoro di J. Buriollo, *gestione nodi* di A. Cecchinato e *gestione database* di A. Costa. Nel prossimo capitolo ci si concentrerà delle nuove funzionalità di PariDBMS che assieme al collega Alberto Rizzi abbiamo progettato e realizzato.

3 INNOVAZIONI DI PARIDBMS

E’ arrivato il momento di approfondire il lavoro di cui mi sono occupato personalmente. La realizzazione di un DBMS distribuito è un progetto ambizioso che non può che svilupparsi per versioni successive le quali vanno a migliorare e ad aggiungere nuove funzionalità ad una solida base iniziale. Quando sono entrato in PariDBMS esistevano già i tre blocchi di base e la struttura principale descritti nel paragrafo 2.4. Le novità principali che si vogliono ora esporre consistono nel fornire la possibilità di accedere a database remoti e nel controllo degli utenti attraverso username e password. Le due parti prendono rispettivamente i nomi di Accesso Remoto e Login e sono stati progettati da me e da A. Rizzi insieme; ciò perché hanno parecchi aspetti in cui le due parti vanno ad intrecciarsi e richiedono le funzionalità dell’altro per poter essere efficaci. Tuttavia, dopo la fase di progettazione ognuno di noi ha seguito meglio la sua parte per quanto riguarda la stesura del codice. Un’attenzione maggiore in questa parte finale dell’elaborato sarà quindi posta alla parte dell’accesso remoto, senza comunque tralasciare il Login di cui verranno forniti gli aspetti principali.

Accesso remoto e Login sono sicuramente le innovazioni più importanti di questa seconda versione ma non vanno sottovalutati altri miglioramenti che riguardano l’efficienza del plugin. A tale proposito si sono eseguite due integrazioni del nostro plugin con altrettanti moduli di PariPari. Il primo, di cui mi sono occupato personalmente, riguarda l’integrazione con “DiESeL” che ha lo scopo di provare a limitare il numero di messaggi scambiati dai nodi di PariDBMS per monitorare il numero di nodi attivi del plugin, favorendo secondariamente l’abbassamento della congestione della rete. Si vedrà meglio nel paragrafo ad esso dedicato. Il secondo invece è stato curato da A. Rizzi e si tratta dell’integrazione con “Connectivity” e gestione delle socket, altro modulo di PariPari che a richiesta fornisce dei riferimenti di socket java. L’obiettivo era quello di limitare il numero di socket usate perché nella versione precedente queste venivano “sprecate” per una sola sessione di invio o ricezione, dopodiché venivano chiuse; si è quindi adottato un meccanismo che stabilisse per ogni coppia di mittente-destinatario un canale di comunicazione, mantenendo attiva una solo socket per tale coppia finché entrambi fossero rimasti attivi. Un piccolo paragrafo esporrà anche tale aspetto.

3.1 Accesso remoto

3.1.1 Descrizione del protocollo

Nella prima versione del plugin un utente poteva richiedere l'uso di un database solo se fosse stato contenuto direttamente nella memoria di massa del nodo locale. In tal caso era consentito l'accesso ai dati locali, altrimenti veniva stampato a video un messaggio d'errore per far notare all'utente che quei dati non erano in possesso del nodo in uso. Ovviamente è una grossa limitazione, in quanto è perfettamente lecito che un utente potrebbe volersi connettere ad una base di dati creata precedentemente su un altro host. E' una mancanza del plugin quella di non possedere un meccanismo che permetta di agli utenti di eseguire query da remoto. Tuttavia questo processo deve essere trasparente all'esterno: agli utilizzatori si deve dare l'impressione che il tutto funzioni come se si stesse utilizzando un DBMS tradizionale, senza distinzione tra accesso locale e accesso remoto ai dati. Si deve perciò progettare un metodo che renda possibile quanto appena scritto.

In precedenza si è mostrata la figura 2.2 dello schema a blocchi di PariDBMS. Si deve a questo punto osservare il blocco *gestione database* con dettaglio maggiore. Al suo interno esistono due parti: *gestione accesso remoto* e *HSQLDB*. Quest'ultimo è il software di cui si parlava in precedenza il quale gestisce in lettura/scrittura la memoria locale. In questo contesto non ci occupiamo di esso. L'attenzione verrà posta sul primo blocco interno.

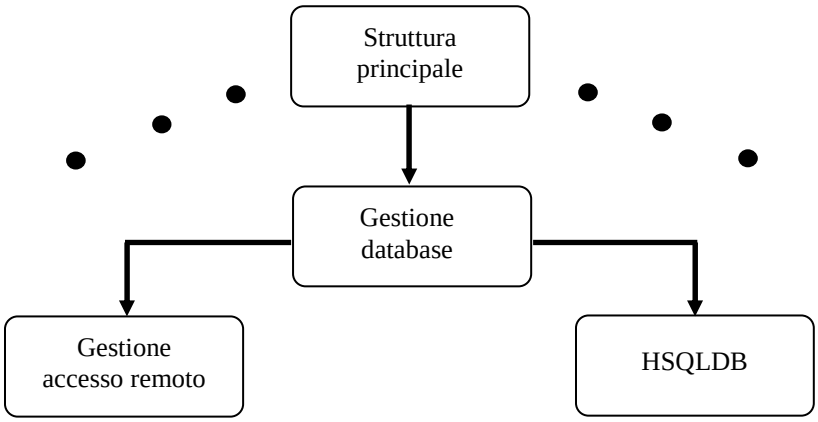

Figura 3.1 : Componenti interne del blocco Gestione database

Il blocco denominato *gestione accesso remoto* è quello che si occupa direttamente di offrire la funzionalità dell'accesso remoto. Come osservato dalla figura 3.1, l'acceso remoto rientra nel blocco *gestione database* perché, come per l'accesso locale, deve comunque garantire la sincronizzazione delle operazioni e seguire l'esecuzione delle query sui database, che in questo caso saranno posseduti da nodi remoti, ed essere capace di fornire i risultati al richiedente. Prima di descrivere tutti i dettagli dell'accesso remoto credo sia importante descrivere brevemente le classi che compongono il blocco *gestione database* visto che nel protocollo che verrà descritto successivamente saranno usati molti riferimenti a queste classi. Tale blocco contiene i seguenti oggetti:

DBSet : classe che gestisce un set di istanze di oggetti Database. E' la classe principale per quanto riguarda la gestione dei database e ogni volta che si ha bisogno di lavorare con una base di dati si deve utilizzare un suo metodo che, una volta fornito il nome del database restituisce l'oggetto Database corrispondente.

Database : è una classe di cui ogni sua istanza rappresenta un database locale del nodo. Possiede un importante campo che rappresenta il suo stato che può assumere uno dei seguenti valori. UPDATING, alla creazione dell'oggetto per indicare che necessità di essere allineato con le altre copie; READY, quando la base di dati è stata aggiornata ed è completamente disponibile per le richieste di accesso; PAUSED, per indicare che questo oggetto sta prendendo parte ad un processo di replicazione e quindi è momentaneamente indisponibile a fronte di richieste di accesso.

RemoteAccessProtocol : è il thread che gestisce il protocollo dell'accesso remoto. Viene lanciato nel momento in cui un utente chiede di utilizzare un database che il nodo si accorge di non possedere nella sua memoria locale. Garantisce anche la disconnessione dallo stesso database quando l'utente lo richiede.

RemoteQueryManager : è il thread che esegue la sessione di "invio query/ricezione risultato" sul client ed "esecuzione query/invio risultato" sul server. Viene lanciato dal thread RemoteAccessProtocol dopo aver stabilito una connessione valida tra il nodo richiedente l'accesso e quello disponibile a concederlo.

Transaction : ogni istanza di questa classe rappresenta una operazione da eseguire sulle basi di dati. Gli oggetti di tipo Transaction vengono aggiunti in una coda apposita chiamata TransactionQueue. Al suo interno contiene un thread privato chiamato TransactionSyncThread che ha il compito di eseguire la sincronizzazione di quella specifica operazione.

TransactionQueue : rappresenta la coda di operazioni da eseguire sui database. Viene sempre estratta da essa l'operazione con timestamp minore.

TransactionSynchronizer : è il thread che gestisce, insieme a TransactionSyncThread, il processo di sincronizzazione delle operazioni. Ha il compito, tra le altre, di inserire in TransactionQueue le operazioni che ha ricevuto dagli oggetti Database.

TransactionExecutor : è il thread che esegue fisicamente le transazioni estratte dalla TransactionQueue solo se il processo di sincronizzazione per queste è terminato correttamente. Attende finché l'operazione da estrarre non è pronta per essere processata.

Inoltre esiste una classe che non rientra in questo blocco, tuttavia è stato molto utile ai nostri scopi. *ConnectedTo* è una classe che permette di gestire tre campi privati: *dbname*, *host* e *status*. Questa classe rappresenta lo stato della connessione del nodo ad un particolare database il cui nome viene salvato con la variabile *dbname*, posseduto dal nodo con indirizzo *host*; *status* può essere NOT_CONNECTED se attualmente non si è connessi a nessun database, CONNECTED viceversa. Esistono poi altri due possibili valori che saranno compresi meglio tra poco, CONNECTING e DISCONNECTING, che rappresentano un nodo che tenta di stabilire una connessione ad una base di dati o che tenta di cancellare la connessione. Si può ora entrare nel dettaglio del protocollo dell'accesso remoto. Esistono tre sottofasi in questo processo. Il primo consiste nella ricerca di un nodo disponibile a concedere l'uso del database richiesto; quando se ne trova uno con queste caratteristiche termina la prima parte. La seconda fase è il tentativo di instaurare una connessione valida tra il nodo locale e quello restituito dalla ricerca della fase precedente. La terza, infine, rappresenta le sessioni di invio query e ricezione del risultato se si osserva dal punto di vista del calcolatore che ha richiesto l'accesso remoto, esecuzione query e invio risultato se si osserva dalla parte di chi ha concesso l'uso dei propri dati. In dettaglio, per le tre fasi, avvengono le seguenti azioni.

Fase 1

Un utente intenzionato ad accedere ad un particolare database, deve scrivere sulla Console il comando "\use DBNAME:USERNAME:PASSWORD". Tale messaggio viene inserito nella coda denominata *GlobalMessageQueue* (*GMQ*). Poiché inizia con il carattere speciale "\" *MessageDelivery* (*MD*) inserisce tale comando nella *MessageQueue* (*MQ*) privata del thread *CommandInterpreter* (*CI*) il quale, come visto in precedenza, deve interpretare i messaggi estratti dalla sua coda ed eseguire le azioni appropriate. C'è da precisare che *MD* inserisce questo specifico messaggio nella coda di *CI* solo se in quell'istante il nodo non è connesso a nessun database; il metodo *getStatus( )* della della classe *ConnectedTo* deve perciò restituire un valore diverso da CONNECTED. Quando *CI* legge il comando di uso, come prima cosa chiama un suo metodo privato (*cmdUse( )*) specializzato nell'interpretare questo messaggio. Tale metodo privato recupera dal messaggio il nome del database e username-password ad esso associato e, sfruttando la ricerca di risorse attraverso il modulo DHT, cerca gli indirizzi di quei nodi che hanno dichiarato di possedere un database di nome DBNAME creato da un utente con nome utente USERNAME. Il risultato di questa ricerca è un array di stringhe, denominato *ip*, proprio perché contenente indirizzi IP. Ovviamente se ha dimensione nulla vuol dire che non esiste nella rete quel database e quindi l'utente ha digitato male il nome della base di dati oppure il proprio username, altrimenti significa che qualche nodo ha precedentemente pubblicizzato quella risorsa. Tuttavia, quel database potrebbe anche essere contenuto direttamente nella memoria locale del calcolatore in uso e quindi il nodo stesso potrebbe averlo reclamizzato; per capire se devo lavorare in locale o in remoto, scorro in un ciclo *while* gli elementi di *ip* confrontando di volta in volta se l'indirizzo IP *i*-esimo coincide con l'indirizzo locale del nodo. Esco dal ciclo o perché ho terminato di scorrere tutti gli elementi di *ip* oppure un confronto durante l'iterazione *i*-esima ha dato esito positivo. Nel primo caso (variabile booleana *found = false*) significa che non ho una copia locale del database richiesto, altrimenti (*found = true*) vuol dire che il nodo locale possiede già il database richiesto. Quindi, *found = false* significa proseguire con la fase 2 del protocollo dell'accesso remoto; *found = true*, al contrario, indica che potrebbe non essere necessario continuare con la fase 2 perché i dati sono direttamente disponibili nella memoria di massa locale. Ho scritto di proposito "potrebbe" per il motivo che, nonostante la base di dati sia memorizzata in locale, nel caso non abbia lo stato READY (pronto all'uso) oppure il nodo stesso risulti già impegnato (stato del nodo diverso da NOT_CONNECTED) allora è necessario comunque proseguire con la fase successiva del protocollo dell'acceso remoto in quanto non si riesce a soddisfare l'utente lavorando in locale. Esiste anche il caso, eseguendo il codice previsto per la parte di accesso locale, in cui fallisce la verifica dei dati personali immessi

dall'utente[8]; potrebbe quindi non esistere la coppia username-password associato a quel database. Tralasciamo questo caso, il quale ovviamente farebbe terminare qualsiasi altra attività stampando a video un messaggio di errore. Per proseguire con la descrizione della prossima fase, ammettiamo quindi che si sia verificato uno di quei casi utili in cui deve iniziare la fase 2. *CI* chiama quindi il suo metodo privato *startRemAccPr( )* (START REMote ACCess PRotocol) passandogli come parametri *dbname, username, password, ip*. Il compito di questo metodo è creare una nuova istanza del thread *RemoteAccessProtocol* e lanciarlo come PariPariThread. A questo punto il thread *CI* si addormenta sulla chiamata sospensiva *wait( )* , togliendo di fatto la possibilità di scrivere nuovi comandi all'utente. Termina quindi la prima fase ed inizia la seconda.

Per chiarezza d'esposizione, distinguiamo con due termini classici il nodo che fa la richiesta di accesso remoto e quello che la riceve. Chiamiamo il primo *client* e il secondo *server*. Non devono trarre in inganno questi due termini associati ai due nodi, in quanto si è più volte scritto che PariPari è una rete peer-to-peer, che per definizione è un insieme di nodi equivalenti. Rinominarli in questo modo aiuta soltanto a non confondersi e sarà più semplice la lettura del protocollo. I due termini derivano dal ruolo assunto dai nodi nella connessione e perché ognuno di essi esegue codice sorgente diverso. La fase 1, quindi, viene svolta solamente dal calcolatore *client*, mentre il *server* ignora la sua esecuzione. Vorrei precisare inoltre che sia il *client* che il *server* lanciano esattamente gli stessi thread che verranno ora descritti, con la differenza che ciò avviene in momenti temporali diversi. Le parti eseguite dalle due parti saranno ora spiegate in maniera differenziata.

Fase 2 – lato client

La fase 2 comincia quando *CI* lancia nel *client* il thread *RemoteAccessProtocol* (*RAP*) e si addormenta indefinitamente. Poiché *RAP* è una classe presente in tutti i nodi di PariDBMS, deve essere in grado di verificare se appartiene all'host *client* o a quello *server* appena viene lanciata perché in base a questo esegue porzioni di codice differenti. La verifica, che in apparenza potrebbe sembrare complicata, è in realtà molto banale: se *RAP* appartiene al *client* significa che è stato *CI* a lanciarlo e i suoi campi privati (*dbname, username, password, ip*) contengono valori validi; al contrario, la presenza di tutti valori nulli in questi campi vuol dire che l'istanza di *RAP* in esecuzione sta svolgendo la sua attività sulla macchina *server*.

Per quanto riguarda il lato *client*, per prima cosa viene eseguito il metodo privato *requestRemoteUse( ).* La sua esecuzione prevede di impostare subito a CONNECTING lo stato del nodo (con il metodo *setStatus( )* di *ConnectedTo*). Ciò serve per bloccare la console in quanto

[8] Il modo in cui funziona la verifica credenziali di accesso sarà spiegato nel paragrafo *Login*.

mentre lo stato del nodo è CONNECTING non si può aggiungere nella coda di *CI* nessun tipo di comando. Successivamente si entra all'interno di un ciclo *while*; qui, si invia al primo indirizzo IP contenuto in *ip* la richiesta di connessione. Tutti i messaggi destinati al thread *RAP* (che esegue la fase 2) cominciano proprio con i caratteri "rap". Infatti, la richiesta inviata dal *client* è: "rap use remote DBNAME:USERNAME:PASSWORD". Dopo aver inviato la richiesta, *RAP* si addormenta in attesa della risposta con un timeout *T* della durata di dieci secondi. La risposta che si attende è la disponibilità ad accettare la connessione o un rifiuto della stessa, quindi "rap acc" nel primo caso o "rap ref" nel secondo (ACCepted o REFused). *RAP* si risveglia o perché è scaduto il timeout o perché ha ricevuto uno dei precedenti messaggi; se il motivo del risveglio è la scadenza del temporizzatore ciò sarebbe sinonimo dell'indisponibilità del *server* oppure che esiste un problema a livello di rete per cui il messaggio inviato potrebbe essersi perso o il livello di congestione è elevato al punto da non permettere uno scambio di messaggi tra *client* e *server* in dieci secondi. In entrambi i casi si decide di tentare di stabilire una connessione con un altro nodo, reiterando il ciclo *while* dopo aver incrementato l'indice di riferimento di *ip*. Altrimenti, se il risveglio è dovuto alla ricezione di uno dei due messaggi attesi esistono ovviamente due possibilità: se è "rap ref" allora si agisce come nel caso di scadenza del timeout e si reitera il ciclo; viceversa se il messaggio ricevuto è "rap acc" significa che il *server* contattato si dichiara disponibile. Esiste poi un altro caso, cioè se il messaggio è "rap user ref"; ciò succede quando il *server* verifica le credenziali di accesso e trova un errore. In questo caso si pone lo stato della connessione a NOT_CONNECTED e viene terminata l'esecuzione di *RAP*, poi viene risvegliato *CI*, che era rimasto in attesa sul *wait( )*, attraverso la chiamata al metodo *notify( )*. Per procedere consideriamo l'unico caso positivo. Quando il *client* riceve il messaggio "rap acc" è consapevole non solo di aver trovato un *server* disponibile ma anche che quest'ultimo ha impostato a CONNECTED il suo stato. Si deve perciò impostare allo stesso valore anche lo stato del *client* non appena si è ricevuta una risposta positiva. A questo punto si può risvegliare *CI* (con la solita chiamata *notify( )*) e far partire la terza fase. Si invoca quindi il metodo privato di *RAP startRQM( )* che ha il compito di creare una nuova istanza della classe *RemoteQueryManager* (*RQM*) e lanciarlo come PariPariThread dopodiché *RAP* si addormenta indefinitamente in attesa di essere risvegliato o da *CI*[9] o da *RQM* stesso. Si vedrà meglio più avanti.

Fase 2 – lato server

Mentre nel *client* si è già conclusa la fase 1 ed è in esecuzione la fase 2, non esiste ancora nessun nodo che abbia il ruolo di *server*. Quando nella seconda fase il *client* invia ad un nodo la richiesta "rap use remote DBNAME:USERNAME:PASSWORD" il nodo che risponde è stato

[9] Vedi *chiusura dell'accesso remoto*.

impropriamente chiamato *server*: in realtà è un *candidato server*. Per poter fungere da *server* deve essere disponibile a cedere l'uso del database richiesto.

Quando un nodo riceve dalla rete un messaggio, la classe *NetworkMessageReceiver* (*NMR*) lo aggiunge nella coda denominata *GMQ*, dalla quale *MD* li estrae per inoltrarli alla parte corretta del plugin (vedi figura 2.3). Nel momento in cui *MD* legge un messaggio che inizia con "rap" sa che il suo destinatario è la classe *RAP*, il responsabile della seconda fase del protocollo dell'accesso remoto. Inoltre se il messaggio inizia con "rap use" allora vuol dire che è arrivata la richiesta di accesso remoto da un *client* e quindi si deve far partire proprio il thread *RAP* perché possa iniziare anche sul nodo locale (il *server* del protocollo) la seconda fase. *MD*, una volta ricevuto un messaggio che inizi con "rap use", crea quindi una nuova istanza del thread *RAP* passandogli nel costruttore tutti valori nulli, successivamente lo lancia come un PariPariThread e poi aggiunge nella sua coda privata il messaggio ricevuto per esteso, contenente anche il DBNAME, USERNAME e PASSWORD.

Parte quindi l'esecuzione di *RAP*. Innanzitutto deve rendersi conto di appartenere all'host con il ruolo di *server* e a tale scopo verifica che tutti i suoi parametri privati siano nulli. Fatto ciò, estrae dalla propria coda il primo (e anche l'unico per il momento) messaggio presente; poiché inizia con "rap use" chiama il suo metodo privato *replyRemoteUse( )* passandogli anche il messaggio per esteso. Questo metodo, per prima cosa, recupera dal messaggio, attraverso un parsing, il nome della base di dati, username e password associati e ha il compito di verificare la disponibilità o meno che sul database richiesto possano essere eseguite query da un host remoto. La verifica si sviluppa in più fasi: innanzitutto deve esistere quel database sul nodo, poi il nome utente e la password associati devono essere validi, lo stato del nodo deve essere NOT_CONNECTED ed infine il database deve avere stato READY. Solo se si avverano queste condizioni allora si risponde con "rap acc" altrimenti con "rap ref". Nel caso fallisse la verifica delle credenziali la risposta sarebbe "rap user ref" la quale farebbe terminare ogni attività anche sul *client*. Come al solito, per poter proseguire con la descrizione del protocollo, consideriamo il caso in cui le verifiche di disponibilità abbiano tutte dato esito positivi. In questo caso, dopo aver rispedito al *client* la risposta "rap acc", il thread invoca il metodo privato *startRQM( )* , il quale crea una nuova istanza del thread *RemoteQueryManager* (*RQM*) e lancia la sua esecuzione. Successivamente *RAP* si mette in un'attesa indefinita sulla chiamata sospensiva *wait( )* in attesa che un evento lo risvegli. Termina così la seconda fase anche anche sul lato *server* e con l'inizio dell'esecuzione di *RQM* si avvia la terza, che verrà descritta tra poco.

Fase 3 – lato client

Questa fase ha inizio non appena *RAP* invoca il metodo *notify( )* sul thread *CI* e si addormenta sul *wait( )* dopo aver lanciato il thread *RQM*. Come già scritto in precedenza, la fase 3 per quanto riguarda il lato *client* è quella che gestisce l'invio delle query e attende la ricezione del risultato dopo la sua esecuzione sul *server*. Risvegliare *CI* è un evento necessario affinché, una volta stabilita la connessione con l'host remoto nella seconda fase, l'utente abbia finalmente la possibilità di scrivere sulla console le query da far eseguire sul database desiderato. Quando *RQM* inizia la sua attività, rimane subito un attesa di ricevere nella sua coda una query da inviare. I messaggi destinati a questo thread devono cominciare con il prefisso "rqm". Ammettiamo ora che l'utente desideri scrivere sulla console la sua prima operazione. Non deve aggiungere nessun prefisso speciale, basta scrivere semplicemente la query che intende eseguire sulla base di dati. Questo messaggio viene inserito da *DBMS* nella *GMQ*. *MD* quando lo estrae da qui verifica che è un messaggio proveniente dalla console, senza il prefisso speciale "\" che identifica i comandi destinati a *CI*, e lo stato del nodo è CONNECTED. Controlla infine che la connessione non sia locale, chiamando il metodo *isLocalConnected( )* della classe *ConnectedTo* che in questo caso restituisce il booleano *false*. Si accorge quindi che siamo nella fase 3 del protocollo dell'accesso remoto e inoltra la query a *RQM*. Quest'ultimo, che era rimasto in attesa, si risveglia non appena riceve la query da *MD*, antepone al messaggio "rqm sql remote" e spedisce, con il metodo *send( )* di *NetworkMessageSender*, la query così costruita; subito dopo, si addormenta con un timeout *Tc1* (Timeout Client 1) della durata di dieci secondi in attesa di una risposta da parte del *server* di query ricevuta. Il suo risveglio è dovuto o alla scadenza del timeout oppure alla ricezione di un ack[10]. Consideriamo per ora l'ipotesi migliore, ovvero che entro i dieci secondi previsti il *server* confermi la ricezione della query; il caso delle scadenze dei vari timeout nella terza fase sarà spiegato nel paragrafo *chiusura dell'accesso remoto*. Abbiamo quindi assunto che *RQM* abbia ricevuto l'ack che ha il seguente formato: "rqm ackq" (ACKnowledge Query). Una volta letto questo messaggio, il *client* sa che il *server* si è dichiarato pronto ad eseguire la query inviata e che attende solo il segnale di partenza per aggiungere tale operazione nella sua *TransactionQueue* (*TQ*). *RQM* invia quindi il messaggio "rqm start" che ha proprio questo significato, poi si addormenta con un timeout *Tc2* (Timeout Client 2) di durata maggiore rispetto al primo, ovvero di due minuti. Questo lasso di tempo rappresenta la durata massima dell'attesa del *client* affinché il *server* abbia il tempo di eseguire la query e rispedire il risultato così ottenuto. Sempre considerando che non scada il timeout, il risultato risveglia *RQM*. Il messaggio ricevuto è "rqm result RISULTATO" dove l'ultima parte è la rappresentazione testuale della vista della tabella ottenuta dopo l'esecuzione dell'operazione sulla base di dati. Si può

[10] Acknowledge. Rappresenta la risposta di avvenuta ricezione di un'informazione completa.

a questo punto stampare a video il risultato ottenuto. Una volta fatto ciò, *RQM* invia la conferma della corretta ricezione spedendo al *server* il messaggio “rqm ackr” e cade nuovamente in un attesa indefinita sul *wait( )* aspettando nuove query. Nel caso l’utente desiderasse eseguirne di nuove, sarebbe sufficiente scriverle sulla console e verrebbero svolte nuovamente tutte le attività della terza fase appena descritte. Nel caso invece si voglia terminare l’uso di quella base di dati è necessario scrivere un comando specifico che verrà spiegato nel paragrafo *chiusura dell’accesso remoto*.

Fase 3 – lato server

L’inizio della terza fase sul *server* avviene, come nel *client*, nel momento in cui *RAP* lancia il thread *RQM* e si mette in attesa invocando il metodo *wait( )*. Appena *RQM* inizia la sua esecuzione, rimane in attesa finché non riceve un messaggio nella sua coda privata. Nel frattempo, nel *client* l’utente potrebbe aver deciso di scrivere sulla console una query da far eseguire in remoto (vedi *Fase 3 – lato client*). Quando questo accade il *server* riceve attraverso *NMR* il messaggio “rqm sql remote QUERY” e lo aggiunge in *GMQ* da dove *MD* lo estrae e, vedendo il prefisso “rqm”, lo inserisce nella coda di *RQM*. Il thread *RQM* si risveglia, legge il messaggio nella sua coda e dal formato del messaggio capisce di aver ricevuto una operazione che si richiede venga eseguita su un proprio database. Recupera dal messaggio la parte relativa alla query e la memorizza in un suo campo privato; non è ancora il momento di far partire la transazione in quanto non è ancora arrivato il segnale di partenza per l’operazione. Dobbiamo prima inviare al *client* l’ack “rqm ackq” ed aspettare con un timeout *Ts1* (Timeout Server 1) della durata di dieci secondi il messaggio di start. Ipotizzando che il *client* spedisca in tempo il messaggio “rqm start”, la sua ricezione mette fine all’attesa del *server*. Quindi, possiamo aggiungere la query nella coda di *SQLInterpreter* (*SQLI*) che ha il compito di interfacciarsi con il blocco *gestione database* di cui si è parlato nel paragrafo finale del precedente capitolo. Questo thread, leggendo il prefisso dei messaggi presenti nella sua coda, riesce a distinguere se l’operazione è arrivata direttamente dalla console (quindi generata in locale) oppure è arrivata per accesso remoto; poiché in questo caso il messaggio letto è “sql remote QUERY” esegue la parte di codice che permette di gestire query da remoto. Viene tolto il prefisso “sql remote” e la parte restante, cioè la query vera e propria, viene aggiunta nella coda privata del database sul quale si vuole eseguire l’operazione. Ci si aiuta quindi con la classe *DBSet* la quale per inoltrare la query alla base di dati richiesta necessita solo di conoscere il suo nome e il nome utente associato (informazioni che si conoscono già dalla chiamata al metodo *getDBName( )* di *ConnectedTo*). La query arriva così al database giusto, il quale la inoltra immediatamente al coda di *TransactionSynchronizer* (*TS*). E’ un altro thread descritto nel blocco *gestione database* che ha il compito di aggiungere le transazione nella *TransactionQueue* (*TQ*) da dove il thread

TransactionExecutor (*TE*) le estrae per eseguirle fisicamente. *TS* inoltre si occupa di sincronizzare le operazioni, come già descritto nel capitolo precedente. Le operazioni che arrivano *TE* sono di due tipi: reperimento (cioè un SELECT) oppure di aggiornamento (UPDATE, INSERT, DELETE). Nel primo caso, dopo aver processato la transazione si deve mostrare la vista così ottenuta dalle condizioni di selezione; nella seconda eventualità, invece, è sufficiente notificare il successo o meno dell'esecuzione di quella operazione. In base a quanto appena scritto, *TE* costruisce una stringa, denominata RESULT, contenente una rappresentazione della vista della tabella ottenuta dalle condizioni di SELECT oppure una stringa che indichi l'andamento dell'esecuzione dell'operazione di modifica. Fatto ciò, *TE* aggiunge nella coda di *RQM* il messaggio "rqm result RESULT" con tipologia da Console (per distinguere il messaggio con identico prefisso che riceve il *client* da Network). A questo punto *RQM* invia tale risultato al *client* e si mette in attesa che questi gli comunichi la sua corretta lettura. Il timeout di questa attesa *Ts2* (Timeout Server 2) è della durata di dieci secondi. Ancora una volta prendiamo in esame l'ipotesi che non venga superato il limite massimo imposto. Il messaggio ricevuto è quindi "rqm ackr" che segna la fine di questa sessione di query. Quando *RQM* riceve questo ack, si addormenta nuovamente sulla chiamata sospensiva *wait( )* in attesa di ricevere nuove query da eseguire; se questo avvenisse, verrebbero nuovamente svolte le azioni appena esposte. Potrebbe tuttavia non arrivare più niente, nel caso l'utente non desiderasse più lavorare con quel database: questa eventualità sarà spiegata nel paragrafo *chiusura dell'accesso remoto*.

3.1.2 Gestione di eventi imprevisti

Nella descrizione appena proposta, si è sempre assunto che si realizzassero le condizioni migliori per proseguire di volta in volta alla fase successiva del protocollo. Tuttavia, non sempre le cose funzionano così bene. I messaggi potrebbero perdersi nella rete oppure potrebbero ritardare di quel tanto che basta perché scada un timeout; potrebbero essere spediti a destinatari errati. Ancora, uno tra il *client* o il *server* potrebbe bruscamente interrompere la sua esecuzione durante una delle tre fasi. Si devono considerare parecchi casi limite. In questo paragrafo, a differenza del precedente, verrà sempre preso in esame l'eventualità peggiore per spiegare i meccanismi con cui il protocollo dell'accesso remoto riesca gestire anche questi casi.

Nella *fase 2 – lato client* conosco una lista di nodi da provare a contattare per tentare un accesso remoto ad un database. L'azione che *RAP* esegue immediatamente è quella di inviare al primo *candidato server* il messaggio "rap use remote..." e poi sospendersi rimanendo in attesa con timeout di dieci secondi; se nel frattempo non ricevo alcuna risposta, al mio risveglio mi accorgo

che è scaduto il timeout perché l'invocazione del metodo *isEmpty( )* mi restituisce valore *true*. Quando questo accade posso ipotizzare che la causa della mancata risposta sia un problema a livello di rete: potrebbe essere congestionata oppure il messaggio potrebbe essersi perso da qualche parte. In tutti e due i casi la scelta migliore è riprovare con il nodo successivo della lista di *candidati server* in quanto nel primo caso la congestione potrebbe non risolversi in breve tempo, rischiando di compromettere anche le scadenze dei timeout successivi, nel secondo caso invece, il collegamento tra i due host potrebbe non essere affidabile.

La *fase 2 – lato server* non prevede particolari casi imprevisti. Riceve da un nodo la richiesta di accesso remoto e, se è possibile, concede l'uso di un proprio database al richiedente.

Per quanto riguarda la fase 3, su entrambi i fronti della connessione potrebbero accadere degli eventi inattesi. Il *client* è quello che invia le query e attende i risultati. Mentre è in esecuzione il thread *RQM*, *RAP* si è sospeso sulla chiamata *wait( )* e attende di ricevere una notifica per risvegliarsi. Questa potrebbe arrivare da *CI* o da *RQM*; il primo caso riguarda la chiusura dell'accesso remoto e per ora lo tralasciamo, il secondo invece è sinonimo di una scadenza di un timeout in *RQM*. Infatti, si è descritto precedentemente che il *client* durante la sua attività possiede due momenti in cui attende con un certo timeout, come si può osservare dalla figura 3.2. Il primo, che è stato chiamato *Tc1*, rappresenta l'attesa dell'ack dal *server* per la corretta ricezione della query; il secondo, *Tc2*, è l'attesa del risultato dell'esecuzione della query. Nel caso, non menzionato nel precedente paragrafo, che scada uno di questi timeout allora viene impostato a *true* la variabile booleana *endTime*. La modifica di questa variabile impedisce la ripetizione del ciclo *while* dentro al quale si sviluppa il codice di *RQM*. Concettualmente, la scadenza di un temporizzatore significa che si è verificato un problema. Bisogna, a questo punto, decidere se adottare un atteggiamento più propenso a credere che i messaggi possano perdersi o ritardare nella rete oppure che i nodi di PariPari si disconnettano improvvisamente. Poiché, come scritto nell'introduzione, stiamo lavorando in una rete peer-to-peer con nodi che possono interrompersi improvvisamente e senza preavviso, si è deciso di fidarsi completamente della rete; quando si verifica una scadenza, si assumerà sempre che il nodo dall'altra parte della connessione non sia più attivo. Quando in *RQM* viene interrotto il ciclo *while*, il *client* deve perciò tentare di far eseguire la query ad un altro nodo; deve prima di tutto stabilire una connessione valida con un altro *server*, ritornando così alla fase 2 del protocollo. Aggiunge nuovamente la query attuale nella sua coda e poi risveglia *RAP* con l'invocazione del metodo *notify( )* passando anche la sua sigla "rqm" in modo che *RAP* al suo risveglio sappia chi sia il risvegliante. *RAP* riprende quindi la sua esecuzione e come prima azione ferma l'esecuzione del thread *RQM* con l'invocazione del metodo *stop( )* il quale termina (ma non uccide) questo thread; poi incrementa l'indice dell'array di indirizzi IP restituiti dalla prima fase ed

esegue nuovamente le attività della *fase 2 – lato client*. Quando un altro nodo si dichiarerà *server* per la connessione verrà lanciato di nuovo il thread *RQM*, che era stato fermato, perché possa tentare sul nuovo *server* l'esecuzione dell'ultima query fallita.

Come il lato *client*, anche la *fase 3 – lato server* potrebbe inciampare in scadenza di timeout. Riprendiamo sempre la descrizione dal momento in cui sul calcolatore *server*, il metodo privato *replyRemoteUse( )* ha lanciato il thread *RQM* cadendo in attesa indefinita sul *wait( )*. Ammettiamo che sia scaduto uno dei due timeout del server. *Ts1* e *Ts2* sono rispettivamente la massima durata dell'attesa del segnale di start e dell'ack di conferma ricezione risultato. Lo sforamento di uno di questi temporizzatori determina, similmente a ciò che accade dalla parte *client*, il cambiamento del valore della variabile booleana *endTime*, con la conseguenza dell'interruzione del ciclo *while* nel quale è inserito il codice principale. Nelle ultime righe di questo ciclo, prima di interrompersi, esegue le ultime due istruzioni: impostare lo stato del nodo a NOT_CONNECTED e risvegliare *RAP* dalla sua attesa, passandogli anche le inizia "rqm" perché questi capisca chi sia il risvegliante. La decisione di dichiararsi disconnesso deriva dal fatto che le scadenze di timeout sono per noi sinonimo che dall'altra parte della connessione il calcolatore non è più in funzione. Appena *RAP* si risveglia, prima ferma *RQM* con la chiamata a *stop( )* e subito dopo lo uccide con *kill( )*. Ormai *RQM* non ci serve più. Successivamente si chiede a *MD* (il detentore del riferimento di *RAP*) di uccidere anche questo thread, perché si è bruscamente interrotto l'accesso remoto.

Anche se ci siamo completamente fidati della rete, potrebbe tuttavia accadere che, per esempio per un errore a livello di bit, un messaggio destinato ad un host venga erroneamente spedito ad un altro. Tale eventualità è gestita in questo modo: ogni volta che si attende una risposta da un nodo specifico, all'arrivo di un messaggio si controlla per sicurezza che il suo mittente corrisponda esattamente con il nodo da cui attendiamo la notifica; se coincidono, come dovrebbe sempre accadere, allora si procede ad eseguire le azioni appropriate, altrimenti non vengono svolte attività. Ad un certo punto scadrà un timeout e si rientrerà in uno delle casistiche precedenti.

3.1.3 Chiusura dell'accesso remoto

Quando un utente non desidera più eseguire query su un database remoto deve far partire una procedura che permetta ad entrambe le parti di accordarsi sul rilascio della risorsa. Finché è in svolgimento il protocollo dell'accesso remoto, sia il *client* che il *server* hanno stato CONNECTED e non possono svolgere nessun accesso né sui dati locali, tantomeno su quelli remoti. La procedura di chiusura dell'accesso permette proprio di liberare i due nodi dalla connessione. Come per l'accesso, anche l'abbandono della connessione deve partire dal *client* e può avvenire in una

qualsiasi delle tre fasi. Il *server*, invece, non può autonomamente prendere la decisione di terminare il rapporto tranne che per una scadenza di un suo timeout.

Iniziamo quindi a descrivere ciò che accade sul lato *client*. L'utente intenzionato a chiudere il protocollo deve scrivere sulla console il comando "\disconnect", il quale, dopo essere stato letto da *MD*, viene inserito nella coda di *CI*. Questo thread, una volta interpretato il comando, chiama il suo metodo privato *cmdDisconnect( )* che, appena invocato, controlla se il nodo è connesso in locale o in remoto. Nel primo caso, imposta semplicemente a NOT_CONNECTED lo stato del nodo; nel secondo, quello che ci interessa, chiama il metodo *notify( )* sul thread *RAP* lanciato precedentemente nel momento della richiesta dell'accesso remoto e attende lui stesso; gli passa anche come parametro le iniziali "ci" per indicare che è lui il risvegliante. *RAP* stava dormendo su un *wait( )* o perché aspettava la risposta di accettazione o rifiuto del nodo contattato o perché ha lanciato correttamente il thread *RQM* in attesa dell'arrivo di un evento. Il *notify( )* invocato da *CI* risveglia *RAP* che leggendo anche il nome del risvegliante ("ci") capisce che il motivo per cui è stato chiamato in causa non è un problema verificatosi per scadenza di un timeout nella terza fase ma è la volontà dell'utente di terminare l'accesso remoto. Esegue quindi il suo metodo privato *requestEndRemoteUse( )* . Qui, lo stato del nodo viene temporaneamente impostato a DISCONNECTING, poi viene inviata la notifica al *server* e senza aspettare risposte di nessun tipo si soddisfa la volontà dell'utente, impostando lo stato del nodo a NOT_CONNECTED. Non è necessario aspettare un ack da parte del *server* in quanto, anche se il messaggio si perdesse e non arrivasse al mittente, la scadenza di un temporizzatore garantirebbe la corretta uscita anche sull'altro versante della connessione. Dopo aver impostato a disconnesso lo stato, *RAP* uccide eventualmente il thread *RQM* se si è già raggiunta la terza fase; in ogni caso, successivamente, chiama il *notify( )* di *CI* in modo che questi riprenda l'attività. Appena si risveglia, *CI* uccide con il metodo *kill( )* il thread *RAP* e termina così sul *client* ogni attività riguardante l'accesso remoto. Tutti i thread relativi ad esso sono stati uccisi e gli altri sono di nuovo operativi.

Dall'altra parte, sul *server*, l'arrivo di un messaggio di terminazione della connessione viene come al solito letto da *MD*. Essendo destinato al thread *RAP*, il messaggio è "rap end use"; *MD* lo smista quindi a *RAP* il quale, quando lo legge, prima ferma il thread *RQM* e poi lo uccide completamente. Dopodiché chiama il metodo privato *replyEndRemoteUse( )* che prevede semplicemente ad impostare lo stato a NOT_CONNECTED, senza inviare alcun messaggio al *client* perché si è consapevoli che di là si è già effettuata la disconnessione. A questo punto si chiede, attraverso il metodo *killRap( )* , a *MD* di uccidere *RAP* per far sì che tutti i thread relativi all'accesso remoto vengano correttamente terminati anche sul *server* della connessione.

Ora, l'unico modo per accedere di nuovo ad un database è scrivere sulla console il comando "\use DBNAME:USERNAME:PASSWORD". Nel caso la base di dati richiesta non fosse presente nella memoria locale del nodo, ricomincerebbe nuovamente il protocollo dell'accesso remoto dalla fase 1 descritto nel paragrafo 3.1.1.

Vediamo ora una rappresentazione grafica del protocollo, prima dal punto di vista di RAP e poi di RQM.

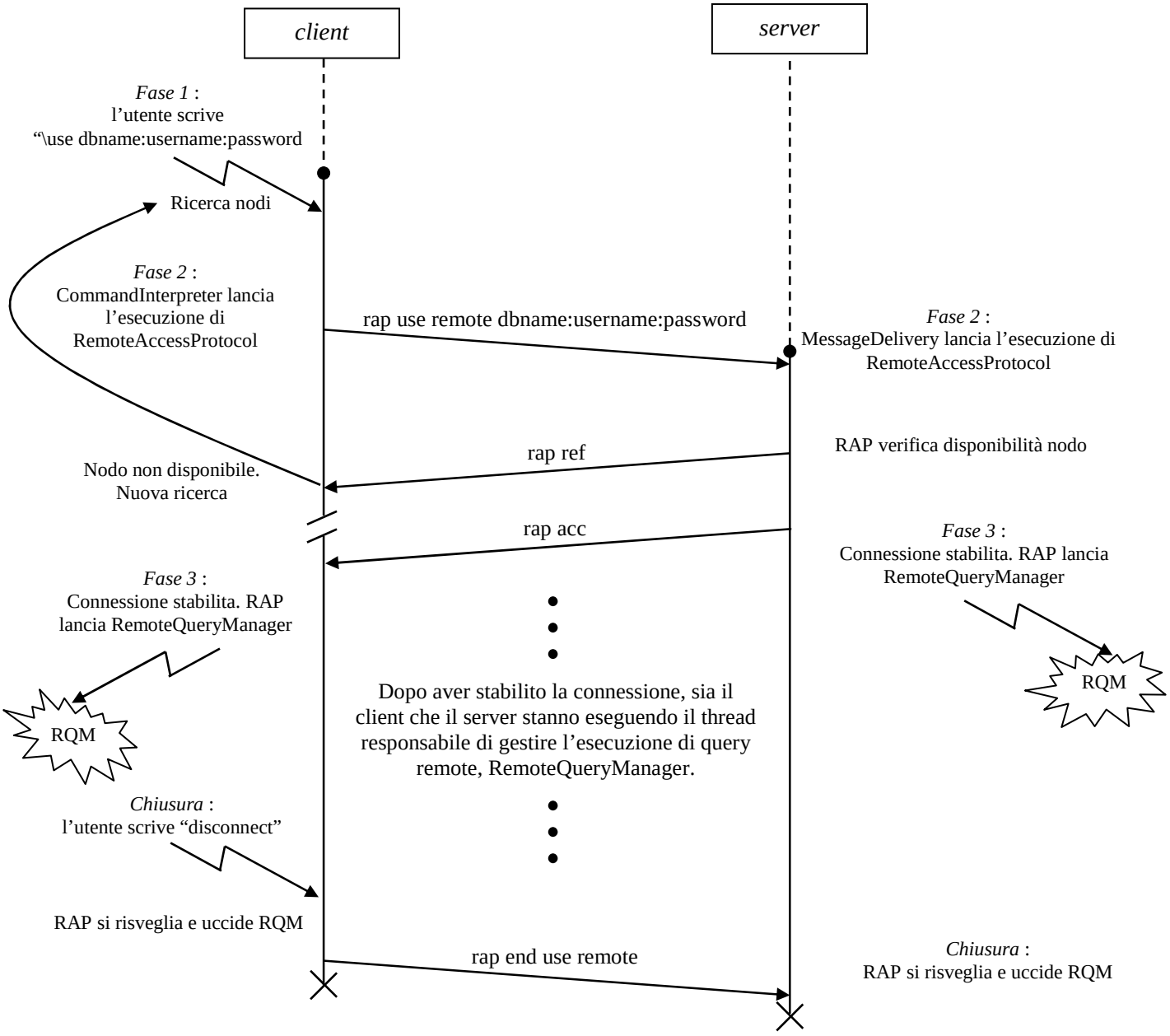

Figura 3.2 : Le tre fasi del protocollo dell'accesso remoto e la sua chiusura, osservata dal punto di vista di RemoteAccessProtocol.

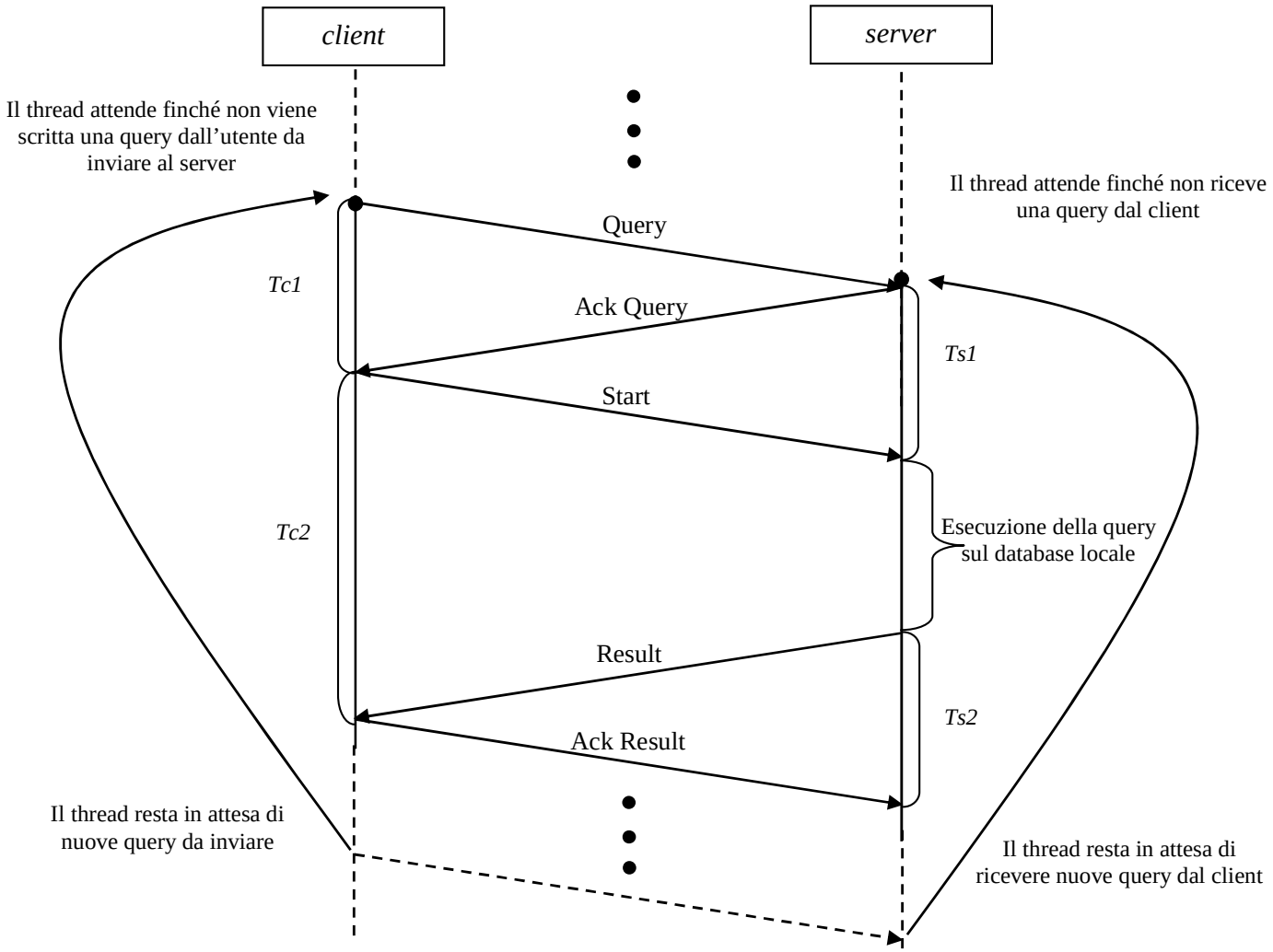

Figura 3.3 : Fasi dell'esecuzione di una query sul server, gestiste dal thread RemoteQueryManager. Rappresenta la parte lasciata in sospeso nella figura 3.2.

3.2 Login

Il meccanismo denominato *login* è l'oggetto della tesi di A. Rizzi. Tuttavia, questa parte si deve integrare perfettamente con quella dell'accesso ai database, poiché viene permesso l'uso dei dati solo dopo aver verificato che le credenziali di accesso siano validi.

Nel database di supporto (*SupportDB)* che ogni nodo di PariDBMS possiede, è stata aggiunta la tabella USERS contenente tre campi: *dbname*, *username* e *password*. Quando l'utente desidera creare sul nodo un nuovo database, deve scegliere, oltre al nome della base di dati, anche i suoi dati personali, ovvero username e password. Le azioni che il plugin esegue quando riceve la richiesta di creazione servono per garantire che non esistano, prima di tutto, utenti con lo stesso username e, in

secondo luogo, che uno specifico utente non crei, anche se su host diversi, database omonimi. Queste verifiche sono possibili grazie al modulo DHT di PariPari. Basta una semplice ricerca usando come chiavi *username* oppure *dbname+username* per controllare esattamente che non si verifichino le eventualità non permesse. Nel caso in cui le verifiche abbiano esito positivo, si può procedere con la creazione sul nodo locale del database; si deve memorizzare la sua struttura logico-relazionale nelle tabelle DATABASES, TABLES e FIELDS, mentre per quanto riguarda i dati personali, va aggiunta una nuova tupla nella tabella USERS inserendo il nome della base di dati, username e password dell'utente. Oltre alle precedenti verifiche, essendo la chiave primaria di USERS la coppia di attributi *dbname*, *username*, siamo ancor più certi che non possano esistere database omonimi abbinati allo stesso nome utente.

Il controllo dei dati personali garantisce uno dei requisiti principali di un qualsiasi DBMS: la privatezza dei dati. Ogni utente, identificato mediante *username* e *password*, ha diritto ad accedere in lettura o in scrittura solo ai database creati da lui stesso. Quando viene scritto sulla console il comando "\use DBNAME:USERNAME:PASSWORD", prima di concedere l'uso della base di dati, si deve verificare che al database di nome DBNAME sia associata la coppia *username*, *password*. Si deve perciò consultare la tabella USERS, eseguendo su di essa una operazione di selezione per capire se esiste o meno la tupla con i tre valori digitati dall'utente. In caso positivo si può permette il suo utilizzo, altrimenti l'accesso viene negato. Si adopera lo stesso meccanismo anche nel caso di un accesso remoto, con la differenza che sarà il *server* a verificare l'esistenza o meno nel proprio *SupportDB* di una tupla così composta nella tabella USERS.

Per aggirare il problema dell'indisponibilità, PariDBMS replica quei dati che possiedono poche copie nella rete. Quindi, quando si recuperano le informazioni riguardanti un database da replicare, ora vengono anche aggiunte quelle relative alla tabella USERS, per non perdere anche nelle copie la possibilità di verificare le autorizzazioni di coloro che fanno la richiesta di accesso al database in questione.

In questo elaborato è sufficiente questa spiegazione ridotta del meccanismo di *login*. Per i dettagli si consiglia la lettura della tesi di A. Rizzi.

3.3 Integrazione con il modulo DiESeL

Ogni nodo di PariDBMS, deve monitorare il numero di copie attive dei database locali. Questa necessità è stata spiegata nel processo di replicazione. I dati che corrono il rischio di indisponibilità devono essere replicati in altri nodi per aggirare questo problema; ogni calcolatore

deve perciò tenere sotto controllo quante replicazioni possiede la base di dati in pericolo; se si scende sotto al valore di soglia, si deve avviare la replicazione. Nella prima versione di PariDBMS, il sistema di monitoraggio era molto semplice: esisteva un thread, chiamato *NodePing*, il quale ogni cinque minuti inviava a tutti i nodi presenti nella *NodeSet* un messaggio di saluto IAMALIVE (I'm alive). Ricordiamo brevemente che la *NodeSet* è una lista nella quale sono presenti degli oggetti di tipo *Node*, ciascuno dei quali rappresenta un nodo che possiede una replicazione di un database locale. Quando *NMGInterpreter* riceveva il messaggio di saluto da un calcolatore, aggiornava il temporizzatore riferito ad esso all'istante di clock attuale del sistema. Un altro thread tuttora presente, *NodeController*, aveva il compito di verificare, sempre ogni cinque minuti, che i temporizzatori relativi ad ogni oggetto presente nella *NodeSet* fossero stati resettati al massimo dieci minuti fa. Per farlo, è sufficiente eseguire la differenza tra il clock attuale e il valore impresso da *NMGInterpreter* alla ricezione del messaggio di saluto; se questa differenza era superiore a dieci minuti, si assumeva che il nodo in questione si fosse scollegato dalla rete e si provvedeva a cancellare l'oggetto *Node* che lo rappresenta nella *NodeSet*. Questo meccanismo di monitoraggio delle copie attive, per quanto semplice ed efficace, ha un grave difetto di efficienza. Ad ogni esecuzione di *NodePing* vengono inviati tanti messaggi nella rete quanti sono i nodi rappresentati nella *NodeSet*. Con *n* nodi presenti in questa lista, l'ordine di grandezza del numero di messaggi scambiati da tutti i nodi di PariDBMS è $O(n^2)$, perché, a parole, i messaggi sono spediti da "tutti verso tutti".

Il modulo DiESeL di PariPari risulta molto utile a tale scopo. Per integrare il proprio plugin con DiESeL è necessario includere il modulo in una cartella (libreria) e implementare semplicemente una sua interfaccia, *IServerLayer*. La classe che in PariDBMS implementa tale interfaccia è stata chiamata *DBMSServerLayer*. Consideriamo questo modulo come una scatola chiusa; quello che ci serve sapere è che ci offre, tra le altre cose, anche un modo per poter monitorare i nodi attivi di PariDBMS (chiamato Outage Detection). Infatti, ogni nodo dello stesso plugin che lanci l'esecuzione di DiESeL, viene riconosciuto dal modulo come entità facente parte dello stesso servizio e, attraverso algoritmi che non verranno qui spiegati, riesce a costruire in più passi una tabella di indirizzi dei "vicini", poi dei "vicini dei vicini", e così via... DiESeL crea dei canali di connessione tra i nodi eroganti lo stesso servizio, ad un livello logico più basso del nostro "gestore del nodo". Questi canali possono non corrispondere esattamente a quelli che noi possediamo nella *NodeSet*, ma non importa. L'utilità del modulo diventa chiara una volta spiegata che esiste un metodo utile da implementare nell'interfaccia *IServerLayer*: *signalOutage( )*. Tale metodo viene invocato dal modulo quando, durante l'esecuzione di un qualche suo algoritmo simile

al nostro *ping*, si accorge che uno dei nodi che lui sapeva di far parte del servizio PariDBMS si è scollegato. A questo punto è sufficiente togliere tale nodo dalla propria *NodeSet*.

Tuttavia, anche questa integrazione potrebbe essere migliorata. DiESeL, dal punto di vista globale della rete, agisce con la modalità multicast perché crea dei canali di collegamento tra i nodi che erogano lo stesso tipo di servizio, come PariDBMS, PariIRC, PariWebServer, etc; dal punto di vista di un particolare plugin, invece, il modulo agisce con modalità broadcast in quanto un messaggio che DiESeL invia nel suo canale di comunicazione, arriva a tutti i nodi che erogano quel servizio. Considerando per esempio il segnale di disconnessione di un nodo, quando il modulo si accorge che uno di essi si è scollegato invia la notifica a tutti nodi di quel plugin. Ricevuto questo avviso, si tenta di cancellare l'oggetto *Node* corrispondente dalla *NodeSet*. Per i nostri scopi, tale avviso interessa in realtà solo a quelli che possedevano delle replicazioni nel nodo che ha appena interrotto la sua esecuzione. Tutti quelli che lo ricevono ma non hanno rapporti con il nodo scollegatosi non sono poi interassi di questo avviso. Ricordiamoci sempre che stiamo parlando del meccanismo di monitoraggio delle copie attive. Proviamo a fissare le idee con un esempio: ammettiamo che esistano 100 nodi sui quali è attivo il servizio PariDBMS. Ognuno di essi possiede certi database; alcuni nodi posseggono poi delle repliche presenti anche su altri host. Nella *NodeSet* di ogni nodo saranno presenti gli oggetti *Node* rappresentanti quei nodi che possiedono copie di un database locale. Consideriamo per semplicità che sia presente un solo database su ogni nodo; poiché questi devono essere replicati per aggirare l'indisponibilità, esistono molti nodi che posseggono esattamente lo stesso database di un altro. Tutti i nodi che hanno nella memoria la stessa base di dati si conoscono reciprocamente attraverso le proprie *NodeSet*. La disconnessione di un nodo provoca che negli altri 99 venga comunque tentata la cancellazione dell'oggetto *Node* corrispondente dalla *NodeSet*. Per la maggior parte di esse, però, non sarà nemmeno necessaria la rimozione in quanto questo nodo non era nemmeno presente visto che non aveva una copia del proprio database. Ci si può ora rendere conto dell'attuale inefficienza.

Si potrebbe chiedere a questo punto del perché si sia scelto di utilizzare questo modulo visto che è comunque inefficiente. Le motivazioni sono due: il primo è per il fatto che DiESeL è un modulo ben testato ed è perfettamente funzionante e inoltre i suoi progettisti hanno garantito che abbia il grande pregio di ridurre l'overhead sulla rete e migliorare le prestazioni finali rispetto ad altri algoritmi tradizionali come il classico "*ping-pong*". Per overhead intendiamo in questo contesto spreco di banda utile a favore di messaggi "inutili" dal punto di vista degli utenti. In secondo luogo, si era consapevoli che a breve sarebbe stata aggiunta al modulo DiESeL la funzionalità di creare dei sottogruppi anche all'interno dello stesso servizio, il che risponderebbe

esattamente alle nostre esigenze. Nella prossima revisione, si dovranno creare i gruppi di nodi in modo che tutti quelli replicanti un certo database facciano parte dello stesso gruppo.

3.4 Integrazione con il modulo Connectivity e gestione delle socket

Nella prima versione di PariDBMS, un'altra inefficienza è data dallo spreco delle socket java. Ogni volta che un messaggio di rete viene inviato o ricevuto, è utilizzata una nuova socket ciascuna volta. Si vorrebbe invece adottare un metodo che permetta di tenere traccia dei riferimenti della coppia di nodi che hanno già scambiato messaggi, cosicché la prossima volta debbano dialogare nuovamente venga riutilizzato la socket creata la prima volta.

Esiste un modulo di PariPari, chiamato Connectivity. Il suo compito è quello di restituire il riferimento di una nuova socket ogni volta che se ne richiede una. Anche se creare le socket utilizzando direttamente gli oggetti Java dedicati non sia sbagliato, uno degli obbiettivi di PariPari è garantire la modularità. Per ogni necessità viene sviluppato un modulo specifico che ha il compito di soddisfare quella categoria di bisogni. Nel caso delle socket, appunto, esiste il modulo Connectivity che le fornisce. Sarebbe quindi proibito procurarseli senza interpellare questo modulo di PariPari.

Potrebbe non sembrare un problema importante lo spreco delle socket. In realtà, ogni risorsa che si utilizza, come lo spazio di memorizzazione, il numero di socket, ecc, deve essere tenuto sotto controllo, perché PariPari verifica la "bravura" dei suoi plugin nella gestione delle risorse che sono state a loro concesse. In base a questo controllo si decide di concedere o meno al plugin richiedente la risorsa desiderata. Quindi, anche le socket devono essere gestite in maniera efficiente.

L'integrazione di PariDBMS con il modulo Connectivity e la gestione delle socket sono stati seguiti dal collega A. Rizzi. Brevemente e senza entrare nei dettagli implementativi, per fare in modo che ogni volta le socket non vengano chiuse e poi create di nuovo, si deve memorizzare in una struttura dati, opportunamente creata, il riferimento di quelle socket già aperte. Nel caso si dovesse nuovamente inviare a quel nodo un messaggio, sarà sufficiente recuperare il riferimento della socket creata alla prima instaurazione della connessione per quella copia. Per il ricevente invece, le cose sono più complicate; ad ogni messaggio da un mittente sconosciuto crea una nuova socket per la ricezione e memorizza tale riferimento sempre in una struttura dati. Controlla poi periodicamente se in questa struttura qualche socket ha il buffer pieno per capire se ha ricevuto dei messaggi da loro. Alterna quindi fasi in cui attende nuove connessioni a quelle in cui verifica se dai nodi che già conosce ha ricevuto messaggi. Si consiglia la lettura della tesi di Rizzi per maggiori dettagli.

4 CONCLUSIONI E SVILUPPI FUTURI

La realizzazione di un DBMS distribuito è un progetto ambizioso e complesso. Si devono fornire agli utenti tutte le funzionalità di un DBMS tradizionale e bisogna saper gestire nella maniera più efficace ed efficiente le basi di dati che vengono create. PariDBMS, per il momento, supporta soltanto le principali funzioni, come la creazione di database ed esecuzione di query sia di reperimento che di aggiornamento su di essi. Riesce poi a garantire, attraverso la replicazione, la disponibilità dei dati, eseguendo anche di fatto un backup automatico per essi. Nella sua prima versione, una delle principali debolezze era la mancanza di un metodo che permettesse di poter accedere a dati non memorizzati direttamente nella memoria del calcolatore in uso. Con la progettazione dell'accesso remoto, ora ciò è stato reso possibile. Inoltre, si è aggiunto anche il meccanismo chiamato login, e con esso si raggiunge un altro obiettivo fondamentale per un DBMS: la privatezza dei dati.

Un progetto così impegnativo, non può certo considerarsi concluso dopo due sole versioni. Di limiti il plugin ne ha tuttora tanti; vi è parecchio da lavoro ancora da svolgere. Un primo possibile miglioramento potrebbe essere quello di gestire la memoria locale dei nodi sfruttando il plugin Local Storage della cerchia interna; attualmente stiamo forzando l'utilizzo di HSQLDB passandogli come parametro il percorso di una nostra cartella privata, essendo ben consapevoli di violare le regole di PariPari. In seguito sarà necessario migliorare anche l'integrazione con il modulo DiESeL. Come accennato nel paragrafo 3.3, l'utilizzo di questo modulo rappresenta già una prima miglioria di efficienza per quanto riguarda l'overhead sulla rete, anche se si potrebbe fare di meglio. E' attualmente in fase di progettazione una nuova funzionalità di DiESeL e perciò a breve sarà possibile definire dei gruppi di nodi di un certo plugin appartenenti allo stesso servizio, che nel nostro caso potrebbe essere "replico il database X"; tutti quei nodi che posseggono una sua replicazione dovranno decidere di entrare in questo gruppo perché lo scambio di messaggi tra di essi sia molto più efficiente di quello che è attualmente. Infine si prevede in futuro di utilizzare il sistema di crediti di PariPari, realizzato da un modulo del Core che si chiama, appunto, Crediti. Tale modulo distribuisce dei token virtuali ai plugin e le risorse da essi richieste hanno un costo. Bisogna perciò essere intelligenti nel loro uso, gestendo le situazioni di spreco. A tale scopo, infatti, il collega A. Rizzi ha studiato un metodo per gestire le socket richieste al modulo Connectivity.

Infine, una possibile modifica, che però appare ancora remota, potrebbe essere il partizionamento dei database. Si vorrebbe in futuro dividere orizzontalmente e/o verticalmente le basi di dati e memorizzare i frammenti così ottenuti in diversi calcolatori; si potrebbe a questo punto permettere l'accesso contemporaneo su frammenti diversi. Attualmente, il collega F. Haymar sta studiando una possibile realizzazione di un tale meccanismo con il supporto del professore del corso di "basi di dati" Luca Pretto. Tuttavia, il lavoro è agli albori ed è solo in fase di studio; la sua reale implementazione richiederebbe una pesante modifica di molte parti del plugin, rendendolo però molto più efficiente a fronte di richieste simultanee di accesso.

PariDBMS è un plugin che, una volta finita la fase di sistemazione e aggiunta la gamma di funzionalità ancora mancanti, potrebbe davvero rappresentare un'ottima soluzione per tutti coloro che necessitano di gestire collezioni di dati, senza acquistare né hardware né software specifici.

5 BIBLIOGRAFIA

[1] *Reti di calcolatori* – Larry L. Peterson, Bruce S. Davie; seconda edizione, Apogeo 2008.

[2] *Sistemi di basi di dati : fondamenti* – Ramez Elmasri, Shamkant B. Navathe; quinta edizione, Pearson – Addison Wesley 2007.

[3] *Basi di dati : modelli e linguaggi di interrogazione* – Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone; seconda edizione, McGraw – Hill 2006.

[4] *PariPari DBMS : Garanzie di servizio* – Andrea Cecchinato; tesi di laurea triennale, Università degli Studi di Padova 2008.

[5] *PariPari DBMS : Re-inizializzazione e churn* – Jacopo Buriollo; tesi di laurea triennale, Università degli Studi di Padova 2008.

[6] *PariPari DBMS : Sincronizzazione* – Alessandro Costa; tesi di laurea triennale, Università degli Studi di Padova 2008.

[7] Sito internet: www.wikipedia.org.

[8] Sito internet: http://java.sun.com/javase/6/docs/api/ - Documentazione di Java.