Università degli Studi di Bologna

FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA
Fondamenti di informatica L-B

SVILUPPO DI UN PROTOTIPO PER L'INTERAZIONE CON BASI DI DATI MEDIANTE UN'INFRASTRUTTURA DI COORDINAZIONE

Tesi di Laurea di:
Luca Tonini

Relatore:
Prof. Enrico Denti

Anno Accademico 2002-2003

Sommario

INTRODUZIONE

I sistemi di messaggistica sono innumerevoli e possono essere sviluppati in modi differenti, utilizzando GUI, console, o altri mezzi per interagire con l'utente. Anche le architetture possono essere le più diverse. Fra queste una delle possibilità è costituita da architetture ad *agenti*, intendendo con questo termine componenti software (eventualmente eterogenei) caratterizzati da un certo grado di autonomia e con compiti ben precisi da svolgere (task).

Un problema critico nella costruzione dei sistemi consiste nel poter utilizzare adeguatamente alcuni servizi "legacy", che sono servizi preesistenti, progettati al di fuori dell'architettura di riferimento che si adotta, ma che si desiderano comunque mantenere e, per quanto possibile, integrare. Esempi rilevanti sono sistemi di mail, sistemi di trasferimento di file, sistemi di accesso alle basi di dati.

Sebbene questi servizi siano utilizzati solitamente da persone, renderli accessibili anche ad agenti software può aprire nuovi scenari verso l'automatizzazione, soprattutto se si considerano sistemi interattivi complessi.

Risolvere il problema particolare, costruendo ogni volta uno specifico agente che fissi esattamente le competenze richieste e la conoscenza del protocollo di comunicazione, è chiaramente un approccio insoddisfacente: una soluzione più interessante consiste nell'incapsulare ogni servizio in un agente che possa essere riutilizzato in contesti differenti. In questo modo un approccio è ben descritto e documentato, e può essere ampiamente adottato da infrastrutture ad agenti.

Naturalmente, il tipo di infrastruttura, e in particolar modo il modello di coordinazione adottato dall'infrastruttura, influisce significativamente sul progetto e sullo sviluppo di questi sistemi.

In questa tesi si assume come base per lo sviluppo di tali agenti l'infrastruttura TuCSoN[TUC] che adotta un modello di interazione mediato: questo rende il progetto, lo sviluppo e la fruizione di servizi ben diversi dal progetto, sviluppo e schema del modello convenzionale basato sull' interazione diretta (peer-to-peer).

L'idea fondamentale è che ogni agente che faccia parte del contesto di coordinazione TuCSoN – sia che si tratti di un semplice agente inserito in qualche dispositivo, sia che si tratti di un agente mobile/intelligente complesso – possa richiedere questi servizi in modo semplice e immediato, semplicemente interagendo con le opportune astrazioni di coordinazione TuCSoN (Centri di Tuple), senza necessità di conoscere i dettagli tecnici riguardanti lo specifico servizio utilizzato.

In questo scenario, ogni comunicazione – spedire una mail, fare il download di un file, richiedere informazioni ad una base di dati, ecc.. – dovrebbe essere effettuata, e indirettamente eseguita, solo inserendo/prelevando le opportune informazioni, espresse come tuple logiche, nel/dal Centro di Tuple opportuno.[GIUS]

Lo scopo di questa tesi è dunque realizzare un agente in grado di utilizzare l'infrastruttura di coordinazione TuCSoN come sistema di comunicazione e tramite delle richieste espresse sotto forma di messaggi rendere disponibile per gli utenti, sia umani che software, un servizio di accesso a basi di dati, indipendentemente dal modo con cui la base di dati a cui si accede è stata realizzata e viene gestita. Si procederà quindi a:

- analizzare lo stato dell'arte dei sistemi di gestione delle basi di dati, considerando i metodi di accesso al gestore della base di dati (DBMS), di descrizione delle strutture logiche contenute nelle basi di dati e di ricerca di informazioni.

- progettare il linguaggio di comunicazione che servirà per far dialogare, secondo delle specifiche, l'agente e gli utenti del servizio.
- progettare l'agente integrandolo nell'infrastruttura TuCSoN per rendere disponibile agli utenti dell'infrastruttura un servizio di accesso a basi di dati. L'agente dovrà consentire ad un utente proprietario di una base di dati di poterla condividere quando e come preferisce. Dovrà inoltre permettere ad un utente qualsiasi di ricercare delle informazioni contenute nelle basi di dati che sono state condivise.
- realizzare, almeno a livello di prototipo, l'agente, collaudarlo e definirne eventuali possibili sviluppi.

CAPITOLO 1
ANALISI

1.1 Analisi dei Database Management System

Un DBMS è sostanzialmente uno strato software che si frappone fra l'utente ed i dati veri e propri. Grazie a questo strato l'utente e le applicazioni non accedono ai dati così come sono memorizzati effettivamente ma ne vedono solamente una rappresentazione logica.

Ciò permette un elevato grado di indipendenza fra le applicazioni e la memorizzazione fisica dei dati.

La maggior parte dei database attualmente utilizzati appartiene alla categoria dei database relazionali.

I motivi di questo successo (anche commerciale) vanno ricercati nel rigore matematico, nell'espressività del modello relazionale, nella semplicità di utilizzo e nella disponibilità di un linguaggio di interrogazione standard (SQL, vedi appendice C) che, almeno potenzialmente, permette di sviluppare applicazioni indipendenti dal particolare DBMS relazionale utilizzato.

Indipendentemente dal tipo di database, le funzionalità principali che ci si deve aspettare da un DBMS sono quelle di:

- consentire l'accesso ai dati attraverso uno schema concettuale, invece che attraverso uno schema fisico;
- permettere la condivisione e l'integrazione dei dati fra applicazioni differenti;
- controllare l'accesso concorrente ai dati;
- assicurare la sicurezza e l'integrità dei dati.

Grazie a queste caratteristiche le applicazioni che vengono sviluppate possono contare su una sorgente dati sicura, affidabile e generalmente

scalabile. Tali proprietà sono auspicabili per applicazioni che usano la rete Internet come infrastruttura e che hanno quindi evidenti problemi di sicurezza e scalabilità.

SQL è un linguaggio di interrogazione dei database indipendente dallo specifico DBMS. Teoricamente un DBMS che adotta il linguaggio SQL dovrebbe poter essere sostituito senza alcuna difficoltà da un altro tipo di DBMS che adotti anch'esso SQL: tuttavia, ciò non è sempre vero, perché nella maggior parte dei casi ogni ditta produttrice di DBMS ha adottato solo il nucleo centrale dell'SQL standard, estendendolo come meglio credeva e come risultava più utile nel caso specifico. Si è cosi giunti allo stato in cui i DBMS che adottano SQL permettono tramite questo di:

- definire una nuova base di dati;
- definire le tabelle in essa contenute con le relative proprietà;
- definire le colonne contenute nelle tabelle e le loro specifiche;
- modificare la struttura delle tabelle e delle colonne;
- inserire, modificare e eliminare informazioni all'interno della base di dati;
- realizzare delle richieste per ottenere informazioni contenute nella base di dati;

In questo linguaggio non è stato definito uno standard per l'accesso al DBMS, cioè non si specifica come poter effettuare dall'esterno del DBMS le richieste nel formato SQL e non è stato neanche definito uno standard per ricavare successivamente la descrizione della struttura di una base di dati già esistente. Mentre per risolvere il primo problema sono stati fatti degli studi e delle ricerche e sono stati definiti degli standard a parte, come ODBC o JDBC, per il secondo problema ogni produttore di DBMS ha intrapreso la propria strada definendo un linguaggio appropriato per ogni caso specifico. Nella pratica questa differenziazione impone, ad un utente

che vuole richiedere informazioni, la conoscenza della struttura della base di dati a cui si rivolge altrimenti non riuscirebbe a realizzare la richiesta nel formato corretto. Per conosce la struttura della base di dati l'utente deve o richiederla a chi l'ha realizzata o conoscere il formato delle istruzioni per averne la descrizione (implica sapere con che DBMS è gestita).

1.2 L'infrastruttura di coordinazione TuCSoN

TuCSoN[ROS] è un'infrastruttura per la coordinazione di agenti su Internet basata su un'astrazione di coordinazione chiamata *Centro di Tuple*. Una *Tupla* logica è una struttura dati costituita da un nome e da una lista di argomenti.

Il Centro di Tuple è uno spazio di tuple arricchito con una specifica di comportamento che assume la forma di reazione agli eventi di comunicazione; programmando tale comportamento si possono realizzare le politiche desiderate.

Il modello di coordinazione di TuCSoN definisce una doppia percezione dello spazio di interazione di TuCSoN:

- come uno spazio globale fatto unicamente di mezzi di coordinazione (Centri di Tuple)
- come una collezione di spazi locali associati ai nodi Internet.

Questo si adatta ad entrambi i ruoli degli agenti Internet, vale a dire, come entità network-aware o entità network-unaware.

È possibile interagire con l'infrastruttura inserendo e recuperando le tuple dai Centri di Tuple di un certo nodo e cambiare il loro comportamento. È possibile fare ciò attraverso un programma Java o Prolog oppure tramite un'interfaccia grafica che permette di invocare direttamente le primitive del

linguaggio di coordinazione su un dato Centro di Tuple residente su un dato nodo della rete.

Le primitive si distinguono in:

- primitive per la manipolazione delle tuple (`out`, `in`, `rd`, `inp`, `rdp`);
- primitive per cambiare il comportamento (`get_spec e set_spec`).

Inserendo o rimuovendo le tuple di comportamento è possibile programmare il sistema affinché esegua determinate azioni in risposta ad una specifica azione; le tuple di comportamento devono essere espresse nel linguaggio di specifica ReSpecT.

Le primitive di manipolazione delle tuple hanno i seguenti compiti:

`out`: inserisce la tupla specificata nel Centro di Tuple;

`in`: rimuove la tupla che fa "match" con la tupla specificata;

`rd`: legge la tupla che fa "match" con la tupla specificata;

`inp:` rimuove, se presente, una tupla che fa match con la tupla specificata, se non è presente, l'operazione fallisce;

`rdp`: legge, se presente, una tupla che fa match con la tupla specificata, se essa non è presente, l'operazione fallisce;

Esempi:

```
out(p(1,hello))
```

Inserisce la tupla `p(1,hello)` nel Centro di Tuple.

```
in(p(_,hello))
```

Rimuove la tupla il cui nome è p, ignorando il primo argomento e il cui secondo argomento è hello.

```
rd(p(X,Y))
```

Legge una tupla che ha due argomenti; X e Y sono variabili a cui saranno associati i valori degli argomenti della tupla con cui è stata verificata la corrispondenza.

Le primitive per il comportamento hanno i seguenti compiti:

`set_spec`: specifica il comportamento del Centro di Tuple. La tupla deve essere un programma ReSpecT valido nella forma di stringa atomica o di lista di reazioni.

`get_spec`: ritorna il comportamento del Centro di Tuple.

Esempi:

```
set_spec('reaction(out(X),(out_r(backup(X))))')
```

Questa primitiva dice al Centro di Tuple di inserire una tupla di backup ogni volta che una tupla viene inserita nel centro di tuple.

È possibile verificare ciò eseguendo l'operazione out:

```
out(array(index(1),value(tomato)))
```

e quindi provando a rileggere la tupla di backup che dovrebbe essere stata generata:

```
rd(backup(X))
```

La risposta generata sarà:

```
answer:backup(array(index(1),value(tomato)))
```

Se le primitive devono essere applicate alle tuple di un Centro di Tuple che non è quello di `default` residente sulla macchina su cui si sta lavorando, bisogna specificare, prima della primitiva, il nome del Centro di Tuple ed eventualmente l'indirizzo dell'host su cui esso risiede, secondo la sintassi:

TCName @ TCAddress ? **op** (Tuple)

dove *TCName* rappresenta il nome del Centro di Tuple e *TCAddress* l'indirizzo o il nome dell'host su cui risiede il nodo TuCSoN.

CAPITOLO 2
PROGETTO

2.1 Specifiche

Volendo realizzare un sistema in grado di agganciare delle basi di dati alla infrastruttura di coordinazione TuCSoN si ritiene utile svincolare l'utente che richiede informazioni, dalla necessità di conoscere in dettaglio la struttura della base di dati al quale accede, lasciandogli solo il compito di esprimere quali informazioni desidera ed eventualmente inserendo anche qualche vincolo (es. il Nome che di cerca deve essere uguale a Topolino).
D'altro canto si vorrebbe anche che un utente che desidera interfacciare una base di dati all'infrastruttura di coordinazione non fosse legato alla necessità di doverla realizzare (o convertire se già presente) con un determinato formato, ma che fosse libero di fare come meglio crede; al momento della connessione, è però necessario descriverne la struttura e il modo per accedervi. Si ritiene inoltre opportuno stabilire che il sistema debba essere in grado di gestire contemporaneamente più basi di dati in modo da permettere, a chi sta ricercando delle informazioni, di reperirne il maggior numero possibile da fonti diverse. Il tutto deve essere realizzato secondo le specifiche imposte da TuCSoN per lo scambio di informazioni fra i vari soggetti collegati all'infrastruttura.
Tali specifiche impongono che il soggetto che desidera effettuare delle richieste di servizio all'interno dell'infrastruttura deve seguire la seguente procedura:

1. Prelevare un ID unico che identificherà univocamente la richiesta.
2. Preparare la tupla (`service_request`) della richiesta di servizio inserendovi i dati relativi al servizio che richiede e i dati relativi al suo profilo (`user_profile`).

3. Inserire la tupla appena creata nel Centro di Tuple competente (`services`).
4. Attendere la tupla (`service_result`) di risposta al servizio richiesto sul Centro di tuple `incoming` o `outgoing` a secondo del servizio desiderato.
5. Se ce ne sono prelevare i dati provenienti dal servizio.

Il servizio, seguendo le specifiche, una volta verificato che c'è una richiesta di servizio deve eseguire le seguenti operazioni:

1. Prelevare dal Centro di Tuple specifico (`services`) le informazioni inviategli, attraverso la `service_request`, dall'utente che richiede il servizio.
2. Leggere dal Centro di Tuple dei profili (`profiles`) quello indicatogli nella richiesta, relativo ad un utente, e selezionare i dati a lui necessari.
3. Eseguire le operazioni necessarie per soddisfare il servizio che gli è stato richiesto di svolgere.
4. Realizzare le tuple contenenti le risposte da inviare al richiedente.
5. Depositare la tupla `service_result` nell'apposito Centro di tuple `incoming` o `outgoing`.

In questo scenario e considerando che servirà:

- un servizio di connessione delle basi di dati al sistema di coordinazione;
- un servizio per richiedere informazioni contenute nelle basi di dati gestite;
- un servizio per disconnettere la basi di dati che non si vogliono più far gestire al sistema;

devono essere progettati i formati delle tuple per le `service_request`, `service_result` e `user_profile`.

2.2 Progetto del formato delle tuple

Occorre innanzitutto sottolineare che già altri servizi (mail, ftp, http...) sono stati realizzati o modificati per essere integrati nell'infrastruttura. Per evitare che ognuno di questi utilizzasse un formato di comunicazione differente dagli altri è stata costruita un'architettura di base, definita nel documento "Integrating and Orchestrating Services upon an Agent Coordination Infrastructure"[IOSACI], che fornisce le specifiche per il formato delle tuple e la modalità con cui devono essere utilizzate. Nel realizzare nuovi servizi occorre attenersi alle specifiche dettate da questa architettura.

Nel definire il formato delle tuple occorre inoltre fare in modo che tutte le informazioni necessarie per gestire una base di dati siano presenti nella richiesta e siano nel formato corretto. Le specifiche del documento impongono che ogni tupla di richiesta (`service_request`) contenga:

- un campo che identifichi il tipo del servizio richiesto;
- uno per il profilo dell'utente che richiede il servizio;
- uno che sia identificativo della richiesta;
- un ultimo campo che specifichi il servizio richiesto, all'interno del quale ci saranno tutti i dati necessari per soddisfare tale servizio.

In tutte le tuple di richiesta si sceglie di usare come campo per identificare il tipo di servizio la costante `dB`; per i campi dove si vuole indicare il servizio richiesto si sceglie:

`connectDB` se si richiede il servizio di connettere un database al sistema;

`getInfoDB` se si richiede il servizio di prelevare dati dal database;

`disconnectDB` se si richiede il servizio per disconnettere un database.

La specifiche del documento impongono che per ogni servizio vi sia un profilo specifico associato ad un utente. Un profilo è concettualmente uno specifico record che fornisce tutte le informazioni, dell'utente, necessarie per effettuare un dato servizio.

Per quanto riguarda le tuple del profilo (`user_profile`) di un utente (che si identifica attraverso il campo `ProfileName`) si sceglie di usare:

`dB/connectDB`	per i profili associati alle richieste di connessione;
`dB/getInfoDB`	per i profili associati alle richieste di informazioni;
`dB/disconnectDB`	per i profili associati alle richieste di disconnessione.

All'interno delle tuple dei profili, nel campo `info` vanno inserite tutte le informazioni riguardanti l'utente.

Inoltre, sempre le specifiche dell'architettura impongono che ad ogni richiesta di servizio corrisponda una tupla di risposta (`service_result`) con lo stato del servizio. Si definiscono inoltre altri due tipi di tuple `db_header` e `db_body` con lo scopo di contenere le informazioni da restituire al richiedente del servizio di richiesta di informazioni.

Il formato dei campi segue il formato Prolog, quindi all'interno delle tuple i campi che iniziano con la minuscola (`dB`, `connectDB`, ecc...) indicano delle costanti, quelli con la prima lettera maiuscolo, indicano le variabili.

Si definiscono nel seguente modo i formati delle tuple di richiesta e di risposta ai vari servizi.

Formato della tupla per la richiesta di aggiungere un database all'agente:

```
service_request(dB,ProfileName,ID,
   connectDB(typeOfConnection(TypeOfConnection),
      nameDB(NameDB),fileOptional(FileOptional),url(Url)))
```

Descrizione dei campi :

`ID` Rappresenta l'ID della richiesta di servizio. Sarà legato alla specifica richiesta di servizio dal momento in cui viene effettuata una richiesta fino al momento in cui vengono prodotti i risultati.

`ProfileName` Rappresenta il nome del profilo da cui il servizio deve prelevare le informazioni riguardanti l'utente.

`typeOfConnection(TypeOfConnection)` Rappresenta il tipo di connessione (driver) che l'agente utilizzerà per effettuare il collegamento e accedere al database (ODBC,JDBC,...).

`nameDB(NameDB)` Rappresenta il nome del database che si vuole connettere e rendere disponibile per prelevare informazioni (ogni database deve avere un nome univoco).

`fileOption(FileOptional)` È il percorso all'interno dell'hard-disk di un file, in formato XML (vedi appendice B), che descrive la struttura del database (nomi delle tabelle, nomi delle colonne, formato delle colonne, chiavi primarie, riferimenti). Si è considerato opzionale perché l'agente potrebbe occuparsi anche di andare a ricostruire la struttura del database attraverso le funzioni che ogni DBMS mette a disposizione; mancando un formato standard per tale scopo è necessario che l'agente conosca di che tipo è il database e soprattutto il formato delle istruzioni per ricavare la descrizione del database.

`url(url)` Rappresenta la classe che occorre istanziare prima di connettersi al database tramite il driver scelto. JDBC e ODBC ad esempio richiedono che prima della connessione ad un database venga istanziata una classe specifica.

Questa invece è la richiesta per ottenere informazioni, tramite l'agente, dal database:

```
service_request(dB,ProfileName,ID,
     getInfoDB(TargetTC,coulumnList(CoulumnList),
         boundList(BoundList),option(nameDB(NameDB)) )))
```

Descrizione dei campi:

`ID` Rappresenta l'ID della richiesta di servizio. Sarà legato alla specifica richiesta di servizio dal momento in cui viene effettuata una richiesta fino al momento in cui vengono prodotti i risultati.

`TargerTC` Rappresenta il centro di tuple sul quale l'utente vuole che gli vengano restituiti i risultati della ricerca.

`coulumnList(CoulumnList)` Rappresenta la lista delle informazioni che il richiedente del servizio vuole ottenere (es. nome,mail,...) (Corrisponderà poi ai nomi delle colonne che l'agente inserirà nella query di richiesta).

`bondList(bondList)` Rappresenta la lista dei vincoli che l'agente deve rispettare nel selezionare le informazioni che poi andrà a fornire come risposta (es. nome="Topolino") (corrisponderà alla parte dei vincoli che si andrà ad inserire nella query).

`option(nameDB(NameDB))` se un utente desidera selezionare i dati da un database specifico può inserire il nome qui. È opzionale perché sarebbe meglio che l'agente utilizzasse tutti i database disponili e da quelli che rispondono alle caratteristiche richieste prelevasse le risposte con tutte le possibili alternative.

Questa è il formato delle tuple per richiede la disconnessione dall'agente di un database condiviso:

```
service_request(dB,ProfileName,ID,
   disconnectDB(nameDB(NameDB),options(Options) ))
```

Descrizione dei campi:

`ID` Rappresenta l'ID della richiesta di servizio. Sarà legato alla specifica richiesta di servizio dal momento in cui viene effettuata una richiesta fino al momento in cui vengono prodotti i risultati.

`nameDB(NameDB)` Rappresenta il nome del database di cui si richiede la disconnessione dall'agente.

`options(Options)` Rappresentano eventuali opzioni da soddisfare prima, durante e/o dopo la disconnessione (es. se si vuole che il database sia disconnesso immediatamente o che prima vengano soddisfatte le richieste ancora pendenti).

Le tuple del profilo per la connessione del database hanno il seguente formato:

```
user_profile(dB/connectDB,ProfileName,
     info(host(Host),user(User),pwd(Password),
         typeOfDB(TypeOfDB),options() ))
```

Descrizione dei campi:

`ProfileName` Rappresenta il nome del profilo da cui si ricavano le informazioni personali dell'utente legate a questo servizio.

`host(Host)` Rappresenta l'indirizzo della macchina da cui l'utente effettua le richieste.

`user(User)` Rappresenta lo pseudonimo che l'utente utilizza per connettersi al database.

`pwd(Password)` Rappresenta la password dell'utente.

`typeOfDB(TypeOfDB)` Rappresenta il tipo di database manager con il quale è stato creato e viene gestito il database che si sta inserendo. Serve per la creazione della struttura del database da parte dell'agente nel caso in cui non riceva il file XML con le descrizione e le specifiche del database.

`options()` Rappresenta eventuali opzioni.

Le tuple del profilo per la richiesta delle informazioni da database hanno il seguente formato:

```
user_profile(dB/getInfoDB,ProfileName,
     info( host(Host),user(User),pwd(Password),options()))
```

Descrizione dei campi:

`ProfileName` Rappresenta il nome del profilo da cui si ricavano le informazioni "personali" dell'utente legate a questo servizio.

`host(Host)` Rappresenta l'indirizzo della macchina da cui l'utente effettua le richieste.

`user(User)` Rappresenta lo pseudonimo che l'utente utilizza per connettersi al database.

`pwd(Password)` Rappresenta la password dell'utente.

`options()` Rappresenta eventuali opzioni relative all'utente.

Le tuple del profilo per la richiesta di disconnessione di un database hanno il seguente formato:

```
user_profile(dB/disconnectDB,ProfileName,
      info(host(Host),user(User),pwd(Password),options() ))
```

Descrizione dei campi:

`ProfileName`	Rappresenta il nome del profilo da cui si ricavano le informazioni "personali" dell'utente legate a questo servizio.
`host(Host)`	Rappresenta l'indirizzo della macchina da cui l'utente effettua le richieste.
`user(User)`	Rappresenta lo pseudonimo che l'utente utilizza per connettersi al database.
`pwd(Password)`	Rappresenta la password dell'utente.
`options()`	Rappresenta eventuali opzioni relative all'utente.

Tutti e tre i servizi, dopo aver svolto il loro compito, istanziano una tupla, indicante l'esito del servizio svolto, che è del tipo:

```
service_result(ID,Status)
```

dove i campi rappresentano:

`ID`	L'ID della richiesta di servizio che è stato elaborato.
`Status`	Stato del servizio elaborato.

La tupla viene poi inserita nel Centro di Tuple di competenza (Centro di Tuple `outgoing` per i servizi di `connectDB` e `disconnectDB` e Centro di Tuple `incoming` per il servizio di `getInfoDB`).

In risposta alla richiesta di informazioni oltre che alla tupla `service_result` vengono anche istanziate e depositate nel centro di tuple indicato nella richiesta, tante tuple `db_header` quanti sono i database, gestiti dall'agente, che contengono le informazioni desiderate e per ognuna di queste tante tuple `db_body` quanti sono i dati trovati in quel database.

Formato delle tuple in risposta alla richiesta di informazioni:

```
db_header(ID,numberOfResult(number),coulumnList(List),
     numberOfDB(NumberOfDB))
```

Descrizione dei campi:

`ID` Rappresenta l'ID corrispondete all'ID della richiesta.

`numberOfResult(number)` indica quanti record sono risultati dalla ricerca all'interno di uno specifico database. È utile se si decide di verificare che tutti i dati giungano effettivamente a destinazione o per effettuare dei controlli per la ricerca di guasti.

`coulumnList(List)` Rappresenta la lista delle informazione in risposta alla richiesta. L'ordine presente all'interno di questo campo fornisce l'ordine per i valori contenuti nel campo `resultList()` delle tuple `db_body`. (es. se qui si trova Nome, Cognome dentro al `resultList` si troverà Luca, Tonini e non viceversa)

`numberOfDB(NumberOfDB)` numero del database da cui si ottengono le informazioni. È un identificativo associato ad un determinato database e ad ogni tupla `db_body` contenente dei dati di esso sarà associato questo numero.

Formato delle tuple contenenti i dati delle risposte:

```
db_body(ID,resultList(List),count(Count),
     numberOfDB(NumberOfDB))
```

Descrizione dei campi:

`ID` Rappresenta l'ID corrispondete all'ID della richiesta.

`resultList(List)` contiene i dati selezionati nell'ordine indicato dal campo coulumbList dell'header.

`count(Count)` numero della tupla. Serve quando ci si trova in situazione di errore nella ricezione e quindi si effettuano politiche di rispedizione (non occorre rinviare tutto ma solo quello che manca).

`numberOfDB(NumberOfDB)` numero del database da cui si ottengono queste informazioni. È associato al `numberOfDB` della tupla `db_header`

2.3 Struttura dell'Agente

Per realizzare un agente il più possibile modulare e quindi anche il più possibile aggiornabile il progetto pensato è il seguente:

- un componente principale che si occupi di:
  - ricercare periodicamente nel centro di tuple `services` le richieste di servizio (la `getInfoDB` e la `disconnectDB` occorre controllarle solo nel caso ci sia almeno un database collegato all'agente);
  - ricavare, una volta trovata una richiesta, dal centro di tuple `profiles` il profilo dell'utente;
  - smistare poi le operazioni da svolgere fra i vari componenti;
  - costruire e inserire le tuple di risposta nei rispettivi centri di tuple;
- ✓ un componente che si occupi di un singolo database e di questo ne conosca il formato di comunicazione, la struttura delle tabelle, quali sono le colonne che contiene e tutto quello che lo riguarda;
- ✓ un componente che contenga i collegamenti a tutti i database già connessi, che si occupi di inizializzare e interfacciare i nuovi database che si vorranno connettere e di disconnettere quelli che ne fanno richiesta;
- ✓ Un componente che, una volta ottenute le informazioni che si vogliono cercare, generi, selezionando i database che hanno le caratteristiche cercate, la query per ottenere le informazioni volute;

2.4 I componenti in dettaglio

AgentDB:

È il componente principale che viene eseguito all'avvio dell'applicazione. Si occupa di:

1) Interfacciarsi con l'infrastruttura TuCSoN e di permettere la comunicazione tra l'agente e il resto del mondo.
2) Istanziare un componente *DataBaseManager* che si occupi di gestire i database connessi al sistema.
3) Istanziare un componente *SelectGenerator* che si occupi di trovare all'interno dell'agente i dati per rispondere ad una richiesta di ricerca di informazioni.
4) Ricercare nei Centri di Tuple assegnati a questo agente le `service_request` dei servizi che è in grado di soddisfare. Appena ne trovatene almeno una, eseguirà le seguenti operazioni:
   a) Se è una richiesta di ricerca di dati:
      i) Attraverso il componente *SelectGenerator*, ricavare i dati necessari per rispondere alla richiesta;
      ii) Analizzare i dati che sono stati trovati, per vedere che non siano vuoti o errati;
      iii) Predisporre le tuple con i valori trovati nel formato specifico per ogni riposta;
   b) Se è una richiesta di connessione/disconnessione di un database:
      i) Effettuare, con i parametri corretti, la richiesta di connessione/disconnessione di un database componente *DataBaseManager*.
      ii) Predisporre le tuple di risposta in base a come è andata l'operazione.

c) Depositare le tuple nei corretti Centri di Tuple.

5) Ricercare nuove richieste di servizio (Punto 4)

SelectGenerator:

Si occupa di:

1) Ricevere dall'*AgentDB* il tipo di informazioni che si devono ricercare ed i vincoli che si devono imporre.
2) Richiedere al componente *DataBaseManager* solo i database che potrebbero contenere le informazioni richieste (se un database non ha una colonna "Nome" non lo seleziono se nella lista delle informazioni da cercare c'è "Nome").
3) Analizzare la sua struttura di ogni database candidato a contenere le informazioni richieste e generare la corretta stringa SQL per interrogarlo.
4) Richiedere al database di effettuare l'interrogazione e di fornire i risultati.
5) Inviare tutti i risultati ottenuti all'*AgentDB*

DataBaseManager

Si occupa di soddisfare le richieste:

1) Del componente *AgentDB* di effettuare la connessione ad un nuovo database e quindi di tenerne il riferimento per utilizzarlo in seguito.
2) Del componente *SelectGenerator* di selezionare solo i database che hanno determinate caratteristiche.
3) Del componente *AgentDB* di effettuare la disconnessione di un database.

DataBase

È il componente che rappresenta un singolo database e ha i seguenti compiti:

1) Conoscere la struttura del database, delle sue tabelle, delle colonne contenute in esse e informare il componente *DataBaseManager*, quando ne fa richiesta, sulla possibilità che il database possa contenere determinate informazioni.
2) Fornire al componente *SelectGenerator* le informazioni sulla sua struttura affinché questo possa generare una stringa SQL per l'interrogazione.
3) Effettuare sul database le ricerche di dati tramite la richiesta fatta dal componente *SelectGenerator*.

CAPITOLO 3
IMPLEMENTAZIONE

3.1 Scelte implementative

Per realizzare l'agente si sceglie di utilizzare come linguaggio di sviluppo JAVA[SUN] in quanto è un linguaggio freeware, tecnologicamente evoluto e disponibile per piattaforme differenti. In più l'infrastruttura TuCSoN gia implementa metodi per l'accesso da classi realizzate con tale linguaggio. Per quanto riguarda il DBMS di riferimento si adotta MySQL[MY] in quanto è anche quest'ultimo un prodotto freeware, in continua evoluzione e di diffusione molto ampia. Quest'ultima scelta però non è vincolante, purché si adottino alcuni accorgimenti. Per quanto riguarda il formato delle tuple di richiesta e risposta si assume che tutti gli `ID` siano dei numeri progressivi associati alla richiesta. In conformità con la struttura delle directory adottate dal progetto TuCSoN e dai servizi già presenti, si definisce un package *agentDB* contenete tutte le classi dell'agente sviluppato.

3.2 Gestione della struttura del database

Per scelta, in questa versione dell'agente, vengono imposti alcuni vincoli sulla struttura del database e sulle conoscenza di questa, da parte degli utenti che vogliono condividere una loro base di dati.

Si richiede infatti che il database non contenga, in tabelle diverse, una colonna con lo stesso nome (se cosi non fosse bisognerebbe aspettarsi di ricevere un numero di colonne maggiori in risposta ad una richiesta con la ricerca di una sola informazione) e che se una tabella ha più colonne che riferiscono a colonne di altre tabelle, sarà al massimo una colonna che fa riferimento ad una determinata tabella. Si richiede inoltre che una tabella

non contenga colonne che referenzino colonne di se stessa. Questi vincoli sono dovuti al fatto che si è scelto, in fase implementativa, di creare un sistema semplice, che non sia specifico per un determinato modello di database e quindi la funzione di generazione di query si occuperà dei casi più comuni. Effettuare delle ricerche all'interno di tabelle che hanno riferimenti a se stesse, non è cosa semplice e comporta la definizione di query complesse o query innestate. Tali scelte non vincoleranno un possibile sviluppo dell'agente per renderlo in grado di gestire anche queste situazioni e per fare ciò si potranno seguire due strade differenti ma non esclusive:

1. espandere il componente predisposto alla generazione di query in modo che sia in grado di effettuare query complesse;
2. più semplicemente fornendo, al momento della connessione e attraverso il file di descrizione della struttura del database, anche le query già predisposte per ricercare i dati all'interno delle tabelle che prima ponevano dei vincoli.

La struttura del database viene gestita in modi differenti a secondo del tipo di database che si sta utilizzando.

a) Database generici

La classe *DataBase* è in grado di gestire un qualunque tipo di database purché nella richiesta di connessione all'agente venga anche fornito un file in formato XML contenente la descrizione di tutte le tabelle e di tutte le colonne appartenenti al database. La descrizione delle colonne dovrà anche indicare quali colonne fanno parte delle chiavi primarie e quali fanno riferimento a colonne di altre tabelle. La struttura di questo file dovrà essere la seguente (in formato DTD):

```
<!DOCTYPE DATABASE [
<!ELEMENT DATABASE (TABLE+)>
```

```
<!ATTLIST DATABASE
        nome CDATA #REQUIRED
        type CDATA #IMPLIED>
<!ELEMENT TABLE (COLONNA+,PRIMARY_KEY*)>
<!ATTLIST TABLE name CDATA #REQUIRED>
<!ELEMENT COLONNA (REFERENCES*)>
<!ATTLIST COLONNA
        type CDATA #IMPLIED>
<!ELEMENT REFERENCES (TABLE_REFERENCE,COLONNA_REFERENCE)>
<!ELEMENT TABLE_REFERENCE (#PCDATA)>
<!ELEMENT COLONNA_REFERENCE (#PCDATA)> ]>
```

o più semplicemente potrà essere realizzata modificando e aggiustando il seguente esempio in formato XML:

```
<DATABASE nome="nome del database" type="tipo di database">
  <TABLE name="nome della 1ª tabella">
    <COLONNA>nome della 1ª colonna</COLONNA>
    <COLONNA>nome della seconda colonna</COLONNA>
    <PRIMARY_KEY>nome della colonna primary</PRIMARY_KEY>
  </TABLE>
  <TABLE name="Nome della 2ª tabella">
    <COLONNA>nome della 1ª colonna
      <REFERENCES>
        <TABLE_REFERENCE>nome della tabella referenziata</TABLE_REFERENCE>
        <COLONNA_REFERENCE>nome colonna referenziata</COLONNA_REFERENCE>
      </REFERENCES>
      ...
    </COLONNA>
    <COLONNA>nome della 2ª colonna</COLONNA>
    ...
    <PRIMARY_KEY>nome della colonna primary</PRIMARY_KEY>
  </TABLE>
  ...
</DATABASE>
```

Se invece si desidera demandare la costruzione del database al sistema, occorre che venga creata una classe che estenda *DataBase* e che nel momento in cui viene istanziata, contenga al suo interno la funzione di creazione della struttura.

b) Database di MySQL

Avendo scelto come DBMS di test MySQL si realizza per un qualunque database di questo tipo una classe denominata appunto *DataBaseMySql*. Questa implementa al suo interno anche la funzione di definizione della struttura. Purtroppo, a causa del fatto che MySql non gestisce da sé i referimenti, se non per particolari database, occorre comunque utilizzare un file XML per descrivere le colonne che sono chiavi primarie e le colonne che fanno riferimento a colonne di altre tabelle.

Il file XML sarà simile a quello di descrizione di un database normale con l'unica differenza che le colonne che non fanno riferimenti ad altre colonne e le colonne che non fanno parte delle chiavi primarie possono essere tralasciate.

In fase implementativa si sceglie di convertire tutti i valori dei nomi delle tabelle, delle colonne, delle chiavi esterne e delle chiavi primarie in minuscolo. In questo modo l'utente non è costretto a conoscere anche il formato (maiuscolo/minuscolo) dell'informazione che ricerca perché, anche dal suo lato, tutti i valori delle informazioni che richiede vengono convertiti in minuscolo.

Da parte dell'utente è necessario che, quando desidera effettuare una richiesta di ricerca di dati, si ricordi che i valori inseriti nella `service_request` corrisponderanno al nome delle colonne dal database. Se voglio ricercare degli indirizzi di posta elettronica e inserisco nella `service_request` come parametro il valore email il sistema selezionerà solo i database che hanno al loro interno una colonna che si chiama email. Per ovviare a problemi di formato in tutti i campi prelevati dalla `service_request` e dal `user_profile` vengono eliminati, se presenti, gli apici inseriti dall'infrastruttura ad inizio e fine stringa.

3.3 Dettegli dei parametri per la richiesta di connessione

Per la richiesta di connessione di un database al sistema, i parametri sono tutti di tipo stringa con l'accortezza che:

- il parametro `typeOfConnection` deve essere il tipo di driver usato per connettersi al database.
- se vi è il `FileOptional` deve essere un percorso di directory valido che riferisca al file di descrizione.
- il parametro `Url` deve essere il driver usato per connettersi al database.

Esempio.

```
service_request(dB,ProfiloLuca,7,connectDB(typeOfConnection(
odbc),nameDB(rubrica),fileOptional("c:/descrizioni/DB.xml"),
url("sun.jdbc.odbc.JdbcOdbcDriver"))))
```

3.4 Dettagli dei parametri per la richiesta di informazione

I parametri fondamentali da inserire nella tupla di richiesta sono `coulumnList(CoulumnList)`, `boundList(BoundList)` mentre il parametro `TargetTC`, che indica su quale centro di tuple si desidera deporre i dati trovati, può essere anche vuoto in quanto l'agente nel caso in cui non trovi questo valore usa di default il centro di tuple `incoming`.

`CoulumnList` deve essere composto da una lista di stringe (almeno una o non ha senso la richiesta) che rappresentano le informazioni che si desidera far ricercare dall'agente all'interno dei database che gestisce. Ogni stringa rappresenta un'informazione.

Esempio:

`coulumnList([CF,Nome,Cognome,Email])` indica che si vuole che l'agente prelevi dai suoi database tutti quelli che contengono delle colonne

CF, Nome, Cognome, Email e risponda con i dati che ogni riga del database contiene.

`BoundList` può essere composto da una lista di stringhe che formeranno i vincoli da imporre nella ricerca dei dati. Ogni stringa può essere un pezzo o tutto l'intero vincolo. La struttura del vincolo deve essere in formato SQL in quanto, a parte alcuni accorgimenti tecnici, viene inserito cosi com'è direttamente nella query. Occorre che un vincolo sia imposto su un campo della richiesta (non si può richiedere il Nome e imporre come vincolo che il Cognome sia uguale a qualcosa). Se i vincoli sono più di uno occorre inserire tra uno e l'altro anche l'operatore logico di congiunzione tra essi (and,or,...).

Esempio:

```
boundList([Nome,like,'%uca%',and,"Cognome like '%onin%'"])
```

indica che nella ricerca dei dati voglio che mi vengano restituiti solo quelli che hanno un Nome composto da qualcosa che contiene "uca" e da un Cognome composto da qualcosa che contiene "onin".

Esempio:

```
service_request(dB,ProfiloLuca,8,getInfoDB("datitrovati",
coulumnList([CF,Nome,Cognome,Via]),boundList([Nome,like,"'%a'"])
,option("")))
```

3.5 Dettaglio dei risultati ottenuti da una richiesta

I risultati di una richiesta vengono inviati all'utente depositandoli nel centro di tuple che ha specificato nella richiesta. Le tuple di risposta sono di due tipi: `db_header` e `db_body`

I due tipi di tuple sono legati perché uno contiene la descrizione dei dati contenuti nel secondo. Per ogni database contenente delle informazioni

viene restituito una tupla db_header e tante tuple db_body quanti sono i dati trovati.

I campi importanti nella tupla db_header sono :

numberOfResult(Number),coulumnList(List),
numberOfDB(numberOfDB))

Number è un numero intero che indica quanti dati sono stati trovati nel database e corrisponde al numero di tuple db_body che saranno associate a questo db_header.

List è la lista di stringhe delle informazioni che si erano inserite nella richiesta. È molto importante come campo perchè fornisce all'utente l'ordine di come ricavare i dati contenuti nel campo List dei db_body associati.

numberOfDB è un numero intero che corrisponde ad una risposta derivata da uno specifico database. Tramite questo valore è possibile associare le corrette tuple db_body e prelevare i dati.

I campi importanti nella tupla db_body sono:

resultList(List),count(Count),numberOfDB(NumberOfDB))

List contiene la lista di stringhe dei dati prelevati da una riga del database nell'ordine indicato dal db_header.

Count è un numero progressivo di ogni tupla. Va da 1 al numero di dati trovati all'interno del database.

NumberOfDB è il numero del database da cui sono stai prelevati i dati contenuti nella tupla. È associato al NumberOfDB della tupla db_header in modo che ogni tupla db_body può avere una sola tupla db_header di riferimento.

CAPITOLO 4
INSTALLAZIONE E COLLAUDO

4.1 Installazione e configurazione di MySql.

Nello sviluppo del prototipo e nelle successive fasi di collaudo si utilizza come database manager l'applicazione MySql versione 12.20 con distribuzione 4.0.13 per sistemi operativi Win95/Win98. Affiancata a questa applicazione verranno usati anche i driver, per connettersi al database tramite ODBC, MyODBC, nella versione 3.51.06, messi a disposizione dalla stessa casa produttrice. Installate entrambe le applicazioni bisogna rendere disponibile un database, tramite il driver ODBC, aggiungendolo nel componente "Origine Dati" del sistema operativo.

Per effettuare quest'ultima operazione occorre seguire i seguenti passaggi: aprire "Origine dati (ODBC)", nella finestra che appare selezionare la cartella "DNS utente" e selezionare "Aggiungi". (Figura 1)

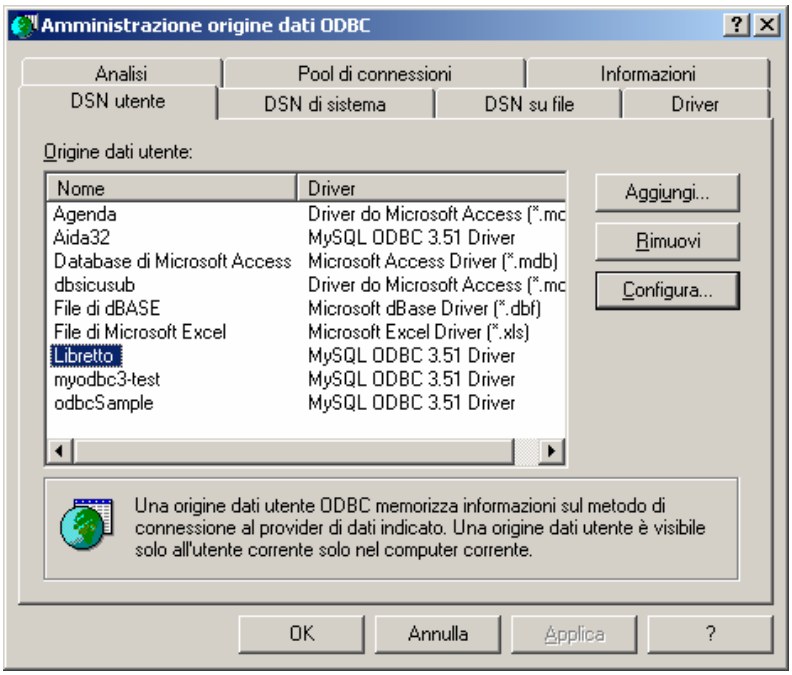

Figura 1 Finestra del sistema operativo di "Origine dati ODBC" del sistema operativo

Nella successiva finestra selezionare il driver ODBC corretto per connettersi alla base di dati scelta e in questo caso è quello di MySQL ODBC 3.51 DRIVER e premere su “Fine”.(Figura 2)

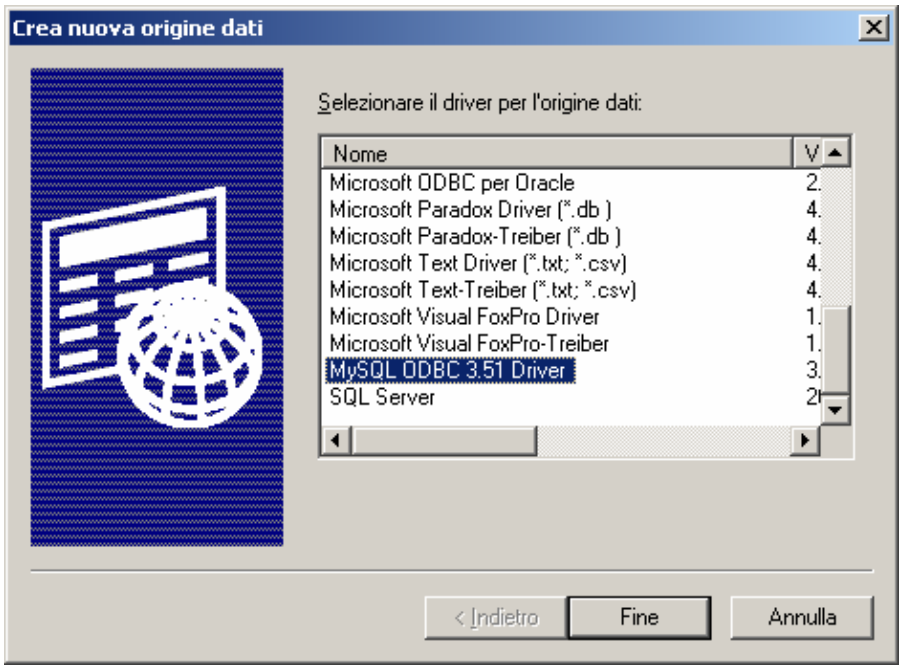

Figura 2 Finestra per creare una nuova origine dati

Nella finestra di MySQL DRIVER inserire i parametri che indicano su quale host si trova il database, il nome del database ed eventuali username e password per accedervi.(Figura 3)

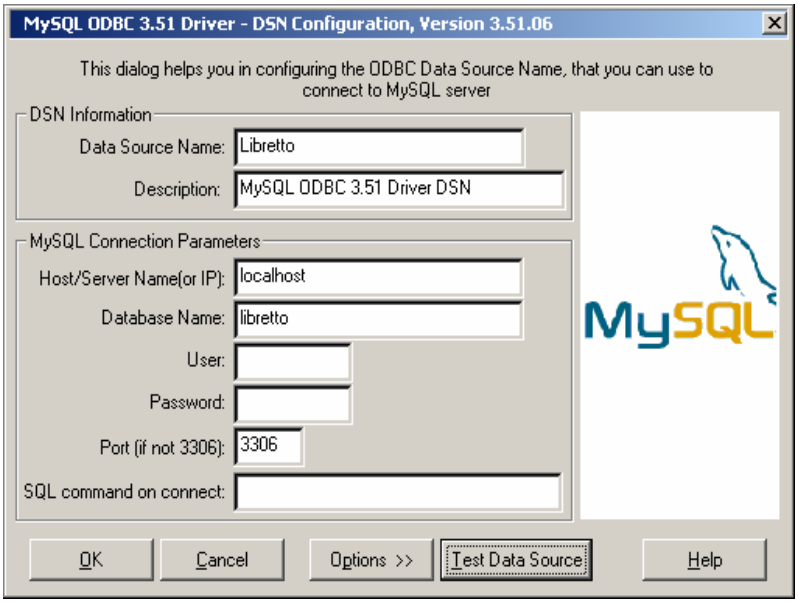

Figura 3 Finestra di MySql ODBC Driver

4.2 Il file XML di descrizione del database

A questo punto conoscendo la struttura del database appena creato si deve realizzare il file XML di descrizione (di tipo MySql) da aggiungere alla richiesta di connessione del database.

Si osservi il seguente esempio:

```
<?xml version="1.0"?>

<!DOCTYPE DATABASE [
<!ELEMENT DATABASE (TABLE+)>
<!ATTLIST DATABASE
        nome CDATA #REQUIRED
        type CDATA #IMPLIED>
<!ELEMENT TABLE (COLONNA+,PRIMARY_KEY*)>
<!ATTLIST TABLE name CDATA #REQUIRED>
<!ELEMENT COLONNA (REFERENCES*)>
<!ATTLIST COLONNA
        type CDATA #IMPLIED>
<!ELEMENT REFERENCES (TABLE_REFERENCE,COLONNA_REFERENCE)>
<!ELEMENT TABLE_REFERENCE (#PCDATA)>
<!ELEMENT COLONNA_REFERENCE (#PCDATA)>
]>

<DATABASE nome="libretto" type="MySql">
        <TABLE name="studente">
                <COLONNA>CF</COLONNA>
                <PRIMARY_KEY>CF</PRIMARY_KEY>
        </TABLE>
        <TABLE name="recapito">
                <COLONNA>CF_EXT
                        <REFERENCES>
                                <TABLE_REFERENCE>studente</TABLE_REFERENCE>
                                <COLONNA_REFERENCE>CF</COLONNA_REFERENCE>
                        </REFERENCES>
                </COLONNA>
                <PRIMARY_KEY>ID</PRIMARY_KEY>
        </TABLE>
        <TABLE name="esame">
                <COLONNA>Cod</COLONNA>
                <PRIMARY_KEY>Cod</PRIMARY_KEY>
        </TABLE>
        <TABLE name="risEsame">
                <COLONNA>CF_STUD
                        <REFERENCES>
                                <TABLE_REFERENCE>studente</TABLE_REFERENCE>
                                <COLONNA_REFERENCE>CF</COLONNA_REFERENCE>
                        </REFERENCES>
                </COLONNA>
                <COLONNA>Cod_Esame
                        <REFERENCES>
                                <TABLE_REFERENCE>esame</TABLE_REFERENCE>
                                <COLONNA_REFERENCE>cod</COLONNA_REFERENCE>
                        </REFERENCES>
                </COLONNA>
                <PRIMARY_KEY>ID</PRIMARY_KEY>
        </TABLE>
</DATABASE>
```

4.3 Test dell'Agente

Per collaudare l'agente è necessario, per prima cosa, avere installato sul sistema locale un nodo TuCSoN, un Database Manager e i relativi driver per gestirlo tramite ODBC.

Per effettuare il test bisogna innanzitutto creare un database di riferimento e popolarlo con dei valori. Si osservino le seguenti richieste in formato SQL che sono state utilizzate per creare e popolare un database di esempio che simula il libretto con i voti degli esami di uno studente.

```
create database libretto;

create table Studente (CF char(16) ,Nome varchar(20),Cognome Varchar(20), Matricola
        Integer(10),primary key(CF) );

create table Recapito(ID Integer(10) AUTO_INCREMENT,CF_EXT char(16),Via varchar(30),Telefono
        varchar(10),Email Varchar(25),Fax Varchar(10),Sito Varchar(30),primary key(ID),foreign
        key(CF_EXT) references Utente(CF));

create table RisEsame(CF_STUD char(16),Cod_Esame Integer(10),Voto Integer(3),ID Integer(10)
        AUTO_INCREMENT,primary key(ID),foreign key(CF_STUD) references Utente(CF),foreign
        key(Cod_Esame) references Esame(Cod));
create table Esame(Cod integer(10) AUTO_INCREMENT,Titolo varchar(50),primary key(Cod) );

insert into Studente(CF,Nome,Cognome,Matricola) values ('PPRPPR88A01H100$','Paperon','De
        Paperoni',1010101010);
insert into Studente(CF,Nome,Cognome,Matricola) values
        ('PLNPPR75B03H376D','Paolino','Paperino',2020202020);

insert into Recapito(CF_EXT,Via,Telefono,Email,Fax,Sito) values('PPRPPR88A01H100$','del dollaro',
        100,'paperone$$@dollaro.do',100,'www.vivaidollari.com');
insert into Recapito(CF_EXT,Via,Telefono,Email,Fax,Sito) values('PLNPPR75B03H376D','dello schiavo
        dello zio ', 010,'paperino@schiavo.com',010,'www.paperino.com');

insert into Esame(Titolo) values('Lo Sfruttamento dei nipoti');
insert into Esame(Titolo) values('Onesta sul lavoro');
insert into Esame(Titolo) values('Farsi dal nientè);
insert into Esame(Titolo) values('Togliersi dai Debiti per semprè);
insert into Esame(Titolo) values('Metodi per far guai');

insert into RisEsame(CF_STUD,Cod_Esame,Voto) values('PPRPPR88A01H100$',1,30);
insert into RisEsame(CF_STUD,Cod_Esame,Voto) values('PPRPPR88A01H100$',2,18);
insert into RisEsame(CF_STUD,Cod_Esame,Voto) values('PPRPPR88A01H100$',3,30);
insert into RisEsame(CF_STUD,Cod_Esame,Voto) values('PLNPPR75B03H376D',4,18);
insert into RisEsame(CF_STUD,Cod_Esame,Voto) values('PLNPPR75B03H376D',5,30);
insert into RisEsame(CF_STUD,Cod_Esame,Voto) values('PLNPPR75B03H376D',1,18);
```

A questo punto occorre avviare un nodo dell'infrastruttura e inserire attraverso un agente messo a disposizione dalla infrastruttura TuCSoN, *CLIAgent*, le seguenti tuple dei profili nel centro di tuple profiles.(Figura 4)

```
profiles?out(user_profile("dB/connectDB",ProfiloLuca,info(
      host(localhost),user(Luca),pwd("luca"),typeOfDB(MySql),
      options(""))))
profiles?out(user_profile("dB/getInfoDB",ProfiloLuca,info(
      host(localhost),user(""), pwd(""),options(""))))
profiles?out(user_profile("dB/disconnectDB",ProfiloLuca,info(
      host(localhost), user(Luca),pwd("luca"),options(""))))
```

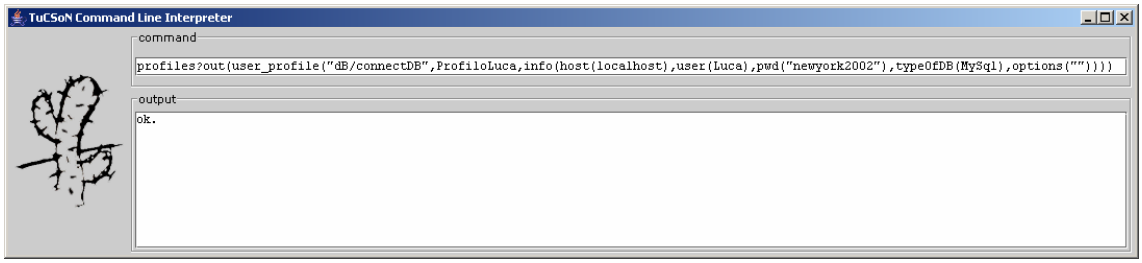

Figura 4 Inserimento di una tupla di profilo

Effettuate queste procedure bisogna avviare, attraverso il file batch AgentDB.bat precedentemente realizzato, il client AgentDB che una volta interfacciato al sistema TuCSoN si pone in attesa di una nuova richiesta di connessione di un database.(Figura 5)

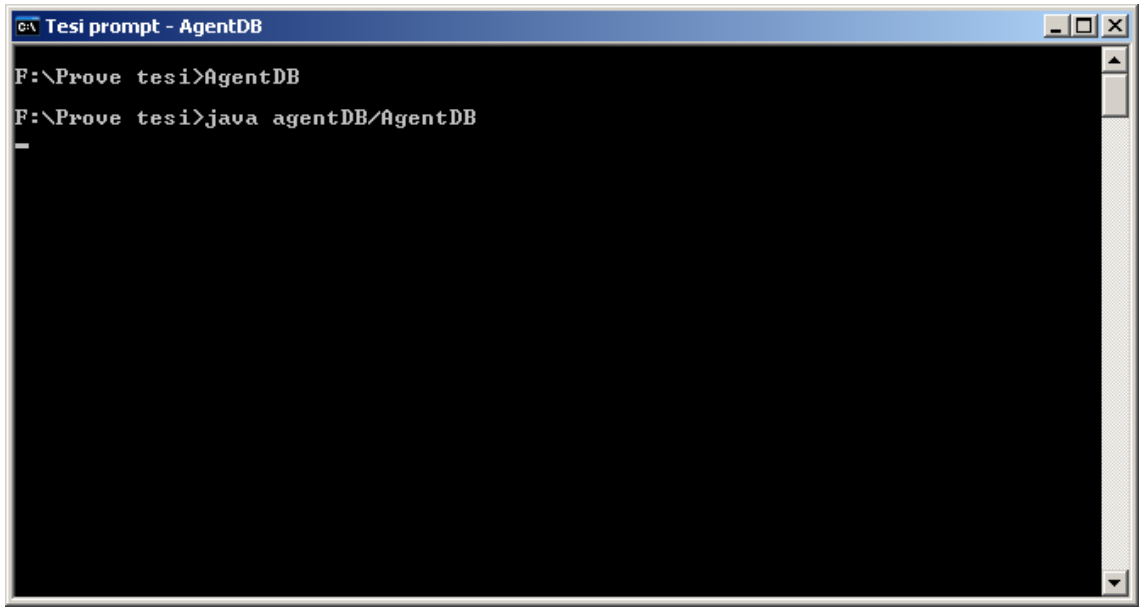

Figura 5 AgentDB in attesa di nuove richieste di servizio

Attraverso lo stesso *CLIAgent* usato in precedenza si deve poi simulare, inserendo la seguente richiesta, un utente che desidera connettere la sua base di dati per renderla disponibile agli utenti dell'infrastruttura:

```
services?out(service_request(dB,ProfiloLuca,1,connectDB(
       typeOfConnection(odbc),nameDB(libretto),
       fileOptional("f:/tesi/DBlibretto.xml"),
       url("sun.jdbc.odbc.JdbcOdbcDriver"))))
```

Con l'applicazione Inspector si può poi verificare sul centro di tuple `outgoing` la tupla `service_result` che l'agente ha comunicato in risposta al servizio richiesto.(Figura 6)

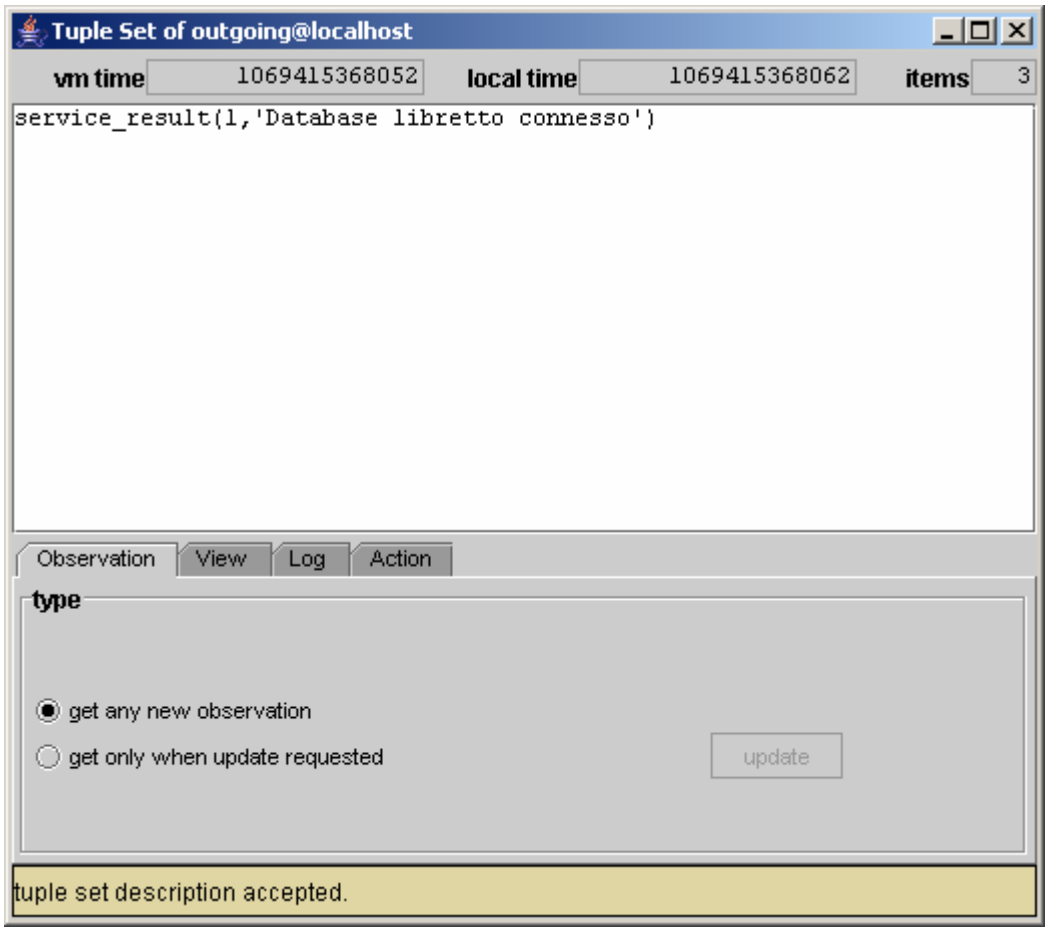

Figura 6 Centro di Tuple outgoing in risposta alla richiesta di connessione database

Si passa poi alla simulazione di un utente che richiede all'agente delle informazioni contenute nelle basi di dati che gestisce (in questo caso una) inserendo la seguente `service_request` nel centro di tuple `incoming`:

```
services?out(service_request(dB,ProfiloLuca,2,getInfoDB(
       "risposta",coulumnList([CF,Nome,Cognome,Telefono,Titolo,
       Voto]),boundList( [Nome,like,"'paper%'"]),option(""))))
```

si verifica poi cosa risponde l'agente nel centro di tuple `incoming` e successivamente nel centro di tuple `risposta` specificato nella `service_request`.(Figura 7 e Figura 8)

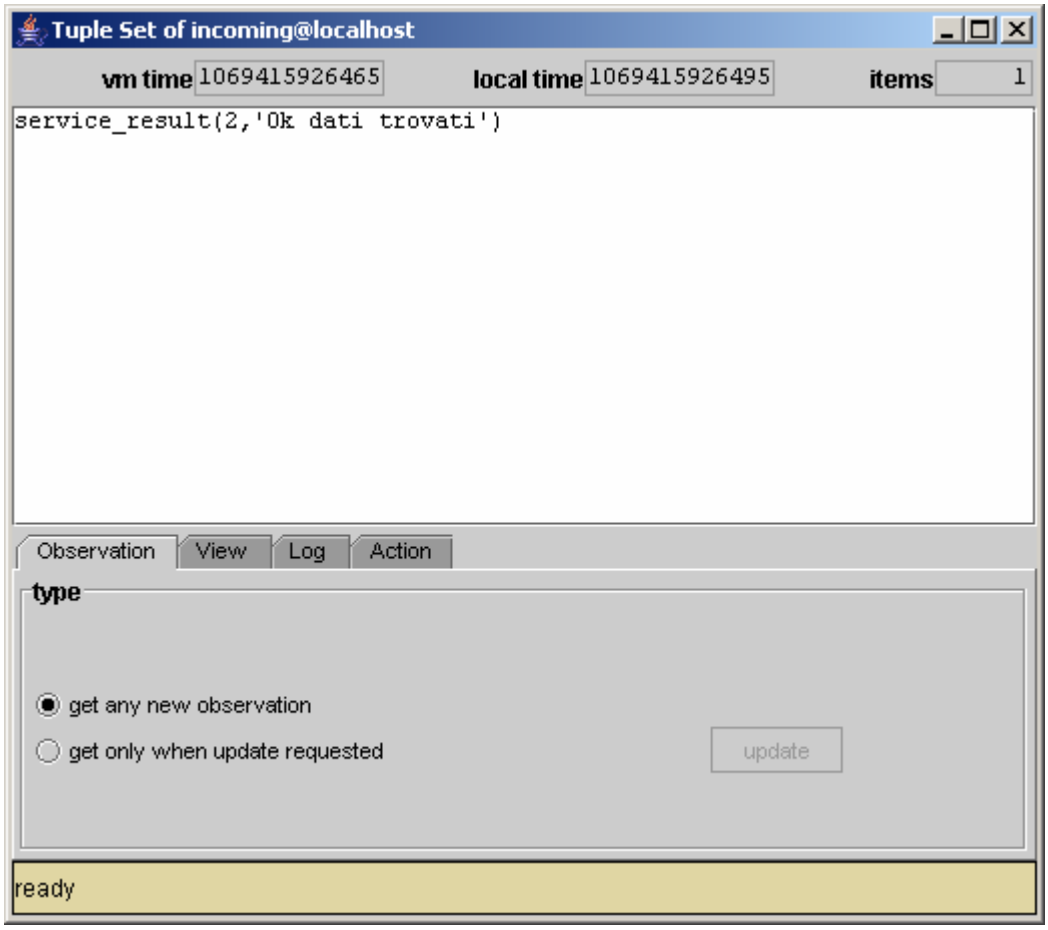

Figura 7 Centro di tuple incoming in risposta alla richiesta di informazioni

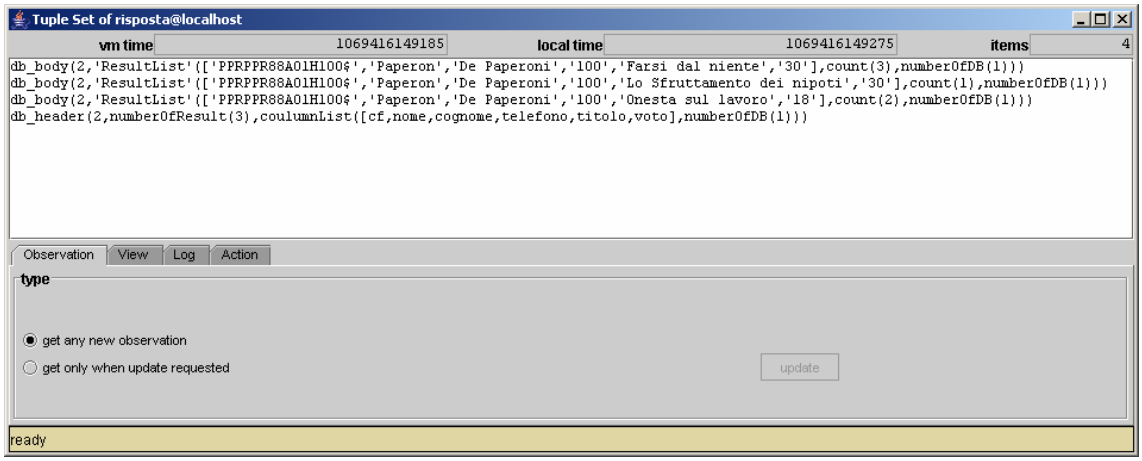

Figura 8 Centro di tuple "risposta" contenete i risultati della richiesta di informazioni

Infine si simula una richiesta di disconnessione di un database tramite la tupla:

```
services?out(service_request(dB,ProfiloLuca,3,disconnectDB(nameD
      B(libretto), options("") )))
```

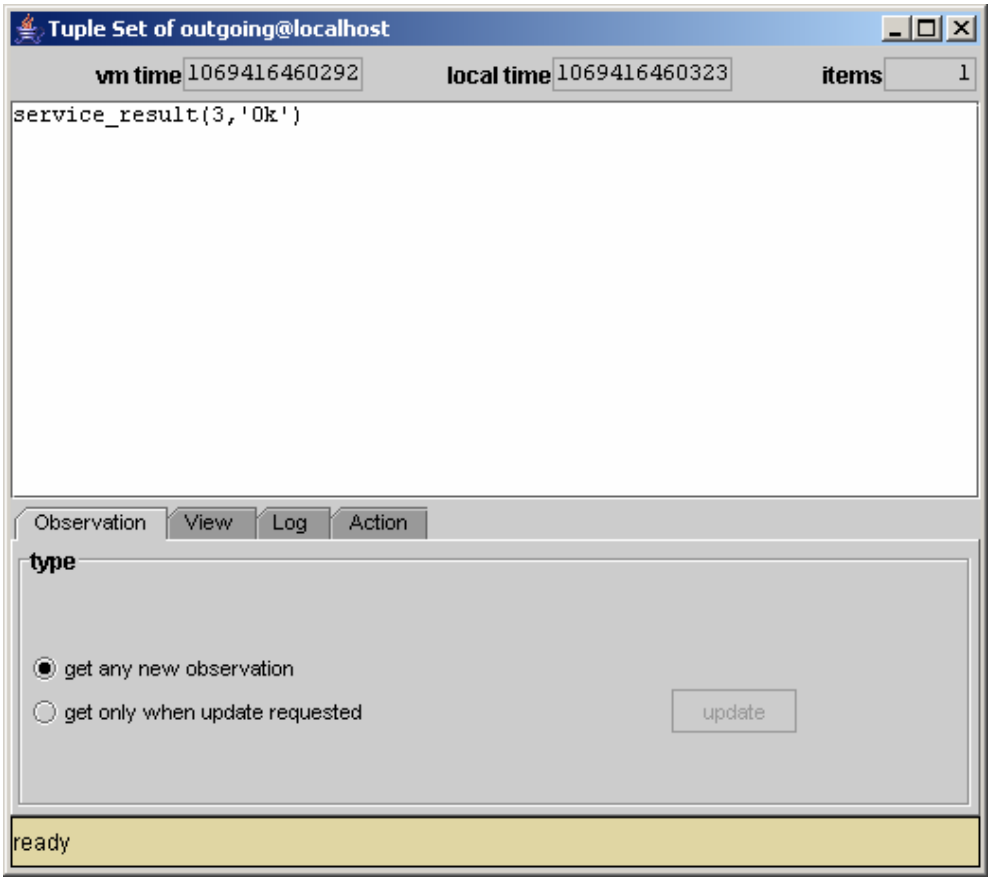

Figura 9 Centro di tuple outgoing in risposta ad una richiesta di disconnessione database

CONCLUSIONI

Nel corso di questa tesi si sono definite le specifiche per la progettazione e l'implementazione di un agente in grado di rendere disponibile, per tutti gli utenti dell'infrastruttura di coordinazione TuCSoN, un servizio di accesso a basi di dati evitando all'utente l'onere di conoscere i dettagli della struttura dati da cui si prelevano e dei metodi necessari per accedervi. Nella situazione attuale il prototipo permette di avviare l'agente collegandolo ad un nodo TuCSoN e, attraverso delle richieste effettuate secondo le regole dettate dall'infrastruttura, è in grado di prelevare e soddisfare i servizi fondamentali:

- Connessione di un database, generico o realizzato con MySql. La descrizione del database dovrà essere effettuata, con modalità differenti per i due tipi, tramite file XML. L'accesso fisico al database viene gestito tramite driver JDBC/ODBC.
- Ricerca di dati all'interno dei database gestiti dall'agente. Le informazioni richieste dovranno essere contenute nella richiesta assieme ai vincoli da imporre nella ricerca.
- Disconnessione di uno o più database.

Sia la struttura dell'agente, sia le specifiche e il formato delle tuple non sono vincolate alle scelte attuali, ma permettono di estendere le funzioni attualmente implementate. Seppur predisposto a livello progettuale, l'agente nella situazione attuale non si occupa di applicare una politica di sicurezza nell'accesso ai dati: di conseguenza un qualunque utente dell'infrastruttura può richiedere informazioni ai database interfacciati. Uno sviluppo interessante dell'implementazione attuale consisterebbe nel permettere al proprietario di un database di fornire all'agente delle query già preparate. Questo sviluppo non richiede numerosi

cambiamenti dell'agente in quanto il file XML, che viene passato al momento della connessione, potrebbe contenere anche delle query già preparate e basterebbe cambiare la classe che preleva i dati dal file per prendere anche le query e modificare la classe che gestisce i database per farle gestire anche le query già fatte. Il vantaggio di questa estensione risulterebbe evidente nel caso in cui, per ricercare dei dati all'interno di un database, servano delle query complicate che l'agente non è in grado di realizzare dinamicamente e che quindi verrebbero scartate. In questo modo invece le query complicate verrebbero già fatte preparare al proprietario del database e le query normali verrebbero gestite tranquillamente dell'agente.

Un ulteriore sviluppo dell'agente potrebbe essere quello di fornire un servizio che permetta ad un utente non solo di richiedere informazioni ma anche di richiedere l'esecuzione di query realizzate per uno specifico scopo. Questo servizio infatti permetterebbe oltre a fare delle ricerche mirate, con select anche complesse, anche di modificare o aggiungere dati alla base di dati. Infine risulterebbe indispensabile, in caso di grandi quantità di dati richieste e scambiate con gli utenti (e quindi tempi di elaborazione elevati), permettere all'agente di analizzare più richieste contemporaneamente (attraverso processi multipli) in modo da accorciare i tempi di attesa.

APPENDICE A: CODICE

AgentDB

```
import javax.sql.*;
import agentDB.database.*;
import agentDB.service.*;
import alice.tucson.api.*;
import alice.logictuple.*;

public class AgentDB{
    //Stringe utilizzate per i centre tuple
    private static String tupleCentreOutgoing="outgoing";
    private static String tupleCentreIncoming="incoming";
    private static String tupleCentreProfile="profiles @ '127.0.0.1'";
    private static String tupleCentreServices="services @ '127.0.0.1'";
    private static String tupleCentreFromRic;
    private static String localhost="'127.0.0.1'";
    private static String profileName;
    private static int ID=0;
    private static String typeOfConnection;
    private static String nameDB;
    private static String fileOptional;
    private static String urlDB;
    private static String typeOfDB;
    private static String passDB;
    private static String userDB;
    private static String options;
    private static String host;
    private static String coulumnList;
    private static String boundList;
    private static String [] infoRichieste;
    private static String [] boundRichieste;

    ...

/**Ricerca sull'infrastruttura di coordinazione di nuovi Database da aggiungere o di nuove richeste da
elaborare */
public static void main(String argv[]) throws Exception{

    TucsonContext cn = Tucson.enterDefaultContext();
    //Centre tuple dove andare a prelevare le richieste di servizio
    TupleCentreId tidServ = new TupleCentreId(tupleCentreServices);
    //Centre tuple per i Profili
    TupleCentreId tidProf = new TupleCentreId(tupleCentreProfile);
    //Centre tuple per le risposte!!!Attenzione viene usato per mandare le risposte a chi mi chiede il
servizio
    //Viene inizializzato di volta in volta con i parametri che mi vengono richiesti dall'utente

    TupleCentreId tidRisp=null;
    //Tuple Centre per le risposte da mandare nel centro di tuple Incoming(richieste di informazioni)
    TupleCentreId tidInc=new TupleCentreId(tupleCentreIncoming);
    //Tuple Centre per le risposte da mandare nel centro di tuple Outgoing(richieste di connessione e
disconnessione)
    TupleCentreId tidOut=new TupleCentreId(tupleCentreOutgoing);
```

```
//Agancio il gestore dei database e il generarore di Select
DataBaseManager dBManager = new DataBaseManager();
SelectGenerator vSG = new SelectGenerator(dBManager);
LogicTuple tRic=null;
LogicTuple tProf=null;
TupleArgument ta=null;
TupleArgument tb=null;

for(;;){
   try{
      //lettura tupla da Services per richiesta agganci database
      tRic=cn.inp(tidServ, new LogicTuple("service_request",new Value("dB"),new
          Var("Profile"),new Var("ID"),new Value("connectDB",new
          Value("typeOfConnection",new Var("TypeOfConnection")),new Value("nameDB",new
          Var("NamDB")),new Value("fileOptional",new Var("FileOptional")),new Value("url",new
          Var("Url")))  ));
      //lancio gestore database con nuove richieste di aggancio(gli passo i dati del database)
      if (tRic!=null){
          //Ho trovato una richiesta di aggancio di un database.
          //Prelevo i dati dalla richista
          profileName=tRic.getArg(1).toString(); //Prelevo il profilo

          ...

          //prelevo i dati dal Profile indicatomi
          tProf=cn.rd(tidProf,new LogicTuple("user_profile",new Value("dB/connectDB"),new
                Value(profileName),new Value("info",new Value("host",new Var("Host")),new
                Value("user",new Var("User")),new Value("pwd",new Var("Password")),new
                Value("typeOfDB",new Var("TypeOfDB")),new Value("options",new
                Var("Options"))) ) );
          ta=tProf.getArg(2);//ho prelevato la parte info del user_profile

          ...

          //risposta affermativa se agganciato negativa se non agganciato
          if (dBManager.addDataBase(nameDB,userDB,passDB,urlDB,typeOfConnection,typeOfDB,
              fileOptional)){
              //il database è stato agganciato correttamente
              cn.out(tidOut,new LogicTuple("service_result",new Value(ID),new Value("Database
                    "+nameDB+" connesso") ));
              System.out.println("Agganciato al sistema il database: "+nameDB);
          }else{
              //c'è stato un errore nell'aggancio al database
              cn.out(tidOut,new LogicTuple("service_result",new Value(ID),new Value("Errore nella
                    connessione al Database "+nameDB) ));
              System.out.println("Errore!! Non Agganciato al sistema il database: "+nameDB);
          }
      }
      //Se c'almeno un database collegato al sistema
      if (dBManager.conteinDB()){
          //lettura tupla incoming per ricerca di nuovi dati(gli passo i dati che cerco)
          tRic=cn.inp(tidServ,new LogicTuple("service_request",new Value("dB"),new
                Var("ProfileName"),new Var("ID"),new Value("getInfoDB",new Var("TargetTC"),new
                Value("coulumnList",new Var("CoulumnList")),new Value("boundList",new
                Var("BoundList")),new Var("Option") )));
          //Se c'è una richiesta di ricerca di informazioni
          if(tRic!=null){
              //prelevo i dati dalla richiesta
```

```
profileName=tRic.getArg(1).toString();//prelevo il profilo di chi mi ha fatto la richiesta

...
//prelevo il tuple centre sul quale l'utente vuole che io risponda
tupleCentreFromRic=ta.getArg(0).toString();
tupleCentreFromRic =tupleCentreFromRic.replaceAll("\"","");//tolgo gli apici
//se la stringa è vuota rispondo su incoming
if (tupleCentreFromRic.equals("")) tupleCentreFromRic="incoming";

...

//Adesso vado a prendere le informzioni dal Profilo dell'utente
tProf=cn.rd(tidProf,new LogicTuple("user_profile",new Value("dB/getInfoDB"),new
        Value(profileName),new Value("info",new Value("host",new Var("Host")),new
        Value("user",new Var("User")),new Value("pwd",new Var("Password")),new
        Value("options",new Var("Options"))) ) );

...

//lancio generatore di selec
ResultSet rs[]=vSG.getInformation(infoRichieste,boundRichieste);
if (rs!=null){

    ...

    //devo rispondere con i dati che ho trovato
    for(int i=0;i<rs.length;i++){
        //ci sono dei database che hanno dato delle risposte
        numeroDatiFind=0;
        Vector vNomeColonna=new Vector(5,2);
        Vector vDatiColonna=new Vector(5,2);
        if(rs[i]!=null){
            //le risposte sono diverse da nulle

            ...

        }
    }
    if(vValueNome.size()!=0){
        //ho trovato dati
        //rispondo per prima cosa con l'ok
        cn.out(tidInc,new LogicTuple("service_result",new Value(ID),new Value("Ok
                dati trovati") ));
        //tuple centre per la risposta delle informazioni indicatomi dalla request
        tidRisp=new TupleCentreId(tupleCentreFromRic+" @ "+localhost);
        for(int n=0;n<vValueNome.size();n++){
            Value momDati[]=(Value [])vValueDati.get(n);
            Value momNome[]=(Value [])vValueNome.get(n);
            cn.out(tidRisp,new LogicTuple("db_header",new Value(ID),new
                    Value("numberOfResult",new Value(numOfDatiForDB[n])),new
                    Value("coulumnList",(TupleArgument)(momNome[0]),new
                    Value("numberOfDB",new Value(n+1))) ));
            for (int t=0;t<numOfDatiForDB[n];t++){
                        cn.out(tidRisp,new LogicTuple("db_body",new Value(ID),new
                        Value("resultList",(TupleArgument)momDati[t],new
                        Value("count",new Value(t+1)),new
                        Value("numberOfDB",new Value(n+1)) )));
            }
```

```
                    }
                }else{
                    //ho trovato i database ma non i dati che cervavo
                    cn.out(tidInc,new LogicTuple("service_result",new Value(ID),new
                            Value("Database trovati ma non contengono alcun dato")));
                }
            }else{
                //devo rispondere che non ho trovato niente
                cn.out(tidInc,new LogicTuple("service_result",new Value(ID),new
                        Value("Informazioni non trovate") ));
            }
        }
        //leggo se di sono richieste di disconnessione di database
        tRic=cn.inp(tidServ, new LogicTuple("service_request",new Value("dB"),new
                    Var("Profile"),new Var("ID"),new Value("disconnectDB",new
                    Value("nameDB",new Var("NameDB")),new Var("Options")) ));

        if (tRic!=null){
            //Ho trovato una richiesta di disconnessione di un database.
            //Prelevo i dati del database da disconnettere dalla richista
            profileName=tRic.getArg(1).toString(); //Prelevo il profilo

            ...

            //Prendo i dati dal profilo
            tProf=cn.rd(tidProf,new LogicTuple("user_profile",new Value("dB/disconnectDB"),new
                    Value(profileName),new Value("info",new Value("host",new Var("Host")),new
                    Value("user",new Var("User")),new Value("pwd",new Var("Password")),new
                    Var("Options") ) ));

            ...

            if (dBManager.disconnectDB(nameDB)){
                //il database è stato disconnesso e rispondo ok
                cn.out(tidOut,new LogicTuple("service_result",new Value(ID),new Value("Ok") ));
                System.out.println("Sconnesso dal sistema il database: "+nameDB);
            }else{
                //il database non è stato disconnesso rispondo errore
                cn.out(tidOut,new LogicTuple("service_result",new Value(ID),new Value("Errore di
                        disconnessione") ));
                System.out.println("Errore nella disconnessione dal del sistema database:
                        "+nameDB);
            }

        }

    }
  }catch(Exception e){System.out.println("Errore di tipo "+e);}
 }

}
```

SelectGenerator

```
package agentDB.service;
import agentDB.database.*;
import java.sql.*;
import javax.sql.*;
import java.util.*;
public class SelectGenerator{

    public SelectGenerator(DataBaseManager databasemanager){
        dbm=databasemanager;
    }

    public ResultSet [] getInformation(String [] informationFind ,String [] boundFind){
        try{
            String colSelect="";
            String boundSelect="";
            String fromSelect="";
            Tabella tableSelect [];
            Colonna tmpCol;
            Vector listSelect;
            Vector listTableFound;
            Vector listBound;
            boolean thereIsTable;
            ResultSet rsFinal[];

            DataBase listDb[];
            listDb=dbm.getDataBase(informationFind);
            if (listDb==null)return null;//non è stato trovato nessun database che contiene le informazioni
                richieste
            rsFinal=new ResultSet[listDb.length];
            for (int numDat=0;numDat<listDb.length;numDat++){
                //Per ogni database che risponde alle specifiche richieste faccio i seguenti passaggi
                colSelect="";
                fromSelect="";
                boundSelect="";
                listSelect=new Vector(informationFind.length,3);
                listTableFound=new Vector(5,2);
                listBound=new Vector(5,2);
                for (int numInfo=0;numInfo<informationFind.length;numInfo++){
                    //per ogni informazione rihiesta faccio le seguenti operazioni
                    //prelevo le tabelle che contengono l'informazione
                    tableSelect=listDb[numDat].getTableOfColumn(informationFind[numInfo]);
                    for (int numTab=0;numTab<tableSelect.length;numTab++){
                        //per ogni tabella che contiene l'informazione
                        //prelevo la colonna che contiene l'informazione
                        tmpCol=tableSelect[numTab].getColName(informationFind[numInfo]);
                        //inserisco la colonna nella lista di colonne che poi andranno a fare la select
                        listSelect.add(tableSelect[numTab].getTableName()+"."+tmpCol.getColName());
                        //controllo che la tabella che sto inserendo sia gia inserita nella lista per i from
                        thereIsTable=false;
                        for (int nTabFound=0;nTabFound<listTableFound.size();nTabFound++){
                            if (  ((String)listTableFound.get(nTabFound)
                                ).equals(tableSelect[numTab].getTableName())) thereIsTable=true;
                        }
                        if (!thereIsTable) listTableFound.add(tableSelect[numTab].getTableName());

                    }
                }
```

```
//Se ho due tabelle controllo che ci sia almeno un vincolo
if(listTableFound.size()>=2){
    //devo cercare il collegamento fra le tabelle
    for (int i=0;i<listTableFound.size();i++){
        //per ogni tabella prelevo le colonne che hanno delle referenze
        Colonna []
            colReferenced=listDb[numDat].getTable((String)listTableFound.get(i)).getRefere
            nced();
        //per ogni colonna che ha delle referenze guardo se referenzia una tabella che sta
        nella lista delle tabelle del from
        for (int j=0;j<colReferenced.length;j++){
            for (int y=0;y<listTableFound.size();y++){
                if ( (colReferenced[j].getTableReferences()).equals(
                    (String)listTableFound.get(y)) ){
                    //La tabella che sto analizzando ha una colonna che referenzia una
                    colonna di una tabella che sta nel from
                    //quindi inserisco il vincolo
                    listBound.add(
                        colReferenced[j].getTableReferences()+"."+colReferenced[j].getColRe
                        ferences()+"="+(String)listTableFound.get(i)+"."+colReferenced[j].get
                        ColName() );
                }
            }
        }
    }
}
//Qui devo costruire la stringa che poi andra nella parte di select
for (int i=0;i<listSelect.size();i++){
    if(i==0){
        colSelect=(String)listSelect.get(0);
    }else{
        colSelect=colSelect+","+(String)listSelect.get(i);
    }
}
//Qui devo costruire la stringa che poi andra nella parte di from
for (int i=0;i<listTableFound.size();i++){
    if(i==0){
        fromSelect=(String)listTableFound.get(0);
    }else{
        fromSelect=fromSelect+","+(String)listTableFound.get(i);
    }
}

//Qui devo costruire la stringa che poi andra nella parte di where(è la parte di collegamento
fra le tabelle)
for (int i=0;i<listBound.size();i++){
    if(i==0){
        boundSelect=(String)listBound.get(0);
    }else{
        boundSelect=boundSelect+" and "+(String)listBound.get(i);
    }
}
//Alla parte di collegamento delle tabelle aggiungo le vere clausule nella stringa che andra
nella parte di where
for (int i=0;i<boundFind.length;i++){
    if(!boundFind[i].equals("")){//nel caso che non ci siano vincoli da parte dell'utente
        if (i==0){
```

```
                        //al primo giro ci devo aggiungere un and(solo se c'è gia una clausula di
                        congiunzione) e le parentesi
                        if (!boundSelect.equals("")){
                            boundSelect=boundSelect +" and ("+ boundFind[i];
                        }else{
                            boundSelect="("+ boundFind[i];
                        }
                    }else{
                        //ai giri normali aggiungo solo i bound
                        boundSelect=boundSelect + boundFind[i];
                    }
                    if(i==boundFind.length-1){
                        //all'ultimo giro chiudo le parentesi
                        boundSelect=boundSelect +")";
                    }
                }
            }
            if (boundSelect.equals("")){
                //non ci sono clausule di where
                System.out.println("select "+ colSelect +" from "+fromSelect);
                rsFinal[numDat]=listDb[numDat].select("select "+ colSelect +" from "+fromSelect);
            }else{
                //Ci sono delle clausule di where
                System.out.println("select "+ colSelect +" from "+fromSelect+" where "+boundSelect);
                rsFinal[numDat]=listDb[numDat].select("select "+ colSelect +" from "+fromSelect+"
                        where "+boundSelect);
            }
        }
        return rsFinal;
        //return listDb[0].select("select Nome from Utente");
    }catch(Exception e){System.out.println("Errore nella ricerca dei database conteneti determinate
colonne "+e);}
    return null;
  }
}
```

DataBaseManager

```
package agentDB.service;
import java.util.*;
import agentDB.database.*;
import java.sql.*;
import java.io.*;
import javax.sql.*;
public class DataBaseManager{
    /**Vettore di Database*/
    private Vector vectorDB=new Vector(5,3);

    public DataBaseManager(){
    }

    public boolean addDataBase(String nameDB,String userDB,String passDB,String driverDB,String
            typeConnectionDB,String typeDB,String fileOptional){
        try{
            //verifico che non ci sia gia un database con lo stesso nome
            for (int numdb=0;numdb<vectorDB.size();numdb++){
                if(((DataBase)vectorDB.get(numdb)).getDbName().equals(nameDB)){
                    System.out.println("Errore nell'inserimento di un database: Nome di database già
                        usato");
                    return false;
                }
            }
            DataBase db =
                DBService.getDataBase(nameDB,userDB,passDB,driverDB,typeConnectionDB,typeDB,
                fileOptional);
            if (db!=null){
                vectorDB.add(db);
                return true;
            }
            return false;
        }catch(Exception e){System.out.println("Errore in DataBaseManager" +e);return false;}
    }

    public DataBase getDataBase(String name){
        try{
            if (vectorDB.size()!=0){
                for (int i=0;i<vectorDB.size();i++){
                    if ( ((DataBase)(vectorDB.get(i))).getDbName().equals(name)){
                        return (DataBase)vectorDB.get(i);
                    }
                }
            }
        }catch(Exception e){System.out.println("Errore nella ricerca del getDataBase(name) "+e);}
        return null;
    }

    public DataBase [] getDataBase(String [] colName){
        try{
            Vector vDbWithCol=new Vector(3);
            for (int i=0;i<vectorDB.size();i++){
                if (((DataBase)vectorDB.get(i)).conteinCol(colName)){
                    vDbWithCol.add((DataBase)vectorDB.get(i));
                }
```

```
            }
            if (vDbWithCol.size()!=0){
                //se ho trovato almeno un db che contiene le informazioni richieste rispondo con i db
                        richiesti
                DataBase listDb[]=new DataBase[vDbWithCol.size()];
                vDbWithCol.copyInto(listDb);
                return listDb;
            }else{
                //altrimenti rispondo con un null
                return null;
            }
        }catch(Exception e){System.out.println("Errore nella ricerca di database conteneti determinate
                 colonne in DataBase Manager "+e);}
        return null;
    }

    public boolean conteinDB(){
        if (vectorDB.size()>0)return true;
        return false;
    }

    public boolean disconnectDB(String nameDB){
        if(vectorDB.size()>0){
            for (int i=0;i<vectorDB.size();i++){
                if ( ((DataBase)(vectorDB.get(i))).getDbName().equals(nameDB)){
                    ((DataBase)(vectorDB.get(i))).close();
                    vectorDB.removeElementAt(i);
                    return true;
                }
            }
        }
        return false;
    }
}
```

APPENDICE B: XML

L'XML[XML] è un linguaggio di markup aperto e basato su testo che fornisce informazioni di tipo strutturale e semantico relative ai dati veri e propri. Questi "dati sui dati", o *metadati*, offrono un contesto aggiuntivo all'applicazione che utilizza i dati e consente un nuovo livello di gestione e manipolazione delle informazioni basate su Web.

L'XML, derivazione del noto linguaggio SGML (Standard Generalized Markup Language), è stato ottimizzato per il Web, diventando potente complemento dell'HTML basato su standard. L'importanza dell'XML nel futuro delle informazioni sul Web potrebbe pertanto giungere ad eguagliare quella dell'HTML agli albori.

L'Extensible Markup Language (XML) è un metalinguaggio che permette di creare dei linguaggi personalizzati di markup; nasce dall'esigenza di portare nel World Wide Web lo Standard Generalized Markup Language (SGML), lo standard internazionale per la descrizione della struttura e del contenuto di documenti elettronici di qualsiasi tipo; ne contiene quindi tutta la potenza, ma non tutte le complesse funzioni raramente utilizzate. Si caratterizza per la semplicità con cui è possibile scrivere documenti, condividerli e trasmetterli nel Web.

L'XML può essere utilizzato come piattaforma per lo scambio di dati tra le applicazioni, come mostra la Figura 10; ciò è possibile perché è orientato alla descrizione dei dati.

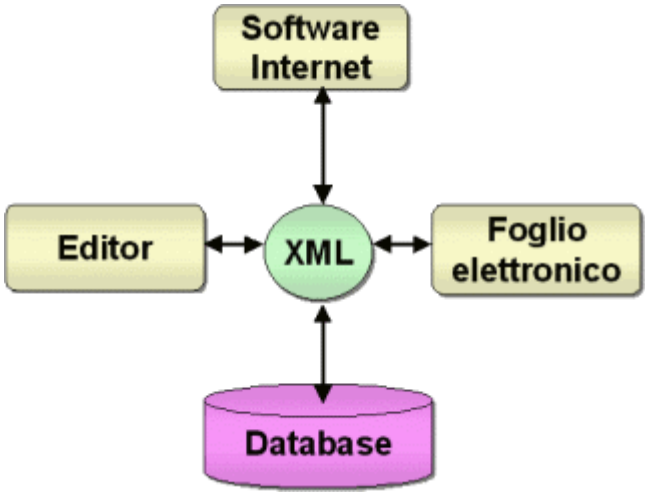

Figura 10 L'XML può essere utilizzato come piattaforma per lo scambio di dati

Un documento XML è costituito da dichiarazioni, elementi, istruzioni di elaborazione e commenti. Alcuni componenti sono opzionali, altri sono necessari.

PROLOGO

Il primo elemento strutturale di un documento XML è un *prologo* opzionale, costituito da due componenti principali anch'essi opzionali: la *dichiarazione XML* e la *dichiarazione del tipo di documento*.

DICHIARAZIONE XML

La dichiarazione XML identifica la versione delle specifiche XML a cui è conforme il documento. Sebbene la dichiarazione XML sia un elemento opzionale, deve sempre essere inserita in documento XML. Il documento inizia con una dichiarazione XML di base:

```
<?xml version="1.0"?>
```

Una dichiarazione XML può inoltre contenere una *dichiarazione di codifica* (encoding) e una *dichiarazione di documento autonomo* (standalone). La dichiarazione di codifica identifica lo schema di codifica dei caratteri, ad esempio UTF-8 o EUC-JP. Schemi di codifica diversi assegnano formati di caratteri o linguaggi diversi. La dichiarazione di

documento autonomo identifica l'esistenza delle dichiarazioni di markup esterne al documento. Questo tipo di dichiarazione può assumere valore *yes* o *no*.

DICHIARAZIONE DEL TIPO DI DOCUMENTO

La dichiarazione del tipo di documento è costituita da codice di markup che indica le regole grammaticali o la definizione del tipo di documento DTD per una particolare classe di documenti. Questa dichiarazione può anche essere diretta a un file esterno che contiene tutta o parte della DTD e deve essere visualizzata dopo la dichiarazione XML e prima dell'elemento Document. Queste stringhe di codice aggiungono una dichiarazione del tipo di documento all'esempio:

```
<?xml version="1.0"?>
<!DOCTYPE Wildflowers SYSTEM "Wldflr.dtd">
```

L'ELEMENTO DOCUMENT

L'elemento Document contiene tutti i dati di un documento XML inclusi tutti i sottoelementi nidificati e le entità esterne. Può essere considerato simile all'unità C: del computer. Tutti i dati del computer sono memorizzati in questa singola unità in cui le cartelle e le sottocartelle contengono le singole parti di dati in una struttura logica e di semplice gestione. Queste stringhe di codice aggiungono un elemento Document, in questo caso l'elemento Plant all'esempio:

```
<?xml version="1.0"?>
<!DOCTYPE Wildflowers SYSTEM "Wldflr.dtd">

<PLANT>
        <COMMON>Columbine</COMMON>
         <BOTANICAL>Aquilegia canadensis</BOTANICAL>
</PLANT>
```

La ***nidificazione*** è il processo che consente di incorporare un oggetto o un costrutto l'uno all'interno dell'altro. Un documento XML può ad esempio contenere elementi nidificati e altri documenti. Ogni elemento secondario, cioè un elemento diverso dall'elemento Document risiede interamente all'interno del relativo elemento principale, così :

```
<DOCUMENT>
    <PARENT1>
        <CHILD1></CHILD1>
        <CHILD2></CHILD2>
    </PARENT1>
</DOCUMENT>
```

IL TAG DI ELEMENTO VUOTO

Il linguaggio XML supporta un collegamento per elementi vuoti, il *tag di elemento vuoto*. Questo tag unisce i tag di apertura e di chiusura per un elemento senza alcun contenuto. Viene utilizzato un formato speciale: <NOMETAG/>. In questo caso la barra segue il nome del tag, il che non è possibile nel linguaggio HTML.

Gli *attributi* consentono di associare valori a un elemento senza che siano considerati parte del contenuto dell'elemento stesso. Ad esempio osserviamo un comune elemento HTML e l'utilizzo di un attributo:

```
<A HREF = "http://www.microsoft.com">Microsoft Home Page</A>
```

In questo caso l'elemento Anchor indicato dal tag <A> contiene un attributo denominato HREF. Il valore dell'attributo è http://www.microsoft.com. Mentre il valore non viene mai visualizzato dall'utente, l'attributo contiene importanti informazioni relative all'elemento e fornisce la destinazione dell'ancoraggio. Questo formato del nome e del valore mostra il modo in cui sono utilizzati gli attributi nel linguaggio XML.

Questo esempio aggiunge un attributo a uno degli elementi del documento:

```
<?xml version="1.0"?>
 <!DOCTYPE Wildflowers SYSTEM "Wldflr.dtd">

 <PLANT ZONE=3>
  <COMMON>Columbine</COMMON>
  <BOTANICAL>Aquilegia canadensis</BOTANICAL>
 </PLANT>
```

L'attributo ZONE del tag di apertura <PLANT> segue il formato del nome e del valore.

Il linguaggio XML possiede due caratteristiche fondamentali, la capacità di fornire una struttura ai documenti e di rendere i dati auto-descrittivi. Queste caratteristiche non sarebbero di alcuna utilità se non si potessero far rispettare le regole strutturali e grammaticali.

APPENDICE C: SQL

Un[SQL] database in un sistema relazionale è composto da un'insieme di tabelle, che corrispondono alle relazioni del modello relazionale. Nella terminologia usata nell'SQL non si fa accenno alle relazioni, così come non viene usato il termine attributo, ma viene usata la parola colonna, e non si parla di tupla, ma di riga. Nel seguito verranno usate indifferentemente le due terminologie, quindi tabella varrà per relazione, colonna per attributo, riga per tupla, e viceversa.

In pratica la creazione del database consiste nella creazione delle tabelle che lo compongono. In realtà prima di poter procedere alla creazione delle tabelle normalmente occorre creare in effetti il database, il che di solito significa definire uno spazio dei nomi separato per ogni insieme di tabelle. In questo modo per un DBMS è possibile gestire più database indipendenti contemporaneamente, senza che ci siano dei conflitti con i nomi che vengono utilizzati in ciascuno di essi. Il sistema previsto dallo standard per creare degli spazi dei nomi separati consiste nell'utilizzo dell'istruzione SQL "CREATE SCHEMA". Di solito tale sistema non viene utilizzato (o almeno non con gli scopi ed il significato previsti dallo standard), ma ogni DBMS prevede una procedura proprietaria per creare un database. Normalmente viene esteso il linguaggio SQL introducendo un'istruzione non prevista nello standard: "CREATE DATABASE".

La sintassi utilizzata dai più diffusi DBMS, è la seguente:

CREATE DATABASE nome_database

Una volta creato il database è possibile creare le tabelle che lo compongono. L'istruzione SQL preposta a questo scopo è:

CREATE TABLE nome_tabella (nome_colonna tipo_colonna [clausola_default] [vincoli_di_colonna][, nome_colonna tipo_colonna [clausola_default] [vincoli_di_colonna] ...][, [vincolo_di tabella] ...])

nome_colonna: è il nome della colonna che compone la tabella. Sarebbe meglio non esagerare con la lunghezza degli identificatori di colonna, dal momento che l'SQL Entry Level prevede nomi non più lunghi di 18 caratteri. Si consulti comunque la documentazione dello specifico database. I nomi devono iniziare con un carattere alfabetico.

tipo_colonna: è l'indicazione del tipo di dato che la colonna potrà contenere. I principali tipi previsti dallo standard SQL sono:

- *CHARACTER(n)*
  Una stringa a lunghezza fissa di esattamente n caratteri.
  CHARACTER può essere abbreviato con CHAR
- *CHARACTER VARYING(n)*
  Una stringa a lunghezza variabile di al massimo n caratteri.
  CHARACTER VARYING può essere abbreviato con VARCHAR o CHAR VARYING.
- *INTEGER*
  Un numero intero con segno. Può essere abbreviato con INT. La precisione, cioè la grandezza del numero intero che può essere memorizzato in una colonna di questo tipo, dipende dall'implementazione del particolare DBMS.
- *SMALLINT*
  Un numero intero con segno con precisione non superiore a INTEGER.
- *FLOAT(p)*
  Un numero a virgola mobile, con precisione p. Il valore massimo di p dipende dall'implementazione del DBMS. È possibile usare FLOAT senza indicazione della precisione, utilizzando quindi la precisione di default, anch'essa dipendente dall'implementazione. REAL e DOUBLE PRECISION sono dei sinonimi per un FLOAT

con una particolare precisione. Anche in questo caso le precisioni dipendono dall'implementazione, con il vincolo che la precisione del primo non sia superiore a quella del secondo.

- *DECIMAL(p,q)*
  Un numero a virgola fissa di almeno p cifre e segno, con q cifre dopo la virgola. DEC è un'abbreviazione per DECIMAL. DECIMAL(p) è un'abbreviazione per DECIMAL(p,0). Il valore massimo di p dipende dall'implementazione.

- *INTERVAL*
  Un periodo di tempo (anni, mesi, giorni, ore, minuti, secondi e frazioni di secondo).

- *DATE, TIME e TIMESTAMP*
  Un preciso istante temporale. DATE permette di indicare l'anno, il mese e il giorno. Con TIME si possono specificare l'ora, i minuti e i secondi. TIMESTAMP è la combinazione dei due precedenti. I secondi sono un numero con la virgola, permettendo cosi' di specificare anche frazioni di secondo.

clausola_default: indica il valore di default che assumerà la colonna se non gliene viene assegnato uno esplicitamente nel momento della creazione della riga. La sintassi da utilizzare è la seguente:

DEFAULT { valore | NULL }

dove, valore è un valore valido per il tipo con cui la colonna è stata definita.

vincoli_di_colonna: sono vincoli di integrità che vengono applicati al singolo attributo. Sono:

- *NOT NULL*, che indica che la colonna non può assumere il valore NULL.
- *PRIMARY KEY*, che indica che la colonna è la chiave primaria della tabella.
- una definizione di riferimento, con cui si indica che la colonna è una chiave esterna verso la tabella e i campi indicati nella definizione. La sintassi è la seguente:

REFERENCES nome_tabella [(colonna1 [, colonna2 ...])]
[ON DELETE { CASCADE | SET DEFAULT | SET NULL }]
[ON UPDATE { CASCADE | SET DEFAULT | SET NULL }]

Le clausole ON DELETE e ON UPDATE indicano quale azione deve essere compiuta nel caso in cui una tupla nella tabella referenziata venga eliminata o aggiornata. Infatti in tali casi nella colonna referenziante (che è quella che si sta definendo) potrebbero esserci dei valori inconsistenti. Le azioni possono essere:

- *CASCADE*: eliminare la tupla contenente la colonna referenziante (nel caso di ON DELETE) o aggiornare anche la colonna referenziante (nel caso di ON UPDATE).
- *SET DEFAULT*: assegnare alla colonna referenziante il suo valore di default.
- *SET NULL*: assegnare alla colonna referenziante il valore NULL.

- un controllo di valore, con il quale si permette o meno l'assegnazione di un valore alla colonna, in base al risultato di un espressione. La sintassi da usare è:

  CHECK (espressione_condizionale)

  dove espressione_condizionale è un'espressione che restituisce vero o falso.
  Ad esempio, se stiamo definendo la colonna COLONNA1, definendo il seguente controllo:

  CHECK (COLONNA1 < 1000)

  in tale colonna potranno essere inseriti solo valori inferiori a 1000.

vincolo_di_tabella: sono vincoli di integrità che possono riferirsi a più colonne della tabella. Sono:

- la definizione della chiave primaria:

  PRIMARY KEY (colonna1 [, colonna2 ...]) Si noti che in questo caso, a differenza della definizione della chiave primaria come vincolo di colonna, essa può essere formata da più di un attributo.

- le definizioni delle chiavi esterne:

  FOREIGN KEY (colonna1 [, colonna2 ...]) definizione_di_riferimento

  La definizione_di_riferimento ha la stessa sintassi e significato di quella che può comparire come vincolo di colonna.

- un controllo di valore, con la stessa sintassi e significato di quello che può essere usato come vincolo di colonna.

Col termine "popolazione del database" si intende l'attività di inserimento dei dati al suo interno. In un database relazionale ciò corrisponde alla creazione delle righe che compongono le tabelle che costituiscono il database. Normalmente la memorizzazione di una singola informazione corrisponde all'inserimento di una o più righe in una o più tabelle del database.
L'istruzione SQL che effettua l'inserimento di una nuova riga in una tabella è INSERT. La sintassi con cui essa viene usata più comunemente è:

INSERT INTO nome_tabella [(elenco_campi)]VALUES (elenco_valori)

nome_tabella è il nome della tabella in cui deve essere inserita la nuova riga.

elenco_campi è l'elenco dei nomi dei campi a cui deve essere assegnato un valore, separati fra loro da una virgola. I campi non compresi nell'elenco assumeranno il loro valore di default o NULL se non hanno un valore di default. È un errore non inserire nell'elenco un campo che non abbia un valore di default e non possa assumere il valore NULL. Nel caso in cui l'elenco non venga specificato dovranno essere specificati i valori di tutti i campi della tabella.

elenco_valori è l'elenco dei valori che verranno assegnati ai campi della tabella, nell'ordine e numero specificati dall'elenco_campi o in quello della definizione della tabella (se elenco_campi non viene specificato). I valori possono essere un'espressione scalare del tipo appropriato per il campo o le keyword DEFAULT o NULL, se il campo prevede un valore di default o ammette il valore NULL.

Vedremo ora invece le istruzioni che occorrono per estrarre da un database i dati che interessano. L'istruzione SQL preposta a tale scopo è SELECT. Dal momento che l'interrogazione è forse la funzionalità più usata di un database, le opzioni che l'istruzione SELECT sono numerose e a volte piuttosto complicate. La sintassi con cui l'istruzione SELECT deve essere utilizzata è la seguente:

SELECT [ALL | DISTINCT] lista_elementi_selezione
FROM lista_riferimenti_tabella
[WHERE espressione_condizionale]
[GROUP BY lista_colonne]
[HAVING espressione_condizionale]
[ORDER BY lista_colonne]

L'istruzione SELECT produce una tabella ottenuta applicando il seguente procedimento (per lo meno dal punto di vista logico, ogni DBMS ottimizza l'esecuzione delle interrogazioni secondo le proprie strategie):

1. produce una tabella ottenuta come prodotto cartesiano delle tabelle specificate nella clausola FROM. Ogni elemento della lista_riferimenti_tabella segue la seguente sintassi:

   riferimento_tabella [[AS] alias_tabella]

   Il riferimento può essere il nome di una tabella o un' espressione (posta fra parentesi tonde) il cui risultato è una tabella, quindi, ad esempio, anche un'altra SELECT. L'alias è un nome che serve per indicare in breve un riferimento di tabella. Nel caso in cui il riferimento di tabella sia un espressione, è obbligatorio specificare un alias.

2. dalla tabella precedente elimina tutte le righe che non soddisfano l'espressione condizionale (cioè le righe per cui l'espressione condizionale restituisce falso come risultato) della clausola WHERE.

3. (se è presente la clausola GROUP BY) le righe della tabella risultante dal passo 2 vengono raggruppate secondo i valori presenti nelle colonne specificate nella clausola GROUP BY. Righe con valori uguali vengono accorpate in un'unica riga. Le colonne non comprese nella clausola devono contenere delle espressioni con funzioni di aggregazione (come ad esempio AVG, che calcola la media) che quindi vengono calcolate producendo un singolo valore per ogni gruppo.

4. (se è presente la clausola HAVING) dal risultato del punto 3 vengono eliminate le righe che non soddisfano l'espressione condizionale della clausola HAVING.

5. Vengono calcolate le colonne presenti nella clausola SELECT (quelle nella lista_elementi_selezione). In particolare vengono calcolate le colonne con le funzioni di aggregazione derivanti dal raggruppamento avvenuto al punto 3. Ogni elemento della lista_elementi_selezione segue la seguente sintassi:

   espressione_scalare [[AS] alias_colonna]

   Un espressione scalare è un'espressione che produce come risultato un valore scalare. I tipi di dato scalari del linguaggio SQL sono principalmente quelli descritti precedentemente, esclusi INTERVAL, DATE, TIME e TIMESTAMP.
   Le espressioni scalari degli elementi della SELECT normalmente coinvolgono le colonne della tabella risultante dal punto 4. Nel caso in cui siano presenti delle ambiguità a causa di colonne con nomi uguali in due o più tabelle comprese nella clausola FOR, esse possono essere risolte prefissando il nome o l'alias della colonna con il nome o l'alias della tabella, separati da un punto. Ad esempio, T.C indica la colonna C della tabella T. L'alias di colonna è il nome che viene assegnato alla colonna.
   L'intero elenco delle colonne di una tabella può essere specificato utilizzando il carattere '*'.

6. (se è presente l'opzione DISTINCT) vengono eliminate le righe che risultano duplicate. Nel caso non sia presente né ALL né DISTINCT, viene assunto ALL.

7. (se è presente la clausola ORDER BY) le righe della tabella vengono ordinate secondo i valori presenti nelle colonne specificate nella clausola. La sintassi da utilizzare è la seguente:

   ORDER BY nome_colonna [ASC | DESC] [, nome_colonna [ASC | DESC]...]

L'ordinamento di default è quello ascendente. Nel caso si desideri effettuare quello decrescente bisogna specificare l'opzione DESC. Se non viene specificata la clausola ORDER BY, la tabella è da considerarsi senza un particolare ordinamento, infatti per la definizione di relazione del modello relazionale, le righe della tabella formano un insieme nel senso matematico e per gli elementi di un insieme non è definita nessuna proprietà di ordinamento. Nella pratica, invece, l'ordine che si ottiene non specificando la clausola di ordinamento è quasi sempre quello che rispecchia la loro memorizzazione fisica e quindi, di solito, quello secondo cui le righe sono state inserite nella tabella.

La sequenza di operazioni appena presentata deve essere considerata valida solo dal punto di vista concettuale. Infatti non è detto che esse vengano eseguite esattamente in questo modo e in questo ordine, dal momento che ogni DBMS ottimizzerà le interrogazioni secondo le strategie più opportune.

Indice delle figure

BIBLIOGRAFIA

TUC. TuCSoN:
www.lia.deis.unibo.it/Research/TuCSoN - Tucson Home Page;

GIUS TESI dell'Ing Giuseppe Tomaiuoli:
"Integrazione di servizi di messaggistica intelligenti basati su una infrastruttura di coordinazione", Ateneo di Bologna, Facoltà di Ingegneria, Corso di Laurea: Ingegneria Informatica; Relatore E. Denti; Anno Accademico 2002-2003;

ROS TESI dell'Ing. Rossella Rubino:
"Sviluppo Prototipale di un Sistema di Posta Elettronica Intelligente Basato su una Infrastruttura di Coordinazione", Ateneo di Bologna, Facoltà di Ingegneria, Corso di Laurea: Ingegneria Informatica; Relatore: E. Denti; Anno Accademico: 2001-2002;

IOSACI "Integrating and Orchestrating Services upon an Agent Coordination Infrastructure"
di Enrico Denti, Alessandro Ricci e Rossella Rubino

SUN www.java.sun.com Java Home Page

MY www.mysql.com MySQL Home Page

XML www.html.it

SQL www.html.it