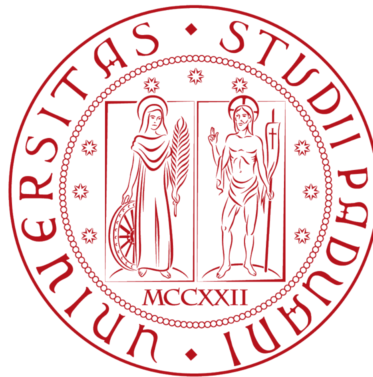
PARIDBMS: OTTIMIZZAZIONE THREAD E MESSAGGI

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
CORRELATORE: Ing. Paolo Bertasi
LAUREANDO: Alessandro Panciera

Corso di laurea in Ingegneria Informatica

A.A. 2010-2011



UNIVERSITÀ DEGLI STUDI DI PADOVA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA

PARIDBMS: OTTIMIZZAZIONE THREAD E MESSAGGI

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: Alessandro Panciera

A.A. 2010-2011

Alla mia famiglia.

Indice

Sommario **1**

Introduzione **3**

1 PariDBMS **7**

1.1 Blocco Base 9
1.2 Blocco Sincronizzazione 12
1.3 Blocco UpdateManager e Replication 13
1.4 Blocco RemoteAccessProtocol 18
1.5 Blocco NodeManager e integrazione DiESeL 18
1.6 Testing 20

2 Ottimizzazione Thread e messaggi **22**

2.1 DBMessage 23
2.2 RemoteAccessProtocol 29
2.3 Gestione Thread 45

3 Conclusioni **49**

4 Sviluppi Futuri **51**

Bibliografia **52**

Elenco delle figure **55**

A Ringraziamenti **56**

Sommario

La facilità sempre crescente di ottenere accesso ad internet e il diffondersi di apparecchiature elettroniche, tipicamente di bassa e media potenza di calcolo, rendono l'architettura peer-to-peer una scelta vincente nella programmazione di nuovi sistemi software. L'assenza di Server permette di eliminare il collo di bottiglia delle architetture tradizionali e sopperire alla necessità di dispositivi con capacità di memorizzazione e calcolo sempre maggiori. In quest'ottica si inserisce PariDBMS con lo scopo di sfruttare tutte le potenzialità dei *Database Management System* (DBMS) distribuiti nonostante la dinamicità dei nodi nella rete. Le innovazioni introdotte nell'ultimo semestre accademico dal gruppo, frutto di un attento studio delle prestazioni, sono state tutte pensate al fine di eliminare o minimizzare lo spreco di risorse del plugin. In particolare la ridefinizione della struttura di base e un restiling completo di alcune classi hanno permesso di ottenere un notevole incremento della velocità di esecuzione delle funzionalità in dotazione. Innanzitutto verrà presentato il sistema in generale PariDBMS, per poi passare alla spiegazione in dettaglio delle differenze strutturali e le motivazioni che ci hanno portato a pensare a delle ottimizzazioni del codice e dello schema interno.

Introduzione

In questo elaborato verranno illustrate la progettazione e la realizzazione delle modifiche attuate a `PariDBMS` per il progredimento a questa nuova versione. La prima versione ad opera di Jacopo Buriollo, Andrea Cecchinato e Alessandro Costa ed una seconda prodotta da Deniz Tartaro e Alberto Rizzi sono state ampiamente documentate nelle loro tesi di laurea triennale; quest'ultime sono letture consigliate per prendere piena coscienza di quelle parti di PariDBMS che non vengono trattate in questo testo o per approfondire le diversità introdotte in questa terza versione del progetto.
L'attenzione verrà in particolare focalizzata sulle parti riguardanti le modifiche introdotte al sistema di smistamento dei messaggi all'interno del plugin e della rete costituita dai nodi di `PariDBMS`, al refactoring completo del Protocollo di Accesso Remoto e all'evoluzione dei Thread nell'ottica del risparmio al fine di migliorarne le prestazioni.

`PariDBMS` si pone come progetto ambizioso di unire l'efficienza di gestione di grandi moli di dati strutturati, affidati ai *Database Management System* (DBMS), alla natura distribuita di una rete basata su architettura peer-to-peer (P2P), come *PariPari*, cercando di risolvere tutte le problematiche che questa unione comporta.

PariPari

In particolare *PariPari* è una rete multifunzionale progettata in Java che fornisce le funzionalità dei programmi P2P maggiormente diffusi (file sharing, storage e backup distribuito...) e i servizi tipici di internet (IRC , IM , Voip, Web Server, DNS...). Ma la vera forza di *PariPari* sta nell'erogare tutti questi servizi garantendo una cooperazione tra di loro e cercando di evitare il più possibile interferenze di funzionamento. La natura serverless permette ad ogni nodo (peer)

di funzionare sia da client che da server all'interno della struttura permettendo di richiedere, e al contempo erogare, una certa quantità di servizi predisposti dalla rete. La facilità di condivisione di risorse e la loro disponibilità non vincolata dai limiti di scalabilità e prestazioni, limiti tipici dei server dedicati, ha permesso il diffondersi di queste architetture che, in favore di una notevole flessibilità, richiedono una maggior complessità dei meccanismi di costruzione degli algoritmi di gestione della rete.
La struttura all'interno del singolo nodo della rete di *PariPari* è costituita da plugin[1]. Ogniuno di essi può essere visto come una black box a cui affidare l'elaborazione di dati e ottenere dei risultati senza entrare in dettaglio sul reale funzionamento del modulo. In questo modo, attraverso le *API*[2] fornite, chiunque è in grado di interfacciarsi con i servizi offerti permettendo di espanderne le funzionalità; finchè le *API* non vengono modificate, eventuali cambiamenti, aggiunte e sostituzioni all'interno dei moduli risultano totalmente trasparenti al resto del sistema.
La struttura modulare di *PariPari* è costituita principalmente da due gruppi di tipologie di plugin, la *cerchia interna* e la *cerchia esterna*. Fanno parte della cerchia interna *Connectivity*, il quale amministra gli accessi alla rete , *Local Storage*, che gestisce il file system, *Crediti*, un modulo ancora in sviluppo che dovrebbe regolamentare l'utilizzo di risorse, e *DHT* (Distributed Hash Table), quest'ultimo in particolare risulta essere fondamentale per la ricerca di risorse,servizi e nodi all'interno della rete *PariPari*. Questi plugin sono da considerarsi basilari e vengono utilizzati dai moduli della cerchia esterna per fornire i propri servizi.
Questi plugin si appoggiano ad un organo di sostegno denominato *Core*; infatti, come si evince dal nome, è il vero e proprio nucleo di tutto *PariPari*. Si occupa di assegnare risorse, soddisfare le richieste inoltrate tra i vari plugin e monitorare che quest'ultimi lavorino correttamente.

[1]programmi non autonomi che interagiscono con altri programmi per ampliarne le funzioni;

[2]Application Programming Interface - sono ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per un determinato compito. È un metodo per ottenere un'astrazione, di solito tra l'hardware e il programmatore, o tra software a basso ed alto livello. Le API permettono di evitare ai programmatori di scrivere tutte le funzioni dal nulla. Le API stesse sono un'astrazione: il software che fornisce una certa API è detto implementazione dell'API;

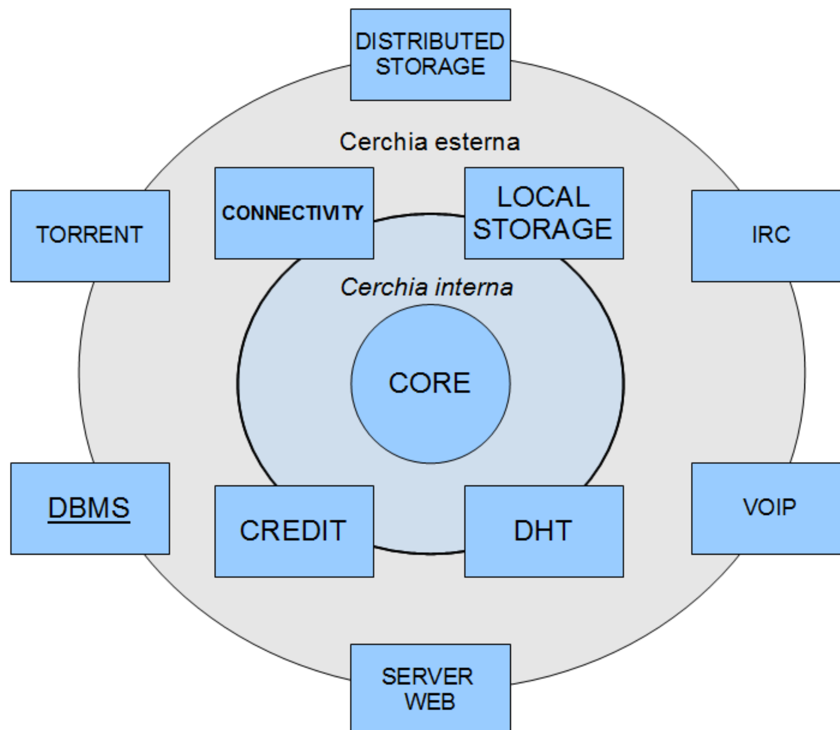


Figura 1: Rappresentazione grafica della struttura di PariPari;

PariDBMS

Il plugin DBMS appartiene alla cerchia esterna e ha il compito di riuscire a gestire un sistema di basi di dati distribuito all'interno della rete di *PariPari*. Qualsiasi altro DBMS presente sul mercato, sia esso strutturato con l'architettura client-server tradizionale o sia esso formato da una architettura distribuita, si basa sul principio che i nodi a cui viene affidato il compito di contenere il database sono macchine dalla grossa potenza di calcolo e perennemente connessi alla rete; inoltre la struttura dei database, ovvero la definizione del suo schema, contenuta è nota a priori; quest'ultimo punto è un privilegio di cui il nostro sistema DBMS distribuito non gode e risulta perciò uno degli ostacoli maggiori nella progettazione. *PariPari*, a differenza delle realtà appena accennate, è formata da una rete di dispositivi che potrebbero connettersi e sconnettersi a loro piacimento, ed è questa importante differenza a creare ulteriori problematiche nella progettazione. La peculiarità di PariDBMS consiste nel mantenere attiva e aggiornata una base di dati distribuita a fronte di una situazione della rete dinamica come quella appena descritta. L'obiettivo non è stato pienamente raggiunto e, per ora, il plugin permette di mantenere aggiornate nei nodi solamente repliche intere di ciascuna

base di dati inserita.
Questa scelta progettuale è stata effettuata per permetterci di creare una iniziale struttura di base solida da cui partire per poi introdurre, con l'implementazione di uno studio teorico delle basi di dati distribuite, un sistema DBMS totalmente distribuito e serverless attualmente non presente nel mercato informatico.
I vantaggi di un sistema così pensato sono innumerevoli, questo a patto che vengano rispettate le proprietà fondamentali definite *ACID*[3] dei DBMS in commercio; attualmente noi garantiamo atomicità ,consistenza e isolamento mantendo le copie all'interno della rete aggiornate fra loro tramite un sistema di sincronizzazione a *Timestamp*[4] distribuito simile a quello di Lamport e grazie all'utilizzo di un DBMS per la gestione delle transazioni in locale; la durabilità viene garantita da un monitoraggio costante delle copie salvate all'interno della rete che, se scendono al di sotto di una certa soglia numerica, inizieranno a riprodurre ulteriori copie.
`PariDBMS` inoltre utilizza un plugin dedicato per mantenere una connessione maggiore tra le copie, ovvero *DiESeL*[5]. Le API di questo modulo ci permettono di avere un controllo più efficiente sui nodi sparsi nella rete che contengono le copie di un certo database. Questa integrazione verrà dettagliata maggiormente nella tesi di laurea specialistica di Dan Bogatu.
Nelle successive pagine verrà descritta in dettaglio la struttura di `PariDBMS` evidenziando le modifiche apportate, verrà poi discusso in dettaglio il Protocollo di Accesso Remoto da me rielaborato e la ridefinizione di quelle classi che maggiormente hanno richiesto una rivisitazione del codice e della teoria di base.

[3]deriva dall'acronimo inglese Atomicity, Consistency, Isolation, e Durability (Atomicità, Coerenza, Isolamento e Durabilità)

[4]marca temporale per accertare l'effettivo avvenimento di un certo evento

[5]Distributed Extensived Server Layer - Il modulo DiESeL è una libreria specifica per gestire le applicazioni di tipo Server in modo indipendente alle funzioni particolari del programma che intende usufruirne

Capitolo 1

PariDBMS

In questo capitolo si scenderà più in dettaglio sulla struttura con cui `PariDBMS` è stato concepito e implementato offrendo una panoramica della rivoluzione interna subita. La struttura del DBMS a cui il progetto fa affidamento rimane immutata, infatti la scelta di utilizzare HSQLDB[1] come motore di management per basi di dati relazionali risulta essere pratica e conveniente a fronte anche di future modifiche al codice del DBMS stesso; essendo scritto in Java e avendo licenza di distribuzione opens source è possibile apportare risistemazioni utili al codice senza incorrere in violazioni del copyright. In proposito, allo stato attuale, `PariDBMS` non si affida al *Local Storage* per la scrittura del database su file, ma richiede semplicemente il percorso della cartella di riferimento per poi affidare ai metodi dell'HSQLDB la scrittura in base al percorso ottenuto; sono attualmente in corso delle modifiche alla struttura del supporto di DBMS, ma queste sono ancora in una fase non testabile e ritengo doveroso presentare solamente le modifiche completate.
E' ad HSQLDB stesso che ci si è affidati per mantenere memorizzato tra una sessione e l'altra lo stato e la struttura dei database contenuti; questo avviene tramite scrittura su un database di supporto (*SupportDB*) invisibile agli utenti che memorizza al suo interno i dati fondamentali del plugin come le query eseguite, la lista dei database creati sul nodo, i nomi delle tabelle per ogni base di dati e tutte quelle informazioni temporali sulle modifiche effettuate ai database. In questa nuova versione si è deciso di ridisegnare parzialmente lo *schema ER*[2]

[1] DBMS relazionale nato dalle ceneri di Hypersonic SQL Project scritto interamente in java e con codice sorgente distribuito con una licenza simile alla BSD

[2] Schema Entity-Relationship ,è un modello per la rappresentazione concettuale dei dati ad un alto livello di astrazione,uno degli approcci più solidi per la modellazione di domini applicativi in ambito informatico

del *SupportDB* definendo meglio l'ordine sequenziale con cui i parametri di ciascun database devono essere inseriti. La tabella cardine è *USERS*, la quale serve a sottolineare il fatto che un database può essere inserito se e solo se esiste un utente, con relativa password, a cui appoggiarsi; rispetto al precedente schema, non esisteva questa dipendenza delle tabelle, ma vi era un campo della tabella *DATABASES* con la dicitura User a cui affidarsi. La modifica effettuata serve per permettere di poter creare basi di dati con lo stesso nome ma utente proprietario diverso; ciò non era possibile prima in quanto non si aveva controllo sulla possibilità di definire un utente unico per tutta la rete. In futuro, questo utente, verrà attribuito e assegnato automaticamente mediante l'accesso al plugin *Login*, modulo attualmente ancora in fase di sviluppo. Il resto dello schema non presenta profonde modifiche fatta ad eccezione delle chiavi delle entità deboli che, derivando dalla tabella *DATABASES* possiedono la chiave aggiuntiva Username, chiave primaria di *USERS*. Il resto del modulo verrà ora presentato partendo

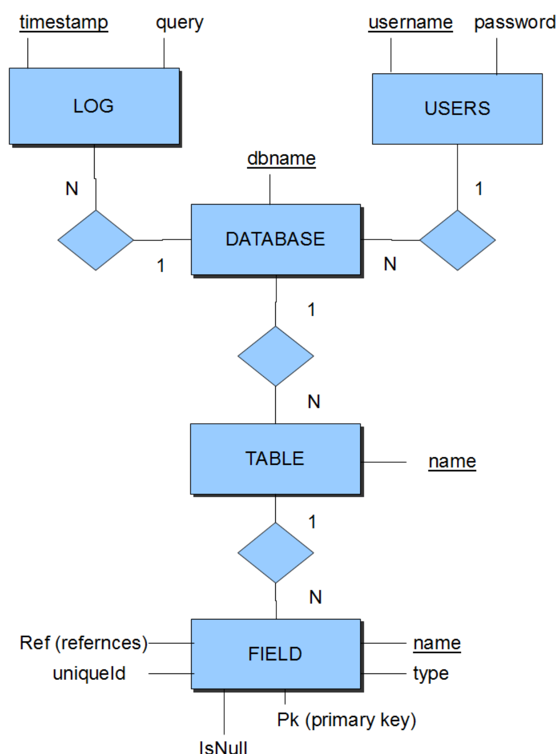


Figura 1.1: Schema ER del SupportDB con evidenziate solamente le chiavi (che godono della proprietà di Unique) della singola tabella; le tabelle ombreggiate indicano che l'entità è debole e eredita le chiavi primarie dalla tabella la cui la relazione 1:N è lato 1;

da una definizione generale e dalla descrizione di funzionamento delle parti che lo compongono, per poi scendere nei dettagli di implementazione sulle parti del codice riguardanti modifiche e migliorie da me personalmente introdotte.
La struttura di `PariDBMS` si suddivide principalmente in cinque macro blocchi, ciascuno di essi si occupa di portare a termine una (o più) delle fasi necessarie al corretto funzionamento del plugin.

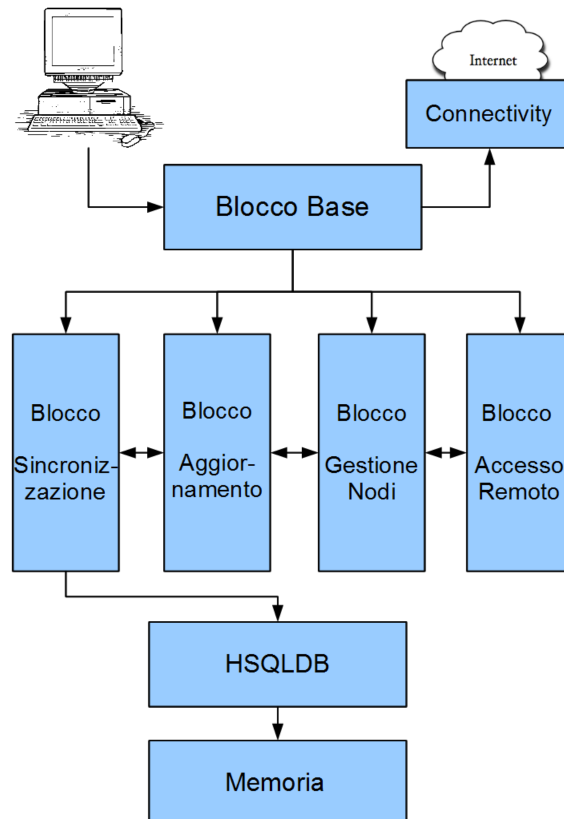


Figura 1.2: Schema dei blocchi del plugin

1.1 Blocco Base

Il blocco definito Base riguarda tutta la parte del plugin adibita ad interfacciarsi con l'esterno, verso ulteriori nodi e utenti, e allo smistamento dei messaggi all'interno, verso gli altri blocchi del plugin. Notevoli modifiche sono state introdotte al fine di rendere più semplice e diretta la comunicazione fra i nodi; questa avviene tramite due punti di accesso: Console, ovvero attraverso l'inserimento dei comandi da parte dell'utente, e Connectivity, in cui i messaggi provengono da altri nodi

della rete con cui siamo in collegamento.
La riga di commando della *PariPari Console* prenderà le stringhe inserite e le convertirà in messaggio tramite la classe *DBMessage* che si occupa di confezionarle nel modo corretto e effettuare una prima verifica sui contenuti della stringa; passata questa fase il messaggio verrà inviato alla *GlobalMessageQueue* in attesa di essere processata. Allo stesso modo i messaggi proveniente dalla rete vengono prelevati dalle *Socket*[3], quest'ultime aperte all'ascolto tramite l'ausilio del plugin della cerchia interna denominato *Connectivity*, il quale, nelle intenzioni degli sviluppatori, ha il compito di amministrare tutto quello che riguarda la connettività, ossia centralizzare la creazione dei socket. Il *NetworkMessageReceiver* è il thread incaricato di comunicare e gestire la connessione con questo modulo e, dopo aver prelevato il messaggio, si adopererà, come la console ,ad inserirlo nella *GlobalMessageQueue*; quest'ultima classe rappresenta il punto di raccolta di tutti i messaggi prima dello smistamento ai blocchi.
Il *MessageInterpreter* è la vera novità di questo blocco di `PariDBMS` in quanto ha preso il posto di cinque classi separamente implementate nella precedente versione; potrebbe sembrare ad un primo sguardo che la scelta di accorpare tutte quelle classi in una sola non sia la più concorde ai principi di programmazione *multithreading*[4], in realtà, andando ad analizzare il funzionamento di queste classi, ci accorgiamo che così facendo stiamo evitando uno spreco non indifferente di risorse. Tra le vecchie classi di cui era composto il blocco base, *MessageDelivery* si incaricava di prelevare i messaggi dalla *GlobalMessageQueue*, quando quest'ultima non era vuota, e di smistarle ai vari blocchi sottostanti analizzando semplicemente il primo campo del messaggio (i campi dei messaggi e i loro significati verranno discussi in dettaglio nel prossimo capitolo) e demandando

[3]un'astrazione software progettata per poter utilizzare delle *API* standard e condivise per la trasmissione e la ricezione di dati attraverso una rete, è il punto in cui il codice applicativo di un processo accede al canale di comunicazione per mezzo di una porta, ottenendo una comunicazione tra processi che lavorano su due macchine fisicamente separate. Dal punto di vista di un programmatore, un socket è un particolare oggetto sul quale leggere e scrivere i dati da trasmettere o ricevere;

[4]supporto da parte di un processore di gestire più thread Questa tecnica si distingue da quella alla base dei sistemi multiprocessore per il fatto che i singoli thread condividono lo stesso spazio d'indirizzamento, la stessa cache e lo stesso *TLB*(translation lookaside buffer). Il multithreading migliora le prestazioni dei programmi solamente quando questi sono stati sviluppati suddividendo il carico di lavoro su più thread che possono essere eseguiti in parallelo. I sistemi multiprocessore sono dotati di più unità di calcolo indipendenti, un sistema multithread invece è dotato di una singola unità di calcolo che si cerca di utilizzare al meglio eseguendo più thread nella stessa unità di calcolo.

l'analisi dei campi successivi a dei thread dedicati ciascuno ad un blocco specifico; quest'ultimi però non facevano altro che scandagliare, ciascuno con un proprio metodo, il messaggio in tutte le parti non ancora esplorate e proseguire nel corretto inoltro. Una leggera contemporaneità di funzionamento che poteva verificarsi nello smistamento dei messaggi, come ad esempio un messaggio diretto al blocco sincronizzazione il quale poteva essere elaborato in simultanea ad un messaggio diretto al *Protocollo di Accesso Remoto*, soffriva però dei maggiori rallentamenti causati dai vari cambi di contesto fra i vari thread interpreti in esecuzione; a fronte di questi ragionamenti si è preferito accorpare tutta la sezione dedicata all'inoltro dei messaggi in un unico thread *MessageInterpreter*. Una migliore spiegazione sulla scelta di accorpamento dei thread verrà motivata nel successivo capitolo di approfondimento.
In più questa classe appena accennata sfrutta il nuovo formato dei messaggi basato sugli *Enum*, tipi di dato strutturato che saranno descritti nel capitolo successivo, per ottenere un codice più snello e funzionale accuratamente testato in ogni sua parte. La classe *DBMS* del blocco Base verrà approfondita in dettaglio

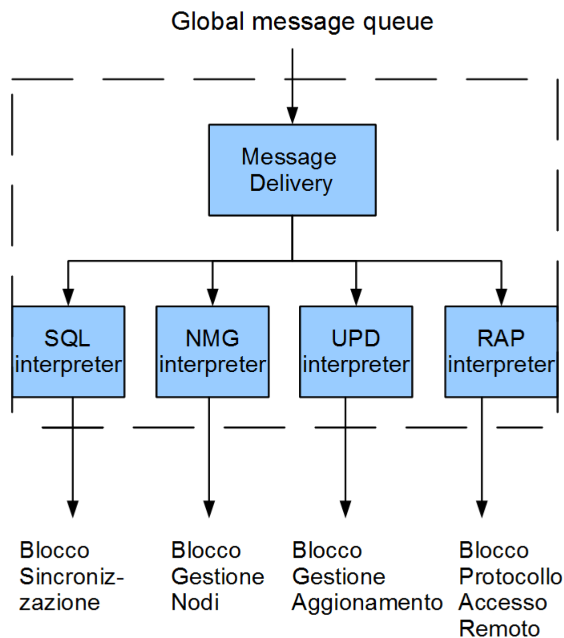


Figura 1.3: Schema di accorpamento da MessageDelivery e interpreti a MessageInterpreter;

nel capitolo successivo in cui si discuterà del funzionamento e sulla scelta delle partenze dei thread durante la fase di caricamento del plugin.

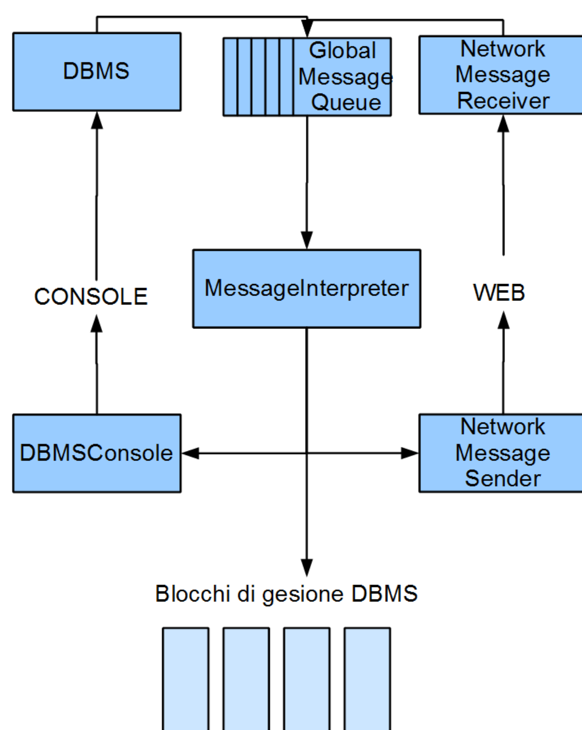


Figura 1.4: Raffigurazione struttura di base

1.2 Blocco Sincronizzazione

Il blocco di sincronizzazione delle basi di dati si occupa di tutto ciò che riguarda l'esecuzione delle operazioni sui database contenuti nel nodo, infatti tutte le istruzioni SQL indirizzate ad una particolare base di dati sono direzionate dal *MessageInterpreter* alla relativa coda messaggi.
All'avvio del plugin verrà creato un set denominato *NodeSet* che si occupa di tenere i riferimenti agli oggetti *Database* le cui credenziali vengono acquisite dall'interrogazione del contenuto del *SupportDB*; questi oggetti contengono i riferimenti alla coda messaggi e a tutti i thread necessari a gestire l'evoluzione della base di dati nel tempo. Un aspetto fondamentale è il coordinare il salvataggio in tutte le replicazioni del database in modo da non avere istanze non correttamente allineate fra loro; è in questa fase che entra in gioco il temporizzatore a *Timestamp* che attribuisce un ordine sequenziale alle istruzioni inserite ed inviate. Ciascun oggetto *Database* ha associato un thread *TransactionSynchronizer* che preleva le istruzioni SQL dalla coda messaggi e inizia ad elaborarle secondo un protocollo di sincronizzazione descritto in dettaglio nella tesi di Alessandro Costa. La sostanziale differenza tra la vecchia versione e la nuova è riferita alle

modifiche attuate alla classe *TransactionExecutor*, un thread precedentemente istanziato per ogni singolo database in locale che si occupava di inserire concorrenzialmente all'interno di HSQLDB una query e verificare che quest'ultima fosse correttamente inserita. Questo thread è sostanzialmente un mero esecutore di query che, oltre a doverle eseguire passandole al gestore, deve visualizzarne anche i risultati correttamente e, in caso di modifiche alla struttura tramite query DDL[5] e operazioni di INSERT, l'*Executor* deve mantenere aggiornato il *SupportDB*. Se osserviamo il comportamento dei singoli *Executor* non fanno altro che contendersi le risorse e la permanenza nella CPU per poi incanalare comunque le risorse in code di esecuzione che seguono l'ordinamento a *Timestamp* imposto; una suddivisione così pensata risulta perciò dispendiosa e non sfrutta le potenzialità del multithreading. La soluziona da me introdotta elimina questi thread, ciascuno dei quali dedicato ad un database salvato, e ne introduce invece solamente una versione generalizzata denominata *TransactionExecutor* che si occupa di accodare e gestire tutte le istruzioni SQL inserite indipendentemente dalla base di dati a cui fanno riferimento; in particolare aggiungendo dei riferimenti alla base di dati di appartenenza dell'istruzione e introducendoli all'interno dell'oggetto *Transaction*[6] posso reperire facilmente le credenziali dbname, username e password necessarie per le modifiche e l'accesso.
Fanno parte di questo blocco anche i metodi di creazione di nuovi database siano essi richiesti da utenti locali o di provenienza esterna al nodo.

1.3 Blocco UpdateManager e Replication

Questo particolare blocco riguarda la fase in cui le basi di dati si riconnettono dopo un periodo di assenza dalla rete; per permettere l'aggiornamento il database inizia sempre il suo nuovo ciclo di funzionamento partendo da uno stato di UPDATE. Questo è un processo delicato della sincronizzazione delle replicazioni perchè necessita una ricerca nella rete di una copia attiva e con lo stato del database READY, stato che certifica che questa copia è aggiornata all'ultimo timestamp; dopo averla trovata e aver instaurato un ponte di comunicazione, inizia una fase di ricerca della situazione del database del nodo. Quest'ultimo per

[5]Data Definition Language - compone una parte del linguaggio SQL(Structured Query Language) e permette di creare, modificare o eliminare gli oggetti in un database

[6]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;

verificare il suo stato effettua un confronto tra il suo timestamp e quelli del nodo con il database in stato READY. Per comodità contraddistingueremo con la dicitura MinTS, per indicare l'ultimo timestamp reperibile nello storico dei Log, e MaxTS, per indicare il timestamp dell'ultima operazione eseguita e confermata; la casistica prevede quattro situazioni distinte:

- **Timestamp >MaxTS:**
  Questo caso indica un erronea trasmissione del *Timestamp* e si procede con la ritrasmissione del valore di MaxTS. Per evitare che la situazione si ripeta ed evitare il rischio di ottenere un valore di timestamp non affidabile, ne viene richiesto l'invio ad opera di un altro nodo con lo stesso database READY contenuto;

- **Timestamp = MaxTS:**
  Il verificarsi di questo caso limite indica che la replicazione in nostro possesso è aggiornata all'ultimo timestamp e perciò può semplicemente diventare una delle copie READY presenti nella rete ed interagire attivamente alla gestione dell'allineamento;

- **MinTS <Timestamp <MaxTS:** Se fino a questo punto le metodologie di soluzione e risposta del programma di aggiornamento del database sono rimaste inalterate, è in corrispondenza del verificarsi di questo particolare evento che si evidenziano le modifiche introdotte. Nel vecchio sistema si effettuava una scansione delle righe di log mancanti e le si inviavano una ad una; conclusasi questa fase, si passava poi alle operazioni non ancora eseguite ma in attesa di conferma (da parte del gruppo di basi di dati in allineamento) contenute all'interno della *TransactionQueue* presente nel blocco sincronizzazione. Nel nuovo sistema introdotto si è deciso di evitare di mandare i messaggi di aggiornamento singolarmente, ma si è preferito accorparli in un unico file di testo che viene, in una successiva elaborazione, compresso, attraverso algoritmi di zip, e inviato. Tanti messaggi nella rete aumentavano le probabilità di errore e di non essere recepiti correttamente dal destinatario nonostante il protocollo FTP[7] con cui questi vengono inviati, inoltre il padding del frame Ethernet necessitava spesso di valori di riempimento diminuendo la percentuale di file di testo utile su pacchetto inviato; il messaggio unico zippato permette di inviare tutte le informazioni e elaborarle avendo la garanzia che l'ordine di inserimento e l'ordine di

[7]File Transfer Protocol - è un protocollo per la trasmissione di dati tra host basato su TCP

ricezione delle righe contenute siano perfettamente rispettati ottenendo una trasposizione delle query nel file di Log e di conseguenza una copia esatta del database in questione.
Alla ricezione il file viene smistato dal *MessageInterpreter* ed inoltrato al relativo *DBUpdateClient* (ovvero il thread adibito allo scopo di aggiornamento del database relativo), nel momento in cui il timestamp, dopo il progressivo inserimento ed esecuzione delle query, ha raggiunto il MaxTS allora la base di dati può ritenersi aggiornata e passare allo stato di READY come nel caso precedente;

- **Timestamp <MinTS:** Anche questo caso richiede un'analisi aggiuntiva e una spiegazione maggiormente approfondita del funzionamento del nuovo protocollo di ripristino introdotto. Nella versione precedente non vi era un sistema implementato a livello di codice, ma semplicemente teorizzato, che prevedeva di copiare l'intera base di dati; questo sistema obsoleto viene rivoluzionato tramite un sistema simile alle funzioni di DUMP supportate in altri DBMS (funzione non prevista da HSQLDB), metodo che consiste nel creare in un file di testo tutto il codice SQL riguardante la definizione e il contenuto dell'istanza della base di dati in questione. Questo avviene tramite un'interrogazione al *SupportDB* e alla definizione della struttura delle tabelle e la popolazione delle sue tuple tramite inerrogazioni SQL opportune; per facilitare la spiegazione e la comprensione del testo verrà definito anche questo metodo non supportato da HSQLDB come DUMP. Come nel caso precedente, il file viene compresso e inviato; i vantaggi di avere una compressione di un file di testo piuttosto che una copia zippata dell'intera base di dati con l'aggiunta di metadati e file di log non è così evidente come potrebbe sembrare, infatti, sperimentando questa funzionalità introdotta, si sono ottenuti spesso file con dimensione maggiore rispetto alla base stessa (un esempio è un database con dimensione 3 MB e un "DUMP file" da 4MB) che potrebbero scoraggiare l'aver intrapreso questa strada. Questo però prima di effettuare la compressione di entrambi i file elencati ottenendo una riduzione di spazio di memoria di un fattore stimato al 4.4 per il file di testo (dimensione di 0.9 MB) contro un fattore 2.1 per quanto riguarda la base di dati intera (dimensione di 1.4 MB); si è perciò deciso di sfruttare la funzione DUMP per l'invio della copia aggiornata della base di dati. La copia così formata del database non è sufficiente ad aggiornarlo nella sua totalità in quanto mancano, come nel caso d'interesse precedente, tutte le istruzioni in attesa di elaborazione all'interno della *Transaction-*

Queue del modulo di sincronizzazione. Se concludessimo l'aggiornamento del nodo dopo quest'ultima istruzione otterremmo, contrariamente a quanto potrebbe sembrare, una discrepanza del Log della copia in nostro possesso del database con quello in possesso nel nodo Server; ciò avviene a causa del fatto che noi otteniamo all'interno della nostra base di dati solo gli effetti delle istruzioni SQL e non il loro codice. L'influsso di questa differenza va a danneggiare il valore di riferimento del timestamp della copia lato Client generando delle inconsistenze; analizzando le possibili casistiche ci si rende conto che il timestamp risultante può presentare solo due tipologie di valori: stesso timestamp o timestamp inferiore rispetto alla copia lato Server; non può superare di dimensioni la versione Server in quanto al più vi saranno un numero di righe pari a quelle riguardanti il DDL della base di dati, in egual misura per il Client, e l'inserimento delle singole tuple, che per definizione della copia devono essere numericamente uguali.
Il caso timestamp inferiore, che sembra verificarsi più frequentemente, è legato al fatto che nella creazione del database tramite DUMP non vengono considerate le modifiche come la cancellazione di tuple o gli aggiornamenti di valori; evento che viene però evidenziato e memorizzato all'interno del Log del *SupportDB*. Il caso timestamp uguale non è da considerarsi migliore di quello appena affrontato in quanto risulta essere una mera casualità numerica l'aver fatto combaciare i due risultati. Viene presentata in figura 1.5 una situazione ipotetica come esempio dell'ultima casistica appena analizzata. La soluzione per ovviare a questo problema è ricopiare l'intero file di Log del nodo Server all'interno del Client ottenendo così una copia uguale in tutto e per tutto alle replicazioni READY nella rete, da questo momento in poi la base di dati è pronta a collaborare alle varie sincronizzazioni. Questa è la soluzione che, attualmente, si è deciso di adottare per mantenere un alto numero di copie nella rete; è prevista, però, un'attenta analisi delle replicazioni e del funzionamento quando la rete di *PariPari* inizierà a diffonsersi su un numero elevato di nodi andando a modificare i parametri intrinsechi di riproduzione del modulo, in particolare si valuterà se è sempre opportuno replicare immediatamente l'intera base di dati nel nodo o se eliminarla semplicemente permettendo la connessione remota in caso di necessità.
Lo stesso procedimento di copia appena descritto può essere utilizzato per replicare le basi di dati nel momento in cui queste scendono sotto una soglia numerica prefissata (attualmente 12, ma nulla esclude che possa variata in

Log dopo DUMP	Log sorgente	timestamp
CREATE TABLE Scuola (codice ...	CREATE TABLE Scuola (codice ...	1
CREATE TABLE Studenti (nome ...	CREATE TABLE Studenti (nome ...	2
INSERT INTO Scuole(...	CREATE TABLE Studentibis (codice...	3
INSERT INTO Scuole(...	DROP TABLE Studentibis	4
INSERT INTO Studenti(...	INSERT INTO Scuole(...	5
INSERT INTO Studenti(...	INSERT INTO Scuole(...	6
INSERT INTO Studenti(...	INSERT INTO Studenti(...	7
INSERT INTO Studenti(...	INSERT INTO Studenti(...	8
	ALTER TABLE Studenti ADD ...	9
	INSERT INTO Studenti(...	10
	INSERT INTO Studenti(...	11

Figura 1.5: Variazione Log in base alla versione e relativo timestamp

base alle necessità della rete o dell'utente, un database prioritario può decidere di avere un valore di soglia maggiore rispetto a quello prefissato). Sempre nell'ottica del risparmio di potenza di calcolo si è deciso di eliminare il thread *UpdateManager* che si doveva incaricare di far partire tutti i thread DBUpdateClient e i DBUpdateServer adibiti all'aggiornamento e spostare i riferimenti all'interno dell'oggetto Database, si occupa a questo punto lui stesso di cercare di portare la propria copia locale dallo stato di UPDATE allo stato di READY;

Questa fase di aggiornamento avviene per ciascuna base di dati ogni volta che quest'ultima si riconnette alla rete, finito di renderla aggiornata e disponibile il thread di update esaurisce i suoi compiti e termina così la propria esecuzione. Eventuali fallimenti che comportano l'eliminazione della base di dati sono dovuti all'assenza di copie READY nella rete (dovuto ad un Churn[8] troppo veloce o al fatto che tutte le copie stanno aggiornando un ulteriore nodo) e sono eventi che devono esserre ancora compresi e gestiti appieno a causa dell'attuale limitata diffusione della rete *PariPari*.

[8] indica il tasso di abbandono definitivo di un servizio da parte di un cliente; nel nostro caso l'entrata-uscita di nodi nella rete

1.4 Blocco RemoteAccessProtocol

Il blocco *RemoteAccessProtocol* (acronimo *RAP*) è stato introdotto come parte del plugin solamente dalla seconda versione a cura di Deniz Tartaro, ma, a causa di errori sulla logica di funzionamento di tutto il plugin, è stato progettato senza tenere conto delle vere necessità e potenzialità che questa sezione poteva offrire. L'introduzione di questa funzionalità è dovuta al fatto che si è deciso di non permettere il controllo all'utente sul luogo in cui salvare e mantenere le repliche della base di dati; l'accessibilità risulta essere così compromessa in quanto, se mi trovo in un nodo che non possiede la replicazione in locale, non posso accedere al database di mia appartenenza (o di cui possiedo i permessi di accesso). Con il *Protocollo di Accesso Remoto* elimino questo punto debole da parte del plugin, infatti, grazie al blocco *RAP*, sarà permesso di effettuare l'accesso a qualsiasi base di dati appartenente a `PariDBMS` conoscendone solamente le credenziali private di accesso che lo identificano univocamente nella rete (nome database e nome utente sono le chiavi primarie di identificazione) e la password riferita ad esso; il resto dell'utilizzo risulta essere identico alla versione di impiego in locale. Rispetto alla versione originale si è cercato di snellire e semplificare il protocollo di comunicazione per renderlo maggiormente utilizzabile e garantendo una certa trasparenza di funzionamento da parte dell'utente.
Questo blocco verrà analizzato in dettaglio nel prossimo capitolo della tesi.

1.5 Blocco NodeManager e integrazione DiESeL

Ciò che ci permette di attuare il protocollo di sincronizzazione e il mantenimento dell'allineamento delle copie è il blocco integrato con il modulo *DiESeL*, questa porzione di codice di `PariDBMS` identificata come NodeManager permette di mantenere una efficiente e ottima connessione con i nodi che contengono la stessa copia di database in locale. Questa parte della definizione del plugin `PariDBMS` è stata attentamente rielaborata a cura del laureando in ingegneria informatica specialistica Dan Bogatu al fine di evitare congestione e ridondanza dei messaggi all'interno della rete; verranno qui di seguito presentati alcuni dei vantaggi ottenuti con l'integrazione e le differenze di funzionamento apportate, per una valutazione approfondita del blocco e del suo funzionamento si rimanda alla lettura della tesi del laureando Bogatu.
IserverLayer è l'interfaccia a cui si fa riferimento per riuscire a comunicare con questo plugin, in particolare *DBMSServerLayer* è l'implementazione nel plugin

DBMS dell'interfaccia di DiESeL e rappresenta lo strato comune utilizzato per comunicare informazioni tra i due plugin; un esempio del suo utilizzo è dato dalla tipologia dei messaggi e funzionalità offerti dallo strato DiESeL che, per mezzo dell'estensione della classe che usufruisce dell'interfaccia, permette l'invio di messaggi appartenenti allo standard "NMG" previsto dall'oggetto *DBMessage* (vedi capitolo successivo per i dettagli). All'avvio del plugin l'interrogazione al *SupportDB* permette di ottenere nel *NodeSet* la lista di oggetti *Database* da creare e configurare per avere il riferimento a tutte le basi di date contenute in locale; ciascuna di esse, dalla creazione, si adopererà ad ottenere, tramite l'avvio dei thread di DiESeL, la lista dei nodi di `PariDBMS` attivi nella rete in cui vi è contenuta una replica (sia essa in stato di READY o UPDATING) del database relativo.
Nella versione precedente esistevano due classi che contenevano reciprocamente, una i riferimenti ai database, identificata con il nome di *DBSet*, e l'altra conteneva la lista dei nodi riferita alle basi di dati, il *NodeSet*; questa separazione causava all'interno del plugin una serie di anomalie dovute ad alcune inconsistenze, in particolare ogni attività che comprendeva la cancellazione o l'aggiunta di database e nodi poteva essere catturata solamente da una delle due classi, quella non influenzata dalla modifica procedeva con il suo lavoro con una versione non aggiornata della lista di riferimenti. Le classi all'interno dei moduli ottenevano così informazioni diverse a seconda del tipo di set interrogato e allo stato in cui verteva offrendo spesso una soluzione errata al problema o richiedendo di intervenire direttamente sull'aggiornamento delle classi.

Attualmente, il *NodeSet*, come descritto nel blocco sincronizzazione, è stato riscritto al fine di contenere i riferimenti ai database e, quest'ultimi, al loro interno, avranno il controllo e il riferimento ai nodi delle replicazioni della rete

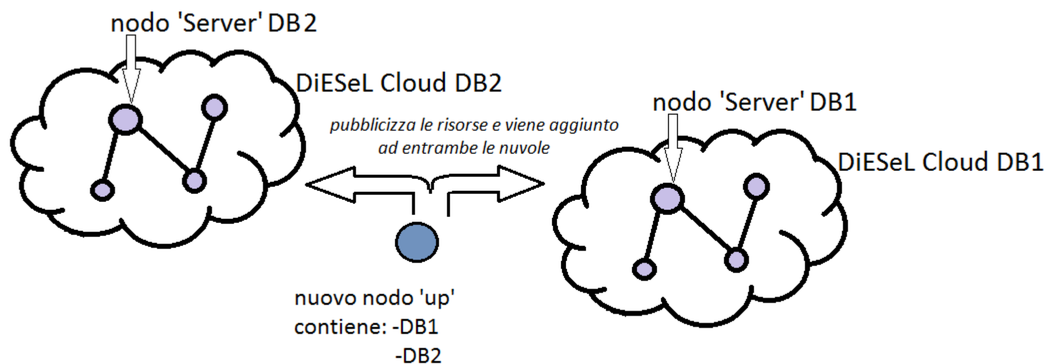


Figura 1.6: Schema di funzionamento delle reti DiESeL

PariPari in collaborazione con DiESeL.
Le informazioni che DiESeL necessita per mantenere il nodo in contatto con il resto della rete sono composte principalmente dalla conoscenza dell'indirizzo Ip e dalla porte assegnate a quel servizio (tipicamente due per database memorizzato, una per l'uscita e una per l'ingresso di messaggi); quando il nodo cade viene notificato al gruppo la sparizione dell'elemento dall'insieme di repliche di database appartenenti al plugin i quali lo rimuovono dalla propria lista.
Un ulteriore innovazione, introdotta con l'integrazione completa con il plugin di gestione dei Server, è la possibilità di mandare messaggi in multicast all'interno di un gruppo di nodi con stesso database; questa funzionalità permette che ogni nodo possa gestire aggiornamenti e cancellazioni semplicemente inoltrandoli al layer di comunicazione con DiESeL.

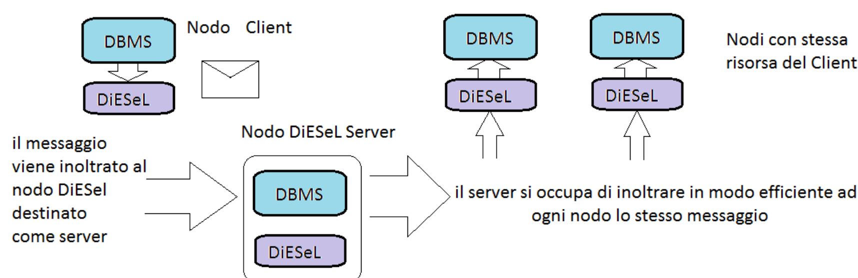


Figura 1.7: Sistema di invio messaggi sfruttando la struttura delle reti DiESeL

1.6 Testing

Ultima parte, ma non per questo la meno importante, è la descrizione dell'attività di testing intrapresa dal plugin in questi ultimi mesi di lavoro. Per verificare il funzionamento delle singole classi del nostro progetto è stato utilizzato lo *unit testing*, la procedura prevede la possibilità di testare la più piccola unità (*Unit*) della programmazione procedurale, nel nostro caso il metodo.
La facile individuazione dei bug rappresenta il beneficio maggiore dato dall'utilizzo di questo tipo di verifica del codice; la metodologia di sviluppo che utilizza maggiormente questo sistema di sviluppo è l'*Extreme Programming*(*XP*) il quale prevede che, ancor prima di scrivere del codice per l'implementazione di progetti, esista una solida base di test in modo da verificare passo passo il funzionamento e la veridicità dello sviluppo delle classi. Lo unit test se progettato correttamente evidenzia le funzionalità richieste, le specifiche e i difetti del progetto ancor prima

che questo sia effettivamente scritto.
Poichè alcuni oggetti possono far riferimento ad altri, i test spesso si propagano anche in altre classi; un esempio di interazione può essere quello affrontato dalle classi che accedevano al database *SupportDB*, in questo caso il testing della classe spesso implicava la scrittura di codice che interagisce con la base di dati. Questo è un problema in quanto lo unit test non dovrebbe mai varcare i confini della classe; la soluzione prevede l'implementazione di interfacce e l'utilizzo dei *mock object*, ovvero delle simulazioni dell'oggetto reale che possono essere configurate e controllate a nostro piacimento; i mock fingeranno di essere l'oggetto che si nasconde dietro l'interfaccia implementata e verranno utilizzati al posto delle istanze reali delle classi che non sono coinvolte direttamente nel test. La situazione iniziale che il progetto `PariDBMS` presentava era supportata solamente per un 5% da classi di test; alla luce dei fatti si è deciso di integrare, parallelamente al restiling del codice, la scrittura delle relative classi di test, arrivando attualmente ad una soddisfacente copertura di metà delle classi sul totale(50%). La decisione di dedicare parte della forza lavoro alla scrittura di test è stata ripagata appieno dalla quantità di bug scoperti e risolti; inoltre, con la necessità di utilizzare i mock nella programmazione di quest'ultimi, si è potuto notare dipendenze e interazioni fra le classi che ad una prima occhiata non si erano presentati obbligando i programmatori a rivalutare schemi che fino ad un attimo prima erano stati considerati attendibili.

Capitolo 2

Ottimizzazione Thread e messaggi

Questo capitolo riguarda il lavoro di ottimizzazione da me pensato e implementato al fine di migliorare le prestazioni del codice per avvantaggiare e semplificare l'utilizzo da parte dell'utente e da parte dei programmatori futuri di `PariDBMS`.
Durante la mia prima fase di approccio al modulo DBMS ho potuto notare come una linea guida generale di progettazione non rispettata potesse creare problemi alla comprensione del codice, infatti ogni blocco precedentemente descritto presentava una propria metodologia di accesso alle risorse, una propria classe da cui estrarre i riferimenti ai database (come la discrepanza *NodeSet* e *DBSet*) e molte altre iniziative individuali e non concordate nel gruppo che hanno ampliato la confusionarietà del codice; è da questa idea che sono partito per rivedere la struttura logica con cui il plugin è stato sviluppato.
Le parti fondamentali trattate qui di seguito riguardano due aspetti determinanti per un buon funzionamento del plugin `PariDBMS`: la comunicazione e il risparmio di risorse; la prima tematica verrà affrontata tramite la semplificazione e la generalizzazione dell'unità fondamentale per la comunicazioni fra nodi e attraverso un'importante riduzione dei tempi di booting grazie al protocollo di accesso remoto; per quanto riguarda il risparmio di risorse verrà analizzato il sistema di partenza dei thread e il loro effettivo utilizzo e comportamento nel sistema.

2.1 DBMessage

La comunicazione fra i nodi di `PariDBMS` è stata concepita e implementata per funzionare tramite stringhe di testo facilmente manipolabili sia dal punto di vista dell'elaborazione da parte del programma, sia per l'utilizzo da parte della componente umana (utente e programmatore) e sia per l'invio all'esterno della macchina come pacchetti nella rete; l'estrema versatilità delle stringhe e il poco controllo che si può imporre su di esse risultano essere, contemporaneamente, un pregio ed un difetto. In un sistema vasto e vario con blocchi dai comportamenti pressochè indipendenti, come quello di `PariDBMS`, l'utilizzo di stringhe può essere vantaggioso per permettere l'introduzione di nuove sfaccettature sulle funzionalità o sulle risposte del sistema, ma deleterio per quanto riguarda la comprensione e l'utilizzo del codice. Se ogni programmatore potesse scrivere e accedere sulle stringhe in modo completamente incontrollato è presumibile che ciascuno di essi assegni delle priorità o nomi diversi ai campi del messaggio; addirittura reputo prematuro introdurre il concetto di campo in un sistema così liberamente strutturato.

Per risolvere il problema della comunicazione è stato necessario strutturare e,

Messaggio programmatore1 blocco Sincronizzazione:

SQL SELECT * FROM Table 1 WHERE Campo1=Campo3

Messaggio programmatore2 blocco NodeManager:

NMG:UPDATE:DBNAME:USERNAME:PASSWORD query HOST TIMESTAMP

Messaggio programmatore3 blocco RemoteAccessProtocol:

RAP USE REMOTE DBNAME:USERNAME:PASSWORD

Figura 2.1: Esempio sistema di messaggistica della vecchia versione

soprattutto, uniformare i messaggi all'interno della classe *DBMessage* partendo da un'analisi completa delle funzionalità e delle necessità di ogni blocco interno al plugin. Sfruttando il sistema già implementato di distribuzione sono riuscito a distinguere i vari gruppi con cui le componenti dei messaggi vanno a ramificarsi all'interno del plugin. Partendo dalla radice si possono contraddistinguere due tipologie di messaggi che vengono aggiunti alla *GlobalMessageQueue*, quelli provenienti da *Console* e quelli che arrivano dalla rete, a questi ho aggiunto una

terza tipologia, ovvero i messaggi in uscita. Quest'ultima è stata presa in considerazione per operare un controllo ulteriore sulla classe *DBMessage* obbligando il programmatore ad etichettare quei messaggi che sono diretti all'esterno del plugin; il *NetworkMessageSender* costringe ad utilizzare quest'ultima dicitura per permettere l'invio dei messaggi, nel caso contrario quest'ultimo risponde lanciando un eccezione e comunicando un messaggio di errore.
Scendendo ad un livello inferiore nella scala gerarchica schematizzata in figura 2.2 troviamo tipicamenti gli identificatori del blocco da cui il messaggio è partito o verso cui è indirizzato, ciascuno di essi ha una sigla che lo contraddistingue permettendo l'inoltro; questa fase dell'instradamento gode il vantaggio di avere una struttura rigida e ben delineata che ha avvantaggiato nell'elaborazione di uno standard dei messaggi. Al livello sottostante lo schema inizia a complicarsi a causa della mancata presenza di un'idea generalizzata di funzionamento della comunicazione all'interno del plugin; in particolare ogni blocco richiede a questo punto una serie di comandi e istruzioni con semantica totalmente diversa l'una dall'altra come si è potuto notare nell'esempio precedente di figura 2.1. Se andiamo a vedere il codice la situazione non può altro che peggiorare, ogni blocco utilizza un metodo diverso di estrazione dei campi della stringa; *substring e split*[1] i metodi più utilizzati. La possibilità di commettere errori di lettura delle componenti aumenta se consideriamo la necessità frequente di ottenere i valori contenuti; si è potuto notare come perfino nelle stesse classi, prodotti dagli stessi progettisti, vi sia la presenza di metodologie di estrazione completamente differenti e poco performanti riferita agli stessi messaggi. Utilizzando le ricerche sulle funzioni dei singoli campi riadattati per ogni blocco e basandomi sulla struttura ad albero in figura 2.2 costruita osservando il codice e le direzioni che assumevano l'inoltro dei messaggi è stata definita un primo esempio dei campi che andranno a costituire la classe *DBMessage*. Procedendo dalla radice il campo che contraddistingue la provenienza (o la direzione) del messaggio è costituito dall'Header, questo presenta solamente tre valori, CONSOLE, NETWORK e SENDER; i primi due stabiliscono la provenienza e sono settati tipicamente dall'omonima classe nel caso di CONSOLE o dall'arrivo dalla rete *PariPari* nel caso di NETWORK; SENDER invece contraddistingue un messaggio costruito in locale che ha come destinazione un nodo esterno alla rete.
Prima di iniziare a entrare nel dettaglio sugli altri campi è necessario fare una premessa sulla tipologia delle variabili assegnate al *DBMessage*; nonostante si

[1] vedi la javadoc al sito internet http://download.oracle.com/javase/1.4.2/docs/api/java/lang/String.html per dettagli funzionali

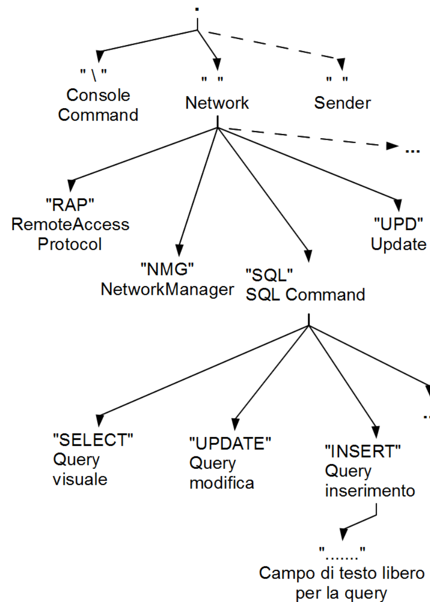


Figura 2.2: Ricostruzione dei percorsi dei messaggi

sia deciso di mantenere le stringhe come tipo di dato da inviare al di fuori del nodo, all'interno del plugin questa decisione causa l'instabilità precedentemente descritta senza porvi una soluzione, se non apparente. Gli *Enum*, riduzione di Enumeration, presentano una soluzione decisiva alla formazione della classe messaggi, in particolare questi sono dei tipi di dato costituiti da set di valori nominali chiamati "elementi"del tipo; gli enumeratori permettono di associare costanti fra loro (ad esempio il numero"01"coincide con la stringa CONSOLE e coindice con il carattere "\").
La classe HeaderEnum permette di segnalare come errore di sintassi la formazione di un messaggio che utilizzi un Header diverso da quelli precedentemente elencati; i programmatori di `PariDBMS` futuri saranno vincolati a rispettare questo elenco offrendo però la possibilità di arricchirlo con ulteriori significativi elementi. Ora che abbiamo uno strumento di controllo sui messaggi che circolano possiamo procedere con la schematizzazione dei campi successivi. Segue in ordine di inserimento TypeEnum, il quale identifica il blocco a cui il messaggio è direzionato, CommandEnum, invece, è il campo che contraddistingue il comando utilizzato in relazione al tipo di blocco a cui è affidato; questo sistema opera un secondo controllo da parte della classe *DBMessage* sull'affidabilità dei token inseriti. Alla creazione, dopo aver confermato l'esistenza del TypeEnum utilizzato, il messaggio viene sottoposto ad un ulteriore controllo di sbarramento per gli errori effettuato

da un SELECT, il quale verifica che il campo successivo relativo al comando sia effettivamente associato al corretto blocco. In ordine di comparsa, procedendo nella lettura del messaggio, seguirebbe il campo Argument che permette l'inserimento di qualsiasi stringa di testo da parte del programmatore; è una libertà concessa per permettere la gestione e l'inserimento di tutte le stringhe di errore, le query di interrogazione e i messaggi di risposta con i risultati; quest'ultimi in particolare possono assumere i più disparati valori da visualizzare all'utente. In coda, tipicamente nei messaggi provenienti dalla rete, viene concatenato al campo Argument l'Host di provenienza del messaggio, campo utilizzato per tenere traccia del nodo origine delle richieste e per poter poi inoltrare le risposte alla corretta destinazione. Nell'utilizzare e riprogettare i token dei messaggi e le varie

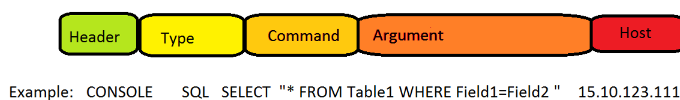


Figura 2.3: Messaggio strutturato con Enum

funzionalità si è notato come all'interno del campo Argument alcune sequenze di caratteri particolari continuino a ripetersi nelle stesse posizioni all'interno della stringa; in particolare mi riferisco al nome del database, all'utente che l'ha creato e alla password di accesso ad esso riferita. Questi tre elementi sono inseriti tipicamente all'inizio del campo contenente l'argomento libero e sono separati dal carattere ": ",essi servono a identificare il database destinazione attraverso i primi due (chiavi primarie della tabella *DATABASES*) e l'ultima che consente e certifica l'accesso alla base di dati richiesta. Nel corso dello sviluppo del codice è apparso sempre più evidente il bisogno di attribuire uno spazio riservato anche a questi campi in modo da migliorarne il riconoscimento e, con un'ispezione maggiore nella creazione del *DBMessage*, operarne dei controlli sui caratteri contenuti per le stringhe di identificazione. Attualmente il passaggio a questo ultimo protocollo di messaggistica non è ancora stato completato a causa della necessità di operare dei test di funzionamento del codice necessari al progredimento di ulteriori parti del progetto e alla correzione di bug di sistema; ciò non esclude però l'enorme utilità dell'introduzione di una struttura ben definita e generalizzata per la creazione dei messaggi. Allo stato attuale, alla creazione del *DBMessage*, il metodo costruttore permetterà di operare un controllo sui campi bloccando im-

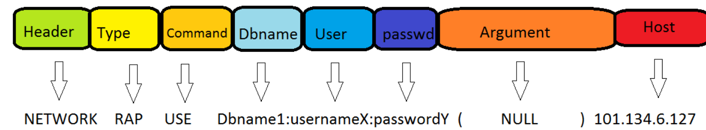


Figura 2.4: Messaggio Enum con l'aggiunta dei campi identificativi

mediatamente eventuali errori di sintassi sui messaggi; da questo punto in poi sarà sicuro e certo che il blocco e il comando relativo al messaggio sono delle funzionalità realmente implementate all'interno del plugin `PariDBMS`. Anche l'inoltro del messaggio gode di ulteriori semplificazioni, infatti, grazie all'introduzione degli Enum come sistema di categorizzazione dei campi, il codice scritto ottiene una maggiore chiarezza con l'utilizzo del costrutto di SELECT per l'instradamento dei messaggi, grazie alla sua capacità di confrontare numeri, in sostituzione al sistema di IF-ELSE precedente che necessitava di continui confronti e non permetteva una facile indentazione e schematizzazione. Gli Enum finora introdotti

```
IDBMessage message = messages.get();
    TypeEnum type = (TypeEnum) message.getType();
    switch (type) {
    case CONS_COM :
        consoleCommandInterpreter(message);
        break;
    case SQL :
        sqlInterpreter(message); //metodo di interpretazione messaggi SQL
        break;
    case NMG :
        nmgInterpreter(message); //metodo di interpretazione messaggi NodeManager
        break;
    case SYNCHRO :
        synInterpreter(message);
        break;
    case RAP :
        rapInterpreter(message);
        break;
    case USER :
        userInterpreter(message);
        break;
    case AUT :
        autInterpreter(message);
        break;
    case UPD :
        updInterpreter(message);
        break;
    default :
        DBMSConsole.println("
                MessageInterpreter : Unhandled message: " + message.toString(), 0);

    }
```

Figura 2.5: Codice del MessageInterpreter per instradare il messaggio a seconda del TypeEnum

e le combinazioni permesse dal *DBMessage* sono qui strutturate con una tabella in figura 2.6 al fine di consulto anche per i programmatori futuri del plugin.

Header	Type	Command	DB:User:Pwd	Arg	Host
CONSOLE	RAP	-USE -END USE -SQL -RESPONSE -REF -USER REF -ACC -CONF	String:String:String	String	X.X.X.X
	SQL	-CREATE TABLE -DROP TABLE -SELECT -UPDATE -INSERT -CREATE DB -REMOTE			
NETWORK	NMG	-ADD NODE -ADD DB -DEL DB -REP REQ -REP REF -REPLICATION -REP FAILURE -REP ACC			
SENDER	SYNCHRO	-CREATED -REQUEST SENT -ACKED -NACK -CONFIRMED -ABORTED -UPDATING			
	UPD	-RES CONFIRM -RES UPDATE -RES TQ -RES END UPDATE -RES END DELETE -RES END ERROR			
	AUT	-ACC -REF			

Figura 2.6: Lista Enum disponibili per la composizione del DBMessage

2.2 RemoteAccessProtocol (RAP)

La necessità di offrire un sistema per permettere all'utente di utilizzare basi di dati, non direttamente contenute in memoria locale, ha portato allo sviluppo del *Protocollo di Accesso Remoto*. Prima di esso non era possibile accedere ad un database che non fosse situato nella memoria locale; essendo impossibile controllare l'evolversi delle replicazioni nei nodi e sapere a priori la destinazione delle copie, se si perdeva il riferimento al proprio database in locale a causa dei sistemi di cancellazione (caso database troppo vecchio) si perdeva anche la possibilità di accedere al proprio database creato. Attraverso il refactoring di quasi tutte le classi di `PariDBMS` e il progredire di alcune di esse, anche la connotazione e lo scopo principe di questo blocco hanno subito delle ridefinizioni notevoli.
Al fine di rendere più chiaro ed esplicativo il passaggio dalla versione antecedente a quella attuale presenterò il funzionamento del blocco RAP sviluppato da Deniz Tartaro evidenziandone i punti salienti e le debolezze di struttura incontrate.
Per completezza e chiarezza espositiva viene presentata anche una classe che partecipa indirettamente ad ogni tipo di connessione, sia essa interna o remota, ad una base di dati; si tratta della classe *ConnectedTo* la quale si occupa di gestire l'esclusività della connessione di un nodo. Come in ogni terminale degli ambienti lavorativi, con cui è possibile accedere ad un'unica base di dati alla volta, allo stesso modo è stato deciso di vincolare le connessioni ai database della rete obbligando l'utente ad effettuare un accesso alla volta; quando questa procedura viene effettuata si passa da uno stato di partenza di disconnessione, con la variabile di stato settata su NOT_CONNECTED, alla fase di connessione aggiornando la variabile al valore di CONNECTED. Questa classe ha subito altrettante modifiche e cambiato il significato di alcune sue variabili nel corso dell'upgrade delle funzionalità; al proseguire della spiegazione del funzionamento passato e futuro verranno messe a confronto le modifiche apportate.

Versione obsoleta

Per spiegare il funzionamento delle varie versioni verrà dettagliato passo passo le procedure attuate dall'utente per accedere ad una base di dati remota e l'effetto sul codice che questo comporta; le classi utilizzate come comprimarie fanno tutte parte della vecchia versione, questo per maggiore coerenza nei confronti del reale utilizzo del protocollo.

Fase 1

L'avvio della connessione avviene tramite l'inserimento delle credenziali nella Console sulla riga di comando con la sintassi di riconoscimento "\ use DB-NAME:USERNAME:PASSWORD ". Il messaggio viene inserito nella *GlobalMessageQueue*(GMQ), quest'ultimo viene notificato all'arrivo del messaggio risvegliando, se sopito, il *MessageDelivery*(MD); il thread estrae dalla coda il messaggio e, poichè inizia con il carattere speciale '\', viene aggiunto alla coda privata del *CommandInterpreter*(CI) il quale procede nell'analisi dei campi successivi del messaggio per eseguire le istruzioni appropriate. Questo avviene solo nel caso in cui il nodo non sia connesso ad alcun database, sia esso locale o meno, tramite il controllo della variabile *Status* all'interno dell'oggetto *ConnectedTo*; verifica che si palesa alla restituzione di un valore diverso da CONNECTED.
Quando CI legge il comando di "use" si attiva a recuperare nella stringa le credenziali di accesso al database composte da una terna di valori separati dal carattere ':'(nessun nome di database , username e password potrà contenere questo carattere al suo interno per evitare errori di lettura); i primi due valori che identificano univocamente la base di dati nella rete, DBNAME e USERNAME, vengono utilizzati come parametri di ricerca di risorse attraverso il modulo della cerchia interna DHT il quale si adopera a trovare tutti i nodi della rete interna di *PariPari* e ritorna come risultato un array di stringhe contenente tutti gli indirizzi IP. Se questo vettore ha dimensione nulla all'utente viene segnalato che il database cercato con quelle credenziali non esiste, altrimenti, per dimensionsi maggiori, viene passato ogni elemento dell'array alla ricerca della presenza del nostro stesso ip. L'esistenza di quest'ultimo significa che la base di dati richiesta è contenuta nel nodo locale e viene, per questo motivo, operato un controllo sullo stato della base di dati; se essa presenta uno stato diverso da READY (stato che indica la replica attiva, aggiornata e funzionante) o il nodo stesso risulta già impegnato in una connessione (ConnectedTo settato ad un valore diverso da NOT_CONNECTED) allora è comunque richiesto il *Protocollo di Accesso Remoto* per accedervi. Esiste anche il caso in cui, supposto trovato il nodo su cui accedere in remoto e appurate le condizioni di connessione, l'accesso alla base di dati venga rifiutato a causa della errata verifica dei dati personali immessi dall'utente (generalmente la password passata nella stringa non corrisponde all'utente designato come possessore del riferimento alla base di dati); questo caso porterebbe, semplicemente, a ricominciare il protocollo di connessione informando l'utente dell'errata digitazione dei parametri di accesso con una comunicazione a video.
Supposto il corretto inserimento delle credenziali, supposto lo stato del nodo coer-

ente alla comunicazione e l'esistenza di un nodo di destinazione accessibile, si può procedere con il *Protocollo di Accesso Remoto*; il *CommandInterpreter* si adopera a chiamare in questa fase il metodo *startRemAccPr()* che si occupa di creare una nuova istanza del thread *RemoteAccessProtocol* e lanciarlo come *PariPariThread* (questa tipologia di thread verrà illustrata in dettaglio nel paragrafo Gestione Thread). A questo punto si conclude la fase di avvio del blocco da parte del lato Client della connessione; per facilitare la comprensione del testo si è deciso di denominare il nodo che richiede l'accesso remoto come Client, mentre il nodo che offre il servizio viene definito Server, la nomenclatura così definita è solo a fini espositivi, ogni nodo funge nella rete PariPari sia da client che da server secondo i principi del peer-to-peer.

Fase 2 lato Client

La seconda fase inizia quando CI lancia nel Client il thread *RemoteAccessProtocol* (RAP) e si addormenta indefinitamente in attesa di una notifica nella sua coda messaggi. RAP è una classe presente in tutti i nodi di `PariDBMS` e contiene al suo interno porzioni di codice che vanno ad essere eseguite in base al fatto che il protocollo sia stato avviato come client o come server; ciò che determina la funzione del nodo è determinata dal tipo di chiamata sul metodo costruttore *startRemAccPr()* effettuata. Se il RAP viene lanciato dal *CommandInterpreter* e con i valori dei campi privati settati e coerenti (diversi da null) allora è eseguito come istanza client, altrimenti, se i campi sono tutti nulli, l'esecuzione è attiva con la funzione di server.
Il Client procede con l'esecuzione del metodo privato *requestRemoteUse()* che si adopera a impostare a CONNECTING lo stato del nodo per bloccare la console e non permettere ulteriori proposte di connessione. A questo punto inizia il primo tentativo di instaurare la comunicazione tramite il messaggio "rap use remote DBNAME:USERNAME:PASSWORD" (questa stringa appartiene alla vecchia versione dei messaggi) che viene inviato al primo indirizzo contenuto nell'array di ip precedentemente acquisito; al termine dell'invio il thread si addormenta in attesa della risposta del nodo destinatario con un timeour T della durata di alcuni secondi.
L'attesa si conclude allo scadere del timeout tramite la ricezione dei messaggi di "ref", in caso di rifiuto e impossibilità del nodo server di procedere nel protocollo, o di "acc", in caso di conferma; l'iterazione dei vari ip procede fintanto che un nodo non risponde con un messaggio che stabilisce la possibilità di

collegamento. Con l'arrrivo del messaggio di "acc" la connessione con la destinazione è confermata e lo stato del nodo può essere impostato a CONNECTED; il Client attiverà a questo punto l'istanza della classe di elaborazione delle query *RemoteQueryManager* (RQM) eseguita anch'essa come PariPariThread.

Fase 2 lato Server

La fase 2 dal lato Server inizia nel nodo destinatario con la ricezione della richiesta "rap use remote DBNAME:USERNAME:PASSWORD"; la classe *NetworkMessageReceiver* aggiunge alla coda *GlobalMessageQueue* il messaggio, dalla quale *MessageDelivery* lo estrae per inoltrarli alla parte correta del plugin. Nel momento in cui MD legge il token "rap" sa che il suo destinatario è la classe omonima; inoltre, se il messaggio prosegue con la stringa "use" allora vuol dire che è arrivata la richiesta di accesso remoto da un Client e quindi si deve far partire il thread RAP perché possa iniziare anche sul nodo locale (il Server del protocollo) la seconda fase. Se il nodo è disponibile a cedere l'utilizzo esclusivo del database richiesto, attraverso vari controlli di verifica, allora il nodo candidato risponderà con un messaggio di "rap acc", altrimenti con un "rap ref" ("rap user ref" nel caso in cui le credenziali di riconoscimento siano errate).
Supposti il database e il nodo a disposizione al proseguimento del protocollo, viene invocato il metodo *startRQM()*, il quale crea un'istanza server del thread *RemoteQueryManager* (RQM) e ne avvia l'esecuzione. A questo punto il RAP lato server si mette in attesa sulla chiamata sospensiva wait() e cede il controllo e il proseguimento della gestione del protocollo al RQM.

Fase 3 lato Client

La fase 3 gestisce lato client l'invio delle query e l'attesa della ricezione del risultato dalla rete; il *CommandInterpreter* risvegliato è da questo punto in poi abilitato a ricevere query da console da far eseguire sul database desiderato. *RemoteQueryManager* inizia la sua esecuzione rimanendo immediatamente in attesa di ricevere nella sua coda le query da inviare; il prefisso che contraddistingue i messaggi appartenenti a questa fase è "rqm". Da parte dell'utente, tuttavia, non è richiesto nulla più che il mero inserimento delle query, sarà *MessageDelivery* che associerà la scrittura a console ad un messaggio diretto ad RQM tramite l'ausilio dei metodi offerti da *ConnectedTo*(un esempio è *isLocalConnected()* che ritorna false nel caso di connessione remota).
Il messaggio viene rielaborato da *RemoteQueryManager* nella forma "rqm sql

remote QUERY ” e spedito tramite *NetworkMessageSender* al nodo server. A questo punto si attende tramite l'ausilio di un timeout di qualche secondo una risposta da parte del server che conferma l'arrivo della query tramite un messaggio di “rqm acks”; se ciò avviene entro il tempo limite il Client procede con l'inviare un secondo messaggio di “rqm start” che segnala l'intenzione del nodo di procedere con l'esecuzione effettiva della query inviata all'inizio del protocollo. L'attesa della risposta alla query è limitata ad un secondo timeout con ordine di grandezza maggiore per compensare i maggiori tempi di elaborazione di un risultato e del suo invio; il messaggio in arrivo ha la struttura “rqm result RISULTATO” dove l'ultima parte è la rappresentazione testuale della vista della tabella ottenuta, un messaggio di errore o la conferma dell'esecuzione dell'istruzione. A questo punto basta semplicemente presentare la soluzione a video e inoltrare una conferma di avvenuta ricezione al Server tramite il messaggio “rqm ackr” e addormentare il thread in attesa della query successiva. Ogni scadenza dei timeout precedentemente elencati a causa di congestione della rete o disconnessione di uno dei nodi comporta la terminazione dell'intero protocollo e necessita, se si desidera riprendere la connessione, di ricominciare dalla fase 2.

Fase 3 lato Server

RemoteQueryManager lato server inizia la sua attività quando riceve nella coda privata un messaggio inoltrato dal nodo Client contenente una query da elaborare; le istruzioni da eseguire corrispondo al campo finale del messaggio “rqm sql remote QUERY” e vengono estratte e salvate in una variabile privata della classe. A questo punto viene inviato al Client un messaggio di ricezione acquisita “rqm ackq” e si attende una risposta contenente il permesso di iniziare l'elaborazione della query. Nel caso in cui non vi sia riscontro da parte del Client allora la query non viene eseguita e il protocollo cade chiudendo la connessione; se però la risposta “rqm start” viene inviata allora il nodo Server procede all'esecuzione della query che verrà affidata al blocco di sincronizzazione per l'elaborazione e, in caso di operazione andata a buon fine, a catturarne il risultato in una stringa da incorporare nel messaggio di risposta. Il blocco di sincronizzazione prevede, in questo caso, di conoscere solamente il nome del database, inoltre la stringa da inserire nell'elaborazione locale non contiene “sql remote” come token di riferimento, ma solamente la sottostringa rimanente; *Transaction* si adopera a costruire la stringa RESULT a seconda non solo della risposta, ma anche dal tipo di query inserita(select o aggiornamento). Al termine di questa operazione viene inviata la risposta(RESULT) al Client inserendola in coda alla stringa “rqm

result" al quale il nodo destinatario risponde a sua volta con un messaggio di acquisizione "rqm ackr" che segna la fine della sessione di query. Questa fase viene ripetuta ogni volta che il nodo Client decide di inserire una query, è quest'ultimo in particolare a decidere quando e come interrompere la connessione instaurata.

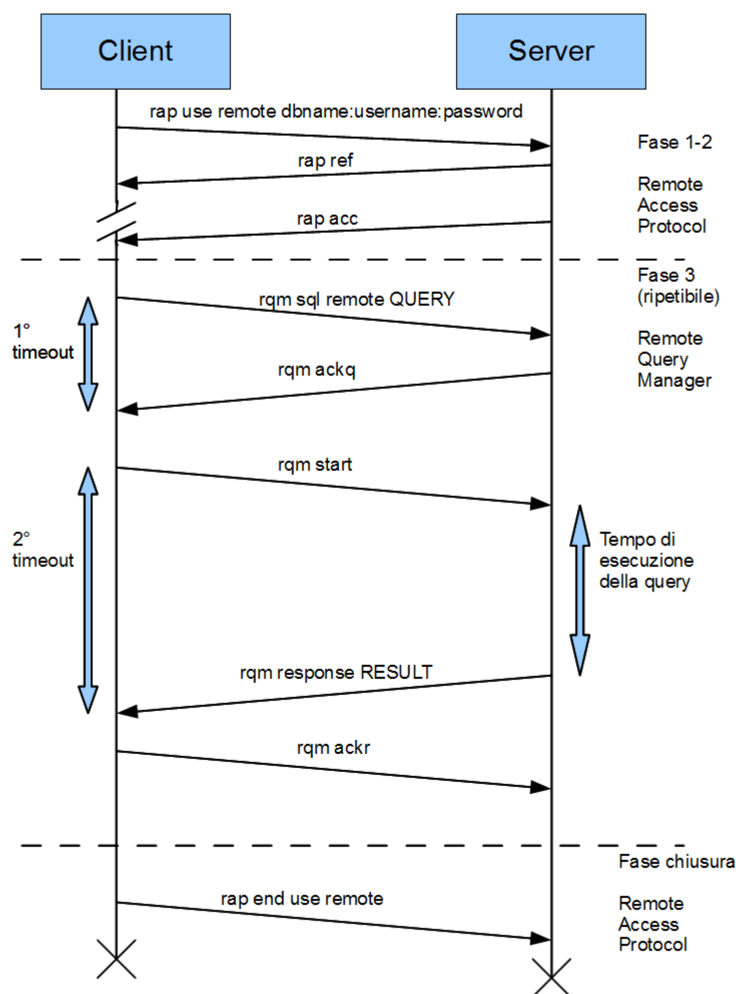


Figura 2.7: Funzionamento vecchia versione di Rap

Analisi funzionale

Analizzando nel dettaglio questa struttura per il *Protocollo di Accesso Remoto* ho riscontrato una serie di punti deboli e inefficienze che ho deciso di evidenziare in queste righe per motivare la scelta di riscrivere l'intero blocco e il relativo

schema. Innanzitutto nella sperimentazione del codice ho notato che la struttura del *ConnectedTo* permette, se opportunatamente rielaborata e utilizzata, di simulare una mutua esclusione all'utilizzo delle risorse dei database. Alla luce dei fatti ho deciso di analizzare meglio nelle varie fasi di richiesta di connessione anche lo stato di transizione CONNECTING, stato che dovrebbe bloccare ulteriori accessi fino al momento in cui decide di passare ad uno stato maggiormente consistente (CONNECTED o NOT_CONNECTED). Da questo punto in poi tutte le connessioni transitano da NOT_CONNECTED a CONNECTING, bloccando tentativi di collegamento altrui, e da CONNECTING a CONNECTED, ufficializzando definitivamente la connessione e settando il campo *ConnectedTo* con i parametri del database e del nodo che ne ha richiesto la connessione. Questa impostazione di base è stata utilizzata per eliminare la presenza di un'ulteriore debolezza del sistema. Ogni connessione remota bloccava, infatti, ben due nodi, sia quello Client che quello Server; entrambi segnavano come attiva la loro connessione all'avvio del protoccol di accesso remoto; il client richiamava il metodo *connectionTo[Remote/Local]DB()* di *ConnectedTo*, mentre il server richiamava il metodo *connectionFromRemoteDB()* che ne segnalava la connessione dall'esterno. Eliminare questo vincolo non ha richiesto molte righe di codice, ma ha preteso un attento lavoro di integrazione e di approfondito studio per fare in modo che ogni nodo potesse offrire un numero di supporti di *RemoteAccessProtocol* lato server non limitato superiormente (pur mantenendo il vincolo di una connessione attiva per nodo come regola generale di funzionamento).
Procedendo nell'analisi del codice ho notato come la non suddivisione e distinzione delle classi Client e Server abbia portato il nascere nel codice di anomalie, soprattutto in avvii consecutivi della stessa porzione di programma. Il thread in alcuni casi in cui doveva riconoscersi come Server, si riconosceva come Client (e viceversa); inoltre la mancata separazione non permetteva di avviare nella stessa macchina il protocollo Server e Client in contemporanea aggravando i vincoli di utilizzo dello stesso RAP. La struttura del codice, benchè non fosse una motivazione di tipologia funzionale, era stata considerata troppo confusionaria e approssimativa per rimanere accorpata sia lato client che server nella stessa classe, perciò, come conseguenza, ho deciso di ampliare la struttura delle classi del protocollo di accesso remoto in *RemoteAccessProtocolClient (RapClient)* e *RemoteAccessProtocolServer (RapServer)*. In contemporanea all'apparire di queste due classi ho provveduto all'eliminazione della classe *RemoteQueryManager* inglobando la parte funzionalmente utilizzata dal Client nel *RapClient* e, rispettivamente, la parte riguardante il Server nel *RapServer*. Questa scelta adottata mira

a mantenere il numero di thread limitato a due (uno operante nel Client e uno nel Server) per il funzionamento del protocollo, in particolare ho considerato che l'utilizzo del RQM diventava pressochè inutile ai nostri scopi, anzi, mantenerlo attivo avrebbe complicato i metodi di instradamento e ricezione dei messaggi. Con la scomparsa del *RemoteQueryManager* ho provveduto alla ridefinizione dei messaggi e della loro struttura nel modo descritto nella tabella 2.6 sfruttando il facile instradamento offerto dagli Enum e dai campi contenenti le credenziali necessarie.
Nella versione precedente ogni errore, caduta di connessione o ritardo per intasamento della rete veniva gestita frettolosamente dal sistema con la chiusura della connessione e la notifica all'utente; la mia intenzione nel nuovo protocollo è stata quella di tentare di recuperare il collegamento cercando di acquisire, nel caso di scadenze dei vari timeout, nuovi nodi con cui riavviare il protocollo e mantenere salvate le informazioni riguardo la risoluzione di query. E', in particolare, la messaggistica e il flusso di comunicazione della parte che precedentemente riguardava l'RQM ad aver subito una maggiore ridefinizione. Nella vecchia versione il sistema di elaborazione delle query necessitava di uno scambio numeroso di messaggi riassunto nella figura 2.7; un totale di cinque messaggi per l'invio di una interrogazione e la ricezione della risposta con tempi di attesa relativamente lunghi. Ho deciso di riscrivere il metodo di risoluzione di query remote utilizzando il metodo più semplice e veloce costituito da due messaggi, "rap query", per l'invio, e "rap response",per la ricezione. Sostituzione intrapresa per motivi riguardanti principalmente le scadenze temporali del sistema; l'attesa dei messaggi intermedi e la loro rielaborazione rallentavano enormemente la comunicazione. Oltre alle tempistiche ho notato che i messaggi intermedi risultavano inutili al protocollo; prima di tutto, non essendo sotto il controllo dell'utente, non potevano essere fermati e quindi il messaggio di "start" rappresentava un secondo avvio senza forma di gestione; in secondo luogo la presenza di questi messaggi non permetteva alcun sistema di recovery delle query per introdurre punti di ripristino. Se un nodo si disconnetteva non c'era modo per sapere se la query era stata eseguita o meno, oppure se era avanzata a qualche grado intermedio; gli effetti erano esclusivamente visibili tramite ulteriori interrogazioni o tramite l'osservazione dei file di log e del progredire del timestamp.
Ulteriori modifiche e migliorie saranno introdotte con la spiegazione del funzionamento dellla versione del protocollo *RemoteAccessProtocol* da me implementata permettendo di chiarire con la descrizione operativa i perfezionamenti introdotti.

Nuova versione

Come per la versione di Tartaro, procedo con la descrizione del funzionamento del protocollo avanzando per fasi che, pur non essendo così distinte l'una fra l'altra, permettono una facile comprensione del flusso di istruzioni eseguito.

Fase 1 Avvio connessione

L'utente procede nell'utilizzo, tramite la *DBMSConsole*, di una base di dati con il solito comando "\use dbname:username:password ", la stringa così introdotta viene immediatamente riscritta sotto la forma più manipolabile di un *DBMessage* e aggiunto alla GMQ; il *MessageInterpreter* (MI) è la classe adibita al corretto instradamento del messaggio, il primo simbolo corrisponde all'HeaderEnum CONSOLE e "use" è il comando della console che richiama il metodo *cmdUse(messaggio)* (il messaggio viene sempre passato tra i vari metodi, virtualmente potrei creare più istanze della classe *MessageInterpreter* per ottenere un maggiore parallellismo di elaborazione); a questo punto ha inizio l'avvio della connessione al database identificato dai parametri inseriti e, perciò, vengono bloccati ulteriori tentativi attraverso il setting delle variabili della classe ConnectedTo che passa a CONNECTING (lock della connessione) .
La connessione al database può avvenire in modalità diverse a seconda delle condizioni del nodo; non è l'utente che richiede una connessione remota, ma è il nodo che, accortosi dell'indisponibilità del database in locale, procede ad effettuare il collegamento tramite il protocollo di RAP. I confronti di disponibilità della risorsa vengono effettuati interrogando il *SupportDB* per verificare la presenza e mettersi in contatto con il relativo riferimento al *NodeSet*; quest'ultimo permette di informarsi sullo stato della base di dati; a meno che non sia READY ogni tentativo di comunicazione locale verrà interrotto.
Questo sistema di gestione della copia locale permette di bypassare i limiti di tempo introdotti con l'utilizzo di DiESeL, infatti quest'ultimo, dopo l'avvio di DHT, necessita di qualche minuto di tempo per connettere il nodo a tutte le nuvole delle basi di dati a cui fa riferimento; inoltre è previsto il procedere dell'aggiornamento delle copie in locale per l'allineamento con quelle in rete, in questo frangente di tempo i database locali sono ad uno stato di UPDATE che non ne permette l'utilizzo. Con l'introduzione del RAP è possibile connettersi ad una qualsiasi base di dati READY mentre il database in locale è ancora UPDATE; una piccola miglioria(non ancora implementata) consiste nel verificare che, alla fine degli UP-

DATE, un database preferisca tornare all'utilizzo locale piuttosto che procedere in remoto diminuendo il traffico di messaggi in rete.
Nel caso di fallimento della connessione o della ricerca in locale, viene rilevata la presenza di ulteriori nodi all'interno della rete `PariDBMS` tramite l'ausilio di DHT con chiavi di ricerca dbname e username; la ricerca si conclude con il ritorno di un array di indirizzi ip che, se diverso da null, indica l'esistenza effettiva del database identificato dalla coppia dbname-username.

Fase 2 lato Client

A questo punto viene fatto partire il thread *RapClient* con il metodo *startRemAccPr()* a cui viene fatto passare la coppia dbname-username e l'iteratore della lista di ip; quest'ultimo elemento viene utilizzato dal protocollo RAP per l'inoltro dei *DBMessage* di richiesta di connessione così costituiti : "RAP USE dbname:username:password ". L'invio di questi non è stato trattato in modo sequenziale come nella versione precedente, ma si è cercato di inserire un certo grado di competività spedendo contemporaneamente a tutti i nodi del vettore ip una copia del messaggio; il primo che risponde con un messaggio di "RAP ACC" viene ufficializzato come HostServer del protocollo di RAP e la connessione può ritenersi confermata con l'invio del *DBMessage* "RAP CONF" il cui utilizzo si vedrà meglio nell'analisi lato server.
Ho cercato di progettare un sistema il più possibile tollerante ai guasti e che si adattasse al rapido variare dei nodi nella rete; per realizzarlo ho previsto una serie di risposte da parte del *RapClient* a seconda dei tempi e dei messaggi della rete. Se nell'attesa il sistema riceve da un host il messaggio "RAP REF" ciò non preclude l'esistenza di ulteriori nodi con cui connettersi e perciò rimane nuovamente in attesa sul timeout T; tutt'altra questione è la ricezione del *DBMessage* "RAP USER REF" il quale rappresenta la notifica di un errore nell'inserimento delle credenziali e quindi l'interruzione completa del RapClient con la notifica a video di errore sulle credenziali. La scadenza del tempo T di attesa di un messaggio da parte degli Host è sempre interpretata come una fine connessione del protocollo per permettere la ricerca di nuovi nodi con DHT.
A questo punto il Client procede nel suo flusso di programma attendendo nuove messaggi da parte dell'utente e concludendo così la fase 2.

Fase 2 lato Server

Il protocollo di accesso remoto si attiva nel nodo lato server con l'arrivo del messaggio "RAP USE dbname:username:password", ma, a differenza della versione precedente, vi è un supporto intermedio che si occupa di gestire le nuove funzionalità introdotte.
Tolto il limite che obbligava i nodi ad offrire un servizio server remoto esclusivo è nato il bisogno di avere un supporto software con capacità di governo ed instradamento sui vari accessi remoti; questo layer intermedio denominato *RapSetServer* è stato da me implementato per supportare la creazione, l'avvio e il bloccaggio dei vari *RapServer*. Il *RapSetServer* contiene i riferimenti delle classi in una lista concatenata e si adopera al corretto inoltro in base alle credenziali del messaggio in arrivo; in particolare il modo di agire della classe è di creare un nuovo *RapServer* e di farne partire il relativo thread se riceve in ingresso un messaggio in cui la terna di parametri di identificazione composta da host, dbname e username non è stata riconosciuta in alcun *RapServer* già avviato in precedenza e contenuto nella lista privata. Se, a differenza di quanto appena scritto, il *RapServer* con le credenziali inviate esiste già ed è avviato il messaggio verrà semplicemente inoltrato nella relativa coda privata. La scelta di utilizzare anche l'host come identificatore è stata introdotta per permettere l'accesso ad una stessa base di dati da parte di più nodi e permettere comunque di ottenere una ricezione del messaggio priva di ambiguità.
Riprendendo dall'arrivo del messaggio dal client di avvio, supponiamo di essere nella condizione iniziale per cui non ho alcun accesso remoto avviato in locale, a questo punto il messaggio instradato tramite i metodi del *MessageInterpreter* viene passato alla classe *RapSetServer* che, estratti i campi, avvia il thread di *RapServer* apposito depositando immediatamente il messaggio nella coda privata appena creata. La connessione, a questo punto, non è ancora stata completata ma necessita un controllo di disponibilità del database locale, in base a quest'ultimo e allo stato in cui si trova si possono ottenere tre distinte risposte da inviare al client in ascolto; nel caso in cui l'esistenza della base di dati non sia più pervenuta (caso di cancellazione non notificata a DHT o DiESeL) la risposta è un messaggio di "RAP REF" che permette al client di attendere le risposte di altri nodi. Nel caso in cui il database esista viene effettuato un controllo sulle credenziali per il riconoscimento degli utenti, in caso di esito negativo viene risposto con il messaggio di "RAP USER REF" che interrompe il funzionamento del client, altrimenti si procede con un "RAP ACC" che indica la disponibilità del nodo a procedere con il protocollo.

Dall'invio di questo ultimo messaggio il thread si mette in attesa con timeout di una risposta del client di “RAP CONF”; la scelta di utilizzo di questo protocollo di 3-way-handshake[2] nasce dal fatto che il *RapClient* ha inoltrato lo stesso messaggio a tutti i nodi della sua lista e il *RapServer* nel nodo in nostro possesso ha bisogno di una conferma dal Client per ritenersi effettivamente coinvolto. Se la conferma non è ricevuta in tempo o non arriva il protocollo lato server sul nodo questo termina il protocollo (probabilmente un altro nodo candidato è stato scelto al posto di questo), altrimenti, nel momento in cui arriva il messaggio, la connessione è effettivamente stabilita.
A questo punto il nodo server è pronto a ricevere istruzioni e rimane in attesa di un messaggio dell'utente contenente query o ulteriori comandi di funzionamento.

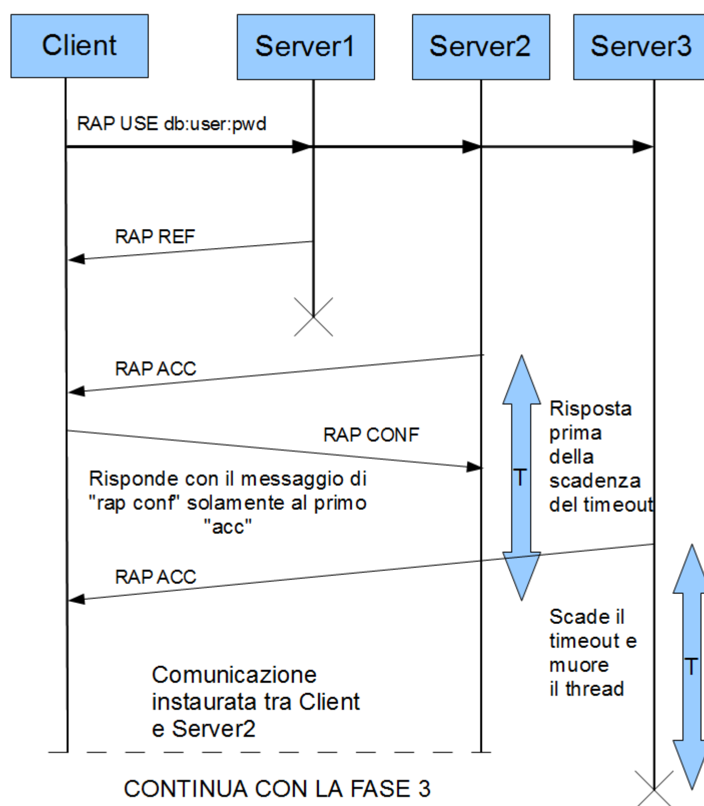


Figura 2.8: Protocollo 3-way handshake per la connessione al Server

[2]protocollo a 3 fasi tipicamente utilizzato nelle connessioni TCP, fase 1 invio messaggio,fase 2 risposta, fase 3 conferma

Fase 3 lato Client

Il lato client della terza fase corrisponde, come per la versione di Tartaro, alla scrittura dell'utente di query e di istruzioni da inviare al server. Essendo scomparsa la classe di *RemoteQueryManager* è compito del *RapClient* gestire sia i messaggi provenienti da console, sia quelli provenienti dalla rete per gestire le query; a tal proposito, essendo uniformate sotto lo stesso Enum, è sufficiente effettuare una ricerca sull'HeaderEnum per individuare la provenienza e agire di conseguenza.
I messaggi attivi(nel senso che implicano una risposta del software) previsti in questa fase sono in sostanza quattro; uno di essi corrisponde alla chiusura della connessione tramite la volontà dell'utente di interrompere la comunicazione; basta la digitazione del comando "disconnect" da console ad attivare l'invio al *RapClient* del messaggio "RAP END USE", questo messaggio, oltre ad avviare la procedura di chiusura locale, permette di notificare al server la fine del protocollo.
Ovviamente un messaggio di quelli permessi in questa fase è dedicato all'invio di query per l'esecuzione a remoto. Per rendere invisibile all'utente la differenza tra accesso remoto e locale si è deciso di utilizzare la stessa sequenza di caratteri testuali per definire l'inserimento di query a console. Il MI elabora la stringa (riconoscendo tramite la classe *ConnectedTo* che la connessione è avviata in remoto) traslandone le istruzioni (la stringa SQL che identifica solitamente il tipo diviene il comando del nuovo messaggio "RAP SQL" traslando di una unità tutti i campi) e inoltrandolo al *RapClient*; quest'ultimo riconosce il messaggio come query SQL e lo inserisce nella propria coda privata di istruzioni.
Si è cercato di mantenere una certa solidità di funzionamento al progetto mante-

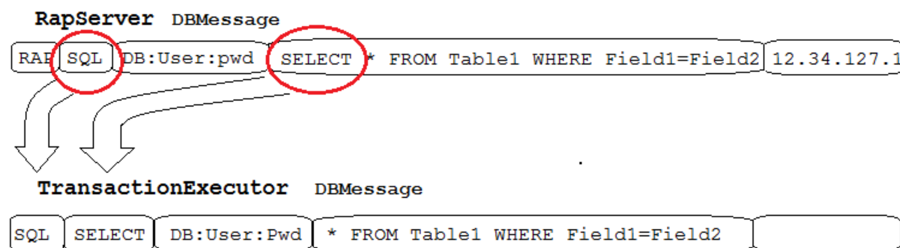


Figura 2.9: incapsulamento della query

nendo l'idea di permettere l'invio a remoto di una query per volta; questo scelta progettuale nasce dal fatto che più messaggi spediti nella rete potrebbero avere un ordine di arrivo diverso da quello che ci aspettiamo ed eseguire le query in tempi

diversi da quelli previsti. Il verificarsi di questo fenomeno e la sua trasparenza all'utente porterebbero ad avere effetti imprevisti e incontrollabili sulla base di dati interrogata.
Un compromesso di flessibilità è stato realizzato con una coda privata che contiene tutte le query inserite dall'utente sequenziate secondo l'ordine di inserimento; per spiegare il funzionamento completo partiamo dall'ipotesi iniziale di avere la coda di query vuota e procediamo con i dettagli di esecuzione del messaggio SQL proveniente da console. Arrivata la query al *RapClient* quest'ultimo controlla, dopo averla inserita nella sua *queryList*, che la sua variabile booleana *sqlStop* sia impostata a false; questa variabile funge da vero e proprio semaforo sull'esecuzione delle istruzioni e quando assume il valore di true blocca l'invio di ulteriori query nel nodo remoto.
Il primo comando digitato in seguito alla conferma di connessione viene sempre mandato immediatamente al nodo remoto tramite il *NetworkMessageSender* aggiungendo ai campi del messaggio le credenziali di accesso necessarie al *RapServer* per gestire la query. A questo punto qualsiasi altro comando sulla base di dati deve attendere nella coda l'arrivo del relativo *DBMessage* di RESPONSE (facente parte di quell'insieme di messaggi validati all'ingresso) che, dopo averne visualizzato a video il contenuto, sblocca la lista e invia il messaggio immediatamente successivo, se esiste, per poi bloccarsi nuovamente in attesa della relativa risposta.
L'ultimo messaggio elaborabile dal *RapClient* è il particolare *DBMessage* "RAP REF" proveniente dal nodo con cui la connessione è già stata stabilita, un messaggio che stabilisce che il nodo è, da quel momento, impossibilitato a rispondere; in questo caso, non essendo una chiusura voluta dall'utente, il *RapClient* provvede a ricostituire la comunicazione con un altro nodo ripetendo il processo della fase 2. Nel ripetere la fase vengono aggiornati i riferimenti ai nodi tramite una nuova interrogazione al plugin DHT; il *RapClient* possiede in questo punto anche un sistema di ripristino che permette di inviare nuovamente la query inserita dall'utente e con la relativa esecuzione in sospeso se questa era già stata inviata al nodo server; il controllo viene effettuato semplicemente tramite l'interrogazione alla variabile *sqlStop*.
Con poche e semplici istruzioni sono state coperte tutte le casistiche di funziomanto e di funzionalità necessarie alla correta esecuzione del protocollo lato client.

Fase 3 lato Server

il *RapServer* è, in questa fase, pronto a ricevere e trattare tre tipi di messaggi : l'arrivo di una query, la notifica della sua esecuzione e l'ordine di chiusura. Partendo dal messaggio contenente la query e identificato dal *DBMessage* "RAP SQL dbname:username:password [SELECT\UPDATE\...] ..." il codice del *RapServer* si adopera a riconvertirlo in un messaggio adatto all'utilizzo in locale pulendolo e ricomponendolo per adattarlo al blocco di sincronizzazione; l'invio alla giusta istanza di *Database* avviene tramite la ricerca sul *NodeSet* utilizzando le credenziali nel messaggio di arrivo (da questo punto in poi non devono essere più inserite nel messaggio), il *DBMessage* è da questo punto in poi formato dai seguenti token : "SQL [SELECT\UPDATE\ ...] ..." (SELECT e gli altri elementi nella parentesi quadra sono tutti i comandi opzionali contenuti all'interno di SQLEnum) e con l'HeaderEnum impostato a SEND per indicare la provenienza esterna e la necessità di rinviare la soluzione al mittente.
Conclusasi l'esecuzione della query, il *TransactionExecutor*, incaricato di elaborarla, ricostituisce un *DBMessage* "RAP RESPONSE dbname:username: RISPOSTA" senza la password relativa e mandata così al *RapServer* tramite l'ausilio del *RapSetServer* che ne possiede i riferimenti; l'introduzione della password avviene in un secondo momento all'interno del *RapServer* a causa di una scelta strutturale atta ad evitare la conoscenza della password.
Per permettere la conoscenza di quest'ultima al *TransactionExecutor* bisognerebbe costruire un metodo apposito di interrogazione del *SupportDB* per ottenere la password, metodo che andrebbe contro all'idea di segretezza e riservatezza con cui il plugin è stato progettato, si è perciò rivolti a questa scappatoia di riscrivere in seguito la password nel messaggio rivolto al client. Conclusasi la fase di risposta alla query, il *RapServer* è pronto a ricevere nuove istruzioni.
L'ultimo messaggio interpretabile è quello che comanda la chiusura del thread "RAP END USE" che, semplicemente, azzera i campi privati del metodo e ordina al *RapSetServer* di terminarlo; se vi è una query in esecuzione riferita a quel client essa viene semplicemente eseguita e la risposta non arriva all'utente.

Il protocollo così ricostituito corregge i difetti presenti nella vecchia versione e ne amplia gli utilizzi risparmiando sull'utilizzo dei messaggi permettendo il recupero delle query in sospeso e gestendo il tutto occultando all'utente il luogo dove avviene il reale funzionamento del protocollo; ogni nodo riesce ad ottenere un accesso sicuro a qualsiasi database della rete senza limiti sulla connessione.

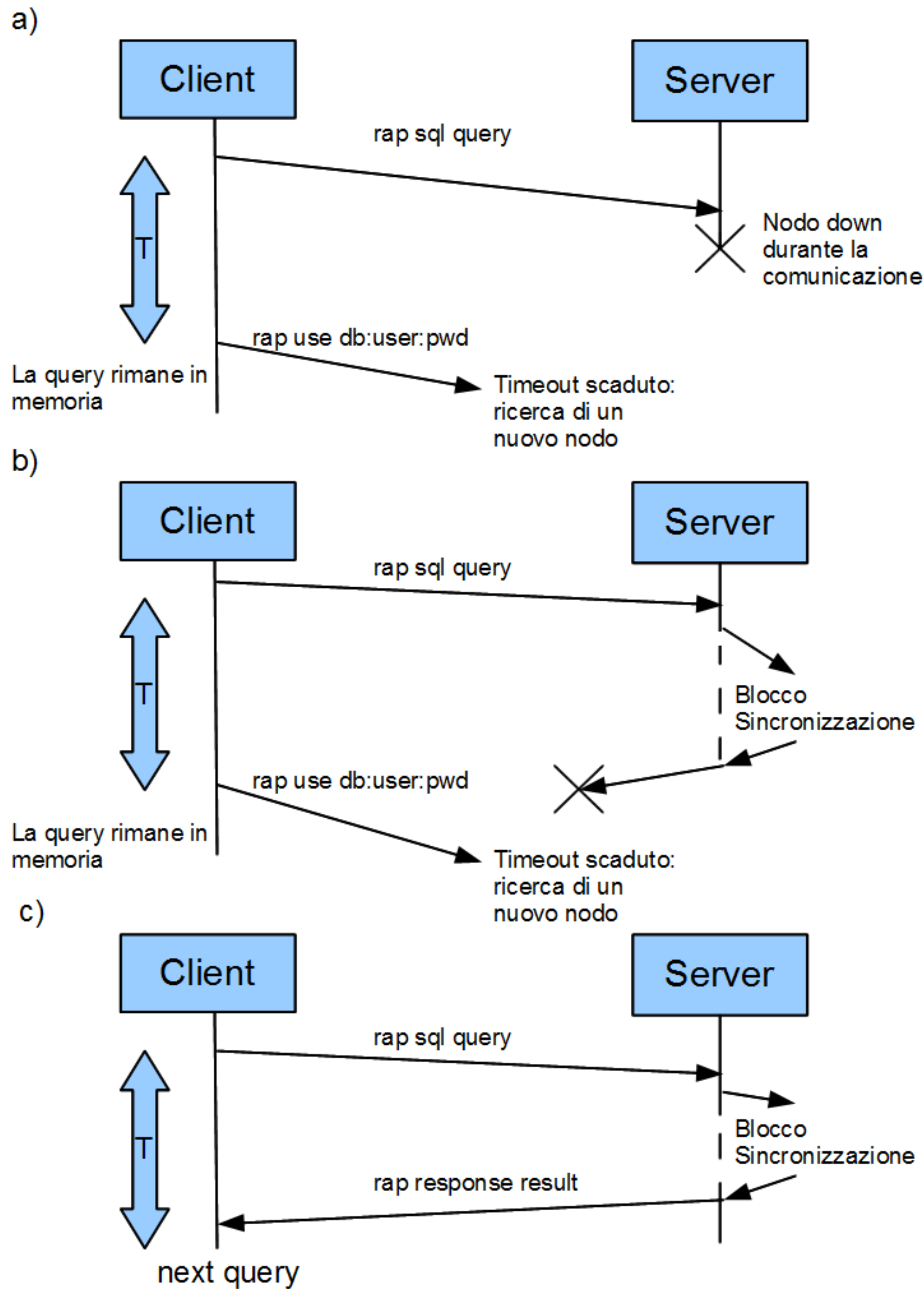


Figura 2.10: Esempio di comportamento del protocollo: a) caso caduta del nodo b) caso scadenza timeout c) caso favorevole

2.3 Gestione Thread

Una base generale più solida e uniforme per governare il funzionamento del plugin ha portato in evidenza anche altri problemi di gestione delle risorse; in particolare mi riferisco alle analisi prestazionali condotte sulla base dei rallentamenti notati all'avvio di `PariDBMS`. Nella fase iniziale di caricamento si è notato lo spropositato numero di thread che venivano caricati dal plugin, essi riguardavano all'incirca la totalità delle funzioni e non rispondevano a nessuno schema gerarchico ben definito.
Per ottenere un miglioramento sui tempi di caricamento si è provveduto in prima fase a ridurre le classi di thread all'essenziale; questo fatto è stato già ampiamente presentato nel capitolo trattante la struttura di `PariDBMS` e riguarda la totalità dei blocchi e un'ampia fascia di classi in essi contenute.
Per migliorare l'efficienza del plugin si è deciso di operare un controllo sul funzionamento e sull'utilizzo dei thread analizzandone gli eventi che ne causano l'avvio e il comportamento durante il loro ciclo di vita. Per evitare che plugin troppo esigenti riescano a bloccare l'intero sistema con thread inutili ciascuno di essi è dotato di un threadpool gestito direttamente da Core: quando un plugin ha bisogno di eseguire un determinato blocco di istruzioni, in parallelo rispetto al thread principale, deve fare una prenotazione di esecuzione; Core è incaricato di raccogliere tutte queste prenotazioni, ed eseguire il codice relativo, su un thread libero del pool assegnato al plugin, attendendo, nel caso fossero tutti occupati. Andrebbero quindi lette sempre in quest'ottica le dichiarazioni di partenza di un thread ma, grazie alle classi *PariPariRunnable* e *PariPariThread*, Core stesso esegue un'ottima astrazione dal sistema di prenotazioni, permettendo di fatto di continuare a parlare di *Runnable* e *Thread*. Nel linguaggio Java la classe *Thread* rappresenta lo stesso concetto di thread già descritto. L'interfaccia *Runnable*, invece, definisce il codice che dovrà essere eseguito da un thread. La relazione di cardinalità tra questi due attori è di tipo N:M in quanto un *Thread* può eseguire più *Runnable* ma pure un dato *Runnable* può essere eseguito da più *Thread*.
La scelta di Java come linguaggio di programmazione apre la strada anche all'utilizzo della sua libreria standard messa a disposizione degli sviluppatori. Essa contiene classi e interfacce che permettono di non dover ogni volta reinventare la ruota : ADT, gestione astratta di stream dati, creazione di interfacce utente e tantissime altre, sono le possibilità che mette a disposizione. Si vorrebbe metter in luce il package *java.util.concurrent* introdotto dove sono contenuti una serie di strumenti di aiuto per i programmatori di sistemi concorrenti come ad esempio

ADT concorrenti, code bloccanti, semafori, lock, e tipi di dato atomici utilizzati ampiamente nella progettazione del codice.
Alla luce dei fatti si è deciso di preferire uno stile di programmazione dei thread che permettano una permanenza maggiore nel pool e un carico di lavoro uniformemente distribuito in modo da ottenere una maggiore efficacia dal sistema di thread-pool supportato; un esempio lampante di questo stile di programmazione è il *MessageInterpreter* che è andato a sostituire il *MessageDelivery* e i vari interpreti(Sql, Nmg, Rap e Upd). L'utilizzo del *MessageInterpreter* non si limita, come si potrebbe pensare dal suo utilizzo fino ad ora accennato, ad instradare nei vari blocchi il corretto messaggio, ma, grazie alla sua locazione gerarchica di punto centrale del plugin, detiene il controllo su alcune funzionalità e thread temporanei come lo stesso *RemoteAccessProtocol* precedentemente illustrato.
Nonostante la funzione del RAP di compensare i tempi lunghi di aggiornamento e caricamento, non vi è ancora un sistema implementato che permette di passare, ad allineamento ultimato, dal database remoto a quello locale; dovrebbe l'utente decidere volontariamente di recidere la connessione e riconnettersi, il plugin a questo punto riconosce come disponibile la base di dati locale e ne effettua il collegamento.
Nella versione precedentemente la classe *init DBMS*, da cui il Core inizia a caricare il modulo, conteneva tutti i thread e li faceva partire tutti contemporaneamente senza tenere conto del loro utilizzo o delle tempistiche di caricamento; a causa di questo si ottenevano thread in esecuzione che superavano di numero la ventina e molti di essi si trovavano ad attendere il loro turno di scheduling nel pool senza avere effettivamente un lavoro da compiere. Un punto determinante nella ridefinizione del codice è stata l'analisi del variare dell'efficienza del plugin in funzione della quantità di database in esso contenuti; in particolare, al crescere del numero di base di dati presenti, si otteneva un aumento di thread di gestione e di mantenimento composti dal *Synchronizer*, dall'*Executor*, dai vari *DBUpdate* e i *DiESeLLayer*; per ridurre l'aumento lineare di quest'ultimi ho cercato di isolare quei blocchi e quelle classi che non dipendevano totalmente dai parametri di identificazione; l'esempio lampante già precedentemente documentato riguarda il *TransactionExecutor* in cui non vi è una cooperazione o una vantaggiosa concorrenza nell'avere un esecutore per basi di dati, questo a causa della presenza del temporizzatore a timestamp che ne obbliga l'inserimento sequenziale.
L'aver eliminato i vari *TransactionExecutor* in favore di uno unico è stato un secondo passo avanti nella riduzione del bisogno di thread; il vero vantaggio, però, è stato nello spostare le partenze di thread in base ai tempi e le necessità reali

del plugin. Come per DiESel che necessita un pieno caricamento di DHT per poter trovare i nodi, così i thread di aggiornamento, come *DBUpdateClient*, necessitano che i nodi di `PariDBMS` siano disponibili alla comunicazione per ottenere un riscontro positivo e farsi mandare i messaggi necessari all'update. La priorità di questo tipo li portano ad ottenere una partenza di esecuzione il più possibile successiva al caricamento di DiESeL; l'update, in particolare, deve essere fatto all'inizio per ottenere delle basi di dati READY e usufruibili in tempi minori (se i tempi si prolungano o il database non si carica subentra il protocollo di accesso remoto per l'utilizzo effettivo); questa funzione viene eseguita all'inizio e, successivamente, l'allineamento della base di dati viene mantenuto dal blocco di sincronizzazione e dai suoi thread.
L'avere una parte dei thread che hanno un proprio ciclo di vita limitato ha reso chiaro a noi progettisti l'ordine di partenza di quest'ultimi; in particolare sia per DiESeL che per *DBUpdate* le loro funzioni si esauriscono nei primi cicli di vita del plugin portando la scelta obbligata di caricare tutti i thread operanti sulle basi di dati contenute; il lavoro degli altri thread sarà così diluito nel tempo e partirà all'attivazione di particolari situazioni (come il passaggio del database da UPDATE a READY o da UPDATE a STOPPED).
Applicando queste tecniche di riduzione si sono ottenuti dei risultati apprezzabili nei tempi di risposta del plugin; mentre nelle versioni precedenti *init DBMS* procedeva con l'avviare una ventina di thread, ora, allo stato in cui verte il plugin, il numero di thread in partenza è confinato alla decina essendo questi strettamente necessari al funzionamento corretto.

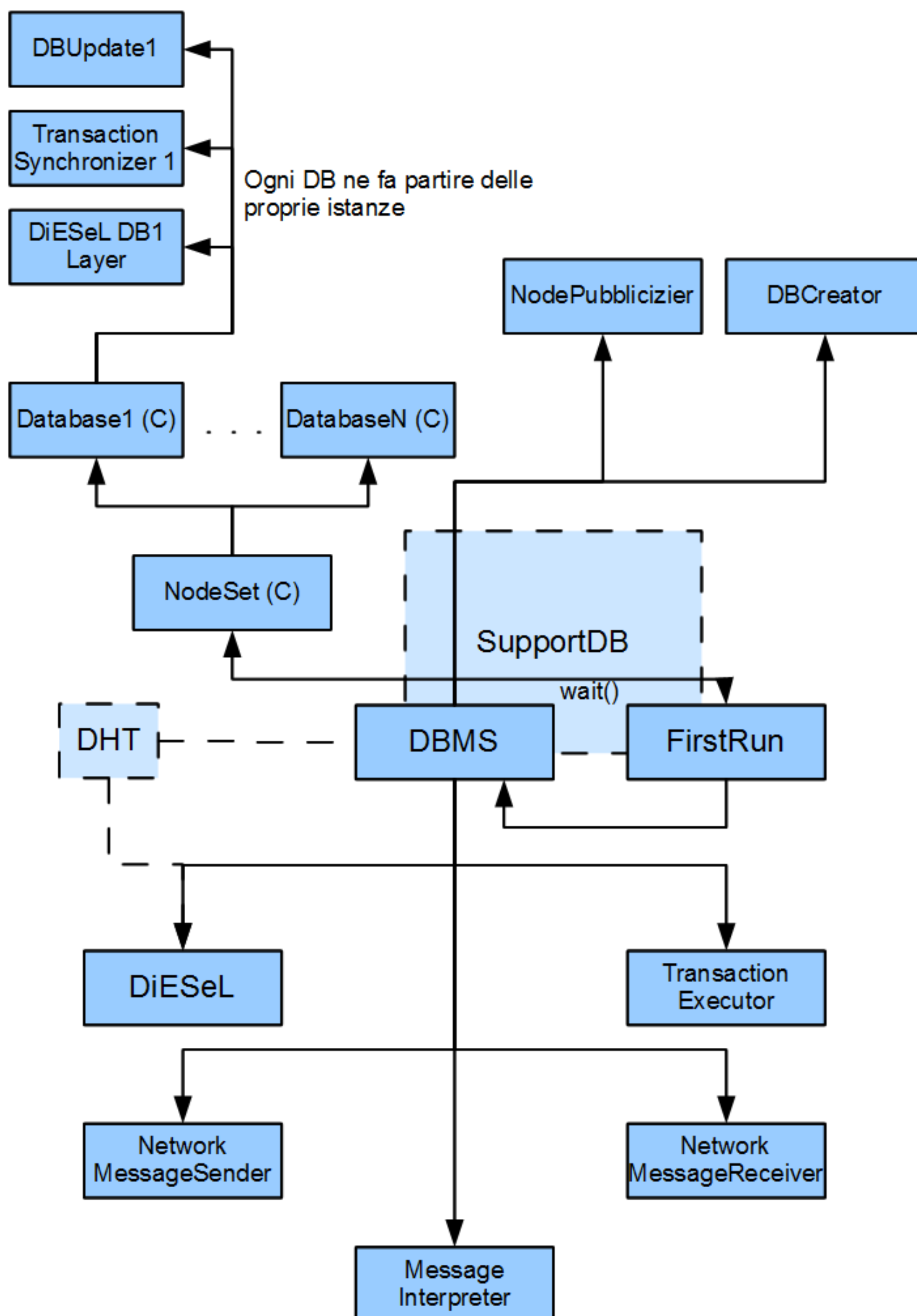


Figura 2.11: Schema di partenza dei Thread del plugin all'interno dell'*init() DBMS*

Capitolo 3

Conclusioni

L'obiettivo che mi sono proposto nello svolgimento di questa tesi è la presentazione di `PariDBMS` nel modo più completo possibile toccando ogni aspetto di funzionamento, sia esso puramente di codice e risoluzione di problematiche, sia esso basato su una migliore interazione macchina-utente. In particolare ho cercato di sottolineare come una condotta di programmazione comune e uno schema di procedimento permettano di semplificare la lettura del codice al punto tale da rendere pressochè minima la necessità di un ampia documentazione; parlo delle ragioni che mi ha portato a strutturare i *DBMessage* e a uniformare l'accesso ai campi in esso contenuti. Un riscontro positivo l'ho potuto notare tramite il corso di ingegneria del software in cui, per gestire la collaborazione di persone diverse, metodi di risoluzione comuni permettevano un più veloce ed efficiente sviluppo del codice. Nonostante un più rigido formalismo i problemi sull'interazione dei blocchi si sono comunque evidenziati soprattutto su quelle parti di codice che facevano riferimento alle prime versioni; il non aver condotto prove dirette sul funzionamento ha influenzato sulla programmazione in modo drastico, infatti, non venivano considerate le latenze e i tempi dilatati della comunicazione della rete nello stile di programmazione adottato. Considerazioni che ci hanno portato a rivalutare classe per classe il funzionamento dei blocchi e operare cambiamenti sulla gestione dei thread e sulle classi che venivano influenzate. `PariDBMS` non risulta, per ora, essere un vero e proprio sistema distribuito, ma funge allo stato attuale da sistema di backup di basi di dati e ne garantisce la perenne accessibilità (grazie anche all'accesso remoto); l'implementazione distribuita, ora più che mai una necessità, richiederà un ulteriore rivoluzione del codice, ciò nonostante il lavoro finora condotto non deve considerarsi come tempo sprecato, esso, infatti, ci ha permesso di mettere chiaramente in evidenza le priorità e i difetti di un sistema così concepito.

Come conclusione finale sottolineo l'importanza di aver partecipato ad un progetto così ampio, qual'è *PariPari*, che ritengo un punto fondamentale nella mia carriera studentesca per migliorare le abilità di programmatore e per imparare a gestire meglio i progetti quando si ha a che fare con un gruppo di lavoro sufficientemente vasto.

Capitolo 4

Sviluppi Futuri

Nonostante l'introduzione di tutte queste migliorie al plugin, `PariDBMS` soffre ancora di difetti strutturali e di collaborazione fra i nodi; solo con continue sperimentazioni è possibile accorgersi dell'effettivo evolversi della rete di database collegati. In proposito si è cercato di rendere la versione attuale il più efficiente e funzionante possibile analizzando le risposte dei nodi, non più a livello teorico, ma attraverso la pratica con l'utilizzo di un numero crescente di macchine dedicate alla sperimentazione. Da questa base di partenza si sta cercando di eliminare tutti i bug che si annidano nel codice causando anomalie e imperfezioni nel flusso di esecuzione del codice. A tal proposito nelle riunioni di `PariDBMS`, di cui attualmente sono il responsabile Team Leader, si è provveduto a stabilire una scaletta lavorativa delle prime necessità qui riproposte come incipit per lo sviluppo futuro del plugin.
In primis la correzione degli errori e l'aumento delle classi di testing è una necessità per dimostrare l'affidabilità e la robustezza di `PariDBMS` cercando di ottenere fiducia di utilizzo nel vasto mondo di *PariPari* e dei DBMS che circolano in rete. Questo primo punto non è una vera e propria miglioria ma serve a tener presente ai futuri programmatori del plugin che un codice testato e funzionante è sempre un'utile base di partenza per revisionare perfezionamenti e nuove funzionalità.
Un bisogno, secondo la mia visione del plugin, necessario è la separazione del timestamp; finora è stato utilizzato un unico temporizzatore per tutte le basi di dati contenute che ad ogni query inserita (sia essa di modifica o di visualizzazione) riferita a qualsiasi database nella rete aumenta costantemente; la difficoltà perciò si presenta nel recuperare il timestamp massimo appartenente ad una base di dati in particolare che ne determini l'ultima istruzione inserita. Al crescere del numero di nodi e di database che circolano è possibile che un aumento significativo del valore incrementale porti a non ritrovare i riferimenti ad alcune interrogazioni

di log troppo vecchie. Altro problema possibile è la conflittualità degli accessi; ogni query, se inserita in tempi troppo vicini ad un'altra con analogo timestamp, rischia di subire problemi di *starvation*[1] favorendo connessioni più vicine e veloci. Un timestamp separato per ogni base di dati, identificato insieme alle chiavi primarie, permetterebbe di procedere più velocemente nelle operazioni, ottenere minori conflitti e supportare un maggiore parallelismo di funzionamento.
Un altro punto discusso nelle riunioni, e già in fase di una prima lavorazione, è l'utilizzo da parte di `PariDBMS` del supporto del plugin di Storage per la scrittura dei file. Questo upgrade necessita di mettere mani sul codi di HSQLDB per dirottarne le API in modo da bloccare la scrittura incontrollata di quest'ultimo.
Ultimo punto è il passaggio al vero obiettivo del nostro plugin, l'implementazione di un DBMS distribuito senza l'ausilio di macchine server che permetta la memorizzazione nella rete di qualsiasi base di dati senza limiti di spazio fisico(limite imposto dalla memoria del calcolatore), con un buon compromesso di velocità di risposta e frammentazione(segmenti troppo piccoli necessitano di una ricerca più lunga e segmenti più grandi richiedono un maggior lavoro di integrazione) e mantenendo tutte le proprietà cardine di qualsiasi DBMS basato su architettura client-server. Questo progetto è ancora in fase di discussione con il professor Peserico a causa della mole di lavoro che richiederebbe operare tale passaggio; le soluzioni che si pensano di adottare richiederebbero, in un caso, la rimozione di HSQLDB e la ridefinizione di un completo DBMS progettato dagli studenti o, nel secondo caso, la creazione di uno spesso strato di software che si adopererebbe nel gestire le tabelle distribuite ad un livello superiore o inferiore rispetto a HSQLDB al quale far credere che l'utilizzo avvenga completamente in locale.
Nonostante questa analisi affrontata tra il team di DBMS, il professore, i ragazzi di ingegneria del software con cui abbiamo collaborato e con il resto dell'alta gerarchia di *PariPari*, nulla esclude l'intervento su altri parti del codice e del sistema di funzionamento qui finora presentato.

[1] l'impossibilità, da parte di un processo pronto all'esecuzione, di ottenere le risorse di cui necessita.

Bibliografia

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

[2] *Distributed System: concept and design* - George Coulouris, Jean Dollimore, Tim Kindberg; quarta edizione, Addison-Wesley 2005;

[3] *PariDBMS: Accesso Remoto* - Deniz Tartaro; tesi di laurea triennale, Università degli studi di Padova 2009;

[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: http://download.oracle.com/javase/6/docs/api/ - Documentazione Java.

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

Elenco delle figure

1 Rappresentazione grafica della struttura di PariPari; 5

1.1 Schema ER del SupportDB con evidenziate solamente le chiavi (che godono della proprietà di Unique) della singola tabella; le tabelle ombreggiate indicano che l'entità è debole e eredita le chiavi primarie dalla tabella la cui la relazione 1:N è lato 1; 8
1.2 Schema dei blocchi del plugin 9
1.3 Schema di accorpamento da MessageDelivery e interpreti a MessageInterpreter; 11
1.4 Raffigurazione struttura di base 12
1.5 Variazione Log in base alla versione e relativo timestamp 17
1.6 Schema di funzionamento delle reti DiESeL 19
1.7 Sistema di invio messaggi sfruttando la struttura delle reti DiESeL 20

2.1 Esempio sistema di messaggistica della vecchia versione 23
2.2 Ricostruzione dei percorsi dei messaggi 25
2.3 Messaggio strutturato con Enum 26
2.4 Messaggio Enum con l'aggiunta dei campi identificativi 27
2.5 Codice del MessageInterpreter per instradare il messaggio a seconda del TypeEnum 27
2.6 Lista Enum disponibili per la composizione del DBMessage 28
2.7 Funzionamento vecchia versione di Rap 34
2.8 Protocollo 3-way handshake per la connessione al Server 40
2.9 incapsulamento della query 41
2.10 Esempio di comportamento del protocollo: a) caso caduta del nodo b) caso scadenza timeout c) caso favorevole 44
2.11 Schema di partenza dei Thread del plugin all'interno dell'*init() DBMS* 48

Appendice A

Ringraziamenti

Approfitto di questo spazio pubblico per porre i miei sentiti ringraziamenti alle persone che hanno segnato uno o più capitoli delle fasi della mia vita .
Ringrazio il professor Peserico e l'ingegner Bertasi per avermi permesso di partecipare al progetto e per il loro sostegno.
Ai miei genitori, ovviamente i più importanti, che mi hanno sempre lasciato la libertà di esprimermi e di fare le mie scelte senza mai imporre le loro, mi hanno insegnato a pensare con la mia testa.
A mia nonna Aurelia che indipendentemente da quello che dicono o pensano gli altri è sempre dalla mia parte, mi ha insegnato ad avere sempre fiducia nelle persone.
A mio nonno Danilo, oramai non c'è più ma rimane sempre una parte di me, lui mi ha insegnato a far valere le mie idee sempre e comunque (a quel (censura) di un ladro di biciclette, sappi che il proprietario originale di quella bici era un ganzo!).
A mia zia Lidia, che per me è una seconda mamma , terza nonna e tutti i gradi di parentela messi assieme, mi ha insegnato che i legami affettivi vincono su tutto e la famiglia ci sarà sempre per te.
A mio fratello, colui che più di tutti mi ricorda ogni giorno i miei difetti e lo ringrazio per questo, siamo più simili di quel che sembra
Alla mia compagnia di Valdagno, la mia seconda famiglia, Berry(come un secondo fratello), Riccio(il mio cugino sbalon, poi repettaro, un giorno chi lo sa...però grasso è più simpatico), Nizz(tranne nel suo periodo con il ciuffo emo e Solange), Bruje(e al suo mono-osso), Netti(e le sue avventure HC in costa rica),Mappa (capo scout a vita), Fabio(il mio maestro di brasilian jujitsu e filosofia), Calghi(il re degli scherzi con cui incrocio volentieri i guantoni), Viso(che conosce BatRoberto e non è poco!), Bindo(detto grezzura, così, per simpatia...), Jonny(il

gestore del racket dei giornali...chissà che finisca prima o poi torre nera...), Keof(il mio matto aggancio a new york), Cekk, Palla, Piz, Pimpi, Valentina, Valeria, Elena, Giulia,... spero di averli elencati tutti; feste e cazzeggio nascondono una grande unità, quando c'è bisogno ognuno c'è per te e per gli altri, non avrei potuto desiderare un gruppo migliore, mi hanno insegnato che basta poco per potersi divertire assieme.
Ai mie amici di Padova, coloro che mi hanno fatto volare questi tre/quattro anni di ingegneria, penso che senza di loro non avrei mai visto la luce in fondo al tunnel: Enri, Albi, Pelo ,Marco , Righi, Miur, Mengheno, Luca , Vic, Momi, Glo, Chiara, Silvia, Giulia, Alessia, Ste, Mario ...ciascuno a modo suo mi ha fatto capire che nel mondo ci sono persone simili a me e dovunque andrò nella mia vita c'è sempre la speranza di trovare amici sinceri.

(Se leggete questo ma il vostro nome non compare è perchè non ho più posto!)

Ai miei coinquilini di Via Perosi 2 che mi hanno sopportato nel disordine, nel mio via vai giornaliero e nel sonnambulismo;
Agli altri miei coinquilini di Via Portello 18, ho approfittato fin troppo dell'ospitalità, grazie di tutto.
Alla mia vecchia squadra e al mio maestro di TaeKwonDo, loro magari non se ne sono resi conto ma questa disciplina e la loro compagnia mi ha reso più forte fisicamente e mentalmente, mi hanno insegnato ad affrontare meglio le sfide.

A tutti voi vi dico grazie.