Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Condivisione di risorse e servizi museali in un sistema Peer-to-Peer

Luca Gatani, Giuseppe Lo Re, Adriano Tamburo, Alfonso Urso

RT-ICAR-PA-04-03 **gennaio 2004**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Condivisione di risorse e servizi museali in un sistema Peer-to-Peer

Luca Gatani[1,2], Giuseppe Lo Re[1], Adriano Tamburo[2], Alfonso Urso[2]

Rapporto Tecnico N. 3: **RT-ICAR-PA-04-03**

Data: **gennaio 2004**

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sezione di Palermo, Viale delle Scienze edificio 11, 90128 Palermo.
[2] Università degli Studi di Palermo, Dipartimento di Ingegneria Informatica, Viale delle Scienze, Edificio 6, 90128 Palermo.

SOMMARIO

Obiettivo del presente lavoro è lo sviluppo di un'architettura *peer-to-peer* per la condivisione di risorse riguardanti un museo di opere d'arte. È stata realizzata un'applicazione, chiamata AlphaTheta, che permette ad un qualsiasi dispositivo multimediale di connettersi ad una rete di *peer* per condividere servizi e risorse.

Per incrementare l'efficienza si è scelto di adottare tecniche avanzate di caratterizzazione delle risorse, facenti uso di schemi e metadati, come proposto nell'ambito del progetto Edutella. I servizi offerti da Edutella consentono la realizzazione di una rete *peer-to-peer* in cui è possibile rappresentare le risorse condivise attraverso metadati RDF (*Resource Description Framework*) e scambiare *query* complesse usando il linguaggio RDF-QEL-i (*RDF-Query Exchange Language*).

L'applicazione è sviluppata sulla piattaforma JXTA, un *middleware* che permette di realizzare sistemi *peer-to-peer* interoperabili, indipendenti dalle particolari architetture di sistema e di rete, e in grado di funzionare su una gamma assai ampia di dispositivi.

AlphaTheta unisce in un'unica applicazione le funzionalità di *consumer* e di *provider*, consentendo ad un utente, attraverso una semplice interfaccia grafica, sia di condividere, che di ricercare risorse relative ad uno specifico dominio applicativo. AlphaTheta utilizza metadati specifici per caratterizzare le risorse da condividere.

Il sistema è stato istanziato su un particolare dominio museale ed ampliamente testato su varie configurazioni di rete.

1. INTRODUZIONE

Il presente lavoro è partito dall'obiettivo di sviluppare un'architettura *peer-to-peer* che permettesse la condivisione di risorse riguardanti un museo d'arte. Considerato che l'applicazione si colloca in un ambito applicativo ben delineato, si è pensato di utilizzare delle tecniche di caratterizzazione delle risorse attraverso l'uso di metadati, utilizzando un approccio simile a quello che viene usato dagli standard dei metadati del *web* semantico. Comunque l'approccio nel *web* è di tipo *client/server*, visto che i browser (*client*) richiedono la ricerca di informazioni sui motori di ricerca (*server*). Per il progetto AlphaTheta, invece, si è scelto un approccio di tipo *peer-to-peer* [1], in cui i metadati all'interno della rete di *peer* sono utilizzati per la ricerca e la localizzazione delle risorse. L'uso di metadati e di schemi è finalizzata alla caratterizzazione delle risorse all'interno della rete, quindi una risorsa viene rappresentata all'interno della rete dai valori che assumono i metadati definiti, permettendo la condivisione di qualsiasi tipo di risorsa. Per affrontare in modo efficiente la possibilità di utilizzazione dei metadati, nel progetto AlphaTheta si è scelto di utilizzare la soluzione proposta nell'ambito del progetto Edutella [2], [3], [4]. Infatti il progetto Edutella, attraverso tecniche avanzate di

caratterizzazione delle risorse mediante l'uso di schemi e metadati RDF (*Resource Description Framework*) [5], [6], mette a disposizione una serie di servizi che permettono di:

- effettuare *query* molto complesse usando un linguaggio basato su RDF (RDF-QEL-i: RDF-*Query Exchange Language* [7]), questo servizio viene anche detto *queryService*;
- condividere qualsiasi tipo di risorsa all'interno della rete Edutella (*providerService*).

Il progetto Edutella, sviluppato utilizzando la piattaforma JXTA, essenzialmente permette ad un qualsiasi *peer* di potersi connettere alla rete Edutella (*EdutellaPeerGroup*) e di poter utilizzare i servizi messi a disposizione.

L'applicazione è stata sviluppata utilizzando la piattaforma JXTA [8], [9], [10], [11], [12]. Il progetto JXTA permette la creazione di sistemi *peer-to-peer* che risultano interoperabili, indipendenti dalla piattaforma e che, in generale, permettono l'ubiquità. JXTA definisce una *suite* di protocolli che definiscono le funzionalità che un qualsiasi sistema *peer-to-peer* deve implementare: localizzazione dei *peer*, scoperta delle risorse, instanziazione di un canale di comunicazione tra due *peer*, etc.. AlphaTheta mette a disposizione una semplice interfaccia utente, che permette ad un qualsiasi utente di effettuare una connessione alla rete Edutella, di condividere risorse e di effettuare ricerche. L'applicazione fornisce sia le funzionalità di *provider*, permettendo di condividere risorse all'interno della rete, sia le funzionalità di *consumer*, permettendo ad un utente di ricercare delle risorse all'interno della rete.

2. JXTA

Con lo sviluppo delle applicazioni di rete di tipo *peer-to-peer* si è avuta la necessità di sviluppare delle piattaforme che ne permettessero una implementazione veloce e robusta. Il progetto JXTA è nato essenzialmente per rendere interoperabili i numerosi protocolli sviluppati per supportare applicazioni *peer-to-peer*. Il progetto è *open source*, e si fonda sull'opera di una vasta comunità di sviluppatori [8], (attualmente l'ultima versione rilasciata è indicata come JXTA 2.0 [11]).

JXTA è costituito essenzialmente da una suite di protocolli che consente a qualsiasi dispositivo connesso alla rete (PC, palmare,..) di comunicare con gli altri in modo paritario [9]. La finalità principale del progetto è stata quella di superare le limitazioni presenti nella maggior parte dei sistemi *peer-to-peer* esistenti, permettendo di creare sistemi *peer-to-peer* interoperabili, indipendenti dalle particolari architetture di sistema e di rete, e in grado di funzionare su una gamma assai ampia di dispositivi.

L'architettura software del progetto JXTA è divisa essenzialmente in tre livelli [9] (si veda figura 1):

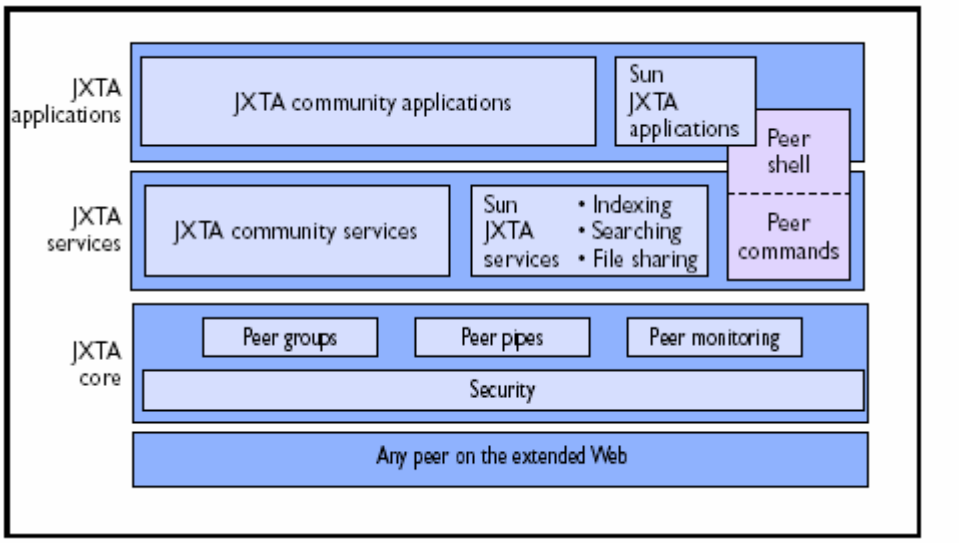

Figura 1: architettura software di JXTA. La tecnologia JXTA fornisce un approccio a livelli per lo sviluppo di nuovi servizi ed applicazioni di tipo *peer-to-peer*.

Questo approccio a livelli permette di utilizzare i servizi centrali definiti dai protocolli (*JXTA core* e *JXTA services*) per l'implementazione di applicazioni *peer-to-peer*. Nella progettazione dell'applicazione *AlphaTheta* sono stati utilizzati i servizi standard definiti dal progetto JXTA, comunque grazie alla struttura modulare della piattaforma è sempre possibile cambiare qualche servizio (per esempio, il servizio di *discovery*) senza la necessità di cambiare l'implementazione dell'applicazione.
I protocolli definiti dal progetto JXTA sono (per maggiori dettagli si veda [12]):

- *Peer Resolver Protocol (PRP)*: fornisce una infrastruttura essenziale di generiche richieste/risposte scambiate dai *peer* all'interno della rete JXTA.
- *Endpoint Routing Protocol (ERP)*: è usato per gestire e determinare le informazioni di *routing* utili ad un *peer* che vuole spedire un messaggio verso un altro *peer*.
- *Peer Discovery Protocol (PDP)*: è usato per scoprire e pubblicare le risorse che i *peer* vogliono condividere sulla rete.
- *Rendezvous Protocol (RVP)*: è usato per propagare i messaggi all'interno di un gruppo.
- *Pipe Binding Protocol (PBP)*: usato per stabilire una *pipe* virtuale tra due *peer* che vogliono comunicare.
- *Peer Information Protocol (PIP)*: usato per ottenere informazioni sullo stato di un peer.

Attraverso il loro uso, JXTA costruisce una rete virtuale *overlay* al di sopra della rete reale [11]. La costruzione della rete virtuale risulta completamente indipendente dalla topologia della rete reale, ciò permette che peer che non hanno connettività nella rete reale possono averla nella rete virtuale, attraverso l'introduzione di *peer* intermedi (*peer* di *relay*). I protocolli definiscono lo scambio di messaggi tra *peer*, fissando il formato dei messaggi che i *peer* si scambiano e gli indirizzi virtuali dei *peer* coinvolti. Lo scambio di messaggi tra due *peer* all'interno della rete JXTA avviene attraverso delle *pipe* (canali di comunicazione virtuale, unidirezionali e asincroni), per esempio nella rete Edutella lo scambio di *query* e risposte tra due *peer* avviene

attraverso l'istanziazione di una pipe. Il *mapping* tra rete virtuale e rete reale è effettuato direttamente dal livello di trasporto dei messaggi, il quale si occupa di stabilire una connessione tra i *peer* (può essere sia TCP che HTTP) e di trasferire i messaggi.
Tutte le risorse in JXTA vengono rappresentate come *advertisement* (strutture di metadati rappresentate come documenti XML): i protocolli permettono lo scambio di *advertisement* tra i *peer*, per scoprire o pubblicare risorse. Anche i servizi forniti all'interno di un gruppo JXTA vengono rappresentati come *advertisement*, per i servizi sono stati definiti tre tipi di *advertisement*: *Module Class* (identifica l'esistenza di un servizio), *Module Specification* (identifica la specificazione di un servizio), *Module Implementation* (ne indica una implementazione). Tale approccio è stato adottato per permettere una maggiore interoperabilità tra i *peer*, garantendo il fatto che diverse implementazioni di uno stesso servizio (possono differenziarsi sia per il sistema operativo utilizzato dal *peer*, sia per il linguaggio con cui è stato implementato) devono essere compatibili.
La rete JXTA è formata da una serie di gruppi (insieme dinamico di *peer* che condividono un insieme di politiche e di risorse). In ciascun gruppo possono essere identificati tre tipi di *peer*.

- *Rendezvous super-peer*: *peer* particolari, che hanno principalmente il compito di conservare gli *advertisement* pubblicati dagli altri *peer* del gruppo, e di propagare le richieste effettuate dai *peer*. Conservano una lista dei *peer* connessi a loro, ed una lista di altri *rendezvous* conosciuti (RPV, *Rendezvous Peer View* [10]).
- *Edge peer*: *peer* normali che sono connessi ai *rendezvous* e che principalmente inviano e rispondono alle richieste propagate nel gruppo.
- *Relay super-peer*: *peer* particolari che mantengono le informazioni di *routing*, permettendo di collegare peer che non hanno connettività fisica diretta. Un *peer* quando deve inviare un messaggio prima guarda nella propria cache locale se ha le informazioni necessarie, in caso negativo invia una richiesta ai *peer* di *relay* conosciuti per ottenere le necessarie informazioni di *routing*.

Quando un *peer* deve unirsi ad un gruppo JXTA deve stabilire almeno una connessione con uno dei *rendezvous* del gruppo. Questo è molto importante, infatti un *peer* che ha in esecuzione l'applicazione AlphaTheta prime di poter interagire nella rete Edutella deve stabilire una connessione virtuale con almeno un *rendezvous* della rete, ciò è fatto all'avvio dell'applicazione: il *peer* cercherà uno tra i *rendezvous* settati durante la fase di configurazione. Tutte le applicazioni che utilizzano la piattaforma JXTA, infatti, prevedono una fase iniziale di configurazione della piattaforma JXTA per l'ambiente di rete [14], per esempio durante questa fase vengono settati gli indirizzi dei *rendezvous* presenti nella rete che sono utilizzati dal *peer* per stabilire la connessione iniziale.
Un *peer* può appartenere simultaneamente a più *peergroup*. *Peer* che appartengono a gruppi diversi non possono comunicare in modo diretto, ma solo usando dei particolari *peer* intermediari (i *peer* di *rendezvous*). I gruppi presentano relazione gerarchica tra di loro. All'inizio ciascun *peer* si unisce a un gruppo radice attivato di *default*, il *NetPeerGroup*. Successivamente un *peer* può creare un nuovo gruppo a cui unirsi. Tale gruppo ha una relazione

padre/figlio con il *NetPeerGroup*, e tutti i servizi pubblicati da lui sono anche pubblicati nel gruppo genitore (ciascun gruppo ha un solo genitore).
Nella progettazione dell'applicazione AlphaTheta è stata utilizzata l'implementazione di JXTA 2.0 con J2SE 1.4.1; questa implementazione abbraccia tutti e sei i protocolli e pochi altri servizi, essa riprende l'architettura modulare mostrata in figura 1 (in [13] sono mostrati una serie di esempi sull'utilizzo di questa implementazione).

3. EDUTELLA

Per permettere un utilizzo efficiente dei metadati si è deciso di utilizzare i servizi forniti da Edutella [2], [3], un *framework* che permette la costruzione di una rete P2P basata su metadati RDF (*Resource Description Language*), sviluppata sopra la piattaforma JXTA.
L'implementazione di Edutella utilizzata per la progettazione dell'applicazione AlphaTheta è quella fornita in [2] (tutte le classi ed i metodi utilizzati si trovano nell'archivio *edutella.jar*); sono state apportate delle modifiche ad alcune classi di questa libreria nell'implementazione dell'applicazione AlphaTheta (nel paragrafo 4.6 vengono mostrate le modifiche che sono state apportate).
I servizi che sono stati utilizzati nel progetto AlphaTheta sono: *queryService* e *providerService*. Attraverso questi servizi è possibile:

- effettuare delle *query* sulla rete Edutella (*queryService*);
- creare dei depositi RDF e rispondere alle *query* ricevute interrogando queste basi di conoscenze (*providerService*).

Per permettere la connessione di *peer* eterogenei, nel progetto Edutella è stato definito l'*Edutella Common Data Model*, il quale definisce la sintassi e la semantica utilizzata per la rappresentazione dei metadati nella rete Edutella. Il progetto Edutella si pone principalmente due obiettivi:

- poter effettuare *query* molto complesse usando il linguaggio delle *query* RDF-QEL-i, capace di interrogare depositi RDF;
- rappresentare le informazioni da condividere nella rete attraverso l'utilizzo di metadati RDF e l'adozione di schemi RDFS arbitrari.

Il linguaggio RDF-QEL è un linguaggio basato sul Datalog, utilizza principalmente la semantica definita dal Datalog, introducendo nuovi costrutti (per esempio, il predicato *Outer Join*, o la definizione dei predicati predefiniti) e modificandone altri (per esempio, la rappresentazione dei risultati). Per permettere l'interazione fra *peer* eterogenei, Edutella segue un approccio a livelli nella definizione del linguaggio RDF-QEL: in questo modo ciascun *peer* nella realizzazione della rete Edutella implementa un determinato livello, il quale definisce le capacità del *peer* di manipolare e rispondere alle *query* ricevute. I livelli correntemente definiti sono cinque: RDF-QEL-1, -2, -3, -4, -5; livelli più alti definiscono la possibilità di eseguire *query* man mano più complesse, presentando livelli di espressività maggiori. Nell'implementazione corrente di

Edutella sono stati definiti tutti gli strumenti necessari per generare tutti i costrutti definiti dal linguaggio RDF-QEL. Tutti le classi che permettono la creazione delle *query* e la manipolazione dei risultati sono state definite nel progetto Edutella all'interno del *package* `net.jxta.edutella.eqm`.

La rete Edutella è formata da un unico gruppo JXTA (*EdutellaPeerGroup*), in cui possono essere individuati tre tipi di *peer* (figura 2):

- *Consumer*: implementano il servizio *queryService* e sono capaci di interrogare i *provider* della rete per l'acquisizione di informazioni.
- *Provider*: implementano il servizio *serviceProvider* (mettono a disposizione le risorse annotate con metadati RDF).
- *Rendezvous*: forniscono un servizio di *rendezvous*, permettono l'interazione tra *consumer* e *provider*).

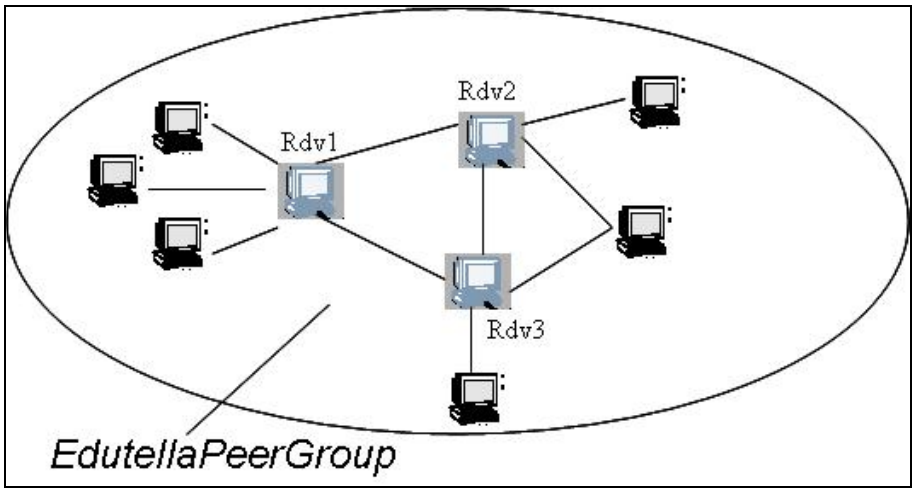

Figura 2: esempio di una rete Edutella.

Un *peer* che inizialmente vuole unirsi al gruppo *EdutellaPeerGroup* deve conoscere almeno uno dei *rendezvous* del gruppo: una volta 'connesso' ad almeno uno di essi, può cominciare ad utilizzare i servizi forniti da Edutella. Ciò può essere fatto semplicemente attraverso la seguente riga di codice:

```
EdutellaPeerGroup pg =
        EdutellaFactory.getEdutellaPeerGroup();
```

Infatti il metodo statico `getEdutellaPeerGroup()` di `EdutellaFactory` (*package* `net.jxta.edutella.peer`) ritorna il gruppo Edutella se questo già esiste oppure lo crea se ancora non esiste, ed inoltre cerca di connettere il *peer* ad almeno uno dei *rendezvous* appartenenti al gruppo.

Un *provider*, all'interno della rete Edutella, pubblica il servizio da esso fornito attraverso la pubblicazione degli *advertisement* necessari (*ModuleClass*, *ModuleSpec*, *ModuleImpl*). Questi *advertisement* vengono pubblicati sui *rendezvous* a cui il *peer* è connesso. Il *provider* definisce

anche la *pipe* (pipeProvider) da cui è possibile accedere al servizio: tutti i *consumer* che vogliono inviare qualche richiesta verso il *provider* devono utilizzare la *pipe* providerPipe pubblicata dal *provider*, infatti il *provider* processa tutte le richieste ricevute attraverso questa *pipe*.
Un *provider* può utilizzare un qualsiasi modello locale di rappresentazione dei dati (per esempio, database relazionali, basi di conoscenza RDF, etc.), ma deve utilizzare dei *wrapper* che permettano di traslare il modello dei dati locali nell'*Edutella Common Data Model*. Infatti un *peer* riceve sempre *query* basate sulla sintassi del linguaggio RDF-QEL, quindi deve traslare la *query* ricevuta nel linguaggio di *query* locale utilizzato (per esempio SQL), interrogare la propria base di conoscenza e traslare i risultati ottenuti sempre in accordo alla rappresentazione dei dati definita dall'ECDM (figura 3). Uno degli obiettivi principali di questo meccanismo è di astrarre dai vari linguaggi delle *query* che possono essere usati a livello di memorizzazione e dai vari modi in cui è possibile memorizzare i dati. Inoltre ciascun *provider* può decidere quale livello del linguaggio RDF-QEL implementare, in questo modo sarà in grado di elaborare solo le *query* che appartengono al livello implementato.

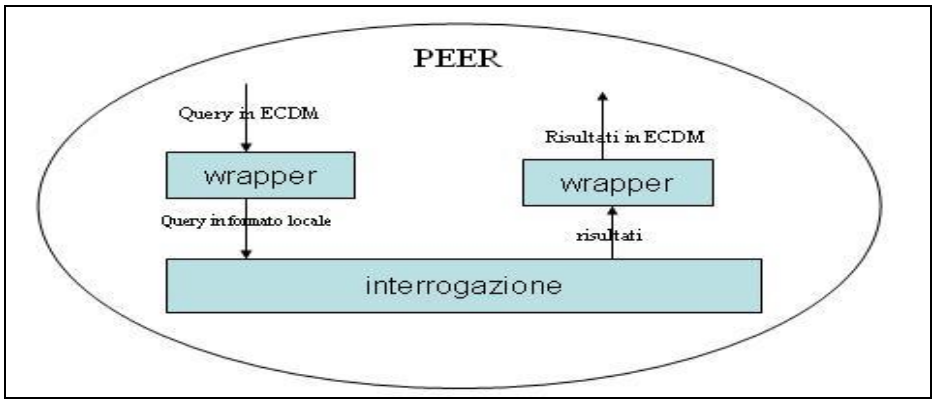

Figura 3: esecuzione di una *query* su un *provider*

La classe fondamentale che gestisce il servizio di *provider* è la classe `EdutellaProvider` (*package* `net.jxta.edutella.provider`). Per far partire il servizio di provider bisogna istanziare un oggetto di questo tipo. Questa classe definisce il seguente costruttore :

```
EdutellaProvider(ProviderPlugin strategy)
```

che accetta un parametro di tipo `ProviderPlugin`, che è una interfaccia JAVA che definisce il metodo:

```
public String processMessage(String type,String message);
```

che si occupa dell'esecuzione della *query* sul *wrapper* definito dal *provider*. La classe che implementa il *wrapper* deve implementare l'interfaccia `ProviderConnection` ed il metodo `executeQuery()`, che si deve occupare dell'esecuzione della *query* come mostrato in figura 3.

Un *consumer*, una volta scoperti i *provider* presenti nella rete (la scoperta avviene attraverso la ricerca degli *advertisement* identificanti il servizio, la classe che gestisce la scoperta dei *provider*

è la classe `ConsumerService`, *package* `net.jxta.edutella.consumer`), può generare la *query* in formato RDF-QEL, ed inviarla verso uno o più *provider* presenti nella rete (l'invio di una *query* verso un *provider* avviene per mezzo della *pipe* pubblicata dal *provider* stesso). In figura 4, è mostrato il funzionamento di un *consumer*: prima di tutto si ha la generazione della *query*, dopo, attraverso il *queryService*, la *query* viene inviata verso uno o più *provider*. Le eventuali risposte vengono gestite dal *ResultListener*.

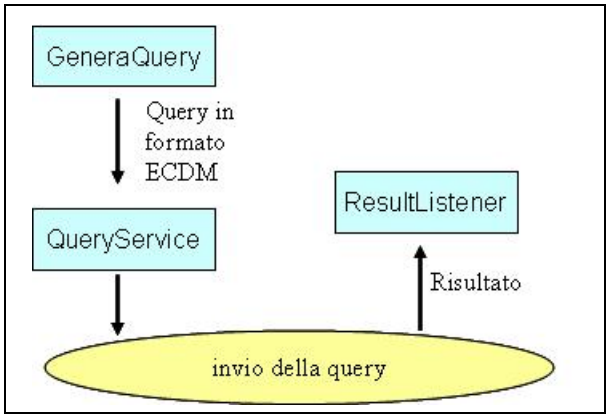

Figura 4: funzionamento di un *consumer*

Nell'implementazione corrente di Edutella le classi che permettono la definizione del *queryService* si trovano nel *package* `net.jxta.edutella.consumer`. La classe principale che implementa le API del servizio è la classe `QueryService`. Mentre l'implementazione del *queryService* è definita dalla classe `QueryServiceImpl`, che implementa il metodo `executeQuery()`, il quale permette di inviare una *query* verso un determinato *provider* o verso tutti i *provider* scoperti dal *peer*. Di seguito è mostrato il codice per estrarre il servizio di query dal gruppo Edutella a cui appartiene il *consumer*:

```
EdutellaPeerGroup
      pg = EdutellaFactory.getEdutellaPeerGroup();
QueryService queryService=pg.getQueryService();
```

Il *consumer* all'interno del messaggio da inviare verso un determinato *provider*, oltre alla *query*, inserisce anche l'*advertisement* della *pipe* in cui il *consumer* si aspetta di ricevere le eventuali risposte (questa *pipe* viene indicata come *pipeConsumer*). Quindi, il *consumer* si mette in ascolto sull'*input pipe* della *pipe pipeConsumer* per aspettare le eventuali risposte inviate dai *provider*.

4. ALPHATHETA

AlphaTheta è un'applicazione che permette la condivisione di risorse riguardanti un museo di opere d'arte in una rete Edutella. Questa applicazione è un buon esempio del funzionamento di Edutella e di come sia possibile velocizzare lo sviluppo di applicazioni *peer-to-peer* utilizzando la piattaforma JXTA e tutti i servizi messi a disposizione da questa. La progettazione di

AlphaTheta è partita dall'analisi delle applicazioni già sviluppate in Edutella [15], cercando di creare un'applicazione *ad-hoc* che ne utilizzasse i servizi messi a disposizione. Rispetto alle applicazioni fornite da Edutella, nel progetto di AlphaTheta si sono volute coniugare in un'unica applicazione le funzionalità di *provider* e di *consumer*, che fino a questo momento in Edutella sono state implementate come due applicazioni differenti. Quindi nel progetto AlphaTheta un *peer* è sia *provider*, visto che è in grado di rispondere alle *query* effettuate da altri *peer*, sia *consumer*, in quanto è in grado di effettuare delle ricerche all'interno della rete Edutella. AlphaTheta mette a disposizione una semplice interfaccia utente che permette ad un qualsiasi utente di effettuare una connessione alla rete Edutella, di condividere risorse e di effettuare ricerche. In figura 5 sono mostrati i diversi moduli che compongono l'applicazione AlphaTheta.

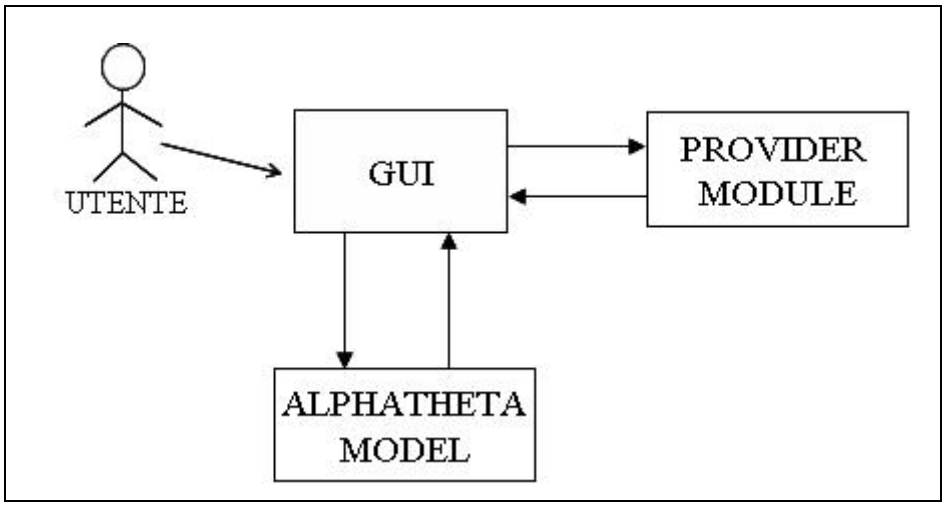

Figura 5: decomposizione modulare dell'applicazione AlphaTheta.

Si possono individuare tre moduli principali che fanno parte dell'applicazione. Il modulo *GUI* permette l'interazione con l'utente. Il *Provider Module* gestisce le funzionalità di *provider* fornite dall'applicazione, permettendo all'utente, attraverso la *GUI*, l'inserimento e l'eliminazione delle risorse da condividere. Il modulo *Alphatheta Model* gestisce le funzionalità di *consumer*, permettendo all'utente di poter effettuare delle *query* nella rete Edutella per la ricerca di risorse: gestisce inoltre le eventuali risposte ricevute. Le risorse condivise vengono annotate attraverso l'utilizzo di metadati RDF (nel paragrafo successivo vengono definiti l'insieme di metadati utilizzati per la caratterizzazione delle risorse). In questa prima versione di AlphaTheta non sono stati utilizzati degli schemi RDFS [6] standard predefiniti, ma è stato introdotto un insieme di metadati *ad-hoc* finalizzato all'annotazione di risorse riguardanti un museo d'arte.

4.1 Definizione dei metadati

Poiché AlphaTheta è nata per la condivisione di risorse riguardanti un museo d'arte si sono definite i seguenti metadati:

- Titolo: individua il titolo della risorsa, può assumere un qualsiasi valore di tipo stringa.

- Autore del documento: individua un autore della risorsa, può assumere un qualsiasi valore di tipo stringa.
- Tipo: individua il tipo di risorsa. Poiché si può effettuare una classificazione sui tipi, si è definita una ontologia dei tipi (*docType:myTypeOnt*); questo metadato è una istanza (cioè un valore) di una delle classi definite nell'ontologia.
- Soggetto: individua il soggetto della risorsa, per esempio se la risorsa è riferita ad un dipinto, ad una scultura, ad un pittore, etc.. Visto che l'applicazione permette di condividere delle risorse riguardanti soggetti di un museo d'arte, si è definita una ontologia sui soggetti di un museo d'arte (*museumOnt:myMuseumOnt*). Il soggetto quindi è una istanza di uno dei soggetti definiti nell'ontologia.
- Periodo d'arte: individua il periodo d'arte a cui fa riferimento la risorsa, per esempio se la risorsa è riconducibile al Rinascimento, all'Impressionismo, etc.. Questo metadato è una istanza di uno dei periodi definiti nell'ontologia *periodOnt:myPeriodOnt*.
- Artista: individua il nome di uno degli artisti a cui fa riferimento la risorsa, per esempio nel caso in cui la risorsa sia una biografia di Michelangelo. Artista può assumere un qualsiasi valore di tipo stringa.
- Lingua: individua la lingua della risorsa. Anche per la lingua si è definita una ontologia delle lingue (*langOnt:myLangOntology*), e quindi questo metadato assume uno dei valori definiti in questa ontologia.

Ciascuno di questi elementi viene rappresentato come una risorsa RDF non anonima [5], di seguito è mostrato la definizione RDF:

- titolo	⟶	`dc:title`
- autore del documento	⟶	`dc:creator`
- tipo	⟶	`type_md:type`
- soggetto	⟶	`art_md:taxon`
- periodo d'arte	⟶	`art_md:period`
- artista	⟶	`art_md:artist`
- lingua	⟶	`lang_md:language`

Dove si sono definiti i seguenti *namespace*:

```
dc = “http://purl.org/dc/elements/1.1/”.
type_md = “http://www.domain.com/resourcetype#”.
art_md = “http://www.domain.com/classification#”.
lang_md = “http://www.domain.com/language#”.
```

I metadati `<dc:title>`, `<dc:creator>`, sono stati ripresi dalla definizione dei metadati Dublin Core [16], gli altri sono stati definiti autonomamente nell'ambito dell'applicazione.

4.2 Definizione delle ontologie

I valori che possono essere assunti dai metadati soggetto, tipo, periodo d'arte, lingua possono essere rappresentati come ontologie in formato RDF. Sono state definite delle ontologie di esempio per permettere di testare l'applicazione. Ciascuna ontologia viene descritta in un file RDF/XML:

- soggetto → se_topic_ontology.rdf;
- periodo dell'arte → se_period_ontology.rdf;
- tipo di documento → se_doctype_ontology.rdf;
- lingua → se_lang_ontilogy.rdf.

L'applicazione AlphaTheta carica questi file all'avvio, quindi se si volesse cambiare una delle ontologie definite, sarebbe sufficiente modificare il file che la definisce senza bisogno di modificare il codice dell'applicazione: in tal modo è possibile aggiungere, eliminare o modificare le classi che compongono una ontologia senza doversi preoccupare di modificare il codice del programma. Di particolare interesse è la definizione dell'ontologia per i soggetti: infatti in questa ontologia si possono avere delle relazioni gerarchiche tra le classi, ciò permette all'utente di definire il livello di dettaglio da utilizzare per la caratterizzazione di una risorsa.
Queste ontologie vengono utilizzate sia per la creazione dei file RDF usati per la caratterizzazione delle risorse, che per la generazione delle *query* da parte dell'utente.
Di seguito è mostrata la definizione di una classe RDF in un file RDF/XML:

```
1. <?xml version='1.0' encoding='ISO-8859-1'?>

2. <!DOCTYPE rdf:RDF [
3.   <!ENTITY namespace "http://www.domain2.com/esempio#">
4. ]>

5.   <rdf:RDF
6.     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
7.     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
8.     xmlns:nameSpace="http://www.domain2.com/esempio#"  >
9.      //definiamo la classe nameSpace:nuovaclasse, il cui label è 'myLabel'
10.     <nameSpace:esempioOntologia rdf:about="&nameSpace;nuovaClasse">
11.       <rdf:value>valore</rdf:value>
12.       <rdfs:label>myLabel</rdfs:label>
13.     </nameSpace:esempioOntologia>
14.  </rdf:RDF>
```

Quindi il metadato può essere una istanza di una delle classi definite nell'ontologia corrispondente; ciascuna classe viene rappresentata come un risorsa RDF non anonima: hanno un *namespace* (sarà un riferimento URI) ed un nome, per esempio nella riga 10 viene definita la classe `"&nameSpace;nuovaClasse"`, viene utilizzato l'abbreviazione del *namespace* (la dichiarazione del *namespace* viene definita nella riga 3).

4.3 Provider Module

Questo modulo si occupa di gestire le funzionalità di *provider* fornite dall'applicazione. Si è già detto che AlphaTheta permette la condivisione di risorse riguardanti un museo d'arte. In realtà

ciascun *peer* condivide le informazioni su tutte le risorse conosciute. Vengono utilizzati i metadati definiti nel paragrafo 4.1 per la classificazione delle risorse, quindi ad ogni risorsa sono associati dei valori assunti da questi metadati, cioè la risorsa ha un titolo, può avere uno o più autori, appartiene ad un determinato tipo, ha uno o più soggetti, si riferisce ad un determinato periodo d'arte, si riferisce ad uno o più artisti, infine può essere associata ad una lingua. Inoltre ogni risorsa viene individuata da un riferimento URI, che specifica l'indirizzo tramite cui si può avere accesso alla risorsa. Queste informazioni sulle risorse vengono rappresentate in formato RDF, ciò viene fatto perché il *provider module* è in grado di caricare le informazioni da condividere da questi file RDF. All'interno di ogni file RDF possono essere presenti informazioni riguardanti più di una risorsa, per ciascuna risorsa si ha una istanze di `<rdf:Description>`. Attualmente AlphaTheta prevede soltanto una modalità manuale di creazione dei file RDF. Attraverso la *GUI* un utente può selezionare la creazione di un nuovo file RDF: apparirà una *form* attraverso cui l'utente può inserire i valori dei metadati e automaticamente viene generato il file RDF corrispondente. Nell'implementazione corrente viene creato un file RDF per ciascuna risorsa, i metadati titolo, tipo, lingua ammettono un unico valore, mentre per gli altri metadati è possibile inserire valori multipli. Inoltre nell'inserire i valori dei metadati soggetto, tipo, periodo d'arte, lingua, vengono mostrata all'utente le istanze definite nelle rispettive ontologie, dunque l'utente sceglierà il valore del metadato selezionando un valore (o multipli valori) dall'insieme di istanze definite nella corrispondente ontologia.
La classe `ProviderModule` implementa le funzionalità di questo modulo. Il costruttore di questa classe si deve occupare di avviare il servizio di *provider* fornito da Edutella (figura 6).

```
1. public ProviderModule(Configurator conf, String filename) {
2.     WrapperAlphaTheta
            providerconn=new WrapperAlphaTheta(filename);
3.     files.add(filename);
4.     connPool = new ProviderConnectionPool();
5.     connPool.setProviderConnectionClass(providerconn);
6.       //configuro il ProviderConnectionPool
7.     conf.finishConfig(connPool);
8.     provider= new EdutellaProvider(
                    new QueryProviderPlugin(connPool));
9.     }
```

Figura 6: costruttore della classe `ProviderModule`.

Il costruttore accetta due parametri:
- `Configurator conf`: oggetto che contiene le informazioni di configurazione del *peer*.
- `String filename`: nome del file RDF che contiene le informazioni da condividere.

Il servizio di *provider* di Edutella viene fatto partire (riga 8). L'applicazione utilizza il *wrapper* implementato dalla classe `WrapperAlphaTheta` (riga 2), che ha il compito di generare il deposito RDF dai file RDF caricati e di processare le *query* ricevute. Questo *wrapper* utilizza

come formato locale RDQL, ciò comporta la creazione, come base di conoscenza del *peer*, di un deposito di triple RDF (sono del tipo soggetto-predicato-oggetto).
AlphaTheta permette di aggiungere (o eliminare) dinamicamente altri file RDF per aggiungere (o eliminare) informazioni contenute nella base di conoscenza. Nella classe `ProviderModule` sono stati implementati i seguenti metodi pubblici:

- `addFile(String filename)`: permette di aggiungere alla base di conoscenza le informazioni contenute nel file "filename";
- `removeFile(String filename)`: elimina dalla base di conoscenza le informazioni contenute nel file "filename";
- `clearAll()`: elimina tutte le informazioni contenute nella base di conoscenza.

Comunque quando l'utente aggiunge un file, nella base di conoscenza vengono aggiunte solamente informazioni non ridondanti (non sono mai presenti due triple perfettamente uguali). Quindi se un utente inserisce file RDF contenenti informazioni ridondanti rispetto alla base di conoscenza posseduta (triple già presenti nella base di conoscenza), queste informazioni non vengono aggiunte.
La classe `WrapperAlphaTheta` permette di aggiungere o eliminare le informazioni contenute nella base di conoscenza attraverso i metodi:

- `addFile(String filename)`: carica dal file passato le triple RDF e le aggiunge alla base di conoscenza.
- `clearAll()`: elimina tutte le triple RDF contenute nella base di conoscenza.

`WrapperAlphaTheta` implementa l'interfaccia `ProviderConnection` (*package* `net.jxta.edutella.provider`) ed il metodo `executeQuery`() (figura 7): riceve la *query* in formato ECDM (riga 1, cioè come un oggetto di tipo `Query`), la trasforma in formato locale (riga 2), interroga la propria base di conoscenza (riga 3) e ritorna i risultati in ECDM (riga 4, cioè come un oggetto di tipo `ResultSet`).

```
1. public ResultSet executeQuery(Query eduquery) {
2.     String constrainRDQL = queryFormat.format(eduquery);
3.     QueryResults constrainQueryResult=
                              executeRDQLQuery(constrainRDQL);
4.     ResultSet resultset=
           importTupleResults(constrainQueryResult,eduquery);
5.     return resultset;
6. }
```

Figura 7: metodo `executeQuery()`.

Il servizio di *provider* viene avviato non appena l'utente aggiunge il primo file RDF contenente le informazioni da condividere, quindi non appena l'utente inserisce il primo file RDF, la *GUI* crea un oggetto di tipo `ProviderModule`, che si occupa del servizio di *provider* fornito dal *peer*:

```
ProviderModule prov=new ProviderModule(conf,filename);
```

4.4 AlphaTheta Model

Questo modulo si occupa della gestione delle funzionalità di *consumer* fornite dall'applicazione, quindi permette la generazione e l'invio di *query* per la ricerca di risorse nella rete Edutella, e la gestione di eventuali risposte ricevute. L'implementazione di questo modulo è stata definita nella classe `AlphaThetaModel.`

Questo modulo viene fatto partire all'avvio dell'applicazione: come primo passo, questo modulo effettua l'estrazione del servizio di *query* fornito dal gruppo Edutella, che di seguito viene utilizzato per l'invio e la ricezione delle richieste. Inoltre nel costruttore di `AlpahThetaModel` (figura 8) vengono caricate le ontologie definite nei file RDF per i metadati soggetto, tipo, lingua, periodo d'arte.

```
1. public AlphaThetaModel() {
     //costruisco un albero per i soggetti.
     //L'ontologia è definita nel file "se_topic_ontology.rdf"
2.   topicsTree=new ResourceLabelTree(
                  "se_topic_ontology.rdf",
                  "http://www.domain2.com/museumontology#myMuseumOnt",
                  TOPIC, true);
     //costruisco una mappa per i tipi.
     //L'ontologia dei tipi è definita nel file "se_doctype_ontology.rdf"
3.   typesMap=new ResourceLabelMap(
                  "se_doctype_ontology.rdf",
                  "http://www.domain2.com/typeontology#myTypeOnt");
     //costruisco una mappa per le lingue.
    // L'ontologia delle lingue è definita nel file "se_lang_ontology.rdf"
4.   languagesMap=new ResourceLabelMap(
                  "se_lang_ontology.rdf",
                  "http://www.domain2.com/languageontology#myLangOnt");
     //costruisco una mappa per i periodi d'arte.
    //L'ontologia dei periodi è definita nel file "se_period_ontology.rdf"
5.   periodsMap=new ResourceLabelMap(
                  "se_period_ontology.rdf",
                  "http://www.domain2.com/periodontology#myPeriodOnt");
6.   try {
      //estraggo il servizio di query dal gruppo Edutella
7.   queryService =
8.         EdutellaFactory.getEdutellaPeerGroup().getQueryService();
9.   } catch (Exception e) {
10.     log.error("Error during init():", e);
11.  }
12. }
```

Figura 8: costruttore della classe `AlphaThetaModel`.

Per le ontologie vengono utilizzate le seguenti strutture dati:

- *net.jxta.edutella.sample.util.ResourceLabelTree*: questa classe permette la costruzione di alberi, i cui nodi sono delle risorse RDF, da ontologie descritte in RDF. AlphaTheta utilizza

questa struttura dati per la rappresentazione dell'ontologia che definisce i soggetti (figura 8, riga 2), infatti si suppone che questa sia l'unica ad avere una struttura di tipo gerarchica.

- *net.jxta.edutella.sample.util.ResourceLabelMap*: questa classe permette la costruzione di liste i cui elementi sono dei nodi RDF, da ontologie descritte in RDF. AlphaTheta utilizza questa struttura dati per la rappresentazione delle ontologie che definiscono il tipo, il periodo d'arte e la lingua (figura 8, righe 3, 4, 5).

Riprendendo il funzionamento di un *consumer* Edutella (figura 4), il modulo a*lphatheta model* si occupa dell'implementazione del modulo che permette la generazione di *query* in ECDM (*generaQuery*) e dell'implementazione del *ResultListener* che ha il compito di gestire i risultati ottenuti. Per l'invio della *query* l'implementazione del servizio *serviceQuery* mette a disposizione due metodi:

- `executeQuery(Query query, ResultListener listener, ServiceInfo service)`
- `executeQuery(Query query, ResultListener listener)`

entrambi i metodi accettano la *query* da inviare e il *listener* che deve elaborare le eventuali risposte ricevute. Il *listener* è un oggetto di tipo `ResultListener` (*package* `net.jxta.edutella.consumer`). La seconda versione del metodo `executeQuery()` accetta anche il *provider* verso cui inviare la *query*. Infatti un *consumer* può decidere di inviare la *query* verso un determinato *provider* invece che verso tutti i *provider* scoperti. Il *consumer* conserva i riferimenti dei *provider* come oggetti di tipo `ServiceInfo` (*package* `net.jxta.edutella.consumer`), quindi al metodo `executeQuery()` si deve passare l'oggetto `ServiceInfo` che mantiene il riferimento del *provider* verso cui inviare la *query*.
La ricerca di una particolare risorsa è realizzata mediante l'invio di una *query* verso i *provider* disponibili sulla rete Edutella in cui vengono richieste tutte quelle risorse che hanno particolari valori dei metadati. Per esempio la richiesta "Ricercare tutte le risorse il cui titolo è *La vita di Michelangelo* e il cui autore è *Bianchi "*, corrisponde ad una *query* con cui vengono richieste tutte quelle risorse con il vincolo che Titolo='La vita di Michelangelo' e Autore='Bianchi'. Nell'implementazione dell'applicazione AlphaTheta sono state definite due modalità di ricerca: una in cui il valore di alcuni metadati viene inserito direttamente dall'utente (*InsertSearch*), l'altro in cui i valori dei metadati vengono selezionati da valori predefiniti (*SelectionSearch*).

InsertSearch

Per questo tipo di ricerca la classe `AlphaThetaModel` fornisce i seguenti metodi:

- `setTitle(String titolo), setArtistName(String artista) setAuthor(String autore), setSubject(String soggetto):` ognuno di questi metodi permette di settare il valore del corrispondente campo.
- `executeInsertSearch(ServiceInfo svc)`: questo metodo genera la *query* corrispondente, e la invia al *provider* `svc`, se `svc!=NULL`, o a tutti i *provider* scoperti, se `svc==NULL`.

La *query* generata utilizzando questo tipo di ricerca, espressa in Datalog, è del tipo:

```
?-(Resource, Property,Object):- qel:s(Resource, Predicate, Object),
   qel:s(Resource,<dc:title>,Titolo),qel:like(Titolo,%“titolo”%),
   qel:s(Resource,<dc:creator>,Autore),qel:like(Autore, %“autore”%),
   qel:s(Resource,<art_md:taxon>,Soggetto),
     qel:like(Soggetto,%“soggetto”%),
   qel:s(Resource,<art_md:artist>,Artista),
     qel:like(Autore, %“artista”%).
```

Cioè vengono ricercate tutte le risorse il cui titolo contiene la stringa “titolo”, il cui autore del documento contiene la stringa “autore”, il cui soggetto contiene la stringa “soggetto”, ed il cui nome dell’artista contiene la stringa “artista”.

SelectionSearch

Un utente ha la possibilità di selezionare i valori dei metadati soggetto, periodo d’arte, tipo e lingua dai valori predefiniti nelle ontologie. Infatti, come già detto, per questi metadati è possibile definire delle ontologie che permettono di classificare le diverse istanze. Ciascuna ontologia viene descritta in un file RDF, l’applicazione carica questi file all’avvio, mostrando all’utente i possibili valori che può assumere ciascun metadato. Quindi l’utente può effettuare la richiesta selezionando per ciascun metadato uno dei possibili valori definiti dalla corrispondente ontologia.

L’ontologia riguardante i soggetti assume una forma gerarchica, cioè possono esserci relazioni di tipo padre/figlio tra le classi soggetto. Nel momento in cui si effettua una ricerca per una risorsa con un determinato soggetto, per esempio “padre”, AlphaTheta permette di ottenere tutte le risorse che hanno come soggetto “padre”, ed anche tutte le risorse che hanno come soggetto una delle qualsiasi classi discendenti dalla classe “padre”: ciò è possibile perché una istanza di una sottoclasse può essere considerata come istanza della classe padre. Questo permette ad un utente di scegliere il livello di dettaglio di una determinata *query*. Considerato il fatto che l’implementazione delle funzionalità di *provider* riesce solamente a gestire *query* di livello RDF-QEL-1 (*query* puramente congiuntive senza regole), il meccanismo descritto prima può essere implementano nel seguente modo: selezionato un soggetto, viene effettuata una *query* inserendo come vincolo il soggetto selezionato, ed inoltre vengono effettuate tante *query* quante sono le sottoclassi del soggetto selezionato, inserendo in ciascuna di esse come vincolo la sottoclasse.

La classe `AlphaThetaModel` mette a disposizione il seguente metodo per effettuare questo tipo di ricerca:

- `executeSelectionSearch(String soggetto, String tipo, String periodo, String lingua, ServiceInfo s)`:

questo metodo genera l’insieme delle *query* corrispondenti ai valori passati (soggetto, tipo, periodo, lingua), e le invia utilizzando il servizio di *query*. Dati i valori dei campi selezionati, viene generato, per ciascuna sottoclasse del soggetto selezionato, il seguente tipo di *query*:

```
?-(Resource, Predicate, Object):- qel:s(Resource, Predicate, Object),
    qel:s(Resource,<art_md:taxon>, ‘subjectValue’),
    qel:s(Resource,<art_md:period >, ‘periodValue’),
    qel:s(Resource,<type_md:type >, ‘typeValue’),
    qel:s(Resource,<lang_md:language >, ‘langValue’).
```

Cioè vengono ricercate tutte le risorse il cui soggetto è ‘subjectValue’, il cui periodo d’arte è ‘periodValue’, il cui tipo è ‘typeValue’ e la cui lingua è ‘langValue’.

Il modulo a*lphatheta model* si occupa anche della gestione dei risultati ottenuti dall’esecuzione di una ricerca. Ciò viene fatto attraverso l’implementazione di un *ResultListener* che ha il compito di processare gli eventuali risultati ricevuti. All’interno della classe `AlphaThetaModel` viene dichiarata la classe `ResultAdder` che implementa l’interfaccia `ResultListener`:

```
public class ResultAdder implements ResultListener{
   }
```

Infatti, quando viene effettuata una ricerca sulla rete Edutella, sia utilizzando la ricerca *InsertSearch* che la ricerca *SelectionSearch*, viene assegnato come *listener* dei risultati ottenuti un oggetto di tipo `ResultAdder`:

```
queryService.executeQuery(query,new ResultAdder(query));
```

La classe `ResultAdder` definisce i seguenti metodi:

- `processEvent()`: viene chiamato ogni volta che viene ricevuto un risultato per una *query* i cui risultati devono essere processati da questo *listener*;
- `stop()`: termina l’esecuzione del processamento dei risultati ottenuti.

I risultati sono visualizzati in una tabella in cui ad ogni riga corrisponde un risultato. Nelle colonne vengono visualizzati i seguenti campi per ciascuna riga:

- *title*: titolo della risorsa;
- *authors*: autori del documento, si suppone che ci possano essere più autori per ogni risorsa, questi vengono visualizzati separati da una virgola;
- *type*: tipo di risorsa;
- *periods*: periodi d’arte di riferimento, anche questo campo può avere valori multipli come *authors*;
- *topics*: argomenti della risorsa, anche questo campo può avere valori multipli;
- *artistnames*: nomi degli artisti a cui la risorsa si riferisce, può avere valori multipli;
- *source*: il nome del *peer* sorgente che ha inviato questa risposta.

Ad ogni risultato, quindi ad ogni riga della tabella, corrisponde una risorsa che soddisfa la *query* effettuata dal *peer*. Ricordando che le risorse sono identificate univocamente da un URI, a ciascuna riga della tabella dei risultati corrisponderà un URI (effettuando un doppio clic sulla riga si attiva il browser con il riferimento URI tramite cui è possibile avere accesso alla risorsa).

4.5 GUI AlphaTheta

Questo modulo permette l’interazione tra l’utente e la rete Edutella. Un utente attraverso la *GUI* può accedere alle funzionalità di *provider*, aggiungendo (o eliminando) file RDF da condividere, e alle funzionalità di *consumer*, effettuando delle ricerche di risorse sulla rete Edutella. Come mostrato in figura 9, la *GUI* è composta da diversi pannelli, ognuno dei quali gestisce una determinata funzionalità dell’applicazione.

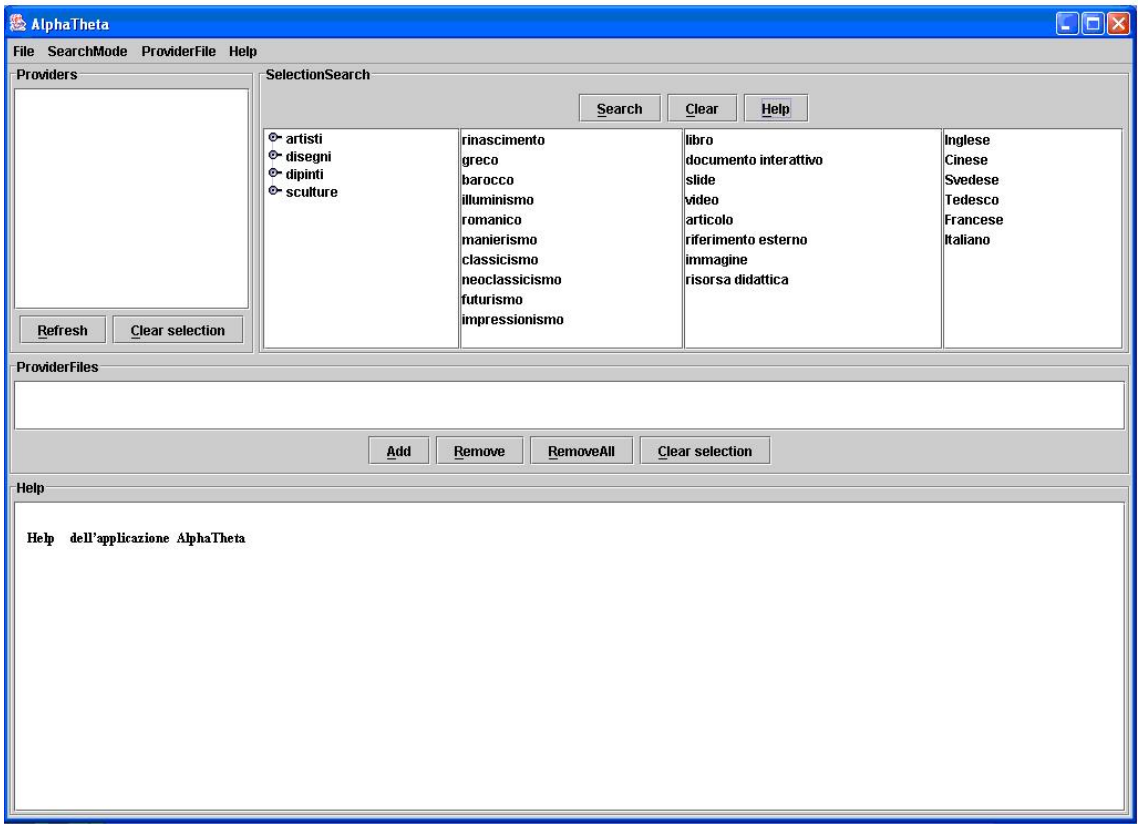

Figura 9: interfaccia utente dell'applicazione AlphaTheta.

In figura 10 è mostrato un diagramma delle classi UML delle varie componenti che formano l'interfaccia utente.

L'implementazione delle componenti *GUIAlphaTheta*, *ElementsInsertSearch*, *ElementsSelectionPanel*, *ProviderFilesSelector* è stata inserita nel progetto AlphaTheta (*package* `alphatheta`), mentre per le altre componenti sono state utilizzate le implementazioni definite nel progetto Edutella (rispettivamente dalle classi: `net.jxta.edutella.consumer.ProviderSelector`, `net.jxta.edutella.sample.util.HelpPane`, `net.jxta.edutella.sample.util.ResultTablePane`).

La classe `GUIAlphaTheta` gestisce l'interfaccia utente completa dei pannelli definiti dalle altre componenti; nella classe `main` per eseguire l'applicazione AlphaTheta bisogna instanziare un oggetto di tipo `GUIAlphaTheta`:

```
public static void main(String[] args) {
        ......................
   AlphaThetaModel consumerModule=new AlphaThetaModel();
    new GUIAlphaTheta(consumerModule,cf);
 }
```

il costruttore della classe `GUIAlphaTheta` accetta l'oggetto `Configurator` contenente le informazioni di configurazione del *peer* ed un oggetto di tipo `AlphaThetaModel` che implementa il modulo che gestisce le funzionalità di *consumer* del *peer*.

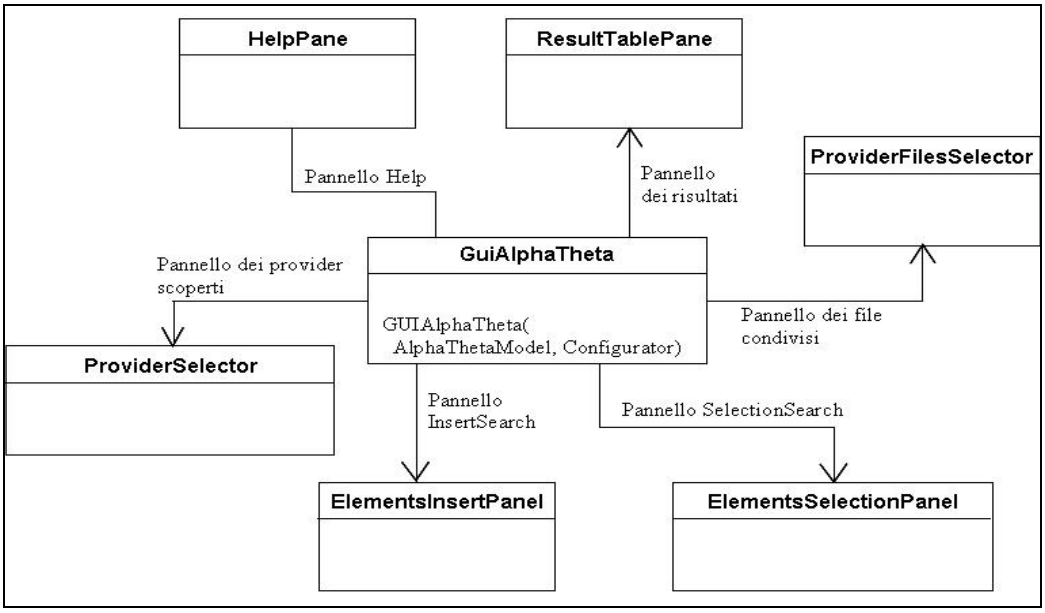

Figura 10: diagramma delle classi UML illustrante le componenti del modulo *GUI*.

4.6 Classi della libreria edutella.jar modificate

Nell'implementazione dell'applicazione AlphaTheta sono state modificate alcune classi della libreria *edutella.jar*, che implementa il *framework* Edutella. Di seguito sono mostrate le modifiche apportate:

Classe net.jxta.edutella.util.Configurator

In questa classe è stato aggiunto il metodo:

```
finishConfig(Configurable obj)
```

che permette di configurare l'oggetto passato. Questo metodo è utile nel caso in cui si voglia configurare qualche oggetto, dopo aver configurato gli oggetti registrati. Nell'implementazione di AlphaTheta questo metodo viene utilizzato nella classe `ProviderModule` per configurare l'oggetto di tipo `ProviderConnectionPool`, che gestisce i *wrapper* per il servizio di *provider*, attraverso l'oggetto `Configurator` passato nel costruttore (figura 6, riga 7). Infatti quando viene fatto partire il servizio di *provider* attraverso l'istanziazione di un oggetto di tipo `ProviderModule` bisogna passare al costruttore di questa classe l'oggetto di tipo `Configurator` che contiene le informazioni di configurazione.

Classe net.jxta.edutella.provider.ProviderConnectionPool

In questa classe è stato aggiunto il seguente metodo:

```
public void setProviderConnectionClass(WrapperAlphaTheta providerAdapter) {
```

```
        this.providerConnection = providerAdapter;
    }
```

Questo metodo permette di settare il *wrapper* gestito da questa classe con l'oggetto di tipo `WrapperAlphaTheta` passato. Questo metodo è utilizzato nella classe `ProviderModule` per settare il *wrapper* che deve essere utilizzato per processare le *query* ricevute (figura 6, riga 5).

Inoltre in questa classe è stato modificato il metodo `createConnection()`, di seguito è mostrata la nuova versione:

```
private PooledConnection createConnection() {
       return providerConnection;
 }
```

Classe net.jxta.edutella.consumer.ConsumerServices.ServiceInit

In questa classe è stato modificato il metodo `addService()`, questo metodo viene chiamato ogni volta che un *consumer* scopre qualche *peer* che fornisce il servizio di *provider* nella rete Edutella, e aggiunge il *peer* scoperto alla lista di *provider* posseduta. Visto che l'applicazione AlphaTheta implementa sia le funzionalità di consumer che di provider, l'implementazione modificata di questo metodo, rispetto all'implementazione originale, aggiunge i *provider* scoperti dal *peer* a meno che non sia il *peer* stesso a fornire il servizio di provider.

Classe net.jxta.edutella.sample.ResourceLabelTree

In questa classe è stato reso pubblico il seguente metodo:

```
public DefaultMutableTreeNode getNode(Resource soggetto){
        ..................
    }
```

infatti questo metodo viene invocato nel metodo `executeSelectionSearch()` per ottenere l'albero dei discendenti del nodo `soggetti` passato al metodo: questi discendenti sono utilizzati per generare l'insieme di *query* da inviare verso i *provider*.

4.7 Download, installazione e avvio dell'applicazione

Come per tutte le applicazioni *peer-to-peer* anche in AlphaTheta si ha la necessità di un *web server* centrale tramite cui sia possibile:

- scaricare l'applicazione;
- accedere ad una lista di *rendezvous* disponibili sulla rete Edutella che permettano ad un qualsiasi *peer* di connettersi alla rete.

Su questo *web server* viene pubblicata una semplice pagina *web* tramite cui un utente può scaricare l'applicazione ed avere accesso alle informazioni sull'applicazione. Attraverso

l'indirizzo URL *http://localhost/alphatheta/index.html* si accede alla pagina iniziale. Nell'area *Download* è possibile scaricare il file di installazione "setup.exe" che permette di installare l'applicazione AlphaTheta. Attualmente viene fornita soltanto la versione per le piattaforme Windows. Per una corretta esecuzione dell'applicazione è necessario possedere la JDK1.4, o comunque versioni superiori; in caso contrario nell'area *Link* del sito viene fornito un collegamento tramite cui è possibile scaricare la JDK opportuna.
Una volta scaricato il file di installazione, è possibile far partire l'installazione dell'applicazione AlphaTheta. L'unica informazione necessaria che l'utente deve inserire durante l'installazione è il percorso della directory su cui installare l'applicazione.
Il programma di installazione crea la directory scelta dall'utente, all'interno della quale vengono inseriti i seguenti file:

- *alphathetaW.exe*: attraverso questo file eseguibile è possibile avviare l'esecuzione dell'applicazione AlphaTheta;
- *edutella.propriertes*: contiene le informazioni di configurazione. Questo file può essere modificato da un qualsiasi utente, comunque la modifica è consigliata solo ad utenti esperti.
- *se_doctype_ontology.rdf, se_topic_ontology.rdf, se_lang_ontology.rdf, se_period_ontology.rdf*: ognuno di questi file contiene la definizione dell'ontologia corrispondente, e viene caricato all'avvio dell'applicazione. Anche questi file possono essere modificati, comunque la modifica è consigliata solo ad utenti esperti;
- *uninstall.exe*: file eseguibile di disinstallazione dell'applicazione AlphaTheta.

Dopo aver istallato l'applicazione un utente può avviarla e può utilizzarne i servizi messi a disposizione.

Configurazione di AlphaTheta
Prima di avviare l'applicazione AlphaTheta bisogna configurarla: cioè inserire tutte le informazioni utili per l'esecuzione dell'applicazione. Le informazioni di configurazioni vengono caricate dal file 'edutella.properties' (figura 11, riga 9) attraverso un oggetto di tipo `Configurator`.
La classe `Configurator` definisci il seguente costruttore:

```
Configurator(String propFile, String[] clArgs)
```

il quale accetta due argomenti:

- `propFile`*:* nome del file in cui sono contenute le informazioni di configurazione.
- `clArgs`*:* argomenti passati dalla riga di comando.

Quindi le informazioni di configurazione vengono estratte dal file definito nel costruttore di questa classe.

```
 1. package alphatheta;
 2. import net.jxta.edutella.peer.EdutellaFactory;
 3. import net.jxta.edutella.peer.util.JxtaConfigBuilder;
 4. import net.jxta.edutella.util.Configurator;
 5. import net.jxta.edutella.util.OutputRedirector;
 6.
 7. public class Main {
 8.   public static void main(String[] args) {
       //le informazioni di configurazioni vengono caricate dal file
       //"edutella.properties"
9.     Configurator cf = new Configurator("edutella.properties", args);
10.    cf.setAppInfo("fornisce una interfaccia utente "+
               " per effettuare ricerche basate su metadati RDF "+
               " e per condividere file RDF nella rete Edutella");
       //vengono registrati gli oggetti da configurare
11.    OutputRedirector con = new OutputRedirector();
12.    con.setTitle("Console Alphateta");
13.    cf.register(con);
14.    JxtaConfigBuilder jxtaConf = new JxtaConfigBuilder();
15.    cf.register(jxtaConf);
16.    f.register(EdutellaFactory.getInstance());
       // configurazione degli oggetti registrati
17.    cf.finishConfig();
18.    //viene creato un oggetto di tipo AlphaThetaModel che ha il
19.    //compito di gestire le funzionalità di consumer
20.    AlphaThetaModel consumerModule=new AlphaThetaModel();

21.    //viene fatta partire l'interfaccia dell'applicazione AlphaTheta
22.    new GUIAlphaTheta(consumerModule,cf);
23.   }
24. }
```

Figura 11: classe `Main`, che gestisce l'avvio dell'applicazione AlphaTheta.

In figura 12 è riportato un tipico esempio che mostra le informazioni contenute ed il formato in cui queste vengono rappresentate in un file "edutella.properties".
Le righe che iniziano con il carattere '#' sono considerate commenti. Ad ogni altra riga corrisponde una determinata informazione. Una informazione è individuata da un nome e da un valore. Per esempio l'informazione sulla porta TCP preferita si chiama `jxta.tcpPort` e può assumere un qualsiasi valore decimale (normalmente un valore compreso tra 9700 e 9780). Le informazioni più comuni che si possono trovare in un tipico file di questo tipo sono:

- `jxta.configDir` : in questo campo bisogna indicare la directory all'interno della quale dovrebbero essere conservati i file di configurazione. Se questa directory è specificata, sostituisce la directory "./jxta", che viene creata normalmente da qualsiasi applicazione che usa JXTA per conservare i file di configurazione.
- `console.swing` : assume valore 'true' o 'false'. Se viene settato a 'true' l'*output* del programma viene scritto su una finestra che apparirà durante l'esecuzione del programma.
- `console.logFile` : possiamo indicare il nome di un file in cui conservare tutte le informazioni di log del programma.

- `jxta.rendezvous` : in questo campo bisogna inserire la lista degli indirizzi dei *rendezvous* conosciuti. Gli indirizzi vengono delimitati con il ';' e iniziano con 'tcp://' o 'http://' a secondo del tipo di trasporto utilizzato.
- `jxta.relays` : inseriamo la lista di *relay* conosciuti.
- `jxta.tcpPort` : indichiamo il numero di porta TCP preferito dal *peer* (se il numero di porta non è disponibile, al *peer* viene assegnato un numero casuale).
- `jxta.httpPort` : indichiamo il numero di porta HTTP preferito dal *peer* (valgono in proposito le stesse considerazioni fatte per la porta TCP).
- `jxta.proxyAddr` : in questo campo inseriamo l'indirizzo ed il numero di porta di un *proxy* che dovrebbe essere usato per gestire il traffico http nel caso in cui il *peer* si trovi dietro un *firewall* o un NAT.
- `jxta.actAsRendezvous` : se settato a 'true', il *peer* si attiva come un *rendezvous*.
- `jxta.actAsRelay` : se settato a 'true', il *peer* si attiva come *relay*.

```
#commento
console.swing=true
console.logFile=mylog.txt
jxta.configDir=myproject1
jxta.autoConfig=true
jxta.rendezvous=tcp://130.75.87.3:9701;http://130.75.87.3:9700;
     tcp://130.75.152.40:9701;http://130.75.152.40:9700;
     tcp://127.0.0.1:9701;http://127.0.0.1:9700
jxta.relays=http://130.75.87.3:9700;http://130.75.152.40:9700;
     http://127.0.0.1:9700
jxta.reconf=true
jxta.tcpPort=9704
jxta.lowPort=9704
jxta.highPort=9780
jxta.proxyAddr=myproxy.mydomain.org:8080
jxta.actAsRelay=true
jxta.actAsRendezvous=true
```

Figura 12: esempio di un tipico file di configurazione.

Download dei *rendezvous* disponibili

Come già detto, il *web server* è necessario per ottenere alcuni degli indirizzi dei *rendezvous* presenti nella rete Edutella. Nell'area *Download* del sito è possibile scaricare il file "rendezvous.cgi" che contiene gli indirizzi dei *rendezvous* disponibili. Un utente esperto può inserire questi indirizzi direttamente sul file di configurazione "edutella.propriertes", come è

stato mostrato precedentemente. Gli utenti meno esperti possono inserire in modo trasparente questi indirizzi attraverso il *JXTA Configurator*, che appare all'avvio dell'applicazione. Infatti, dal pannello *Rendezvous/Relays* del *JXTA Configurator* premendo il bottone 'Download relay and rendezvous lists' appare la finestra 'Load list from URL', in cui bisogna inserire l'indirizzo URL da cui scaricare la lista di *rendezvous* (gli indirizzi dei *rendezvous* messi a disposizione dal *server* vengono sempre rappresentati in file '*.cgi'). Per quanto riguarda l'applicazione AlphaTheta l'utente dovrebbe inserire l'indirizzo *http://localhost/alphatheta/rendezvous.cgi*.

5. TECNICHE DI ROUTING PER RETI P2P BASATE SU METADATI

La funzionalità di *consumer* è stata implementata utilizzando il servizio di *query* definito dal progetto Edutella. Quindi in questa prima versione di AlphaTheta le richieste effettuate da un utente vengono inviate verso tutti i *provider* conosciuti (un utente può anche selezionare un solo *provider* verso cui inviare la *query*). Per migliorare l'efficienza dell'applicazione si possono utilizzare delle tecniche di *routing* delle richieste, basate su metadati [17]. Queste tecniche utilizzano dei meccanismi molto efficienti che permettono di inviare le *query* verso dei particolari *super-peer* che hanno il compito di indirizzarle verso quei *provider* che hanno più probabilità di rispondere, invece di inviarle a tutti i *provider* conosciuti (come avviene attualmente). I *super-peer* operano come *server* nei confronti dei *peer* connessi a loro, fornendo a questi alcuni servizi [18] (i *peer* di *rendezvous* nel progetto JXTA sono in qualche modo dei *super-peer* visto che svolgono delle funzioni di *server* nei confronti dei *peer edge* connessi a loro).
un *consumer* non invia più la *query* verso i *provider* conosciuti, ma la invia verso il *super-peer* a cui è connesso ed è compito del *super-peer* indirizzare la *query* verso i *provider* presenti nella rete. Comunque rimane il *provider* stesso ad inviare le eventuali risposte alla *query*.
Secondo questo approccio i *super-peer* hanno una vera e propria funzione di *routing*: devono, infatti, mantenere degli indici (tabelle di *routing*) che permettano una volta ricevuta la *query* di indirizzarla verso uno (o più) *provider* presenti nella rete.
Per migliorare l'efficienza della rete si può decidere di organizzare i *super-peer* in una topologia con particolari proprietà (ad esempio HyperCuP [19]).
I *super-peer* dovrebbero mantenere due tipi di tabelle di *routing*:
- una prima tabella che contenga gli indici per indirizzare le *query* verso eventuali *provider* connessi direttamente al *super-peer* (*routing SP-P*);
- una seconda tabella che permetta di indirizzare le *query* verso altri *super-peer* (*routing SP-SP*).

Routing SP-P
In questa tabella i *super-peer* mantengono le informazioni sui metadati usati da ciascun *peer* connesso a loro. Ciascun *provider* pubblica sul proprio *super-peer* un *advertisement* contenente le informazioni sugli schemi e sui metadati utilizzati. Nelle tabelle di *routing* si hanno tre tipi di indici (figura 13):

- Indice dello schema: contiene l'identificatore ed i *provider* che supportano questo schema. Le *query* vengono indirizzate solo verso i *provider* che supportano gli schemi utilizzati nella *query*.
- Indice delle proprietà: i *provider* possono scegliere solo determinati metadati (proprietà) per descrivere le proprie risorse. Attraverso questo tipo di indice il *super-peer* identifica quali metadati sono utilizzati nei vari *provider*.
- Indice dei valori delle proprietà: per alcuni tipi di proprietà potrebbe essere utile conservare anche il valore che assume la proprietà su un determinato *provider*. Questo tipo di indice dovrebbe essere usato solo per quelle *proprietà* che vengono usate in modo molto frequente, per non accrescere eccessivamente il livello di traffico sulla rete.

Per assicurare che queste tabelle di *routing* rimangano consistenti, i *provider* dovrebbero notificare degli *update* ai *super-peer*, quando avvengono delle modifiche ai propri contenuti.

Granularità	**Valori**		**Provider**
Schema	dc art_md		Prov1, Prov2 Prov2
Proprietà	dc:title dc:creator art_md:artist		Prov1, Prov2 Prov2 Prov2
Valore delle proprietà	dc:title dc:title	"titolo1" "titolo2"	Prov1 Prov2

Figura 13: esempio di tabella SP-P.

Quando un *super-peer* riceve una determinata *query* estrae le informazioni sui metadati da questa, confronta queste informazioni con quelle possedute nella propria tabella ed invia la *query* verso quei *provider* che hanno più probabilità di risposta.

Granularità	**Query**	
Schema	dc	
Proprietà	dc:title	
Valore delle proprietà	dc:title	"titolo1"

Figura 14: contenuti di una *query.*

Supponendo che il *super-peer* che possegga la tabella in figura 13 riceva la seguente *query* (viene espressa in Datalog per maggiore chiarezza):

?- *qel:s(Resource, Predicate, Object),*
qel:s(Resource, <dc:title>, "titolo1").

le informazioni che il *super-peer* estrae dalla *query* sono mostrate in figura 14.

Basandosi su queste informazioni il *super-peer* indirizza la *query* verso il *provider* Prov1, che eventualmente risponde in modo diretto al *consumer* P1 che ha effettuato la richiesta (figura 15).

Routing SP-SP

Come fatto per i *peer*, per evitare che un *super-peer* indirizzi le *query* in *broadcast* a tutti i *super-peer*, ciascun *super-peer* mantiene delle tabelle di *routing SP-SP* che permettono di propagare efficientemente le *query* tra i *super-peer*. Queste tabelle SP-SP dovrebbero essere estratte e riassunte dalle tabelle locali SP-P: contengono, infatti, le stesse informazioni delle tabelle SP-P ma si riferiscono ai *super-peer* vicini.

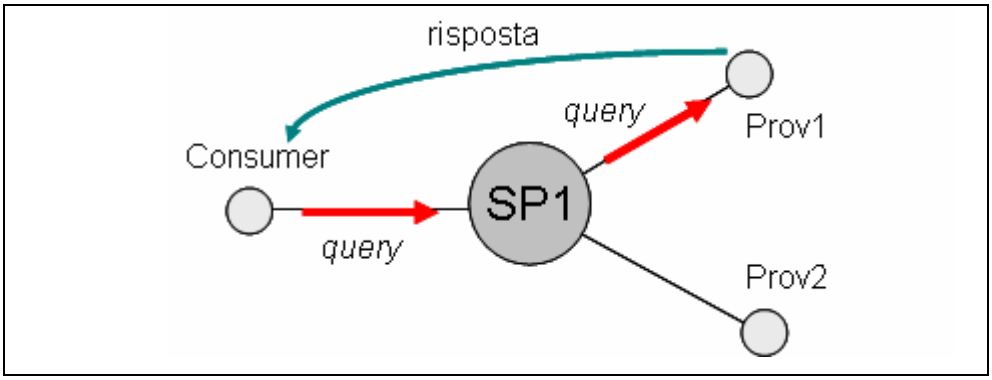

Figura 15: *routing* da parte di un *super-peer* che riceve una *query* da un *consumer*.

Il meccanismo di *routing* appena descritto potrebbe essere implementato come un servizio JXTA: quindi un qualsiasi *peer* che entra a far parte della rete Edutella, se possiede le giuste credenziali, può implementare il servizio di *routing* diventando un *super-peer*. Un *peer* che entra a far parte di una rete Edutella dovrebbe comunque stabilire una connessione con almeno uno dei *rendezvous* presenti; dopo essersi unito alla rete Edutella, dovrebbe scoprire almeno un *super-peer* presente nella rete e stabilire una connessione virtuale con questo.

Strategia di *clustering* basata su metadati

Possono essere sviluppate delle tecniche di *clustering* che organizzano la rete Edutella in zone in cui siano presenti *peer* che abbiano gli stessi interessi. Ciascun *super-peer*, insieme ai *peer* a lui connessi, crea una propria zona di interesse (figura 16).

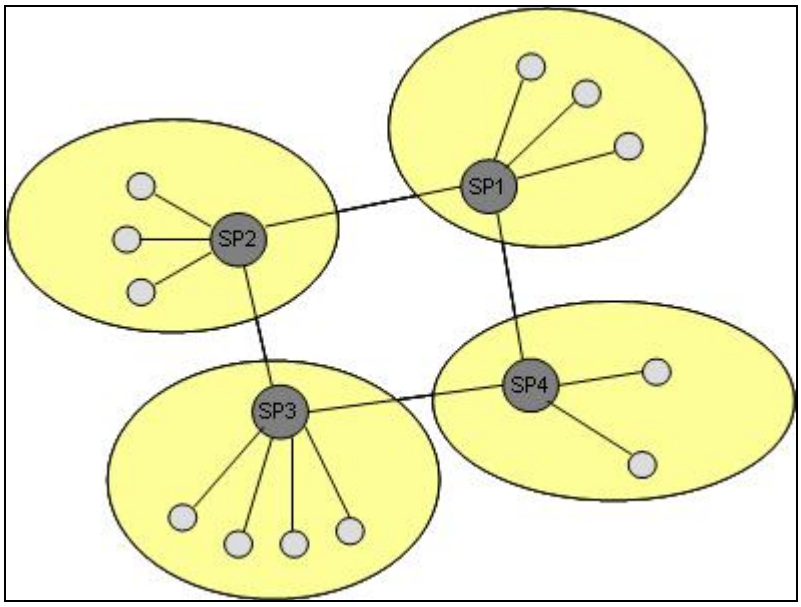

Figura 16: *clustering* sulla rete Edutella

Quando un *peer* invia una ricerca di qualche *super-peer* nella rete Edutella, può includere un *advertisement* contenente le informazioni sui metadati gestiti (cioè, sugli interessi del *peer*). Ciascun *super-peer* che riceve la richiesta, risponde al *peer* includendo una qualche informazione (potrebbe essere anche un semplice numero che misuri il grado di compatibilità), che indichi la compatibilità tra i metadati utilizzati nella zona associata al *super-peer* con i metadati gestiti dal *peer*. Il *peer* dovrebbe scegliere di unirsi alla zona corrispondente al *super-peer* che ha maggiore compatibilità con il *peer*. Ciascun *super-peer* potrebbe calcolare il grado di compatibilità analizzando le tabelle SP-P
e SP-SP. Infatti attraverso le tabelle SP-P è possibile ottenere informazioni sugli interessi dei *peer* connessi al *super-peer* (quindi informazioni riguardante la zona del *super-peer*), mentre con la tabella SP-SP è possibile ottenere informazioni riguardanti i *super-peer* vicini (cioè informazioni sulle zone vicine). Inoltre, per non sovraccaricare un *super-peer,* nel calcolo del grado di compatibilità potrebbe anche essere inclusa la dimensione della zona associata al *super-peer* (cioè del numero di *peer* connessi), facendo in modo che superata una certa dimensione (che risulta la massima sopportabile dal *super-peer*) il grado di compatibilità risulti molto piccolo (se non addirittura zero), in questo modo è possibile bloccare il sovraccarico di un *super-peer*.

6. CONCLUSIONI E SVILUPPI FUTURI

La progettazione dell'applicazione AlphaTheta dimostra come sia possibile utilizzare il *framework* Edutella in un qualsiasi ambito applicativo. Grazie all'utilizzo di Edutella, l'applicazione *peer-to-peer* progettata mostra i vantaggi offerti dall'uso di metadati:

- effettuare *query* molto complesse, grazie all'uso del servizio di *query* di Edutella, che mette a disposizione un linguaggio basato su RDF (RDF-QEL-i [7]) che permette di interrogare depositi RDF.
- condividere qualsiasi tipo di risorsa, grazie all'uso del servizio di *provider* definito nel progetto Edutella, che permette di annotare le risorse da condividere attraverso i valori assunti dai metadati definiti, costruendo depositi RDF che conservano le informazioni sulle risorse condivise. Infatti nella rete essenzialmente ciò che viene condiviso riguarda le informazioni sulle risorse: valori assunti dai metadati e riferimento tramite cui accedere alla risorsa.

Inoltre l'utilizzo della piattaforma JXTA [8] nel progetto AlphaTheta, oltre ad aver permesso uno sviluppo veloce ed efficiente dell'applicazione, la rende interoperabili e permette ad una gamma assai ampia di dispositivi di poter utilizzare l'applicazione.

Possibili sviluppi futuri di AlphaTheta dovrebbero porre particolare attenzione all'implementazione di tecniche di *routing* e *clustering* basate su metadati [17] (l'implementazione, nei futuri sviluppi del progetto AlphaTheta, della tecnica esposta nel paragrafo precedente dovrebbe migliorare di gran lunga l'efficienza e la scalabilità del sistema). Altri sviluppi futuri potrebbero riguardare l'applicazione di tecniche avanzate di localizzazione e collocazione delle risorse, per esempio CAN [20], Chord [21]. Entrambe queste tecniche utilizzano un modello di localizzazione e collocazione basato su degli identificatori assegnati alle risorse condivise ed ai *peer* presenti nella rete: lo spazio degli identificatori viene diviso (la differenza sostanziale tra le varie tecniche è proprio il modo in cui viene diviso questo spazio) tra i vari *peer* presenti, ognuno dei quali ha il compito di mantenere le informazioni sui documenti che hanno un ID appartenente al sottospazio di propria competenza. Le richieste di collocazione e localizzazione di una determinata risorsa vengono indirizzate verso il *peer* il cui identificatore è più vicino all'identificatore assegnato alla risorsa.
Inoltre sviluppi futuri potrebbero riguardare la costruzione dei file RDF utilizzati per annotare le risorse in modo automatico [5], [6]. In questo modo un utente dovrebbe caricare direttamente la risorsa da condividere e in modo automatico si dovrebbero ottenere le informazioni riguardanti la risorsa, cioè i valori assunti dai metadati. Comunque per alcuni metadati dovrebbe essere l'utente ha inserire i valori, in particolare per il titolo e per l'autore del documento, mentre per gli altri metadati (soggetto, periodo d'arte, artista, lingua, tipo) i valori potrebbero essere estratti effettuando un'analisi semantica del documento.

BIBLIOGRAFIA

[1] Milojicic, Kalogeraki, Lukose, Nagaraja, Pruyne, Richard, Rollins, Xu, *Peer-to-Peer computing*, HP Laboratories Palo Alto, Marzo 2002.

[2] Progetto Edutella: http://www.edutella.jxta.org/.

[3] Nejdl, Wolf, Qu, Decker, Stintek, Naeve, Nilsson, Palmer, Risch, *Edutella: A P2P Networking Infrastructure Based on RDF*, Proceedings of the 11th International World Wide Web Conference (WWW 2002), Hawaii, USA, Maggio 2002.

[4] Nejdl, Wolf, Staab, Tane, *EDUTELLA: Searching and Annotating Resources within an RDFbased P2P Network,* Technical Report , December 2001.

[5] Klyne, Carroll, *Resource Description Framework (RDF): Concepts and Abstract Syntax*, W3C Working Draft 08 November 2002, http://www.w3.org/TR/2002/WD-rdf-concepts-20021108/.

[6] Brickley, Guha, *RDF Vocabulary Description Language 1.0: RDF Schema*, W3C Working Draft 12 Novembre 2002, http://www.w3.org/TR/2002/WD-rdf-schema-20021112/

[7] Nilsson, Siberski, *RDF Query Exchange Language (QEL) concepts, semantics and RDF syntax,* http://edutella.jxta.org/spec/qel.html.

[8] Progetto JXTA: http://www.jxta.org/.

[9] Li Gong, *Project JXTA: A Technology Overview*, Technical report, SUN Microsystems, April 2001, http://www.jxta.org/project/www/docs/TechOverview.pdf.

[10] Traversat, Abdelaziz, Pouyoul, *Project JXTA: A Loosely-Consistent DHT Rendezvous Walker*, Sun Microsystems white paper.

[11] Traversat, Arora, Abdelaziz,Duigou, Haywood, Hugly, Pouyoul, Yeager, *Project JXTA 2.0 Super-Peer Virtual Network*, Sun Microsystems white paper.

[12] *JXTA v2.0 Protocols Specification*, http://spec.jxta.org.

[13] *Project JXTA v2.0: Java™ Programmer's Guide*, Sun Microsystems, Maggio 2003.

[14] *JXTA Platform Configuration Instructions,* http://platform.jxta.org/java/confighelp.html.

[15] Applicazioni sviluppate utilizzando la tecnologia Edutella: http://www.jxta.org/edutella/download.

[16] Dublin Core Metadata Initiative, *Dublin Core Metadat a Element Set, Version 1.1*, http://dublincore.org/usage/terms/dc/current-elements/, October 2002.

[17] Nejdl, Wolpers, Siberski, Schmitz, Schlosser, Brunkhorst, Löser, *Super-Peer-Based Routing and Clustering Strategies for RDF-Based Peer-To-Peer Networks*.

[18] Yang, Garcia-Molina, *Designing a super-peer network*, http://dbpubs.stanford.edu:8090/pub/2002-13, 2002.

[19] Schlosser, Sintek, Decker, Nejdl, *HyperCuP—Hypercubes, Ontologies and Efficient Search on P2P Networks*. In International Workshop on Agents and Peer-to-Peer Computing, Bologna, Luglio 2002.

[20] Ratnasamy, Francis, Handley, Karp, Schenker, *A scalable content-addressable network*, Proceedings of the ACM SIGCOMM 2001, Agosto 2001.

[21] Stoica, Morris, Liben-Nowell, Karger, Kaashoek, Balakrishnan, *Chord: a scalable peer-to-peer lookup protocol for internet applications*, Proceedings of the ACM SIGCOMM 2001, Agosto 2001.