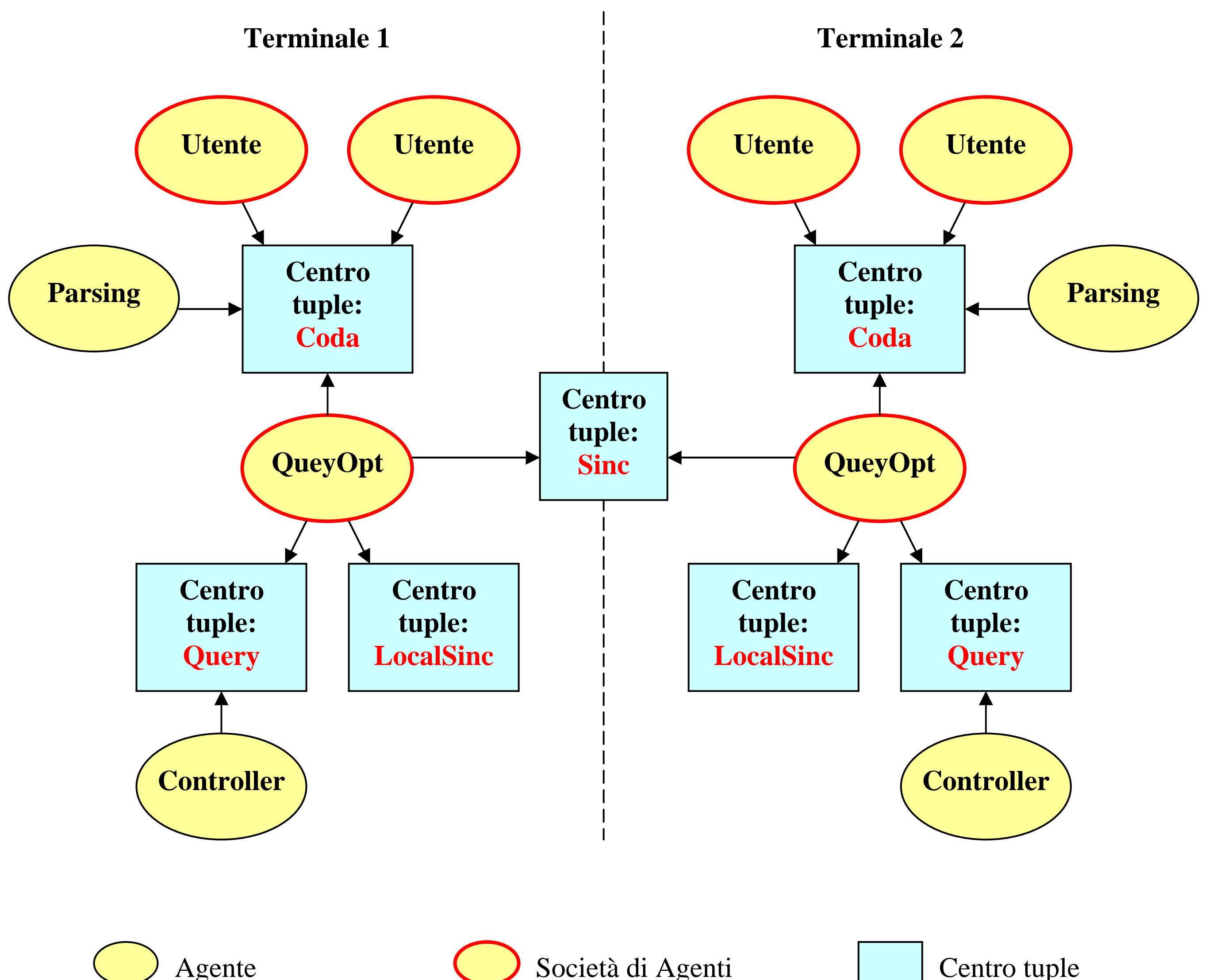
Realizzazione di un database ad oggetti distribuito attraverso l'infrastruttura TuCSon

Lo scopo del progetto è quello di realizzare un database ad oggetti distribuito peer-to-peer utilizzando l'infrastruttura TuCSon per risolvere le query realizzando un ottimizzazione delle stesse in ambiente distribuito.
Queso progetto riguarderà solamente la parte di sistema sovrastante alla base di dati vera e propia nella quale vengono gestite le query in ambiente distribuito (cioe tutta quella parte di struttura che da un lato si interfaccia alle varie applicazioni utente mentre dall'altro si interfaccia al database locale dei singoli terminali). Il resto del sistema verrà simulato attraverso un'apposita classe in quanto non è oggetto di interesse trattandosi di un sistema che opera solamente in locale e quindi non inerente al corso.

Schema di principio

La struttura del sistema è rappresentata nel seguente schema

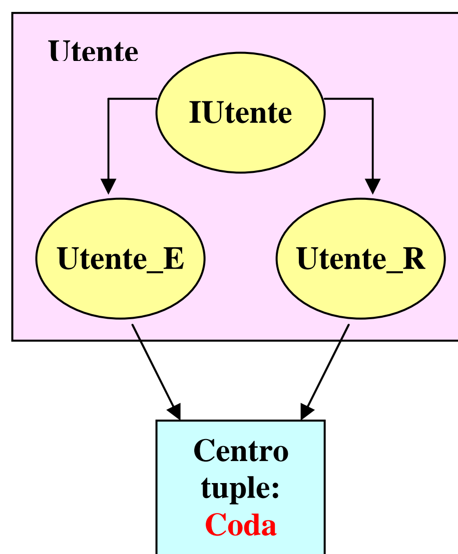


- L'entità **Utente** ha diversi compiti all'interno del sistema. Innanzi fornisce un interfaccia esterna generica che permette di collegare il sistema ad una qualsiasi applicazione utente, inoltre si occupa di immettere la query nel sistema e di prelevarne il risultato una volta pronto.
- Il centro tuple "**Coda**" rappresenta invece una vera e propia coda. Il suo compito è quello di memorizzare le query provenienti dagli utenti ed il loro risultato, qualora fosse disponibile, in attesa che qualcuno li prelevi e li elabori. Inoltre questo centro tuple mantiene traccia attraverso un apposita struttura dati di tutti i terminali che fanno parte del sistema.
- L'agente **Parsing** ha invece il compito di effettuare una elaborazione superficiale delle query in ingresso allo scopo di estrapolare de esse dati ed informazioni che poi saranno utilizzati per la loro ottimizzazione.
- L'entità **QueryOpt** rappresenta il cuore del sistema e racchiude in sé tutta la sottostruttura del sistema che si occupa dell'ottimizzazione delle query in ambiente distribuito e dello smistamento delle stesse e dei vari risultati ad esse associati. In essa sono quindi contenuti i vari algoritmi responsabili dell'ottimizzazione, della politica di risoluzione delle query in ambiente distribuito e della gestione dei risultato provenienti dai vari terminali.
- Il centro tuple **Sinc** svolge un ruolo fondamentale nella sincronizzazione dei vari terminali e mantiene traccia attraverso un apposita struttura dati di tutti terminali registrati nel sistema.
- Il centro tuple **LocalSinc** invece si occupa della sincronizzazione dei componenti appartenenti ad un singolo terminale.
- Il centro tuple **Query** contiene le query che sono in attesa di essere elaborate (risolte) in locale e i relativi risuitati che poi verranno letti e spediti ai vari utenti dal resto del sistema.
- Infine troviamo l'agente **Controller** che rappresenta una semplice interfaccia tra il sistema sovrastante e il database localo vero e proprio. Il suo compito è quindi quello di prelevare le query da eseguire dal centro tuple "Query" e depositare il relativo risultato.

Descrizione dei vari componinti

Utente

Ogni utente è identificato da un intero identificativo dell'utente (univoco relativamente al terminale dove l'utente e registrato) e un secondo intero identificativo del terminale sul quale è registrato.
L'entità Utente è composta da un oggetto che implemeta un Interfaccia *IUtente* e due Agenti.



L'interfaccia *IUtente* prevede due metodi, uno che permette di inserire una nuova query nel sistema e l'altro che permette la registrazione di un qualsiasi oggetto che implementi l'interfaccia *GInterface* il cui compito è quello di "reagire" una volta pervenuti i risultati delle varie query (Quando vi è un risultato disponibile l'entità Utente lo notifica a questo oggetto).
L'oggetto che implementa l'interfaccia *IUtente* si occupa di inserire le nuove query nel sistema immettendo nel centro tuple "Coda" una nuova tupla del tipo

query("serializzazione dell'oggetto rappresentante la query", Uid, Tid, Qid)

dove Uid rappresenta l'identificativo dell'utente, Tid del terminale e Qid rappresenta un identificativo univoco della query (tale identificativo deve essere univoco solamente all'interno del singolo utente).
L' agente *Utente_R* ha il compito di controllare la presenza nel centro tuple di eventuali risposte

result("serializzazione dell'oggetto rapplesentante la risposta", Uid, Tid, Qid)

facendo particolare attenzione a prelevare solo i risultati relativi all'utente che rappresenta (identificato dalla coppia Uid, Qid). Tale agente, una volta informato della presenza di una nuovo risposta indirizzata all'utente che rappresenta, preleva la relativa tupla dal centro tuple e, dopo aver deserializzato l'oggetto "risultato", lo notifica all'interfaccia esterna.
L'agente *Utente_E* ha invece il compito i controllare la presenza nel centro tuple di eventuali messaggi di errore

error(Uid, Tid, Qid, "messaggio di errore")

Centro tuple Coda

Come già accennato, il centro tuple coda contiene al suo interno una struttura dati che serve a tener tracia di tutti i terminali registrati nel sistema. Tale struttura è composta da un insieme di tuple del tipo

asc("indice di registrazione",Tid)

ed il numero di queste tuple sarà uguale al numero di terminali registrati. Il centro tuple "Coda" inoltre contiene anche un'altra tupla che indica il numero di terminali registrati del tipo

n_asc("numero dei terminali registrati")

L'intera struttura dati viene poi aggiornata ogni volta che vi è una registrazione oppure una disconnessione di un terminale. Tali operazioni sono notificate al centro tuple tramite l'inserimento di una tupla

reg("id nouvo terminale")

per la registrazione, oppure

unreg("id nouvo terminale")

per la disconnessione. La struttura dati viene poi modificata tramite le seguenti *reaction*

```
reaction(out(reg(T)),(
                    in_r(reg(T)),
                    in_r(n_asc(N)),
                    N1 is N + 1,
                    out_r(n_asc(N1)),
                    out_r(asc(N,T)))).

reaction(out(unreg(T)),(
                    in_r(unreg(T)),
                    rd_r(asc(N,T)),
                    in_r(n_asc(NT)),
                    N1 is NT - 1,
                    out_r(n_asc(N1)),
                    S is N + 1,
                    out_r(loopUnreg(S,N1)))).

reaction(out_r(loopUnreg(X,X)),(
                    in_r(loopUnreg(X,X)),
                    X1 is X - 1,
                    in_r(asc(X1,_)),
                    in_r(asc(X,T)),
                    out_r(asc(X1,T)))).

reaction(out_r(loopUnreg(X,Y)),(
                    Y > X,
                    in_r(loopUnreg(X,Y)),
                    X1 is X - 1,
                    in_r(asc(X1,_)),
                    in_r(asc(X,T)),
                    out_r(asc(X1,T)),
                    X2 is X + 1,
                    out_r(loopUnreg(X2,Y))))

reaction(out_r(loopUnreg(X,Y)),(
                    Y < X,
                    N is X - 1,
                    in_r(asc(N,T)))).
```

Un altro compito svolto dal centro tuple “Coda” è quello di reagire all’immissione di tuple del genere

query_e(“Serializzazione della query”, Uid, Tid, Qid, F, W)

generando un insieme di tuple del tipo

query_e(“Serializzazione della query”, Uid, Tid, Qid, F, W, Tid)

in numero pari al numero di terminali registrati (ogni tupla sarà caratterizzata da un Tid diverso relativo ad un diverso terminale registrato). Questo meccanismo permette di far leggere la query, una ed una sola volta, a tutti i terminali, ed è realizzato tramite le seguenti reaction

```
reaction(out(query_e(Q,U,T,QID,F,W)),(
                    in_r(query_e(Q,U,T,QID,F,W)),
                    rd_r(n_asc(N)),
                    out_r(loopQuery(Q,U,T,QID,F,W,N)))).
```

```
reaction(out_r(loopQuery(Q,U,T,QID,F,W,0)),(
                                  in_r(loopQuery(Q,U,T,QID,F,W,0)))).

reaction(out_r(loopQuery(Q,U,T,QID,F,W,N)),(
                                  in_r(loopQuery(Q,U,T,QID,F,W,N)),
                                  N1 is N - 1,
                                  rd_r(asc(N1,TID)),
                                  out_r(query_e(Q,U,T,QID,F,W,TID)),
                                  out_r(loopQuery(Q,U,T,QID,F,W,N1)))).
```

Parsing

L'agente Parsing ha il compito di controllare la presenza nel centro tuple di eventuali nuove query pendenti. Ogni volta che viene inserito nel centro tuple una nuova tupla del tipo

query("serializzazione dell'oggetto rappresentante la query", Uid, Tid, Qid)

questo agente deserializza l'oggetto che rappresenta la query, estrapola da essa le condizioni della query stessa e allega queste informazioni alla query attraverso l'inserimento nel centro tuple di una nuova tupla così composta

query_e("serializzazione dell'oggetto rappresentante la query", Uid, Tid, Qid, F, W)

dove F e W sono degli array che contengono le informazioni relative ai campi *From* e *Where* della query in questione.

Centro tuple Sinc

Come accade per il centro tuple "Coda", anche questo centro tuple contiene al suo interno una struttura dati che serve a tener tracia di tutti i terminali registrati nel sistema. Tale struttura è composta da un insieme di tuple del tipo

asc("indice di registrazione", Tid," nome del terminale identificato da Tid")

ed il numero di queste tuple sarà uguale al numero di terminali registrati. Il centro tuple "Sinc" inoltre contiene anche un'altra tupla che indica il numero di terminali registrati del tipo

n_asc("numero dei terminali registrati")

Analogamente a quanto accade nel centro tuple "Coda", l'intera struttura dati viene poi aggiornata ogni volta che vi è una registrazione oppure una disconnessione di un terminale. Tali operazioni sono notificate al centro tuple tramite l'inserimento di una tupla

addTerminal("nome del nuovo terminale", "id nouvo terminale")

per la registrazione, oppure

remTerminal("id nouvo terminale")

per la disconnessione. La struttura dati viene poi modificata tramite le seguenti *reaction*

```
reaction(out_r(loopA(Name,0,T)),(
                    in_r(loopA(Name,0,T)))).

reaction(out_r(loopA(Name,N,T)),(
                    in_r(loopA(Name,N,T)),
                    N1 is N - 1,
                    rd_r(asc(N1,TID,NameP)),
                    out_r(addT(Name,TID)),
                    out_r(addT(NameP,T)),
                    out_r(loopA(Name,N1,T)))).

reaction(out_r(loopR(Name,0)),(
                    in_r(loopR(Name,0)))).

reaction(out_r(loopR(Name,N)),(
                    in_r(loopR(Name,N)),
                    N1 is N - 1,
                    rd_r(asc(N1,TID,NameP)),
                    out_r(remT(Name,TID)),
                    out_r(loopR(Name,N1)))).

reaction(out(addTerminal(Name,T)),(
                    in_r(addTerminal(Name,T)),
                    in_r(n_asc(N)),
                    N1 is N + 1,
                    out_r(n_asc(N1)),
                    out_r(asc(N,T,Name)),
                    out_r(addT(Name,T)),
                    out_r(loopA(Name,N,T)))).

reaction(out(remTerminal(T)),(
                    in_r(remTerminal(T)),
                    rd_r(asc(N,T,_)),
                    in_r(n_asc(NT)),
                    N1 is NT - 1,
                    out_r(n_asc(N1)),
                    S is N + 1,
                    out_r(loopUnreg(S,N1)),
                    out_r(loopR(Name,N)))).

reaction(out_r(loopUnreg(X,X)),(
                    in_r(loopUnreg(X,X)),
                    X1 is X - 1,
                    in_r(asc(X1,_,_)),
                    in_r(asc(X,T,_)),
                    out_r(asc(X1,T,_)))).

reaction(out_r(loopUnreg(X,Y)),(
                    Y > X,
                    in_r(loopUnreg(X,Y)),
                    X1 is X - 1,
                    in_r(asc(X1,_,_)),
                    in_r(asc(X,T,_)),
                    out_r(asc(X1,T,_)),
                    X2 is X + 1,
                    out_r(loopUnreg(X2,Y)))).

reaction(out_r(loopUnreg(X,Y)),(
                    Y < X,
                    N is X - 1,
                    in_r(asc(N,T,_)))).
```

Queste *reaction* fanno anche in modo che ogni volta qualcuno inserisca nel centro tuple una tupla del tipo

addTerminal("nome del nuovo terminale", "id nouvo terminale")

il centro tuple generi un gruppo di tuple

addT(Name,Tid))

in numero pari al numero di terminali in modo tale che il nuovo terminali sia conosciuta da tutti.
La stessa cosa avviene anche per la disconnessione.
Sono principalmente due le funzioni fondamentali svolte da questo centro tuple. Innanzi tutto esso è responsabile della sincronizzazione tra gli ExecuteAgent dei diversi terminali in qunato contiene al suo interno la tupla di sincronizzazione

Ready

la cui presenza (o assenza) indica se il sistema e pronto a eseguire una nuova query.
In secondo luogo, questo centro tuple gioca un ruolo fondamentale nella risoluzione delle query in ambiente distribuito. Quando un terminale, dopo aver prelevato la tupla di sincronizzazione, esegue una query, notifica al centro tuple "Sinc" l'esecuzione della query inserendo una nuovo tupla del tipo

removeQuery(Qid,Tid)

dove Qid rappresenta l'identificativo della query mentre Tid l'identificativo del terminale dove è stata fatta. A questo punto il centro tuple stesso genera un gruppo di nuove tuple del tipo

remQuery(Qid, Tid)

dove questa volta Tid rappresenta l'identificativo del terminale a cui è diretto il messaggio. Le tuple generate saranno, in generale, di numero pari al numero di terminali registrati meno 1 in quanto tutti i terminali devono essere informati che la query in questione è già stata messa in esecuzione a meno del terminale che ha notificato l'esecuzione della query. Queste tuple quindi, saranno caratterizzate tutte da un diverso valore di Tid in modo tale che tutti i terminali possano consumare la tupla a loro diretta. Questo meccanismo viene realizzato attraverso le seguenti *reaction*

```
reaction(out(removeQuery(Q,T)),(
                    in_r(removeQuery(Q,T)),
                    rd_r(n_asc(N)),
                    N1 is N - 1,
                    out_r(loopQ(Q,T,N1)))).

reaction(out_r(loopQ(Q,T,0)),(
                    in_r(loopQ(Q,T,0)),
                    rd_r(asc(0,Tid,_)),
                    T = Tid)).

reaction(out_r(loopQ(Q,T,0)),(
                    in_r(loopQ(Q,T,0)),
                    rd_r(asc(0,Tid,_)),
                    out_r(remQuery(Q,Tid)))).
```

```
reaction(out_r(loopQ(Q,T,N)),(
                        in_r(loopQ(Q,T,N)),
                        rd_r(asc(N,Tid,_)),
                        T = Tid,
                        N1 is N - 1,
                        out_r(loopQ(Q,T,N1)))).

reaction(out_r(loopQ(Q,T,N)),(
                        in_r(loopQ(Q,T,N)),
                        rd_r(asc(N,Tid,_)),
                        out_r(remQuery(Q,Tid)),
                        N1 is N - 1,
                        out_r(loopQ(Q,T,N1)))).
```

In fine, il centro tuple "Sinc", sfruttando la sua base dati, offre la cossibilita di ricavare il nome di un terminale a partire dal relativo identificatore. A tale scopo basterà inserire nel centro tuple una tupla del tipo

convertToString(TidS,TidC)

dove TidS rappresenta Id del terminale di cui vogliamo sapare il nome mentre TidC rappresenta Id del terminale che effettua la richiesta. Per effetto della seguente reaction

```
reaction(out(convertToString(TidS,TidC)),(
                        in_r(convertToString(TidS,TidC)),
                        rd_r(asc(N,TidS,Name)),
                        out_r(convResult(Name,TidC)))).
```

otterremo come risposta la seguente tupla

convResult(Name,TidC)

dove Name contiene il nome relativo all'Id TidS.

Centro tuple "LocalSinc"

Il centro tuple Localsinc può contenere tre tipi di tuple:

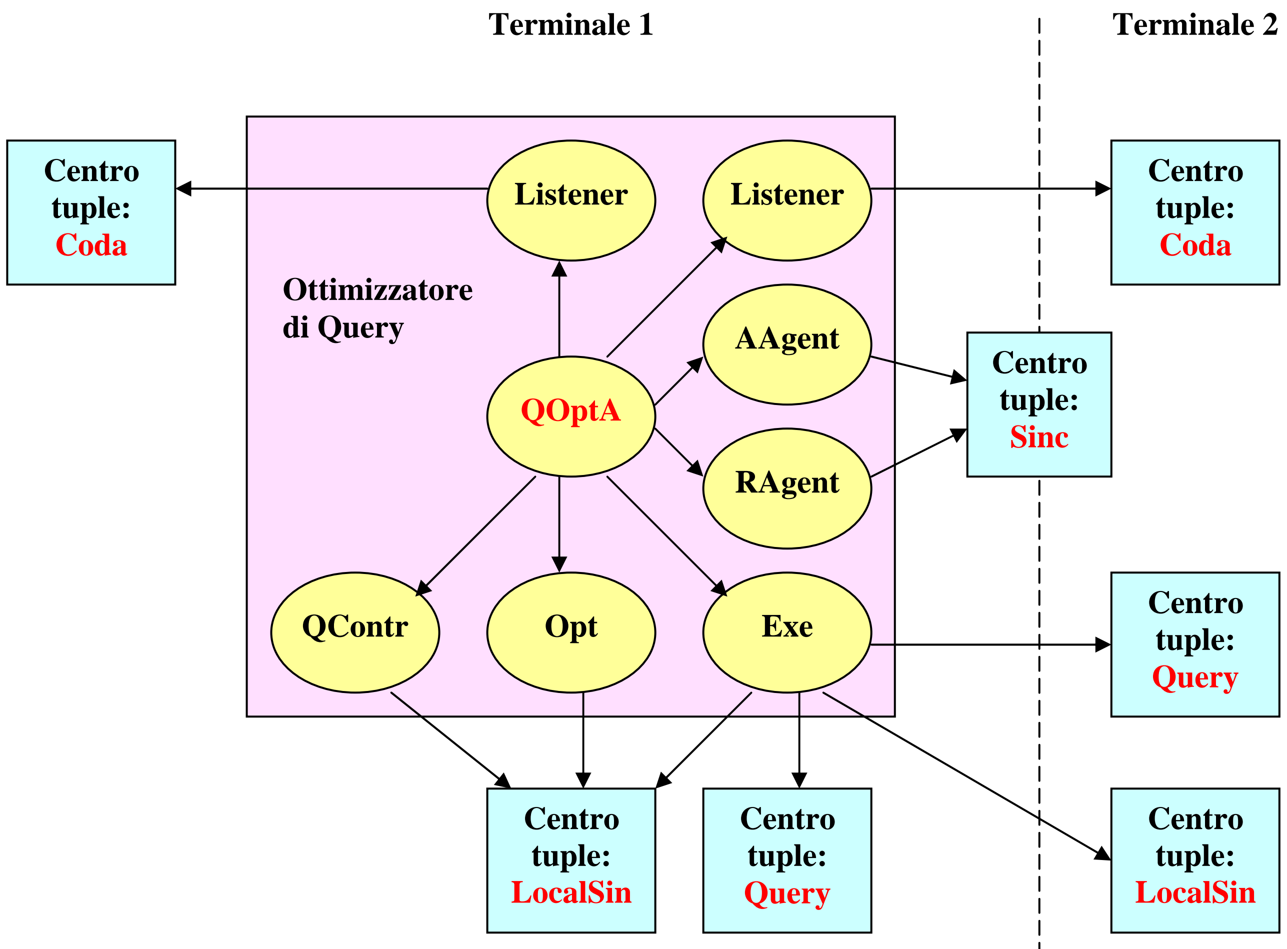
- *NewQuery*
  Questa query indica la presenza di una nuova query nel sistema. L'inserimento di questa tupla provoca un ciclo di elaborazione dell'ottimizzatore
- *Ready*
  Questa tupla indica che l'ottimizzatore ha reso disponibile un set di query pronto per essere eseguito. L'inserimento di questa tupla provoca il "risveglio" dell'agente ExecuteAgent.
- *QuerySinc("0 oppure 1")*
  Questa tupla rappresenta un semaforo, se il valore è 0 allora l'ExecuteAgent può mettere in esecuzione il set di query provenienti dall'ottimizzatore altrimenti si deve fermare in quanto quelle query potrebbero essere già state elaborate da un altro terminale.

QueryOpt

L'entita QueryOpt rappresenta in realtà una socieà di agenti composta dai seguenti elementi:

- un agente **Listener** per ogni terminale attivo nel sistema il cui compito è quello di prelevare le query sulle code locali e notificarle al terminale associato.
- un agente **AccesAgent** che si occupa della registrazione dei terminali.
- un agente **RemoveAgent** che si occupa della disconnessione dei terminali.
- un agente **ExecuteAgent** il cui compito è quello di mettere in esecuzione le query e organizzare la gestione dei relativi risultati gestendo la sincronizzazione tra i vari terminali.
- un agente **OptAgent** che si occupa di elaborare le query cercando elaborare una strategia di esecuzione ottima.
- un agente **ControlQueryAgent** il cui compito è quello di notificare al terminale a cui è associato quando una query è stata eseguita da un altro terminale allo scopo di evitare che la query sia eseguita due volte.
- un oggetto **QoptA** che gestisce l'intero sottosistema.

La struttura del sotto-sistema QueryOpt è rappresentata nel seguente schema



Listener

Ogni terminale possiede un gruppo Listener, uno per ogni coda del sistema, tutti caratterizzati da un Tid pari a quello del terminale stesso. Su ogni coda quindi, si troveranno in ascolto tanti Listener

quanti sono i termiali nel sistema, ciascuno caratterizzato da un diverso Tid relativo al terminale a loro associato.
Un Listener è un agente il cui compito è quello di controolare un determinato centro tuple al fine di reagire ogni volta che in esso vengono depositate tuple del tipo

query_e("serializzazione dell'oggetto rappresentante la query", Uid, Tid, Qid, F, W)

comunicando questa tupla al QOptA del terminale ad esso associato tramite un apposito metodo.

AccesAgent

Ogni volta che un terminale si connette al sistema inserisce un apposita tupla nel centro tuple "Sinc" ed esso reagisce generando un insieme di tuple

addT(Name,Tid))

Il compito dell'agente AccesAgent è quello di prelevare una di queste tuple, e più precisamente, quella avente il Tid uguale al Tid del terminale ad esso associato (ogni tupla ha un Tid diverso dalle altre). Una volta prelevata, l'agente comunica la richiesta di una nuova connessione al QOptA tramite un apposito metodo passandogli il nome del nuovo terminale.

RemoveAgent

Il funzionamento del RemoveAgent è identico a quello dell'AccesAgent. Esso preleva, quando disponibile, la tupla

remT(Name,Tid))

e comunica la disconnessione del terminale al QoptA utillizzando un appositometodo.

OptAgent

L'OptAgent agisce ogni qual volta il QoptA genera una nuova tupla

newQuery

nel centro tuple "LocalSinc" (QoptA genera questa tupla automaticamente quando uno dei suoi Listener gli comunica una nuova query).
Il suo compito è quello di analizzare l'insieme di tuple che rappresentano le query pendenti e elaborare una strategia ottima decidendo quali dovranno essere le prossime query da analizzare. Una volta fatto ciò, l'agente crea la tupla

Ready

in "LocalSinc" e si rimette in attesa.

ExecuteAgent

L'ExecuteAgent è l'agente che si occupa della risoluzione delle query. Esso si attiva ogni volta che l'ottimizzatore genera nel centro tuple di sincronizzazione locale la tupla *Ready* ed il primo passo che esegue è quello di cercare di prelevare la tupla *Ready* nel centro tuple "Sinc". Se questa tupla è presente allora blocca tutti gli altri ExecuteAgent relativi i vari terminali del sistema, esegue le query, comunica agli altri terminali le query eseguite ed infine deposita nuovamente la tupla *Ready* prelevata. Se, invece, la tupla non è presente si mette in attesa e, quando il terminale che ha il controllo restituisce la tupla, decide se continuare o abortire il suo ciclo (in base al valore assunto dalla tupla *QuerySinc(X)*).

QControl

Il compito di questo agente è quello di comunicare al QOptA, attraverso un apposito metodo, tutte le query che vengono eseguite degli altri terminali in modo tale da evitare che alcune query siano ripetute più volte.
Esso si attiva solamente quando sul centro tuple "Sinc" vengono create tuple del tipo

remQuery(Qid,Tid)

(dove Tid deve coincidere con il Tid del terminale ad esso associato Qid rappresenta l'identificatore della query).

QOptA

Il QoptA rappresenta il quore del sottosistema in quanto svolge le funzioni di:
- inserimento di una query
- registrazione di un terminale
- disconnessione di un terminale
- cancellazione di una query già eseguita da un altro terminale

ed inoltre mette a disposizione numerose funzionalità a utili al resto delsistema.

La sincronizzazione

Ipotiziamo di avere due terminali, e di inviare una nuova query da uno di questy.

1. Gli agenti Listener dei due terminali leggeranno entrambi la nuova query e avviseranno l'oggetto QOptA del relativo Terminale che provvederà all'inserimento della tupla "*NewQuery*" sui centri tuple di sincronizzazione locale.
2. Gli ottimizzatori si svegliano ed elaborano ciascuno una strategia di elaborazione (che può essere uguale o diversa) ed inseriscono nel centro tuple di sincronizzazione locale la tupla "*Ready*".
3. Gli esecutori si svegliano e tentano di prelevare la tupla "*Ready*" dal centro tuple di sincronizzazione del sistema "Sinc". Essendo solo una la tupla solo un agente ce la farà mentre l'altro rimarrà bloccato.
4. L'agente che ora ha il controllo, come prima cosa, pone a 1 la tupla "*QuerySinc(X)*" degli altri termnali, poi chiede all'ottimizzatore la sequenza di query da eseguire, la comunica scrivendola nel centro tuple di sincronizzazione del sistema ed, infine, la mette in esecuzione inserendo la query nei centri tuple "Query" dei vari terminali.
5. Dopodichè, attende che tutti rispondano e, quando ciò avviene, formula la risposta, la invia al centro tuple "Coda" da cui la query è stata letta e rigenera la tupla "*Ready*".
6. Nel frattempo, la publicazione delle query eseguite sul centru tuple "Sinc" provoca il risveglio del "QController" relativo al terminale bloccato. Questo comunica le query al terminale che riattiva l'ottimizzatore e pone la tupla "*QuerySinc(X)*" a 0 nel centro tuple di sincronizzazone locale.
7. L'altro agente esecutore si risveglia e consuma nuovamente la tupla di "*Ready*".
8. Come prima cosa controlla tupla "*QuerySinc(X)*", se X=1 allora termina il suo ciclo di esecuzione (in quanto significa che qualcuno a eseguito delle query ma il mio ottimizzatore, per qualche motivo ancora non se né accorto e quindi rischio di rieseguire le stesse query), se X=0 invece continuo in quanto significa che il mio ottimizzatore si è accorto delle query già eseguite ed ha elaborato una seconda strategia (che non contiene le query già elaborate dall'altro terminale).