ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Corso di Laurea Specialistica in INGEGNERIA ELETTRONICA

Impiego di tecnologie semantiche in applicazioni di intelligenza ambientale

Candidato:
Emanuele Montemurro

Relatore:
Chiar.mo Prof. Ing. Tullio Salmon Cinotti
Docente di CALCOLATORI ELETTRONICI LS

Correlatori:
Dott. Luca Roffia
Ing. Alfredo D'Elia
Ing. Francesco Morandi

Anno Accademico 2011/2012 – Sessione I

Ai miei genitori

Indice generale

Introduzione....4
Panorama applicativo....7
1.1 Prefazione....7
1.2 Smart Environments....7
1.3 Sofia Project 8
1.4 Cos'è Smart-M3 ?....8
Semantic Web....10
2.1 Introduzione....10
2.2 Metadati....11
2.3 Architettura....12
2.4 Cenni di XML....14
Smart-M3....19
3.1 Introduzione....19
3.2 Resource Description Framework....19
3.3 RDF ++....22
3.4 Smart-M3 in dettaglio....23
3.4 Smart Space Access Protocol....26
Redland RDF, SPARQL e Python....28
4.1 Redland RDF Application Framework....28
4.2 SPARQL Query Language....31
4.3 Python....34
Primitiva SSAP SPARQL query....37
5.1 Introduzione....37
5.2 Formato SSAP dei messaggi di tipo SPARQL query....37
5.3 Formato XML dei risultati di una query SPARQL....40
5.4 Funzionamento SPARQL query SSAP....45
5.5 Interfaccia utente e test di funzionamento....52
Reasoning....64
6.1 Introduzione....64
6.2 Estensione dei prefissi....64
6.3 Modello RDF++ e sua implementazione....69
Conclusioni....79
Bibliografia....81

Introduzione

Il lavoro, svolto nel corso di questa tesi, si colloca nel campo di ricerca che va sotto il nome di “Intelligenza ambientale”. Questo è un campo che è in continua ascesa grazie al progressivo e regolare sviluppo della tecnologia, nell'ambito della miniaturizzazione dei dispositivi e delle potenzialità dei sistemi *embedded.* Questi sistemi permettono, tramite l'utilizzo sempre maggiore di sensori e attuatori, l'elaborazione delle informazioni provenienti dall'ambiente che ci circonda, al fine di creare ambienti “intelligenti”. Tutto ciò per far sì che l'ambiente esterno risponda automaticamente ai bisogni delle persone, anche senza la necessità di specifiche richieste.

Per poter condividere informazioni tra dispositivi ed entità software sono state implementate varie piattaforme, tra le quali diverse utilizzano web semantico per rendere l'informazione elaborabile dalle macchine. Nel corso di questa tesi, si è deciso di utilizzare il software Smart-M3, ereditato dal progetto europeo SOFIA, conclusosi nel 2011. Smart-M3 permette di creare una piattaforma d'interoperabilità indipendente dal tipo di dispositivi e dal loro dominio di utilizzo.

Il sistema Smart-M3 si compone di due parti:

– un nucleo centrale, che contiene l'informazione e costituisce la base della conoscenza;

– l'insieme dei dispositivi software, che possono inserire e recuperare informazione.

Smart-M3 prevede un metodo di memorizzazione dei dati basato sul modello RDF ed utilizza un protocollo di comunicazione, i cui messaggi sono scritti in formato RDF/XML. Nella sua nuova implementazione, utilizzata per il lavoro svolto, supporta il linguaggio SPARQL, che è un linguaggio di *query* RDF.

Il lavoro, svolto in questa tesi, è diviso in due fasi. La prima fase è consistita nel progettare delle funzioni che permettano ad un qualunque agente software sia di interrogare il nucleo centrale (tramite una *query* SPARQL) che di elaborare l'informazione, in modo da poter riutilizzare i risultati.

La seconda fase del lavoro è consistita nel progettare un meccanismo per relazionare la conoscenza di dominio con l'informazione memorizzata nel nucleo centrale, con lo scopo di aumentare l'informazione presente nella base di conoscenza. Questo meccanismo di *reasoning* è basato su un modello chiamato RDF++, introdotto da Ora Lassila, un ricercatore della Nokia.

Nel dettaglio, l'argomentazione della tesi è strutturata in questo modo:

- nel Capitolo 1 viene fornita una descrizione del *background* e del campo applicativo nel quale è stato sviluppato il lavoro;
- nel Capitolo 2 viene descritto cos'è il *Semantic Web* e qual è il suo scopo;
- nel Capitolo 3 viene introdotto e descritto approfonditamente il software Smart-M3;
- nel Capitolo 4 vengono forniti gli strumenti software utilizzati nel corso del lavoro;
- nel Capitolo 5 viene descritta la prima fase del lavoro, relativa alla primitiva di *query* SPARQL di Smart-M3;

- nel Capitolo 6 viene descritta la seconda fase riguardante il *reasoning*;
- infine ci sono le Conclusioni, in cui si fanno delle considerazioni sul lavoro svolto e in cui si parla di eventuali sviluppi futuri.

Capitolo 1

Panorama applicativo

1.1 Prefazione

I passi avanti nella miniaturizzazione dell'elettronica hanno permesso ai dispositivi di elaborazione ed alle interfacce di diventare parte della nostra vita quotidiana. Attualmente sensori, attuatori ed unità di elaborazione possono essere acquistati a prezzi accessibili. Questa tecnologia può essere collegata ed utilizzata, con la coordinazione di software altamente specializzato, per comprendere gli eventi ed il contesto relativo ad uno specifico ambiente e per prendere decisioni ragionevoli sia in *real time* che a posteriori. Tutto ciò porta a quella che può essere definita come "Intelligenza ambientale"[1], ovvero un ambiente digitale che, attivamente e ragionevolmente, supporta la gente nella sua vita quotidiana. In un prossimo futuro sarà prassi comune quella di muoversi all'interno di ambienti intelligenti (*smart environments*) in grado di soddisfare le esigenze degli individui contenuti in essi senza la necessità di esplicite richieste.

1.2 Smart Environments

Gli *Smart environments* rappresentano il nuovo step nell'automazione di

palazzi, case, servizi,industrie e sistemi di trasporto. Come ogni organismo senziente, lo *smart environment* fa affidamento principalmente sui dati sensoriali dal mondo esterno. I dati sensoriali provengono da sensori multipli di differenti tipologie distribuiti in varie locazioni: *wireless sensor netwoks* (WSN's) distribuite, ovvero reti di acquisizione e distribuzione dei dati, controllate e monitorate da un'unità centrale.

1.3 Sofia Project

SOFIA (*Smart Objects For Intelligent Applications*), come indicato nel sito ufficiale www.sofia-project.eu[2], è un progetto Europeo di ricerca che ha l'obiettivo di rendere le informazioni del mondo fisico disponibili per i diversi dispositivi *smart*, legando il mondo fisico col mondo dell'informazione.

Il progetto ha visto la creazione di una piattaforma di interoperabilità semantica, il cui fattore chiave risiede nei meccanismi comuni e aperti per la memorizzazione e la ricerca delle informazioni, estesi a tutti i sistemi *embedded*, indipendentemente dalla tecnologia implementata.

1.4 Cos'è Smart-M3 ?

Smart-M3 è il nome di un software *open-source* che mira a fornire un'infrastruttura condivisa di informazioni in web semantico tra entità software e dispositivi. Quindi, combina concetti come sistemi distribuiti ed in rete con il web semantico: il tutto per creare uno *smart environment* e collegare il mondo reale a quello virtuale.

Capitolo 2

Semantic Web

2.1 Introduzione

Il web è oggi il maggior contenitore di conoscenza, o, comunque, è quello maggiormente utilizzato dalla gran parte delle persone. Per gli esseri umani combinare informazioni provenienti da fonti diverse, e memorizzate in formati diversi (pagine web, database, fogli elettronici, etc.) per ottenere la risposta adeguata alle proprie esigenze è abbastanza semplice, anche se spesso noioso e ripetitivo. In realtà si vorrebbe che questo compito fosse svolto dalle macchine: si desidererebbe che potessero automaticamente combinare la conoscenza proveniente dalle diverse fonti, e, ancor meglio, derivarne di nuova. Per superare i limiti del web attuale, i ricercatori hanno lavorato intensamente fino alla realizzazione del cosiddetto *Semantic Web*[3], che può essere definito come un'infrastruttura basata su metadati per svolgere ragionamenti sul web. I metadati sono informazioni, elaborabili in modo automatico, relative alle risorse web, identificate univocamente da un URI (*Uniform Resource Identifier*).

Nel *Semantic Web* la conoscenza è rappresentata in maniera elaborabile dalla macchina, e può essere utilizzata da componenti automatizzati, denominati agenti software.

La tecnologia di riferimento per la codifica, lo scambio e il riutilizzo di

metadati è il *Resource Description Framework* (RDF), basato su un modello in cui gli elementi sono rappresentabili come triple. Per evitare che possano essere codificate delle triple sintatticamente corrette, ma prive di senso, è necessario un meccanismo per rappresentare classi di oggetti. Da questa esigenza nasce "RDF Vocabulary Description Language" o "RDF Schema" (RDFS). Per poter effettuare dei ragionamenti, per definire le classi, e per varie altre esigenze, però, RDFS non è sufficiente, e occorre un modo per rappresentare la conoscenza e le regole che permettono di dedurre ulteriore conoscenza: l'ontologia.

Poichè il web è intrinsecamente distribuito occorre un linguaggio che non solo permetta di esprimere dati e regole sui dati, ma che consenta anche di esportare questa conoscenza (ontologia) per renderla disponibile a qualunque applicazione. Il W3C ha definito, per questa esigenza, il *Web Ontology Language* (OWL).

2.2 Metadati

Nel navigare sul web, si susseguono degli URI che identificano quella che formalmente viene detta risorsa. Nel linguaggio comune una risorsa viene anche detta "documento", per mettere in evidenza il fatto che sia leggibile da un essere umano, o "oggetto", per mettere in evidenza che sia leggibile da una macchina. Qualunque sia il termine utilizzato, la risorsa non è una entità a sé stante, ma è accompagnata da informazioni che la descrivono. Le informazioni sulla risorsa vengono generalmente dette metadati. Qundi i metadati sono informazioni su una risorsa web, comprensibili dalla macchina. Di conseguenza, i metadati costituiscono un tipo di

informazione che può essere utilizzata dagli agenti software, per fare un uso appropriato delle risorse, rendendo più veloce il funzionamento del web. Inoltre va sottolineato il fatto che i metadati sono dati, e questo ha alcune conseguenze. La prima è che i metadati possono essere memorizzati come dati, in una risorsa, che può quindi contenere informazioni relative a se stessa o ad un'altra risorsa. La seconda conseguenza del fatto che i metadati sono dati, è che i metadati possono essere descritti da altri metadati.

2.3 Architettura

Il web è uno spazio informativo universale, navigabile tramite gli *Uniform Resource Identifier* (URI), che puntano alle risorse. Nel contesto del *Semantic Web*, il termine semantico assume la valenza di elaborabile dalla macchina e non intende fare riferimento alla semantica del linguaggio naturale e alle tecniche di intelligenza artificiale. Il *Semantic Web* è un ambiente dichiarativo, in cui si specifica il significato dei dati, e non il modo in cui si intende utilizzarli. La semantica dei dati consiste nelle informazioni utili perché la macchina possa utilizzarli nel modo corretto. Il web semantico ha un'architettura a livelli del tipo mostrato in figura 1.

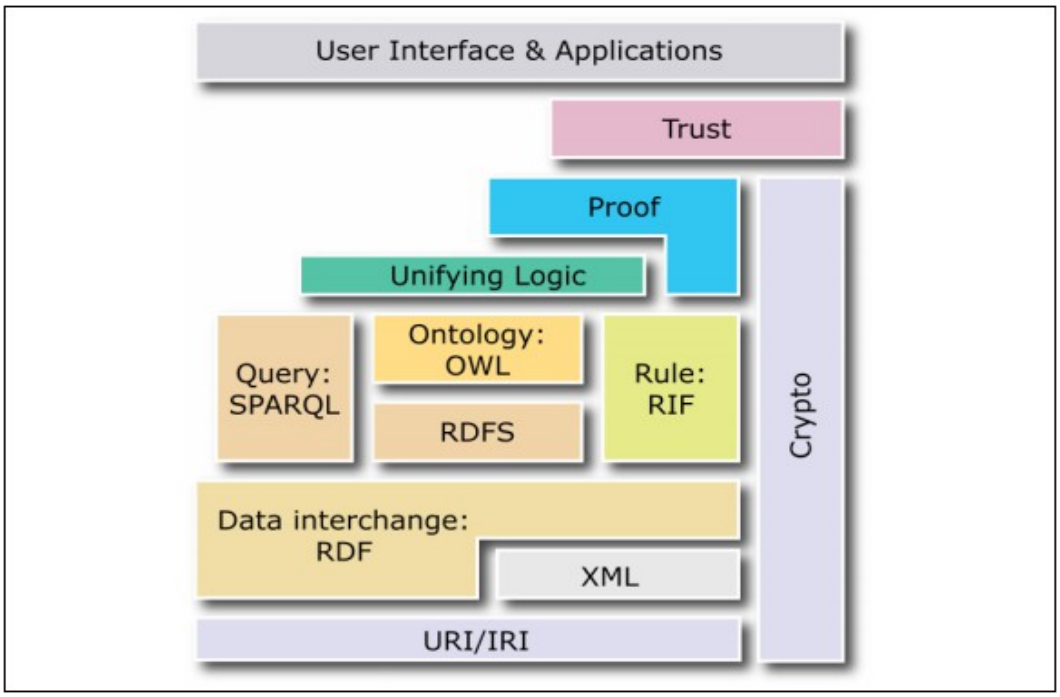

Illustrazione 1: Architettura del Semantic Web

Ad un livello superiore agli URI, di cui si è parlato in precedenza, si trova l'XML, che consente di dare ai documenti una struttura arbitraria. RDF, invece, può essere usato per esprimerne il significato, asserendo che alcuni particolari elementi hanno delle proprietà. Il livello ontologico definisce in modo formale le relazioni fra i termini. Un'ontologia permette di descrivere le relazioni tra i tipi di elementi senza però fornire informazioni su come utilizzare queste relazioni dal punto di vista computazionale. Il linguaggio definito dal W3C per definire ontologie strutturate per consentire una migliore integrazione dei dati tra applicazioni in settori diversi è OWL (*Ontology Web Language*). Il livello logico è il livello immediatamente superiore al livello ontologico. A questo livello le asserzioni esistenti sul web possono essere utilizzate per derivare nuova conoscenza.

2.4 Cenni di XML

Nell'architettura del *Semantic Web* (come si può vedere dalla figura 1) il livello immediatamente superiore agli URI, che puntano alle varie risorse, è l'XML[4](*Extensible Markup Language*), ovvero un linguaggio a marcatori estensibile. Un linguaggio di *markup* è un insieme di regole che descrivono i meccanismi sintattici che consentono di definire e controllare il significato degli elementi contenuti in un documento o in un testo.

La tecnica di composizione di un testo con l'uso di marcatori richiede quindi una serie di convenzioni, ovvero appunto di un linguaggio a marcatori di documenti. I marcatori nel linguaggio XML sono detti *tag*.

XML si dice che è un linguaggio estensibile in quanto, a differenza di HTML, non prevede un set di *tag* fissati. Anche HTML è un linguaggio di *markup*, che permette di descrivere le modalità di impaginazione, formattazione o visualizzazione grafica del contenuto di una pagina web attraverso *tag* di formattazione. Invece, XML non ha come scopo principale quello di descrivere la formattazione e la visualizzazione di un documento sul web, ma semplicemente quello di descrivere la struttura dell'informazione.

Infatti XML è uno strumento per definire i *tag* e le relazioni strutturali tra di loro. Poichè non è presente nessun set di *tag* predefinito, non può esserci nessuna semantica preconcettuale. Non c'è alcun particolare significato intrinseco associato ai *tag*: è un'eventuale applicazione che li interpreta come meglio crede.

Per quanto riguarda la sintattica di un documento XML, i caratteri che formano un documento (che per definizione è una stringa di caratteri) si

dividono in *tag* e contenuti. *Tag* e contenuti possono essere distinti attraverso l'applicazione di semplici regole: tutte le stringhe che costituiscono un *tag* iniziano col carattere '<' e terminano col carattere '>'; le altre costituiscono i contenuti.

La struttura di un tipico documento XML è basata su un modello gerarchico. Esso è composto da componenti denominati elementi (ovvero i *tag*). Ciascun elemento rappresenta un componente logico del documento e può contenere altri elementi o del testo. Gli elementi possono avere associate altre informazioni che ne descrivono le proprietà. Queste informazioni sono chiamate attributi.

L'organizzazione degli elementi segue un ordine gerarchico o arboreo che prevede un elemento principale, chiamato radice, il quale contiene l'insieme degli altri elementi del documento.

La struttura logica di un documento XML dipende dalle scelte progettuali. E' il progettista a decidere come organizzare gli elementi all'interno di un documento XML, poichè non esistono regole universali per l'organizzazione logica di un documento.

Qui di seguito è riportato un esempio di codice scritto in XML riguardante la struttura di un generico articolo.

```
<?xml version="1.0" ?>
<articolo titolo="Titolo dell'articolo">
    <paragrafo titolo="Titolo del primo paragrafo">
        <testo>
          Blocco di testo del primo paragrafo
        </testo>
        <immagine file="immagine1.jpg">
        </immagine>
```

```
    </paragrafo>
    <paragrafo titolo="Titolo del secondo paragrafo">
      <testo>
        Blocco di testo del secondo paragrafo
      </testo>
      <codice>
        Esempio di codice
      </codice>
      <testo>
        Altro blocco di testo
      </testo>
    </paragrafo>
    <paragrafo tipo="bibliografia">
      <testo>
        Riferimento ad un articolo
      </testo>
    </paragrafo>
</articolo>
```

La prima riga del documento lo identifica come un documento XML e ne specifica la versione. La radice corrisponde al *tag* articolo e contiene una lista di elementi che rappresentano i vari paragrafi dell'articolo. Ciascun paragrafo a sua volta contiene del testo, degli esempi di codice e delle immagini. La maggior parte degli elementi di questo albero possiede degli attributi: titolo, tipo, file.

Infine, bisogna sottolineare un concetto importante in XML: i *namespace*. Questi vengono definiti per evitare possibili collisioni che possono avvenire quando si prova a mescolare documenti XML, provenienti da applicazioni differenti. Ad esempio, si possono avere due frammenti di documenti XML, che hanno uno stesso nome assegnato ad un *tag* (in

questo caso tabella), come mostrato di seguito:

1° Documento

```
<tabella>
  <tr>
     <td>mele</td>
     <td>banane</td>
  </tr>
</tabella>
```

2° Documento

```
<tabella>
  <nome>tavolo</nome>
  <larghezza>80</larghezza>
  <lunghezza>120</lunghezza>
</tabella>
```

Se questi frammenti XML vengono mescolati in un unico documento, si avrebbe un conflitto tra gli elementi tabella: infatti hanno un diverso significato. Un *parser* (ovvero l'interprete della sintassi XML) non saprebbe come gestire questa situazione. Ecco perchè sono stati introdotti dei prefissi per distinguere tra elementi, provenienti da diversi vocabolari XML. Questi prefissi sono detti *namespace*. Il *namespace* può essere definito esplicitamente usando l'attributo *xmlns* nel *tag* associato all'elemento. Ad esempio:

```
<h:tabella xmlns:h="http://www.w3.org/TR/html4/" >
  <h:tr>
     <h:td>mele</h:td>
```

```
        <h:td>banane</h:td>
      </h:tr>
    </h:tabella>
```

Capitolo 3

Smart-M3

3.1 Introduzione

L'idea che sta alla base di Smart-M3 è quella di condividere le informazioni tramite un *Semantic Information Broker*[5](SIB) garantendo l'indipendenza del sistema dai dispositivi, dai domini di utilizzo e dai fornitori. Il suffisso *M3* di Smart-M3 deriva infatti dall'inglese "Multi vendor, Multi device, Multi domain".
Smart-M3 è una soluzione *open-source* che permette a dispositivi come smartphone, televisori o laptop di interagire tra loro al fine trarre vantaggio dalle opportunità che gli ambienti di utilizzo offrono.
La tecnologia software Smart-M3 è un'evoluzione del *Semantic Web* con alcune proprietà speciali.

3.2 Resource Description Framework

Indispensabile per proseguire nella descrizione di Smart-M3 è l'introduzione del concetto di *Resource Description Framework*.
Uno degli obiettivi del web semantico è quello di rendere le risorse comprensibili e utilizzabili da agenti software.
Un ruolo molto importante in questo panorama è ricoperto dai metadati,

ovvero le informazioni aggiunte ai dati, per renderli più facilmente utilizzabili. Definendo una sintassi ed una struttura per rappresentare i metadati si ha a disposizione un modo per descrivere le informazioni, relative ad una risorsa sul web, che possono essere comprese da una macchina.

Mentre il linguaggio XML permette di inserire metadati all'interno dei documenti attraverso i *tag*, il linguaggio RDF (*Resource Description Framework*) è lo strumento proposto dal W3C per descrivere i metadati relativi ad una risorsa.

RDF consente l'interoperabilità tra applicazioni che si scambiano sul web informazioni comprensibili dalla macchina. RDF, quindi, non descrive la semantica, ma fornisce una base comune per poterla esprimere, permettendo di definire la semantica dei *tag* XML.

RDF è costituito da due componenti: "RDF Model and Syntax" e "RDF Schema".

La prima componente riguarda la definizione del *data model* RDF (modello dei dati), tramite il quale descrivere le risorse, e la sintassi XML utilizzata per specificare questo modello.

RDF Schema invece permette di definire il significato e le caratteristiche delle proprietà e delle relazioni che esistono tra queste e le risorse descritte nel *data model* RDF.

Una risorsa, identificata univocamente da un URI, viene descritta utilizzando il *data model* RDF.

Questo modello è basato su tre oggetti:

- *Resource* (risorsa): indica ciò che viene descritto mediante RDF e può essere una risorsa web (ad esempio una pagina HTML, un documento XML o parti di esso) o anche una risorsa esterna al web;

- *Property* (proprietà): indica una proprietà, un attributo o una relazione utilizzata per descrivere una risorsa. Il significato e le caratteristiche di questa componente vengono definite tramite RDF Schema;
- *Statement* (tripla): è l'elemento che descrive la risorsa ed è costituito da un soggetto (che rappresenta la risorsa), un predicato (che esprime la proprietà) e da un oggetto (chiamato *Value*) che indica il valore della proprietà. . L' oggetto di uno *statement* può essere un'espressione (sequenza di caratteri o qualche altro tipo primitivo definito da XML) oppure un' altra risorsa.

Un modo per esprimere lo *statement* RDF in forma grafica è quello di rappresentarlo come un grafo etichettato orientato nel quale la risorsa è rappresentata da un'ellisse, la proprietà da un arco orientato, che parte dalla risorsa e punta all'oggetto, e l'oggetto da un rettangolo (come mostrato in figura 2).

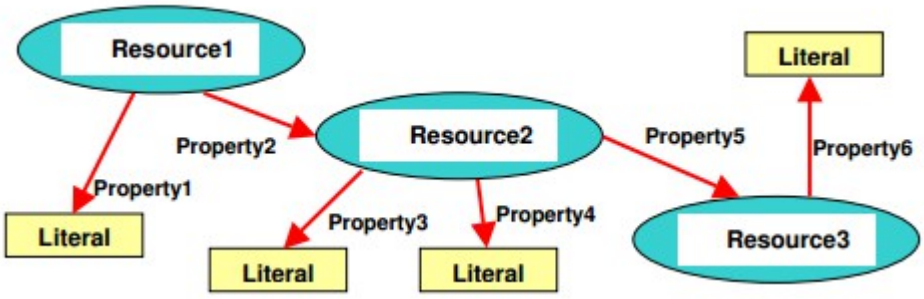

Illustrazione 2: Modello generico di grafo RDF

RDF *data model* non fornisce nessun meccanismo per dichiarare le proprietà, né per definire le relazioni tra queste ed altre risorse.

RDF Schema permette di definire significato, caratteristiche e relazioni di un insieme di proprietà, compresi eventuali vincoli sul dominio e sui valori delle singole proprietà.

Inoltre, implementando il concetto (transitivo) di classe e sottoclasse, consente di definire gerarchie di classi, con il conseguente vantaggio che agenti software intelligenti possono utilizzare queste relazioni per svolgere i loro compiti.

3.3 RDF ++

Come accennato alla fine del precedente paragrafo, il *framework* RDF mette a disposizione un buon numero di strumenti per definire in maniera più precisa le proprietà e le relazioni tra le varie asserzioni, che costituiscono il modello RDF. Nel dettaglio, nel modello RDF, oltre ad essere presenti un certo numero di asserzioni che costituiscono l'informazione posseduta (che fanno parte della cosiddetta *ABox*, dove la A sta per asserzione), sono presenti anche un certo numero di triple assiomatiche che permettono di descrivere le relazioni e le dipendenze tra le risorse indicate sotto forma di asserzioni (quest'ultima parte viene detta anche terminologia o *TBox*[6]).

Molti pacchetti software, per il processamento di un modello RDF, trattano i grafi RDF semplicemente come strutture di dati e lasciano la parte d'inferenza al programmatore dell'applicazione. E' questo quello che accade anche con le librerie di Redland (che è lo *storage* utilizzato come modello RDF), che verranno approfondite in seguito. Lo scopo dell'inferenza, anche detta *reasoning*, è quello di aumentare la conoscenza delle asserzioni presenti nel modello RDF. Aumentare la conoscenza significa inserire delle nuove asserzioni al modello RDF, dedotte dalle triple già presenti nella base di conoscenza rappresentata dalla SIB (*ABox*)

in base alle proprietà e alle relazioni descritte dalla *TBox*.

Nella *TBox* sono definiti concetti come classe, proprietà, dominio e range. La classe consiste in un insieme di istanze che sono correlate poiché condividono certe proprietà. Le classi possono essere organizzate in gerarchie (tramite *rdfs:SubClassOf*). Le proprietà consistono in una serie di attributi che descrivono una risorsa e anch'esse possiedono una gerarchia (tramite *rdfs:SubPropertyOf*). Le proprietà hanno un dominio (*rdfs:domain*) ed un range (*rdfs:range*), i quali servono, rispettivamente, a limitare ed identificare le istanze cui la proprietà può essere applicata (il primo) ed a limitare ed identificare le istanze che la proprietà può assumere come valori (il secondo).

Il *reasoning* era supportato in implementazioni precedenti della SIB, che però non utilizzavano come *storage* RDF lo *storage* di Redland. Scopo di questa tesi è, anche, quello di implementare il *reasoning* allo *storage* Redland utilizzato.

Nel seguito (nel capitolo 6) verrà descritto un modello basato sui criteri dettati da un ricercatore della Nokia, Ora Lassila, che porterà alla definizione di un modello RDF chiamato RDF++. Naturalmente, si è ancora ad un livello inferiore rispetto al livello ontologico, in cui si descrivono concetti più astratti come cardinalità, transitività delle proprietà, etc.

3.4 Smart-M3 in dettaglio

Il grande vantaggio di Smart-M3 è che permette di creare un sistema di

pubblicazione e fruizione delle informazioni comprensibile a tutti i dispositivi software che ne vogliano fare uso. La distinzione tra produttori e consumatori di informazioni non è più vincolante, i contenuti pubblicati sono a disposizione di tutti i consumatori in maniera trasparente ai produttori. Con queste premesse i dati possono essere manipolati in qualsiasi momento, in qualsiasi modo e in qualsiasi contesto da dispositivi di qualunque marca e sviluppati in qualsiasi tecnologia.

La piattaforma d'interoperabilità Smart-M3 è costituita da due componenti principali: il *Semantic Information Broker* (SIB), che è il nucleo del sistema ospitato da un dispositivo fisico, e il *Knowledge Processor* (KP), che può essere un qualsiasi dispositivo in grado di ospitare i software concepiti per inserire o recuperare informazione attraverso la SIB. Uno *smart space* è definito come uno spazio d'informazione con un proprio nome, dove l'informazione è memorizzata in una o più SIB. L'informazione nello *space* è l'unione delle informazioni contenute in tutte le SIB partecipanti, connesse tramite un protocollo. Di conseguenza, ogni KP vede la stessa informazione indipendentemente dalla SIB a cui è connesso. L'informazione nello *smart space* è memorizzata come un grafo RDF, di solito secondo qualche ontologia definita. La struttura logica di Smart-M3 è mostrata in figura 3.

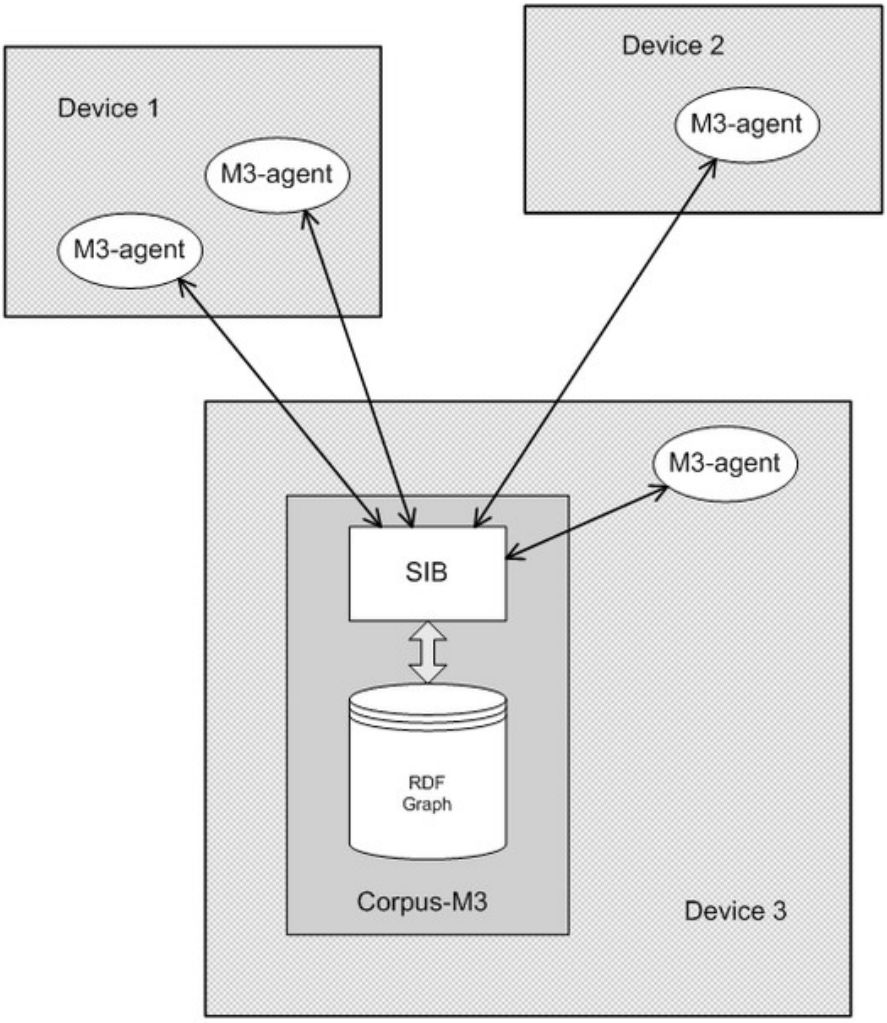

Illustrazione 3: Struttura di Smart-M3

La comunicazione tra KP e SIB può essere implementata usando diversi tipi di protocolli di trasporto (TCP/IP, HTTP, Bluetooth e NoTA) e un buon numero di linguaggi di programmazione (C, C#, Java, Python, etc.). Nel caso analizzato, i KP comunicano con la SIB tramite il protocollo TCP/IP attraverso il canale D-BUS del sistema Linux.

3.4 Smart Space Access Protocol

Lo *Smart Space Access Protocol*[7](SSAP) è il protocollo che i KP usano per accedere ad una SIB. Prevede sette operazioni, le quali sono definite da un insieme di parametri e sono codificate in modi diversi, per esempio in XML o JSON. Il protocollo è *session-based*, assumendo che il KP che vuole accedere allo *smart space* dovrà per prima cosa effettuare un'operazione di *join*. Il KP fornirà le sue credenziali nel messaggio di *join*, e la SIB, ricevendolo, deciderà se il KP potrà accedere allo *smart space*. Dopo la *join*, il KP potrà eseguire le altre operazioni.

Opposta all'operazione di *join*, è l'operazione di *leave*, che permette di abbandonare lo spazio M3. Naturalmente, non sarà possibile eseguire alcun'altra operazione dopo quella di *leave* fino ad una nuova richiesta di *join*.

Le altre operazioni previste dal protocollo SSAP sono elencate di seguito:

- *insert*: permette di inserire delle triple RDF all'interno del dispositivo di *storage* (immagazzinamento);
- *remove*: permette di cancellare delle triple RDF dallo *smart space*;
- *update*: permette di modificare delle triple RDF ed è eseguita come una combinazione di operazioni di *remove* e di *insert*;
- *query*: permette di recuperare informazioni dallo *smart space*, utilizzando un linguaggio di *query* supportato;
- *subscribe*: permette di settare una sottoscrizione (ovvero una *query* permanente) nella SIB; il KP è notificato quando la sottoscrizione risulta cambiata (ovvero quando è cambiato il valore di un dato di

cui si è richiesta la sottoscrizione);

- *unsubscribe*: permette di cancellare una sottoscrizione.

L'SSAP è il punto d'integrazione principale dell'architettura Smart-M3. Tutte le implementazioni delle SIB e dei KP devono supportare tutte le operazioni dell'SSAP: questo garantisce l'interoperabilità tra le differenti implementazioni.

Capitolo 4

Redland RDF, SPARQL e Python

4.1 Redland RDF Application Framework

L'implementazione della SIB, di cui si è fatto uso per il lavoro che verrà descritto in seguito, utilizza come RDF *store* uno *storage* Redland[8] e non uno *storage* Piglet come in implementazioni precedenti, perchè Redland supporta la sintassi delle *query* SPARQL ed inoltre è più veloce (negli inserimenti e rimozioni di triple) e stabile. L'importanza dello SPARQL è dovuto al fatto che è diventato il linguaggio di *query* standard delle W3C.

Redland, realizzato da un ricercatore dell'"Institute for Learning and Research Tecnology" di nome Dave Beckett, è un'implementazione flessibile ed efficiente dell'RDF che fornisce interfacce ad alto livello che permettono di memorizzare, interrogare e manipolare le istanze del modello in C, Perl, Python, Tcl e altri linguaggi.

I linguaggi target che utilizzano Redland sono il C, per uso in applicazioni che richiedono compilazione, e linguaggi usati in molte applicazioni web come Perl e Python. Ciò significa che il progetto deve essere appropriato per linguaggi ad oggetti (Python), non ad oggetti (C) e per quelli come Perl che possono appartenere ad entrambe le categorie precedenti.

Redland è una parte di un sistema a livelli in cui l'applicazione occupa i livelli più alti, che si interfacciano con dei livelli più bassi implementati in Redland. Più in dettaglio, è stato progettato per ricoprire approssimativamente i quattro livelli più bassi del diagramma mostrato in figura 4.

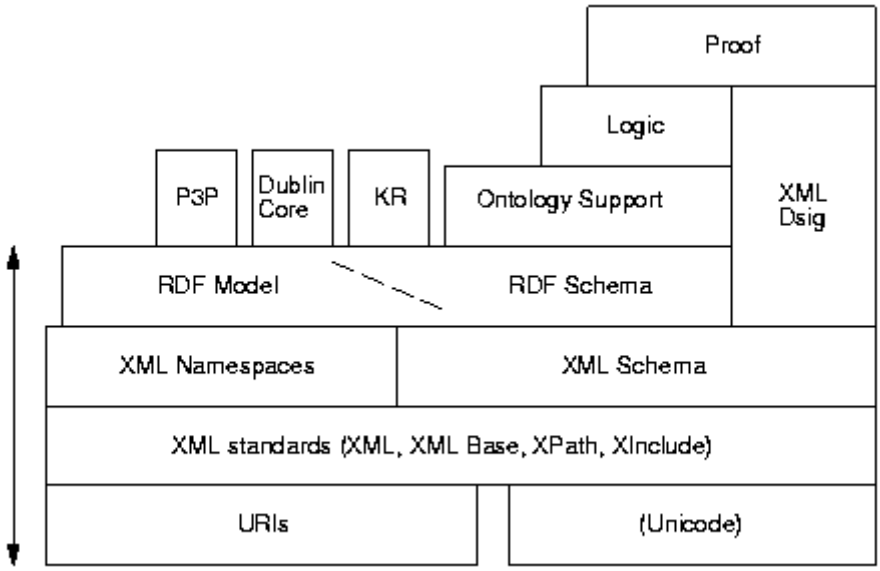

Illustrazione 4: Diagramma dei blocchi implementati in Redland

Redland, quindi, è un set di librerie software scritte in C che fornisce supporto al *Resource Description Framework*, tramite la definizione di un certo numero di classi. Le classi sono associate tra loro come mostrato in figura 5.

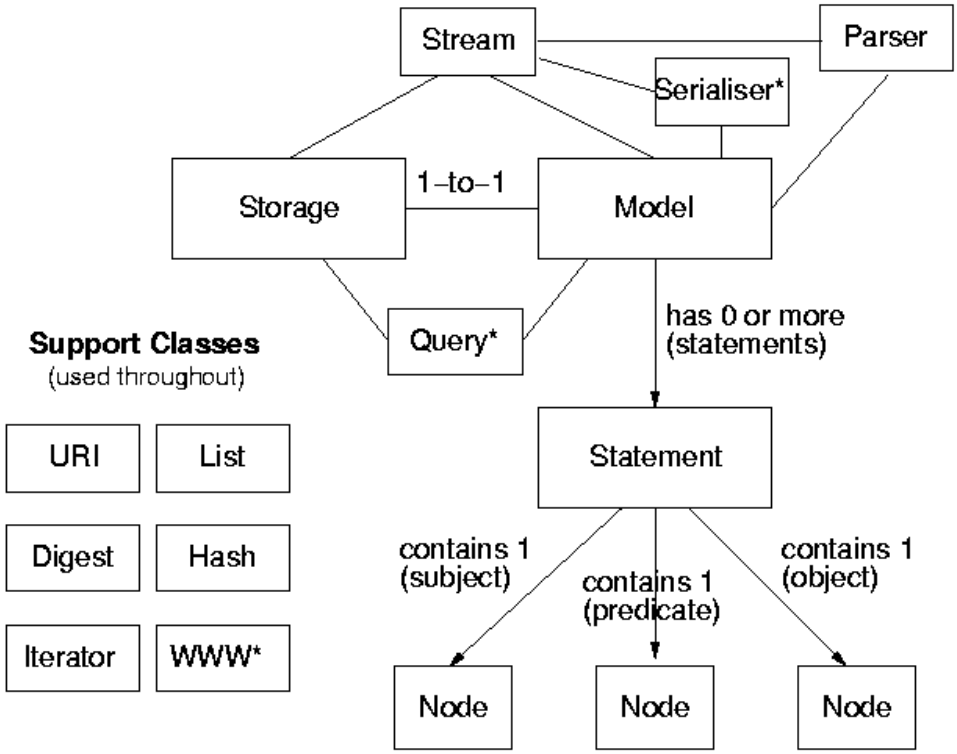

Illustrazione 5: Diagramma delle classi in Redland

La classe *Model* è un set di triple memorizzato di solito in uno *Storage*. In un modello molto semplificato, ogni istanza di tipo *Model* è mappata con *mapping* 1 a 1 ad un oggetto di tipo *Storage* che la rappresenta. Come detto, un'istanza di tipo *Model* può avere da 0 a molti *statement*, i quali a loro volta sono costituiti da istanze di tipo *Node* ovvero soggetto, predicato e oggetto.

La classe *Parser* fornisce un'interfaccia comune ai vari moduli per decodificare sintassi XML al fine di creare un modello RDF. Invece, la classe *Serializer* fa esattamente l'opposto, ovvero genera una sintassi a partire da un modello. La classe *Query* è una classe che supporta l'esecuzione di una *query* in ingresso (in forma di stringa) scritta con una sintassi adeguata. Infine la classe *Stream* fornisce una sequenza di triple a fronte di un'operazione di *parsering* o di *query*.

4.2 SPARQL Query Language

SPARQL[9](*Simple Protocol And RDF Query Language*) è un linguaggio di *query* RDF, ovvero un linguaggio di *query* per database, capace di recuperare e manipolare i dati memorizzati in formato RDF. E' stato reso uno standard dal *Data Access Working Group*, gruppo di lavoro del consorzio W3C.

SPARQL adotta la sintassi *Turtle*, un'estensione di *N-Triples*, alternativa estremamente sintetica e intuitiva al tradizionale RDF/XML.

Le *query* SPARQL si basano sul meccanismo del "pattern matching" e in particolare su un costrutto, il "triple pattern", che ricalca la configurazione a triple delle asserzioni RDF fornendo un modello flessibile per la ricerca di corrispondenze. Il *triple pattern* è come una tripla RDF, ma con l'opzione di una variabile al posto di uno dei termini RDF, ovvero soggetto, predicato e oggetto. Le variabili sono indicate con il costrutto “?” seguito dal nome della variabile oppure “$” seguito allo stesso modo dal nome.

Si possono distinguere i seguenti tipi di *graph pattern*:

– *Group graph pattern*, che sono il caso più generale dei *graph pattern*. Sono composti da:

- *graph pattern* di base;
- condizioni di *filter*;
- *graph pattern* opzionali;
- *graph pattern* alternativi.

– *Pattern* a partire da grafi nominati.

Un *graph pattern* di base (BGP) è un set di *triple pattern* scritto come una sequenza di essi (separati da un punto se necessario). Un BGP viene interpretato come la congiunzione dei suoi *triple pattern*. Un *group graph pattern* è un set di *graph pattern* delimitato da parentesi graffe.

La struttura di una generica *query* include, nell'ordine:

- la dichiarazione dei prefissi, al fine di abbreviare gli URI;
- un costrutto per indicare il tipo di *query*, ovvero per identificare quale informazione viene restituita;
- il *graph pattern*.

Ci sono varie forme di query in SPARQL, introdotte da diverse parole chiave (*keywords*):

- SELECT, il cui risultato della *query* è un set delle variabili che appaiono nella condizione e che naturalmente la verificano;
- CONSTRUCT, il cui risultato della *query* è un grafo RDF specificato da un *graph template*;
- ASK, che può essere usata per testare se un *graph pattern* ha o non ha soluzione; nessuna informazione si ha sulle soluzioni della *query*, ma solo se ne esiste almeno una;
- DESCRIBE, il cui risultato della *query* è un grafo RDF contenente dati riguardanti le risorse; questi sono dati dal processore che esegue la *query* SPARQL.

La clausola WHERE, infine, definisce il criterio di selezione specificando tra parentesi graffe uno o più *triple pattern*: il *matching* dei *triple pattern* con le triple del modello va a selezionare un sottografo del grafo RDF.

Per porre delle restrizioni sui valori da associare alle variabili, viene utilizzato il costrutto FILTER. Quindi, tramite questo costrutto vengono selezionate solo le soluzioni che verificano la condizione interna al FILTER. E' possibile che siano presenti più condizioni di FILTER in un *group graph pattern*: ciò è equivalente ad un unico FILTER con restrizioni congiunte.

A volte è utile avere *query* che permettono di aggiungere informazione alla risposta, ma non di eliminare la risposta se qualche parte del *query pattern* non è in *matching* con la soluzione. Questa capacità è fornita dal costrutto OPTIONAL: se la condizione opzionale non è verificata, non ritorna nessun risultato ma neanche elimina il resto delle soluzioni che erano in *match* con il *pattern* di base.

SPARQL fornisce un mezzo per formare dei *graph pattern* disgiunti, in modo da selezionare tutte le soluzioni che sono in match con almeno un *triple pattern*, indipendentemente dal fatto che verifichino o meno le altre condizioni. La *keyword* che permette ciò è la UNION.

Infine in SPARQL sono presenti altri costrutti, come quelli che permettono di manipolare i risultati come ad esempio:

- DISTINCT, che permette di escludere i valori duplicati;
- ORDER BY, che imposta l'ordine dei risultati della *query*;
- LIMIT, che pone restrizioni al numero dei risultati;
- OFFSET, che permette di saltare un certo numero di risultati.

4.3 Python

In questo paragrafo, è descritto brevemente il linguaggio di programmazione Python perchè i *Knowledge Processor* sono stati progettati con il suddetto linguaggio, nell'implementazione di Smart-M3 utilizzata per il lavoro presentato in questa tesi.

Python[10] è un linguaggio pseudocompilato: un interprete si occupa di analizzare il codice sorgente (semplici file testuali con estensione .py) e, se sintatticamente corretto, di eseguirlo. Quindi in Python, non esiste una fase di compilazione separata (come avviene in C, per esempio) che generi un file eseguibile partendo dal sorgente.

Python supporta diversi paradigmi di programmazione, come quello orientato ad oggetti (con supporto all'ereditarietà multipla) ed offre una tipizzazione dinamica forte. È fornito di una libreria *built-in* estremamente ricca, che unitamente alla gestione automatica della memoria (non esistono specifici costruttori e distruttori) e a robusti costrutti per la gestione delle eccezioni fa di Python uno dei linguaggi più ricchi e comodi da usare.

Per quanto riguarda le variabili, queste sono non tipizzate. Il controllo dei tipi è comunque forte e viene eseguito a *runtime* (*typing* dinamico). In altre parole una variabile è un contenitore al quale viene associata un'etichetta (il nome) che può essere associata a diversi contenitori anche di tipo diverso durante il suo tempo di vita. Essendo Python a tipizzazione dinamica, tutte le variabili sono in realtà puntatori ad oggetto. Gli oggetti sono invece dotati di tipo.

Uno dei grandi vantaggi di Python è soprattutto la sua comodità ma anche semplicità d'uso. Python è nato per essere un linguaggio immediatamente

intuibile. La sua sintassi è pulita e snella così come i suoi costrutti, decisamente chiari e non ambigui. I blocchi logici vengono costruiti semplicemente allineando le righe allo stesso modo (identazione), incrementando la leggibilità e l'uniformità del codice anche se vi lavorano diversi autori.

L'esser pseudointerpretato rende Python un linguaggio portabile. Una volta scritto un sorgente, esso può essere interpretato ed eseguito sulla gran parte delle piattaforme attualmente utilizzate, siano esse di casa Apple (Mac) che PC (Microsoft Windows e GNU/Linux). Semplicemente, basta la presenza della versione corretta dell'interprete.

Se paragonato ai linguaggi compilati a tipizzazione statica, come ad esempio il C, la velocità di esecuzione non è uno dei punti di forza di Python, specie nel calcolo matematico. Le performance di Python sono invece allineate o addirittura superiori ad altri linguaggi interpretati, quali PHP e Ruby, e in certe condizioni può rivaleggiare anche con Java. Non va inoltre dimenticato che Python permette di aggirare in modo facile l'ostacolo delle performance pure: è infatti relativamente semplice scrivere un'estensione in C o C++ e poi utilizzarla all'interno di Python, sfruttando così l'elevata velocità di un linguaggio compilato solo nelle parti in cui effettivamente serve e sfruttando invece la potenza e versatilità di Python per tutto il resto del software. Con opportune accortezze e utilizzando solo moduli standard, in alcuni casi può raggiungere una velocità di esecuzione pari ad un codice equivalente in C.

Infine, Python è un free software: non solo il download dell'interprete per la propria piattaforma, così come l'uso di Python nelle proprie applicazioni, è completamente gratuito; ma oltre a questo Python può essere liberamente

modificato e così ridistribuito, secondo le regole di una licenza pienamente *open-source*.

Capitolo 5

Primitiva SSAP SPARQL query

5.1 Introduzione

La prima parte del lavoro svolto, che verrà descritta nel corso di questo capitolo, è consistita nell'implementare la primitiva *query* di tipo SPARQL del protocollo SSAP, che il KP utilizza per interrogare la SIB.

La SIB risponde a questa operazione di *query* SPARQL, richiesta dal KP, con una stringa contenente il risultato in formato RDF/XML.

Perciò è stata realizzata una funzione che permette al KP di estrapolare i risultati contenuti nella stringa XML e memorizzarli in una struttura dati, che possa consentire un successivo riutilizzo di questi dati.

Adesso verranno analizzati in dettaglio i vari passi con cui si è sviluppato il lavoro, descritto brevemente, fin qui.

5.2 Formato SSAP dei messaggi di tipo SPARQL query

Ci sono due tipi di messaggi SSAP di *query* SPARQL: il messaggio di richiesta di esecuzione della *query* che il KP invia alla SIB ed il messaggio di risposta che contiene i risultati che la SIB invia al KP.

Tutti i messaggi delle primitive che costituiscono il protocollo SSAP sono scritte in XML, comprese le due suddette.

Per primo, di seguito viene mostrata la struttura di un tipico messaggio XML del protocollo SSAP di richiesta di *query* di tipo SPARQL, che il KP invia alla SIB.

```
<SSAP_message>
    <node_id>ID</node_id>
    <space_id>ID</space_id>
    <transaction_id>INTEGER</transaction_id>
    <transaction_type>QUERY</transaction_type>
    <message_type>REQUEST</message_type>
    <parameter name = "type">
        sparql
    </parameter>
    <parameter name = "query">
        QUERY_STRING
    </parameter>
</SSAP_message>
```

Come si può notare, la radice del messaggio, come per tutti i messaggi delle altre operazioni previste dal protocollo, è SSAP_message. Al suo interno contiene i seguenti tag:

- node_id, che è il campo che identifica il KP che richiede l'operazione;
- space_id, che è il campo che identifica lo *smart space*;
- transaction_id, che è il campo che contiene il numero associato all'attuale operazione del KP;
- transaction_type, che è il campo che identifica il tipo di operazione da effettuare (in questo caso una *query)*;

- message_type, che è il campo che indica se il messaggio è inviato dal KP alla SIB, quindi una richiesta (REQUEST), oppure se è una risposta della SIB ad un'operazione richiesta dal KP (CONFIRM).

Questi sono i *tag* presenti in tutti i messaggi SSAP per qualsiasi primitiva. Invece, quelli specifici per l'operazione di *query* sono:

- parameter name=”type”, che è il campo che indica se l'operazione di *query* è del tipo SPARQL oppure RDF-M3 (*query* in cui viene inviato un *pattern* di triple, che viene esplicitato nel messaggio XML). Queste sono gli unici due tipi di linguaggi di *query* supportati dalla SIB utilizzata;
- parameter name=”query”, che è il campo in cui viene inserita la stringa corrispondente alla *query* SPARQL.

Questa messaggio XML viene inviato, tramite protocollo TCP/IP, ed elaborato dalla SIB tramite le API di Redland, il quale poi restituisce il risultato sotto forma di stringa in formato RDF/XML.

Di seguito è mostrata la struttura di un tipico messaggio XML del protocollo SSAP di risposta ad una richiesta di *query*, che la SIB invia al KP.

```
<SSAP_message>
    <node_id>ID</node_id>
    <space_id>ID</space_id>
    <transaction_id>INTEGER</transaction_id>
    <transaction_type>QUERY</transaction_type>
    <message_type>CONFIRM</message_type>
    <parameter name = "status">
                STATUS_CODE
    </parameter>
    <parameter name = "results">
       ...
    </parameter>
 </SSAP_message>
```

Rispetto al messaggio di richiesta *query* analizzato precedentemente, il message_type è di tipo conferma ed inoltre sono presenti i seguenti due campi:

- parameter name=”status”, che indica lo stato della risposta, ovvero l'avvenuta o non avvenuta esecuzione della *query* da parte della SIB;
- parameter name=”results”, che è un *tag* al cui interno è presente la risposta RDF/XML, nel caso in cui la *query* è stata eseguita correttamente.

5.3 Formato XML dei risultati di una query SPARQL

Il risultato della *query* SPARQL ritornato dalla SIB, che si troverà all'interno del *tag* <parameter name=”results”> del messaggio di conferma SSAP, è in formato RDF/XML ed inizia nel modo seguente:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  ...
</sparql>
```

La prima riga della struttura identifica la stringa come un documento in formato XML e ne specifica la versione. Poi viene definito un elemento SPARQL in un *namespace* "http://www.w3.org/2005/sparql-results#" ed all'interno di questo elemento, la struttura dei risultati cambia in base al tipo di *query* (ovvero in base alla clausola presente nella stringa di *query,* che seleziona le variabili da mostrare nel risultato e da cui dipende il tipo

di risultato in uscita).

I casi sono tre:

- SELECT;
- ASK;
- CONSTRUCT.

La DESCRIBE non è implementata in Redland, quindi non è verrà analizzata.

Il primo caso da analizzare è una *query* del tipo SELECT. All'interno dell'elemento SPARQL descritto in precedenza, sono presenti altri due elementi: il *tag* head e il *tag* results.

Il *tag* head contiene una sequenza di elementi che costituiscono il set delle variabili, con i loro nomi, i cui valori sono le soluzioni della *query*, presenti nell'elemento results. Un esempio è il seguente:

```
<head>
  <variable name="a"/>
  <variable name="b"/>
  <variable name="c"/>
</head>
```

In alcune *query*, il *tag* head può anche contenere degli elementi link con attributo href, che contengono un URI che fornisce un collegamento a qualche metadato relativo ai risultati della *query*.

Il *tag* results, invece, contiene l'intera sequenza delle soluzioni della *query*: ogni risultato della *query*, a sua volta, è incapsulato in un *tag* result, figlio dell'elemento superiore results. Ogni result, inoltre, è costituito da altri elementi figli, i *tag* binding, che contengono il valore delle singole variabili che compongono una soluzione. Questa struttura gerarchica è

descritta in basso:

```
<results>
  <result>
     <binding name="a"> ... </binding>
     <binding name="b"> ... </binding>
     <binding name="c"> ... </binding>
   </result>
   <result>
     <binding name="a"> ... </binding>
     <binding name="b"> ... </binding>
     <binding name="c"> ... </binding>
   </result>
   ...
</results>
```

Il valore, associato ad una variabile, può essere:

- un URI;
- un *literal*, che è un tipo di dato che comprende sia le stringhe che i numeri;
- un *blank node*, ovvero un tipo di nodo di un grafo RDF che rappresenta una risorsa per la quale non è presente né un URI né un *literal*.

Il *literal*, a sua volta, può avere degli attributi: linguaggio (*language*) e tipo di dato (*datatype*), quest'ultimo definisce se il *literal* è un tipo numerico. Di seguito, è mostrato un esempio della struttura completa di un risultato in formato XML in risposta ad una richiesta di SELECT.

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="a"/>
    <variable name="b"/>
    <variable name="c"/>
    <variable name="d"/>
  </head>

  <results>
```

```
    <result>
        <binding name="a">
            <bnode>idnode</bnode>
        </binding>
        <binding name="b">
            <uri>http://esempio.org</uri>
        </binding>
        <binding name="c">
            <literal xml:lang=”it”>ciao</literal>
        </binding>
        <binding name="d">
            <literal xml:datattype=”http://www.w3.org/2001/XMLSchema#integer
”>32</literal>
        </binding>
    </result>
    ...
  </results>
</sparql>
```

Il secondo caso da analizzare è quello della risposta XML ad una *query* di tipo ASK. Come si è visto nel capitolo 4, la *query* di tipo ASK ritorna come risultato un valore booleano, che indica se il *pattern* presente nella *query* è presente o meno all'interno dello *storage* RDF. Quindi, per questo motivo all'interno dell'elemento head non è dichiarata nessuna variabile ed al posto del *tag* results è presente un *tag* boolean, in cui al suo interno è presente il valore booleano corrispondente all'avvenuto o non avvenuto *matching* del *pattern* della *query* con una delle triple presenti nel modello RDF. Un tipico esempio di una struttura XML di risposta ad una *query* di tipo ASK è il seguente:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
  </head>

   <boolean>
     true
  </boolean>
   ...
</sparql>
```

Infine rimane da analizzare il caso della CONSTRUCT. Questa tipo di *query*, a differenza delle due precedenti, non ritorna una lista di variabili con il loro valore corrispondente oppure un valore booleano, ma ritorna un nuovo grafo RDF. Di conseguenza, la struttura della risposta XML ad una *query* di tipo CONSTRUCT è completamente diversa. Tutte le risposte a delle *query* di tipo CONSTRUCT sono strutturate in questo modo:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="...">
  ...
  </rdf:Description>
  ...
</rdf:RDF>
```

La prima riga è uguale per tutte le *query* di questo tipo: serve per definire un *namespace* chiamato rdf. All'interno di questo *tag* sono presenti gli elementi che costituiscono il grafo RDF. I vari elementi del grafo sono definiti tramite il *tag* rdf:Description, il quale presenta l'attributo rdf:about, a cui viene associato l'URI corrispondente al soggetto della tripla. A sua volta questo *tag* è strutturato nel seguente modo:

```
<rdf:Description rdf:about="...">
  <ns0:nameURI xmlns:ns0=namespace> … </ns0:nameURI>
</rdf:Description>
```

Come si può vedere, all'interno dell'elemento rdf:Description è definito, come primo *tag* figlio, un *tag* in cui è presente l'URI corrispondente al predicato della tripla, con la definizione del suo *namespace*. A questo punto possono presentarsi due casi:

- se l'oggetto della tripla è un URI, allora questo viene descritto come un attributo di tipo rdf:resource del *tag*, come di seguito

  ```
  <rdf:Description rdf:about="...">
    <ns0:nameURI xmlns:ns0=namespace rdf:resource=URIobject>
  </ns0:nameURI>
  ```

```
</rdf:Description>
```

- se l'oggetto della tripla è invece un *literal*, allora viene descritto come il valore all'interno del *tag*, come si può vedere in basso:

```
<rdf:Description rdf:about="...">
  <ns0:nameURI xmlns:ns0=namespace> nameobject </ns0:nameURI>
</rdf:Description>
```

5.4 Funzionamento SPARQL query SSAP

Come detto in precedenza, lo scopo di questa fase del lavoro è stato quello di implementare la primitiva *query* SPARQL del protocollo SSAP. Questa è un'operazione che il KP richiede alla SIB, per recuperare informazione. Quindi la funzione che realizza l'operazione di *query* SPARQL è stata aggiunta alla libreria che racchiude tutte le operazioni che il KP esegue sulla SIB, il cui nome è *Node.py*. Come si può vedere dall'estensione del file, questa è una libreria completamente realizzata in linguaggio Python.

Per eseguire l'operazione di *query* SPARQL, il KP deve aver già effettuato in precedenza un accesso allo *smart space*, tramite un'operazione di *join*. Dato che nel KP ogni transazione viene registrata con un numero identificativo, la *join* sarà sempre identificata come la prima transazione.

Quindi, la funzione realizzata innanzitutto acquisisce in ingresso una stringa di *query* con sintassi SPARQL e va ad assegnare un identificativo alla transazione (la *query* naturalmente potrà avere un identificativo non inferiore a 2), in base allo stato in cui si trovava la variabile che contiene il numero dell'ultima transizione.

Viene fatto un controllo sulla stringa in ingresso: il motivo è che in XML

sono presenti cinque caratteri predefiniti, che hanno un significato speciale. Se in un documento XML è presente almeno uno di questi cinque caratteri, sarà generato un errore da parte del *parser* (ovvero l'interprete della sintassi XML). Questi caratteri sono detti *entity references* e sono mostrati nella tabella 1.

Carattere	Entità
&	&amp;
<	&lt;
>	&gt;
“	&quot;
'	&apos;

Tabella 1: Tabella Entity References

Quindi, il controllo eseguito sulla stringa di *query* consiste nel cercare questi caratteri e, nel caso in cui vengano trovati, vengono sostituiti con le entità corrispondenti, utilizzando una funzione di Python.

Una volta fatto ciò, si procede alla creazione del messaggio di richiesta di *query* SPARQL, secondo le regole del protocollo SSAP, descritte nel paragrafo 5.2. Questo è realizzato tramite una funzione interna che, una volta acquisiti in ingresso l'identificativo della transazione e la stringa di *query*, genera automaticamente la struttura XML relativa ad una richiesta di *query* SPARQL.

Fatto ciò, viene creata una *socket* (ovvero un particolare oggetto sul quale leggere e scrivere i dati da trasmettere o ricevere). La *socket* è il punto in cui un processo accede ad un canale di comunicazione per mezzo di una porta, ottenendo una comunicazione tra processi che lavorano su due macchine fisicamente separate (in questo caso il KP e la SIB). La creazione

e la successiva connessione alla *socket* vengono implementate tramite l'utilizzo di due primitive di Python. In seguito, viene inviato il messaggio SSAP, tramite la *socket*, alla SIB e una volta che il messaggio è arrivato interamente alla SIB, la *socket* in trasmissione viene disabilitata. Tutto ciò utilizzando le primitive *send* e *shutdown* di Python. A questo punto la *socket* del KP si mette in attesa del messaggio di risposta da parte della SIB: questo è realizzato attraverso l'utilizzo della primitiva *recv* di Python.

Una volta ricevuto il messaggio di risposta della SIB sotto forma di stringa in formato RDF/XML, è stato implementato un meccanismo per poter estrapolare i risultati dalla stringa XML e memorizzarli in una struttura dati, che ne permettesse successive manipolazioni. Il compito di estrarre i risultati è svolto da quello che è definito *parser*, ovvero l'interprete della sintassi XML.

Però, in base alla struttura della risposta RDF/XML della SIB, sono stati implementati due diversi meccanismi di *parsering*. Come è stato descritto nel paragrafo precedente, la struttura di una risposta ad una *query* SPARQL cambia in base al tipo di *query* richiesta. Dato che la struttura della risposta ad una *query* di tipo SELECT è completamente diversa da quella di una *query* di tipo CONSTRUCT, sono state implementate due funzioni diverse per effettuare il *parsering*, con successiva memorizzazione dei risultati in strutture dati differenti. Per quanto riguarda la risposta ad una *query* di tipo ASK, come si è visto, questa è molto simile alla struttura di una *query* SELECT e quindi il *parsering* relativo è stato realizzato all'interno della stessa funzione relativa alla SELECT.

Il *parser* utilizzato per interpretare la sintassi XML è il *parser* Minidom supportato da Python. In entrambe le funzioni implementate, la prima cosa

che viene fatta è convertire la stringa in un oggetto manipolabile da minidom, tramite la primitiva *parseString*, supportata nella libreria di questo *parser*.

Adesso viene focalizzata l'attenzione sul caso della risposta ad una *query* di tipo SELECT.

Prima di analizzare il procedimento con cui si riescono ad estrapolare i risultati, bisogna definire la struttura dati in cui memorizzarli. Si è deciso di utilizzare un vettore di vettori di vettori: più precisamente è un vettore, i cui elementi sono le soluzioni della *query*; ogni elemento a sua volta è un vettore di dimensione pari al numero di variabili presenti nella stringa di *query* SPARQL; ogni elemento di quest'ultimo vettore, a sua volta, è strutturato come un vettore di tre elementi, in cui il primo elemento è il nome della variabile corrispondente, il secondo è il tipo (URI,*literal*, *blank node* o *unbound*) ed il terzo è il valore.

Il metodo, utilizzato per estrapolare i risultati dal documento XML, consiste nella navigazione del grafo RDF, a partire dai nodi padri fino ad arrivare ai nodi figli. In figura 6 è mostrata in maniera schematica la tipica struttura di una risposta RDF/XML ad una *query* di tipo SELECT o ASK.

La primitiva di minidom che permette la navigazione del grafo, ovvero di passare dai nodi padri ai nodi figli, è la *getElementsByTagName*, la quale, in base al nome del *tag* che le viene dato in ingresso, restituisce una lista di elementi il cui *tag* corrisponde al *tag* in ingresso alla primitiva (quindi un sottoalbero).

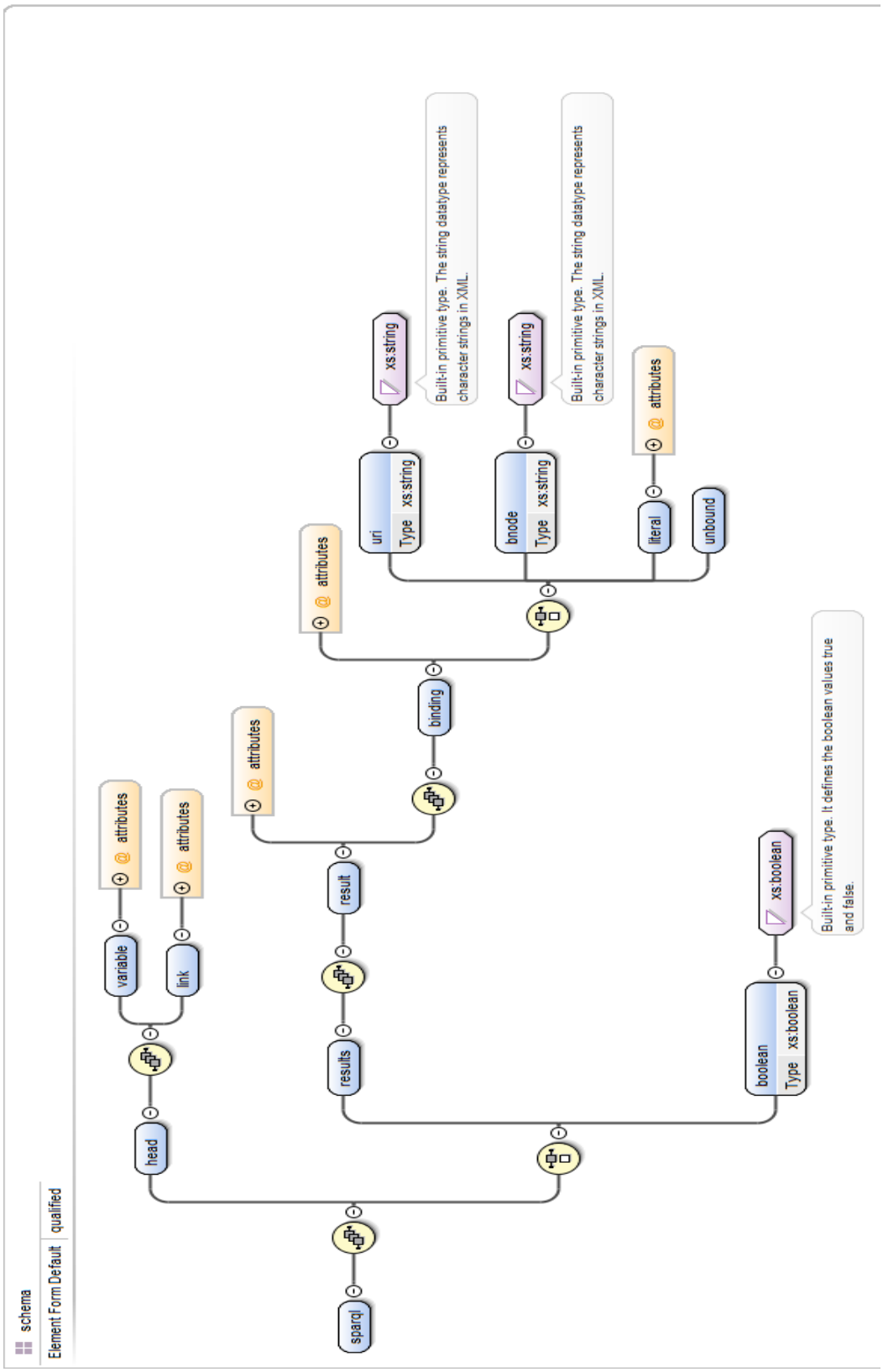

Illustrazione 6: Struttura XML dei risultati di una query SPARQL di tipo SELECT o ASK

Come si può vedere dalla figura, la risposta ad una *query* SPARQL di tipo SELECT o ASK può avere più di un blocco di *head*. Ad ogni blocco di *head* corrisponde un blocco di *results* o *boolean*. Per ognuno di questi blocchi di *results* o *boolean* bisogna estrapolare i risultati.

Quindi la prima cosa che viene fatta nella funzione realizzata è quella di creare una lista di elementi con *tag* uguale ad *head*: ciò serve per conoscere quanti blocchi di soluzioni sono presenti nella risposta RDF/XML, poichè per ogni *tag head* è presente il corrispondente *tag* dei *results* o dei *boolean*. Viene fatto un ciclo per ogni blocco di *head* presente. In ognuna di queste iterazioni, si va a controllare se il risultato è di tipo *results* o *boolean*.

Nel primo caso, si va a creare un sottografo di elementi con *tag* uguale a *result*. Si va a fare un ciclo di navigazione di questa lista e in ogni iterazione viene creato un ulteriore sottografo con *tag* uguale a *binding*. Si va a fare un ultimo ciclo di scorrimento di questo sottografo: in questo step, viene eseguita la vera e propria estrapolazione dei risultati con conseguente memorizzazione. Per ogni ciclo di iterazione, tramite la primitiva di minidom *getAttribute*, che in base al nome dell'attributo in ingresso restituisce il valore corrispondente, viene estratto il valore relativo all'attributo *name* del *tag binding*, che rappresenta il nome di una delle variabili, i cui valori costituiscono le soluzioni della *query*. Per l'estrazione del tipo si utilizza la primitiva *firstChild* (a differenza della *getElementsbyTagName* non restituisce una lista ma solo un elemento), che permette di accedere all'elemento figlio di *binding* e tramite la funzione *tagName*, di estrarre il nome del *tag*. Il tipo può essere di quattro tipi:

- URI;
- *literal*;

- *bnode*, ovvero *blank node*;
- *unbound*, ovvero variabile non vincolata al *graph pattern* della *query*.

Infine viene creato il vettore di tre elementi del tipo [nome variabile,tipo,valore]; ciò viene fatto per tutte le variabili che compongono ogni singolo risultato della *query*; tutte queste triple, relative ad una soluzione della *query*, vengono accorpate in un unico vettore e tutti questi vettori che compongono un singolo risultato vengono aggiunti al vettore generale dei risultati.

Nel secondo caso, ovvero nel caso di risultato di tipo *boolean (query* di tipo ASK) dato che il risultato è unico (*true* o *false*) e nel *tag head* non è presente nessuna variabile, viene memorizzato solamente il valore booleano nel vettore.

Per quanto riguarda il *parsering* della sintassi XML del documento in risposta ad una *query* SPARQL di tipo CONSTRUCT, anche qui l'estrazione delle soluzioni è effettuato tramite navigazione del grafo. In questo caso la struttura dati è un vettore, i cui elementi sono delle triple: ogni tripla costituisce un risultato della *query*.

Si parte dal *tag rdf:Description* per la navigazione del grafo. Come prima, tramite la primitiva *getElementsbyTagName* viene selezionata la lista degli elementi *rdf:Description* che costituiscono le soluzioni della *query*. Si va a fare un ciclo di iterazioni pari al numero dei *tag* uguali a *rdf:Description*. Per ogni iterazione, si va ad estrarre, tramite la primitiva *getAttribute*, il valore del soggetto della tripla contenuta nel *tag rdfDescription*. In seguito, tramite le primitive *firstChild* e *tagName*, si va ad estrarre il predicato della tripla. Per quanto riguarda l'oggetto della tripla, questo viene estratto o

attraverso l'attributo *rdf:resource* o, se non presente questo attributo, attraverso l'elemento interno al *tag*.

Dopo aver estrapolato i risultati di una risposta RDF/XML ad una *query* SPARQL, sono state realizzate due *utility*:

- la prima per estrarre tutti i risultati relativi ad una variabile specificata dall'utente. L'utente inserisce il vettore dei risultati della *query* e la variabile, per discriminare solo le soluzioni relative a quest'ultima, e la funzione restituisce solo i valori della variabile specificata per tutte le soluzioni della *query*. Ciò è stato realizzato banalmente tramite confronto tra la variabile inserita in ingresso e le variabili presenti nel vettore delle soluzioni. Una volta individuata la variabile cercata, la funzione fa un ciclo che seleziona solo la parte interessata tra tutti i risultati.
- La seconda per estrarre i nomi delle variabili presenti nei risultati. L'unico parametro in ingresso a questa funzione infatti è il vettore delle soluzioni. L'obiettivo è stato ottenuto semplicemente selezionando il primo elemento di ogni vettore di cui è formato un singolo risultato della *query*.

5.5 Interfaccia utente e test di funzionamento

In questo paragrafo verrà descritta l'interfaccia utente utilizzata per inserire la stringa di *query* SPARQL da parte dell'utente e per mostrare i risultati corrispondenti. Questa interfaccia è un aggiornamento di una precedente interfaccia, in cui le *query* utilizzate erano scritte con la sintassi delle *query*

di tipo *Wilbur*. Adesso ci si occupa di descrivere la struttura di questo strumento di lavoro.

La prima schermata dell'interfaccia che appare all'utente è quella mostrata in figura 7.

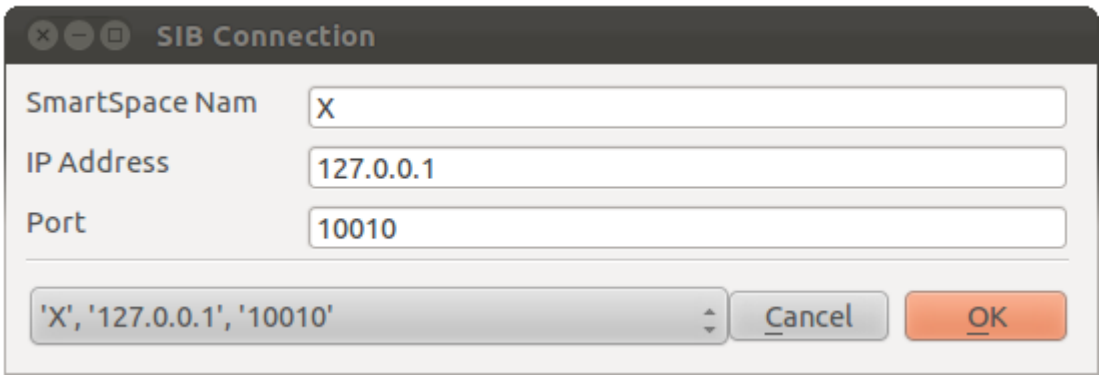

Illustrazione 7: Connessione alla SIB

In questa schermata, viene richiesto all'utente un nome per lo *smart space* e l'indirizzo IP e il numero di porta, necessari per creare una connessione TCP/IP tra KP e SIB. Se lo *smart space* viene individuato, la connessione viene accettata e viene automaticamente effettuata una *join* del KP alla SIB.

A questo punto appare la schermata dove è possibile inserire la stringa di *query* SPARQL, mostrata in figura 8.

La *query* SPARQL va inserita nella barra di testo della schermata: premendo il tasto OK, viene eseguita, ovvero viene richiamata la funzione di libreria, descritta precedentemente, passandole come parametro la stringa di *query*.

Per verificare il corretto funzionamento della funzione di libreria realizzata, verranno mostrati tre esempi di *query* SPARQL, rispettivamente di tipo SELECT,ASK e CONSTRUCT.

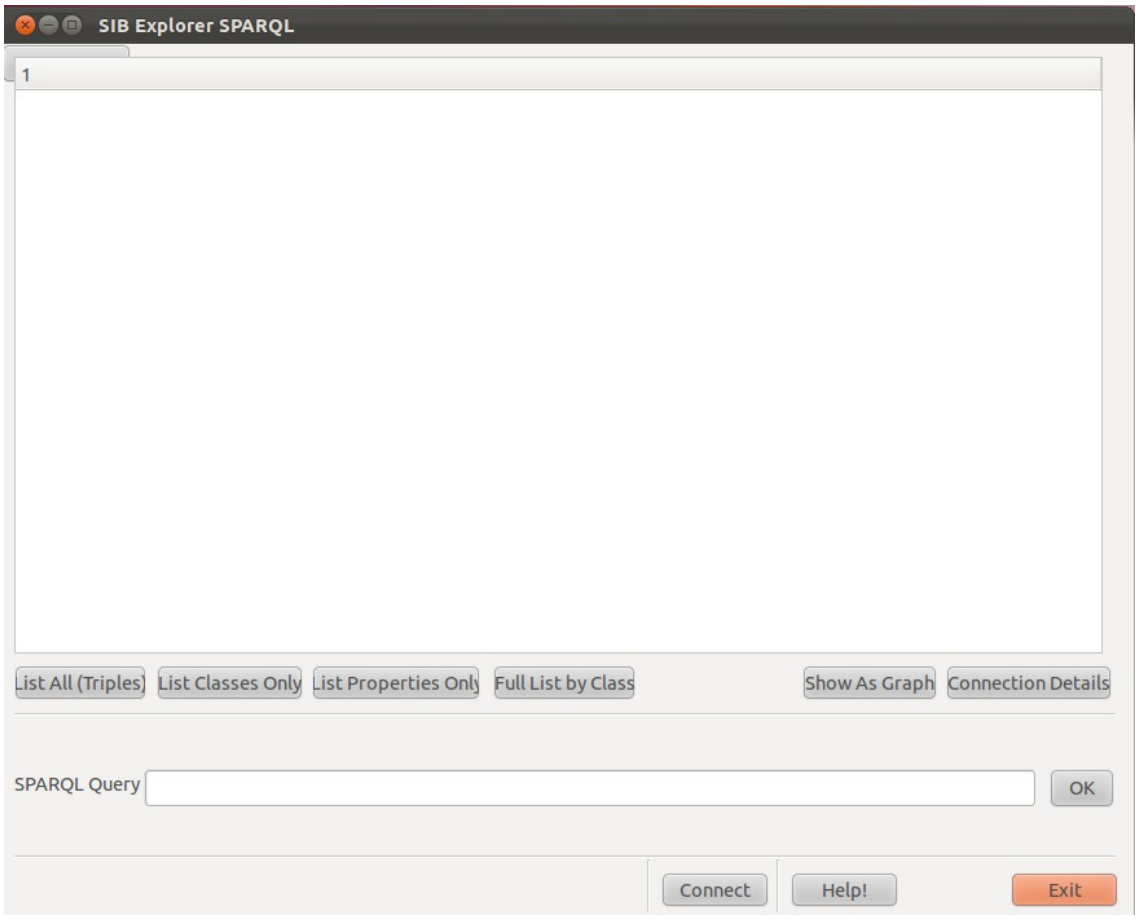

Illustrazione 8: Interfaccia SPARQL

Prima di poter far le *query*, bisogna aver inserito delle triple nello *storage* RDF della SIB. Le triple inserite sono mostrate in figura 9.

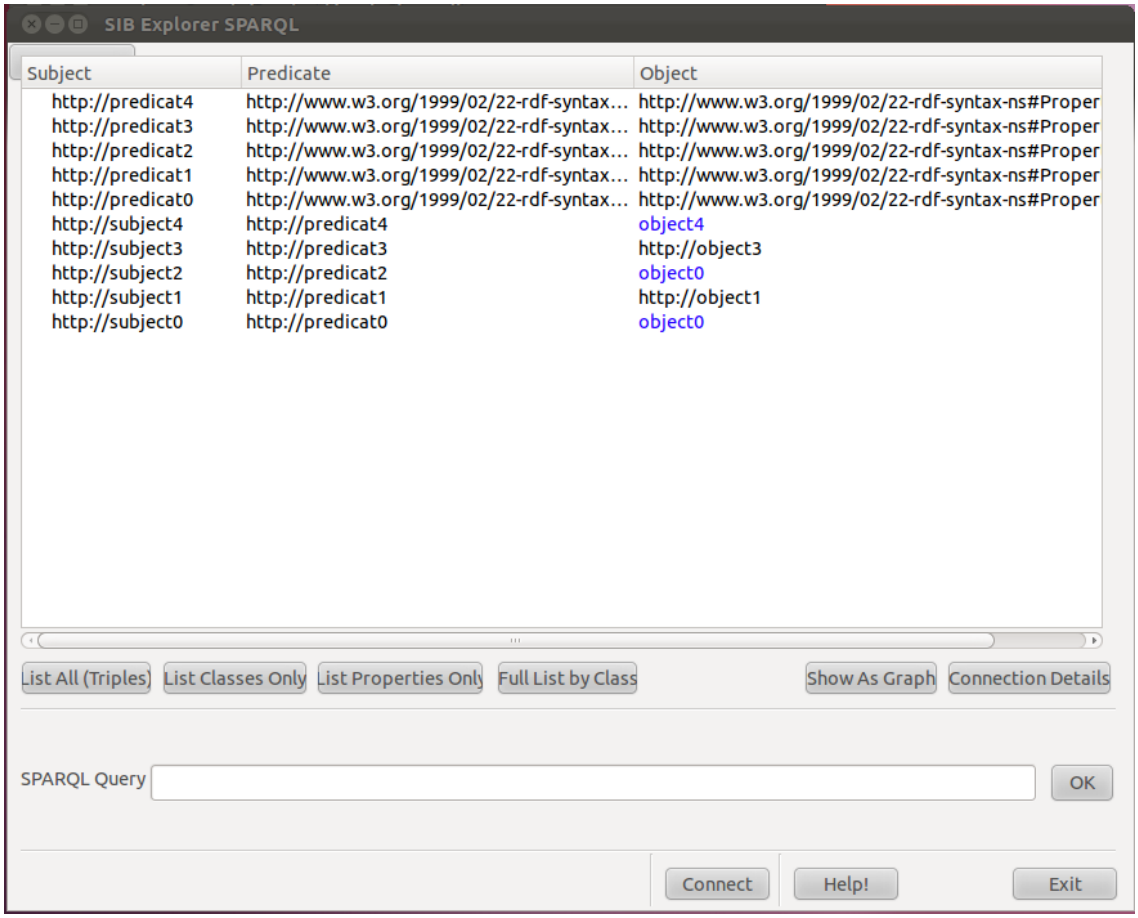

Illustrazione 9: Triple presenti nello storage RDF

Per incominciare è stato testato il funzionamento di una *query* di tipo SELECT. La *query* inserita per effettuare il test è la seguente:

SELECT ?a ?b ?c WHERE{{?a <http://predicat0> ?c FILTER regex(?c,"object0")} UNION {<http://subject1> ?b ?c}} ORDER BY ?c

Questa *query* è costituita da due *graph pattern* disgiunti: il primo ha come variabile il soggetto della tripla e fissa il predicato e, tramite la funzione FILTER regex (obbliga a selezionare le soluzioni che hanno la variabile c uguale al *literal* object0), l'oggetto; il secondo fissa il soggetto e ha come variabili predicato ed oggetto. Inoltre c'è la clausola ORDER BY, che ordina le triple della soluzione in ordine crescente.

Le soluzioni sono mostrate in figura 10.

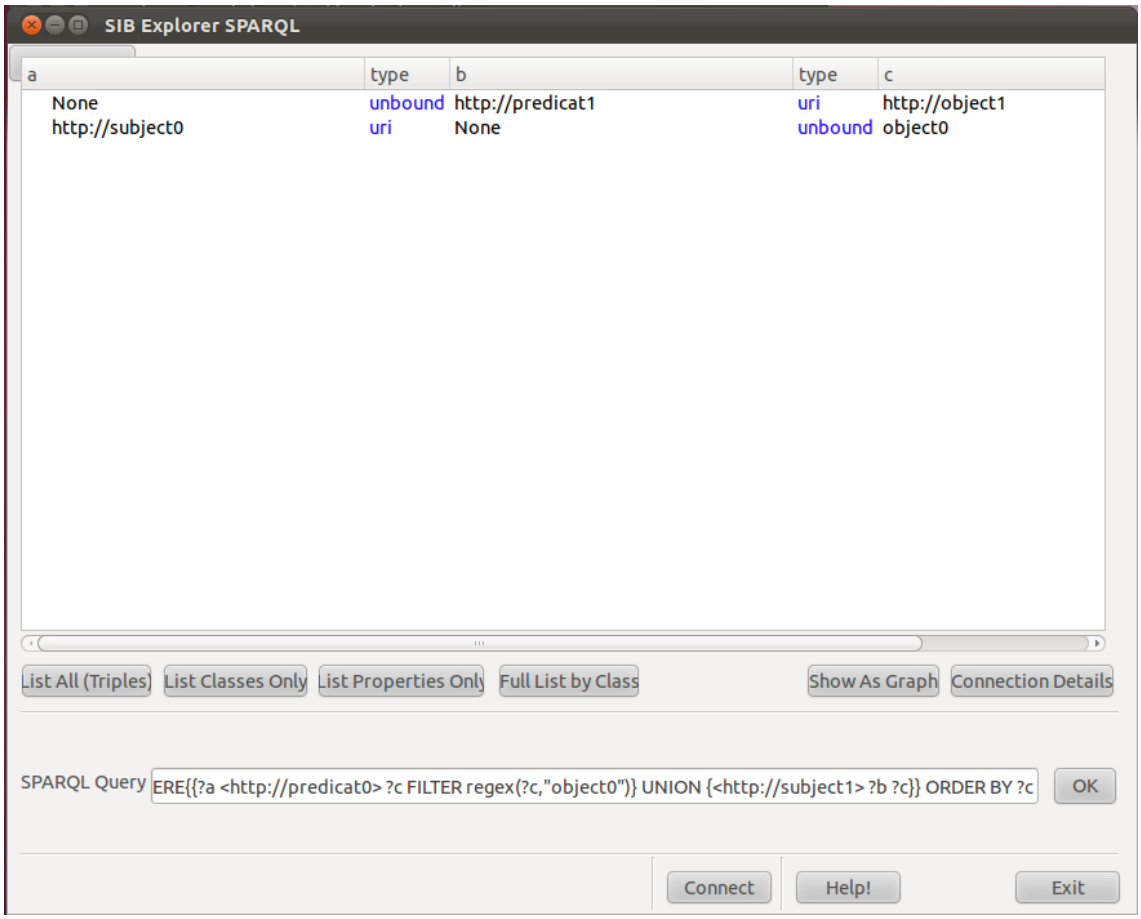

Illustrazione 10: Soluzioni query di tipo SELECT

Come si può vedere dalla figura, le soluzioni della *query* sono due, una che viene selezionata dal primo *graph pattern* e l'altra dal secondo, ordinate in ordine alfabetico secondo l'oggetto delle triple.

Per testare le *query* di tipo ASK, è stata utilizzata la seguente *query*:

ASK {<http://subject2> <http://predicat2> "object0" . <http://subject4> <http://predicat4> "object4" . <http://subject3> <http://predicat3> <http://object3>}

Il risultato della *query* è mostrato in figura 11.

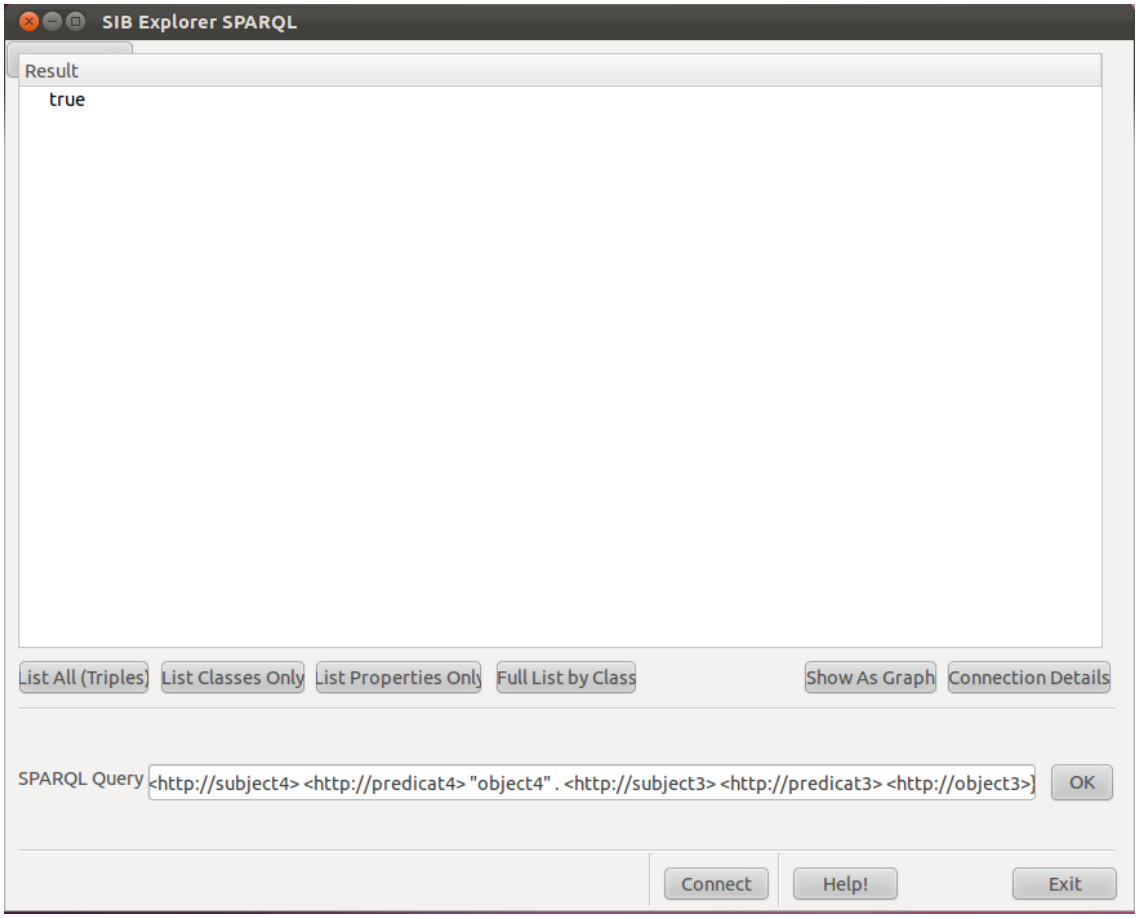

Illustrazione 11: Soluzioni query di tipo ASK

La *query* ha come risultato *true* perchè tutte e tre le triple che costituiscono il *graph pattern* sono presenti nello *storage* RDF della SIB.

Per testare, infine, la *query* di tipo CONSTRUCT, è stata scelta la seguente *query*:

CONSTRUCT {<http://subject10> ?b ?c} WHERE{?b ?c "object0"} LIMIT 1

Le soluzioni della *query* sono mostrati in figura 12.

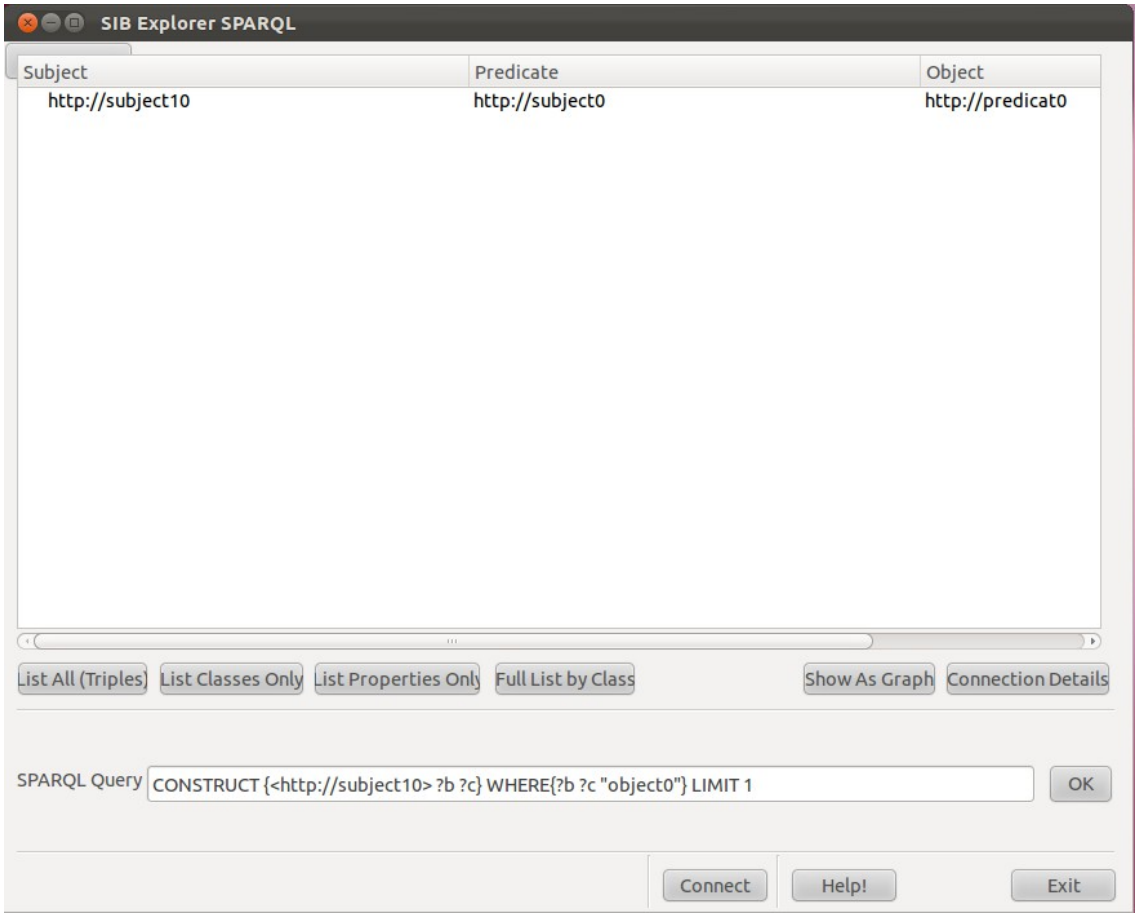

Illustrazione 12: Soluzioni query di tipo CONSTRUCT

Come si può vedere, è stato creato un nuovo grafo composto da una tripla, che ha, come soggetto, <http://subject10> fissato dall'utente e, come predicato e oggetto, rispettivamente il soggetto e il predicato della tripla con oggetto uguale a "object0". In realtà le triple con oggetto uguale a "object0" sono due, ma la condizione di LIMIT 1 ha limitato le soluzioni ad un limite massimo di uno.

Come si può notare dalle figure precedenti, nell'interfaccia utente sono presenti altri bottoni, che implementano altre funzionalità.

Il primo è *List All (Triples)* e serve a mostrare tutte le triple presenti nello *storage* RDF della SIB (attraverso una *query* di tipo RDF-M3).

Poi c'è la funzione *List Classes Only*, che implementa una *query* SPARQL che va a selezionare tutte le triple in cui è istanziata una classe, ovvero tutte le triple che hanno come oggetto un URI del tipo *http://www.w3.org/2000/01/rdf-schema#Class*.

La funzione *List Property Only*, invece, implementa una *query* SPARQL che va a selezionare tutte le triple in cui è definita una proprietà, ovvero tutte le triple che hanno come oggetto un URI del tipo http://www.w3.org/1999/02/22-rdf-syntax-ns#Property.

Infine, la funzione *Full List by Class* seleziona tutte le classi (ovvero le istanze con oggetto *http://www.w3.org/2000/01/rdf-schema#Class*); fatto ciò, per ogni classe cerca le istanze e per ogni istanza di classe cerca le proprietà con i rispettivi valori.

Per verificare il funzionamento di questi tre tipi di *query*, è stato utilizzato un *database* RDF con le triple mostrate in figura 13. Come si può vedere dalla figura, sono state inserite due classi di nome “universitario” e “studente”, due istanze di queste classi, ovvero <http://luca> <rdf:type> <http://universitario> e <http://marco> <rdf:type> <http://studente>, dove rdf è un *namespace* corrispondente a <http://www.w3.org/1999/02/22-rdf-syntax-ns#>, due proprietà di nome “età” e “capelli” e due triple che assegnano alle istanze <http://luca> e <http://marco> le due proprietà con i rispettivi valori. Inoltre è presente una tripla assiomatica, dovuta al *reasoning* implementato e che sarà descritto nel capitolo 6, che afferma che la proprietà è una classe.

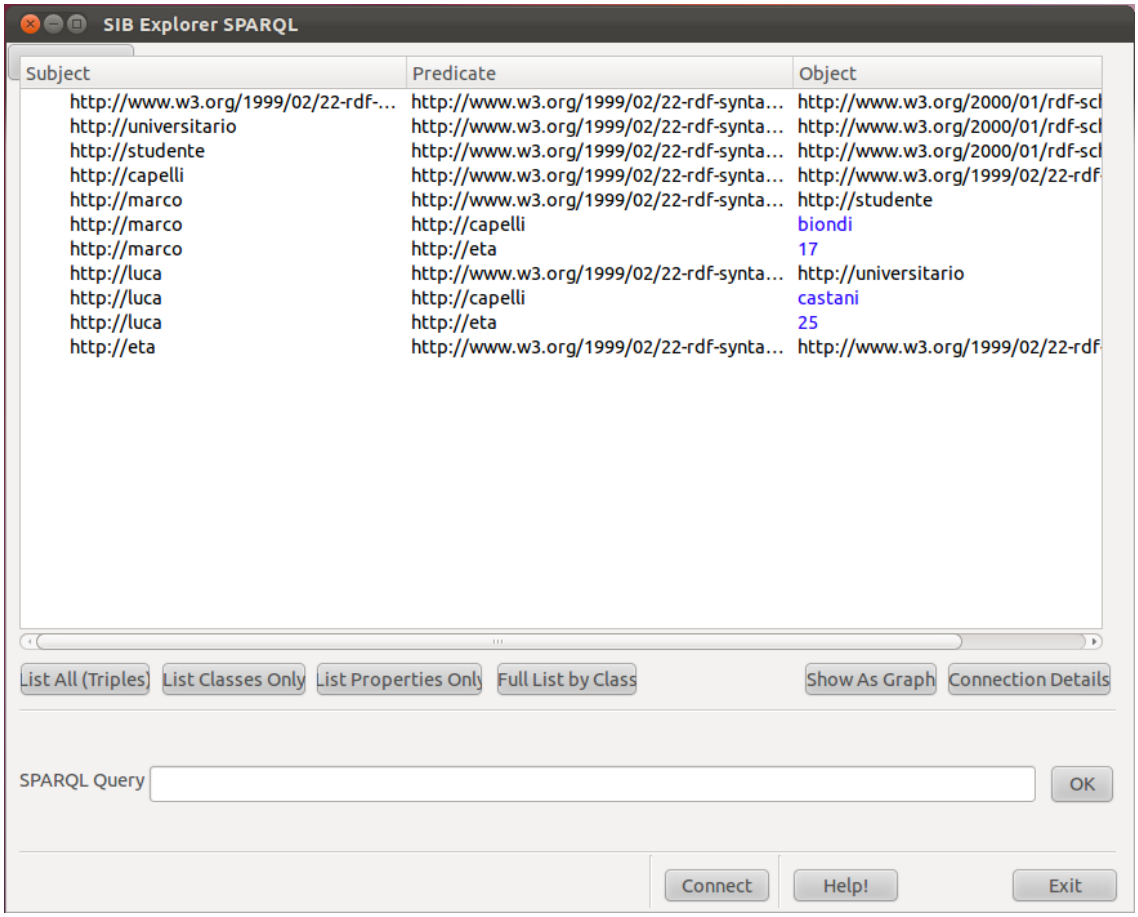

Illustrazione 13: Triple memorizzate nella SIB

Il risultato che restituisce la funzione *List Classes Only* è mostrato in figura 14. Come si nota dalla figura, il risultato è composto da tre elementi, che sono studente, universitario e proprietà (che è una classe assiomatica), come si poteva facilmente prevedere.

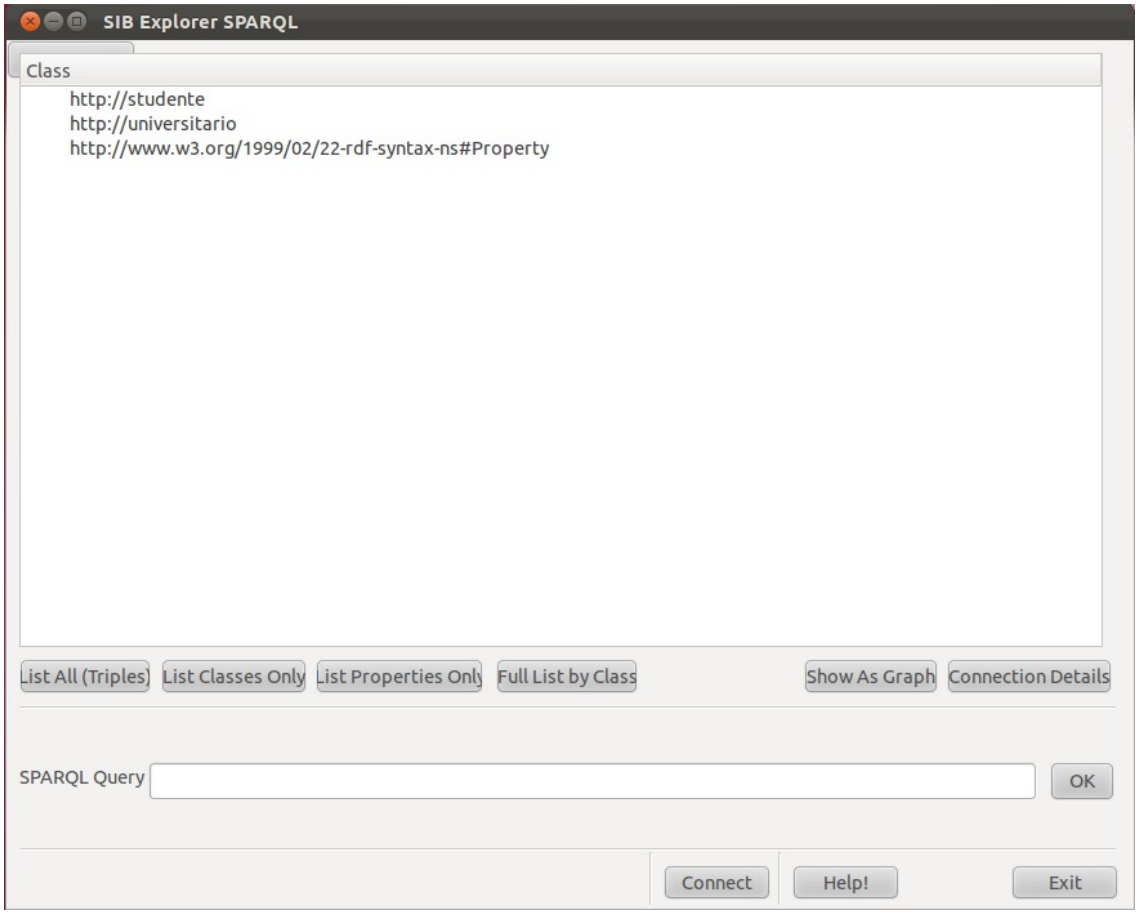

Illustrazione 14: Lista delle classi

Il risultato della funzione *List Properties Only* è mostrato in nell'illustrazione 15. Come prevedibile, la *query* ritorna come risultati due elementi, che sono le proprietà “età” e “capelli”.

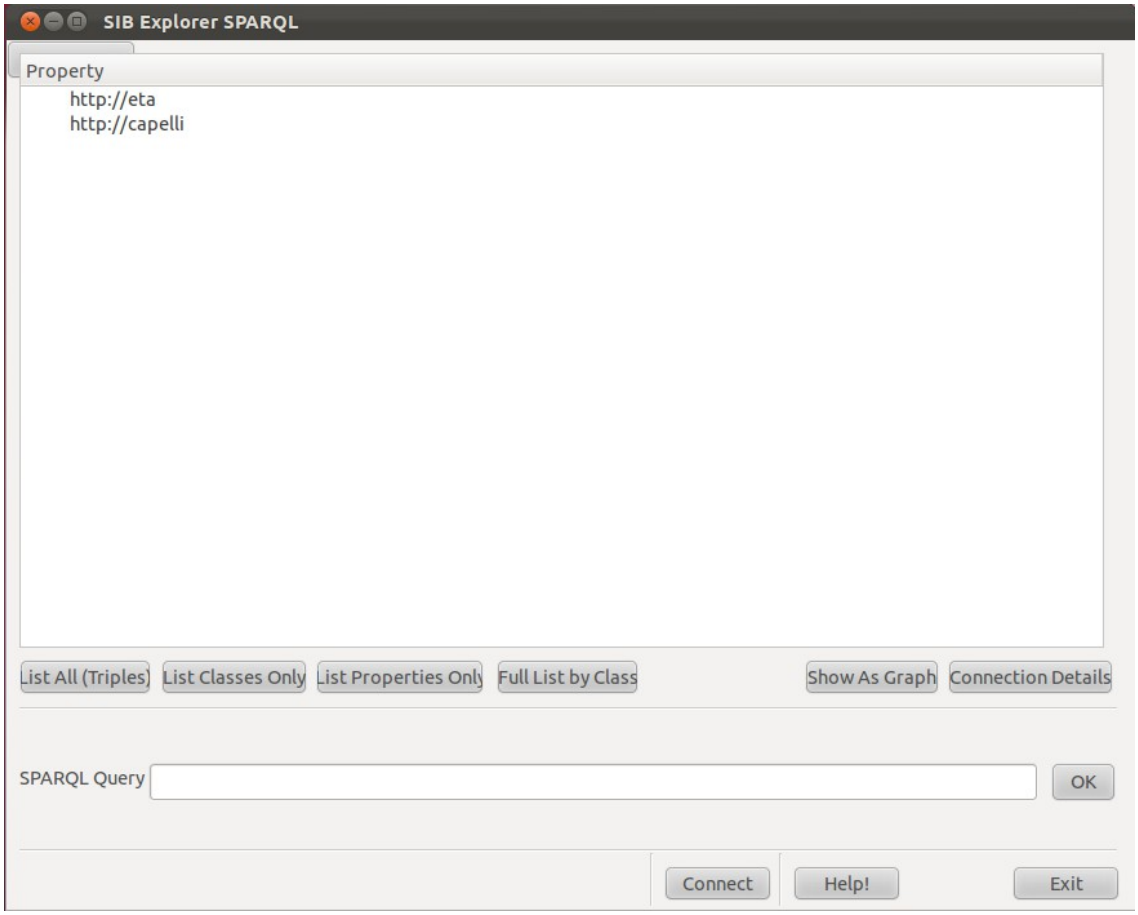

Illustrazione 15: Lista delle proprietà

Infine, la funzione *Full List by Class* mostra i risultati di figura 16. Come si può vedere, la lista comincia con le due classi <http://studente> e <http://universitario>; poi per ognuna di queste classi sono mostrate le istanze, <http://marco> per la prima classe e <http://luca> per la seconda; infine, sono visualizzate le proprietà relative a queste due istanze, ovvero <http://eta> e <http://capelli>, con i valori corrispondenti.

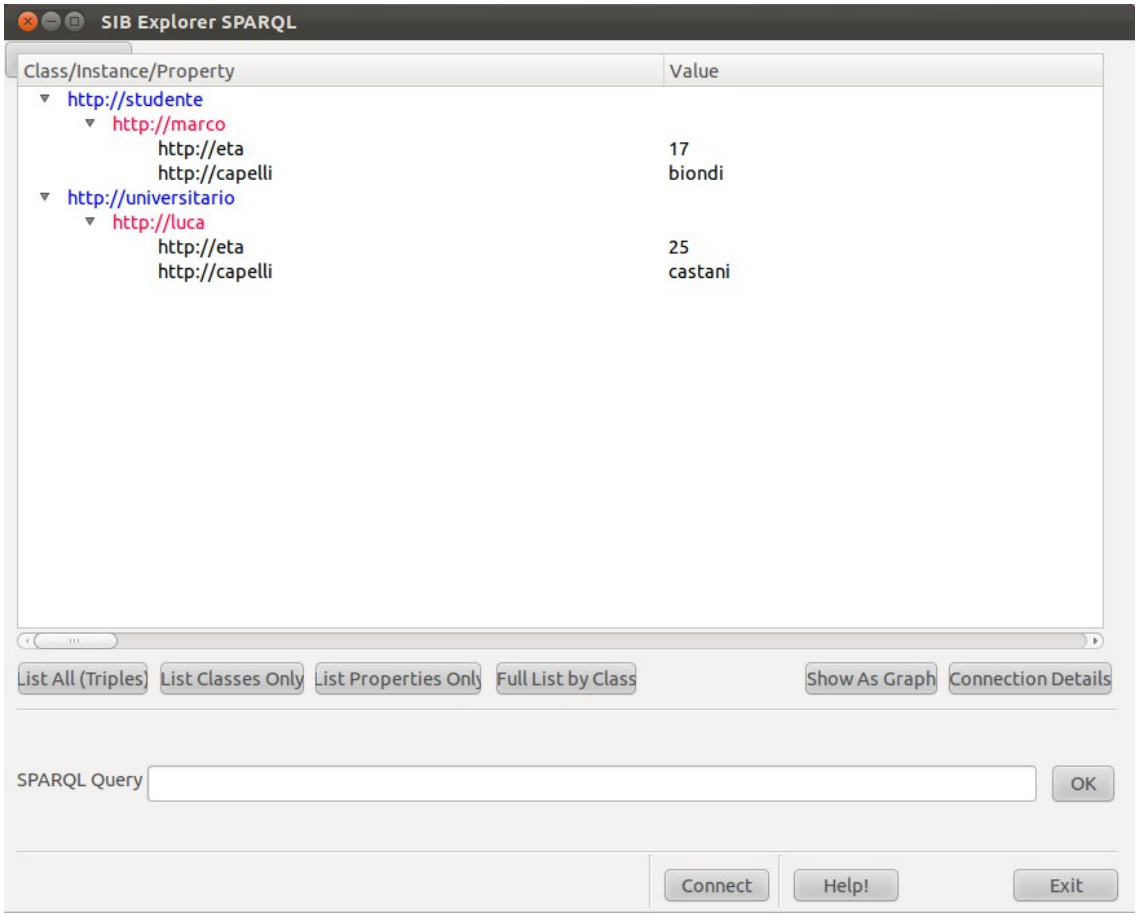

Illustrazione 16:Lista delle classi, istanze di classi, proprietà e valori corrispondenti

Capitolo 6

Reasoning

6.1 Introduzione

In questo capitolo verrà trattata la seconda parte del lavoro svolto.

Questa a sua volta si divide in due fasi: nella prima fase ci si è occupati dell'estensione di alcuni *namespace* standard che si usano nel web semantico; nella seconda parte ci si è occupati del *reasoning* vero e proprio, di cui si è accennato qualcosa nel capitolo 3, ovvero della definizione del modello RDF++. Naturalmente queste due fasi del lavoro sono state implementate direttamente sulla SIB, e quindi realizzate utilizzando interamente il linguaggio C.

6.2 Estensione dei prefissi

Nel *Semantic Web* sono presenti dei prefissi standard, che vengono usati ad esempio per gli URI che indicano classi, proprietà ecc. Sono mostrati nella tabella in basso.

Prefissi	URI
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
xsd:	http://www.w3.org/2001/XMLSchema#
fn:	http://www.w3.org/2005/xpath-functions#

Quindi, un qualsiasi agente software può inserire nella SIB delle triple, in cui possono essere presenti uno o più nodi, che presentano un URI con prefisso abbreviato oppure con prefisso in forma estesa. Ciò implica che, per uno stesso elemento, comunque, la SIB debba memorizzare sempre la stessa informazione nel suo *storage* RDF, sia in caso il prefisso sia abbreviato sia nel caso sia in forma estesa. La stessa cosa deve avvenire anche in caso di rimozione di una tripla, ovvero sia che la tripla selezionata abbia un *namespace* abbreviato oppure che ce l'abbia in forma estesa, l'elemento da eliminare deve essere univoco.

Quindi, un agente software può inserire una tripla con uno o più campi con prefisso abbreviato e successivamente può decidere di rimuoverla selezionandola con i campi col prefisso esteso, e la SIB deve capire che si tratta sempre del medesimo elemento. Stesso discorso vale per le *query* di tipo RDF-M3 che sono implementate nella SIB, le quali si basano su un *matching* con il *pattern* di triple in ingresso (la trattazione di questo tipo di *query* non è compresa nel lavoro svolto). Per quanto riguarda le *query* di tipo SPARQL, questo problema non c'è perchè nel caso in cui si vogliano utilizzare dei prefissi in forma abbreviata, la sintassi prevede, come si è visto nel capitolo 4, una clausola PREFIX, la quale serve ad assegnare dei

prefissi a dei *namespace* specificati dall'utente.

Per evitare questi conflitti da nomenclatura dei campi che costituiscono le triple RDF, è stata implementata una funzione all'interno della SIB, che effettua un controllo sulla tripla, prima di ogni inserimento, rimozione o *query* di tipo RDF-M3.

Questa funzione riceve in ingresso i tre campi di cui si compone la tripla (soggetto, predicato ed oggetto) e per ognuno di questi va a controllare se presenta i prefissi indicati nella tabella precedente. Nel dettaglio, la funzione controlla, dapprima, se i primi tre caratteri del nodo in esame corrispondono a “fn:”: in caso positivo, va a copiare in una stringa d'appoggio la stringa contenente l'URI, ma dal quarto carattere in poi (per non considerare il prefisso “fn:”); viene copiato il *namespace* completo corrispondente al prefisso “fn:” in un'altra stringa, ed infine viene concatenata la stringa precedente a quest'ultima. Nel caso in cui invece non ci sia corrispondenza tra il prefisso “fn:” e i primi tre caratteri della stringa contenente l'URI, viene effettuato un controllo sui primi quattro caratteri della stringa contenente l'URI, per verificare se corrispondono al prefisso “rdf:”. In caso positivo, si ripete lo stesso procedimento logico descritto in precedenza, altrimenti viene effettuato il successivo test di corrispondenza per il prefisso “xsd:”, e se il risultato è ancora negativo, per il prefisso “rdfs:”. Se l'URI non contiene nessun prefisso, questa funzione non esegue nulla e lascia l'URI inalterato.

Mediante questa banale funzione, nella SIB tutte le triple verranno memorizzate con gli URI in forma estesa, per evitare qualsiasi errore di coerenza. E la stessa cosa avviene per la rimozione e le *query* di tipo RDF-M3, che lavoreranno sempre con gli URI in forma estesa.

Come verifica di corretto funzionamento della funzione, si è deciso di riportare un esempio di inserimento di una tripla nella SIB. In figura 17, è mostrata la schermata di un *tool* utilizzato per effettuare delle operazioni sulla SIB (non realizzato nel corso di questa tesi), tra cui l'inserimento di triple, dopo aver effettuato naturalmente un'operazione di *join*.

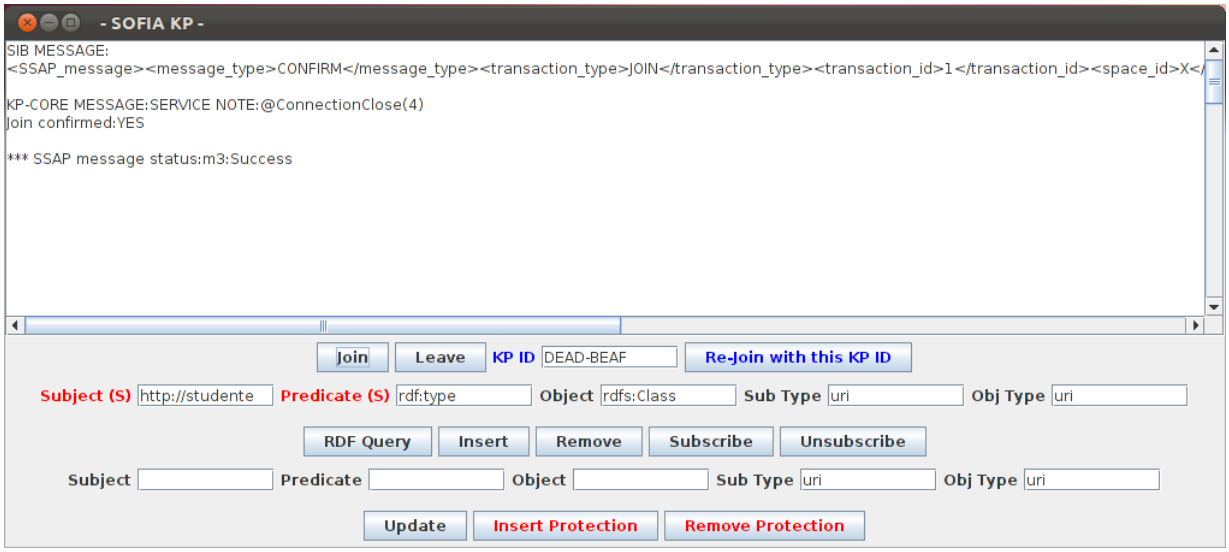

Illustrazione 17: Inserimento tripla

Come si vede in figura 17, nelle tre caselle di testo relative a soggetto, predicato ed oggetto, è stata inserita la tripla <http://studente> <rdf:type> <rdfs:Class> (ovvero è stata aggiunta una nuova classe di nome <http://studente>); le altre due caselle di testo indicano il tipo relativo al soggetto ed all'oggetto, che naturalmente è un URI. Inserendo questa tripla, viene memorizzata nella SIB la tripla in forma estesa, come si può vedere nella figura 18, in cui si è utilizzata l'interfaccia descritta nel precedente capitolo per visualizzare i risultati.

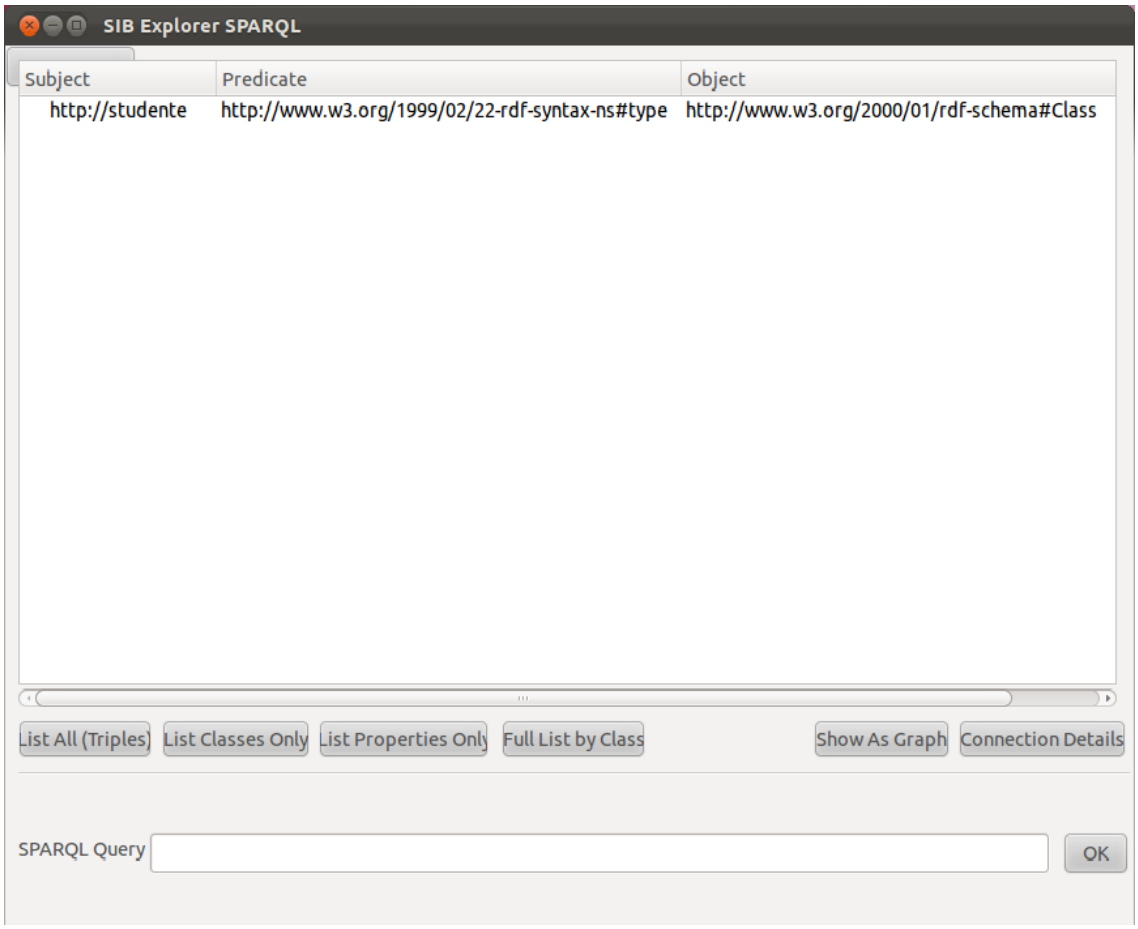

Illustrazione 18: Triple memorizzate in forma estesa

Come si può notare facilmente dalla figura, sia il prefisso presente nel predicato (rdf:) che quello presente nell'oggetto (rdfs:) sono stati estesi; questo verifica il corretto comportamento della SIB.

6.3 Modello RDF++ e sua implementazione

Come precedentemente anticipato nel capitolo 3, nella seconda fase del lavoro ci si è occupati della realizzazione di un modello RDF++, ovvero che preveda il *reasoning*, poiché l'implementazione della SIB, impiegata nel corso della tesi, utilizza come *storage* RDF uno *storage* Redland, che non supportava il *reasoning*.

Il *reasoning* realizzato sulla SIB utilizzata è stato implementato solo per quanto riguarda l'operazione d'inserimento di nuove triple, e non per quanto riguarda la rimozione. Adesso verrà descritto in maniera dettagliata questo modello RDF con *reasoning*.

L'RDF++[11] è un modello RDF realizzato da Ora Lassila, che è un ricercatore presso il centro di ricerca della Nokia a Cambridge, nel Massachusetts. Come detto, questo è un modello RDF che prevede l'inferenza e che quindi prevede l'aggiunta di ulteriori elementi al modello per aumentare la conoscenza, in base a determinate regole dettate da questo ricercatore.

Queste regole sono sei e sono elencate di seguito:

- ogni volta che viene inserita una nuova tripla del tipo <s,p,o>, deve essere inserita anche la tripla che definisce la proprietà p, ovvero la tripla <p,rdf:type,rdf:Property>;
- se viene inserita una tripla che definisce una sottoclasse, del tipo <x,rdfs:subClassOf,y>, di conseguenza bisogna definire x e y come classi, ovvero inserire le triple <x,rdf:type,rdfs:Class> e

<y,rdf:type,rdfs:Class>, e se sono presenti delle istanze della classe x (triple del tipo <z,rdf:type,x>) bisogna rendere quelle istanze anche di classe y (ovvero aggiungere triple del tipo <z,rdf:type,y>). Questo procedimento deve essere fatto poi transitivamente anche per le istanze della classe y, se y è sottoclasse di qualcos'altro, e così via;

- se viene inserita una tripla che definisce una sottoproprietà, del tipo <x,rdfs:subPropertyOf,y>, di conseguenza bisogna definire x e y come proprietà, ovvero inserire le triple <x,rdf:type,rdf:Property> e <y,rdf:type,rdf:Property>. Inoltre per ogni tripla del tipo <o,x,v>, ovvero con predicato uguale a x (i predicati sono delle proprietà), va inserita una tripla del tipo <o,y,v>, poiché x è sottoproprietà di y; questo procedimento va fatto in modo transitivo poi anche per le triple del tipo <o,y,v> se y è sottoproprietà di qualcos'altro, e così via;
- ogni volta che viene inserita una tripla del tipo <p,rdfs:domain,c>, che definisce il dominio di p come c, vanno inserite automaticamente le triple che definiscono p come proprietà (<p,rdf:type,rdf:Property>) e c come classe (<c,rdf:type,rdfs:Class>); inoltre per ogni tripla del tipo <x,p,y>, bisogna definire x come un'istanza della classe c, ovvero bisogna inserire la tripla <x,rdf:type,c>; questo perchè se p è una proprietà che vale in un dominio c, significa che può appartenere solamente a delle istanze di una classe di tipo c;
- ogni volta che viene inserita una tripla del tipo <p,rdfs:range,c>, che definisce il range di p come c, vanno inserite automaticamente le triple che definiscono p come proprietà (<p,rdf:type,rdf:Property>) e

c come classe (<c,rdf:type,rdfs:Class>); inoltre per ogni tripla del tipo <x,p,y>, bisogna definire y come un'istanza della classe c, ovvero bisogna inserire la tripla <y,rdf:type,c>; questo perchè se p è una proprietà che vale in un range c, significa che può valere solamente con delle istanze di una classe di tipo c;

- infine se sono presenti nello *storage* RDF, rispettivamente, una tripla del tipo <x,p,z> , una tripla <y,p,z> ed un'altra del tipo <p,rdf:type,owl:InverseFunctionalProperty>, allora bisogna inserire una tripla del tipo <x,owl:sameAs,y>, che indica che in realtà le due risorse x e y sono la stessa. Questo procedimento deve essere fatto per tutte le triple con proprietà p: questo implica che in realtà queste triple descrivono sempre lo stesso nodo.

Nella SIB tutte queste regole sono state implementate in una funzione, che viene richiamata subito dopo ogni inserimento di una tripla nello *storage* RDF. Questa è una funzione che utilizza le API presenti nella libreria di Redland, dato che si va a lavorare direttamente sullo *storage*.

E' una funzione che riceve in ingresso l'ultima tripla inserita ed effettua una serie di controlli.

Dapprima controlla se la tripla inserita ha come predicato rdf:type e come oggetto un URI diverso da rdfs:Class: in questo caso, inserisce nel modello RDF un'asserzione, che definisce l'oggetto della tripla di partenza come una classe.

Se poi l'oggetto della tripla è l'URI owl:InverseFunctionalProperty, va a cercare tutte le triple che hanno come predicato il soggetto della tripla di partenza, per controllare che siano presenti nel modello RDF dei nodi che coincidono. A questo punto, per ogni elemento trovato si vanno a cercare

tutte le triple che hanno predicato ed oggetto identici a quest'ultimo: se queste triple sono presenti nel modello, significa che sono tutte triple che descrivono lo stesso nodo e quindi per tutte va inserita una tripla del tipo owl:sameAs.

Se, invece, l'oggetto della tripla inserita non è owl:InverseFunctionalProperty, allora si controlla se la classe definita sia sottoclasse di qualcun'altra. In questo caso, si va a creare una nuova istanza della classe superiore, ovvero si inserisce una nuova tripla col soggetto della tripla originaria, il predicato di tipo rdf:type e come oggetto la classe padre. Questo è un procedimento ricorsivo, ovvero si va a controllare se anche la classe superiore è sottoclasse di qualcun'altra ed in caso positivo si va a creare una nuova istanza della nuova classe superiore e così via.

Nel caso in cui, invece, la tripla inserita nello *storage* non sia un'istanza di qualche classe, ovvero non ha il predicato del tipo rdf:type, ma è la dichiarazione di una sottoclasse, con predicato uguale a rdfs:subClassOf, allora per prima cosa si vanno ad inserire due nuove triple in cui si asserisce che il soggetto e l'oggetto della tripla di partenza sono delle classi (come visto nella seconda proprietà del modello di Lassila descritto in precedenza). Inoltre, si vanno a cercare tutte le istanze di tutte le classi, ovvero le triple con predicato rdf:type, e si va a rifare per ognuna il controllo sulle sottoclassi, cioè per ogni istanza si cerca se la classe a cui appartiene è sottoclasse di qualche altra classe superiore ed in tal caso, se non già presente nello *storage*, si va ad aggiungere la stessa risorsa ma come istanza della classe padre. Questo viene fatto in maniera ricorsiva, fino a quando non si giunge ad una classe che non è sottoclasse di nessun'altra.

Se la tripla inserita nello *storage* RDF asserisce una sottoproprietà, ovvero ha un predicato del tipo rdfs:subPropertyOf, allora in primis vengono definiti soggetto ed oggetto della tripla come proprietà (vengono inserite le triple del tipo <soggetto,rdf:type,rdf:Property> e <oggetto,rdf:type,rdf:Property>, come descritto nella terza proprietà del modello). Inoltre, vanno cercate tutte le istanze con proprietà uguale al soggetto della tripla originaria, e se presenti vanno aggiunte le stesse triple trovate ma con predicato uguale all'oggetto della tripla originaria, poiché l'istanza trovata ha una proprietà sottoproprietà di un'altra. Inoltre si va a controllare se la proprietà superiore è sottoproprietà di qualcun'altra, ed in tal caso si aggiungono nuove triple che asseriscono che l'istanza ha come proprietà queste ultime trovate. Il procedimento è ricorsivo come quello per il controllo delle sottoclassi.

Nel caso in cui la tripla inserita nello *storage* definisca un dominio per una proprietà, ovvero è una tripla con predicato uguale a rdfs:domain, per prima cosa vanno aggiunte le triple che asseriscono rispettivamente che il soggetto della tripla originaria è una proprietà e l'oggetto una classe. Inoltre, si va a fare una ricerca delle istanze che hanno come proprietà il soggetto della tripla originaria, e se presenti, il soggetto di queste triple trovate viene istanziato come una classe della tripla originaria (ovvero viene aggiunta una tripla del tipo < ,rdf:type, >). Poichè è stata aggiunta l'istanza di una classe viene effettuato un controllo sulle sottoclassi come nei casi precedenti.

Nel caso in cui la tripla inserita nello *storage* definisca un range di classi per una proprietà, ovvero è una tripla con predicato uguale a rdfs:range, come per i domini, per prima cosa vanno aggiunte le triple che asseriscono rispettivamente che il soggetto della tripla originaria è una proprietà e

l'oggetto una classe. Inoltre, si va a fare una ricerca delle istanze che hanno come proprietà il soggetto della tripla originaria, e se presenti, l'oggetto di queste triple trovate viene istanziato come una classe della tripla originaria (ovvero viene aggiunta una tripla del tipo < ,rdf:type, >). Poichè è stata aggiunta l'istanza di una classe viene effettuato un controllo sulle sottoclassi come nei casi precedenti.

Infine, se la tripla inserita nello *storage* RDF della SIB non appartiene a nessuno dei casi precedenti, allora viene inserita la tripla che definisce il predicato della tripla orginaria come una proprietà (una tripla del tipo < ,rdf:type,rdf:Property>, come descritto nella prima proprietà del modello di Lassila). A questo punto, viene controllato se questa proprietà è sottoproprietà di quancun'altra, se ha un dominio, se ha un range, se è del tipo owl:InverseFunctionalProperty come è stato fatto nei casi precedenti, e seguendo gli stessi procedimenti, e se vengono aggiunte delle istanze di classi, viene fatto un controllo sulle sottoclassi.

Per verificare il funzionamento della funzione di *reasoning* implementata, si è utilizzato un semplice esempio di nucleo familiare.

Sono state inserite in sequenza nello *storage* RDF le seguenti triple:

- <figli,rdfs:subClassOf,parenti>, che definisce la classe “figli” come sottoclasse della classe “parenti”;
- <padri,rdfs:subClassOf,parenti>, che definisce la classe “padri” come sottoclasse della classe “parenti”;
- <parenti,rdfs:subClassOf,persone>, che definisce la classe “parenti” come sottoclasse della classe “persone”;
- <emanuele,rdf:type,figli>, che definisce l'istanza “emanuele” di tipo

"figli";

- <hapadre,rdfs:subPropertyOf,haparente>, che definisce la proprietà "hapadre" come sottoproprietà di "haparente";
- <hasorella,rdfs:subPropertyOf,haparente>, che definisce la proprietà "hasorella" come sottoproprietà di "haparente";
- <nunzio,hapadre,antonio>;
- <hapadre,rdfs:domain,figli>, che definisce il dominio della proprietà "hapadre" uguale alla classe "figli";
- <francesca,hapadre,damiano>;
- <hasorella,rdfs:range,parenti>, che definisce il range della proprietà "hasorella" uguale alla classe "parenti";
- <emanuele,hasorella,francesca>;
- <efigliounico,rdf:type,owl:InverseFunctionalProperty>, che definisce la proprietà "efigliounico" come una proprietà funzionale (cioè ammette un unico oggetto) inversa;
- <maria,efigliounico,barbara>;
- <eva,efigliounico,barbara>.

Il modello RDF, ottenuto con l'inserimento delle triple precedenti, è mostrato nelle figure 19 e 20.

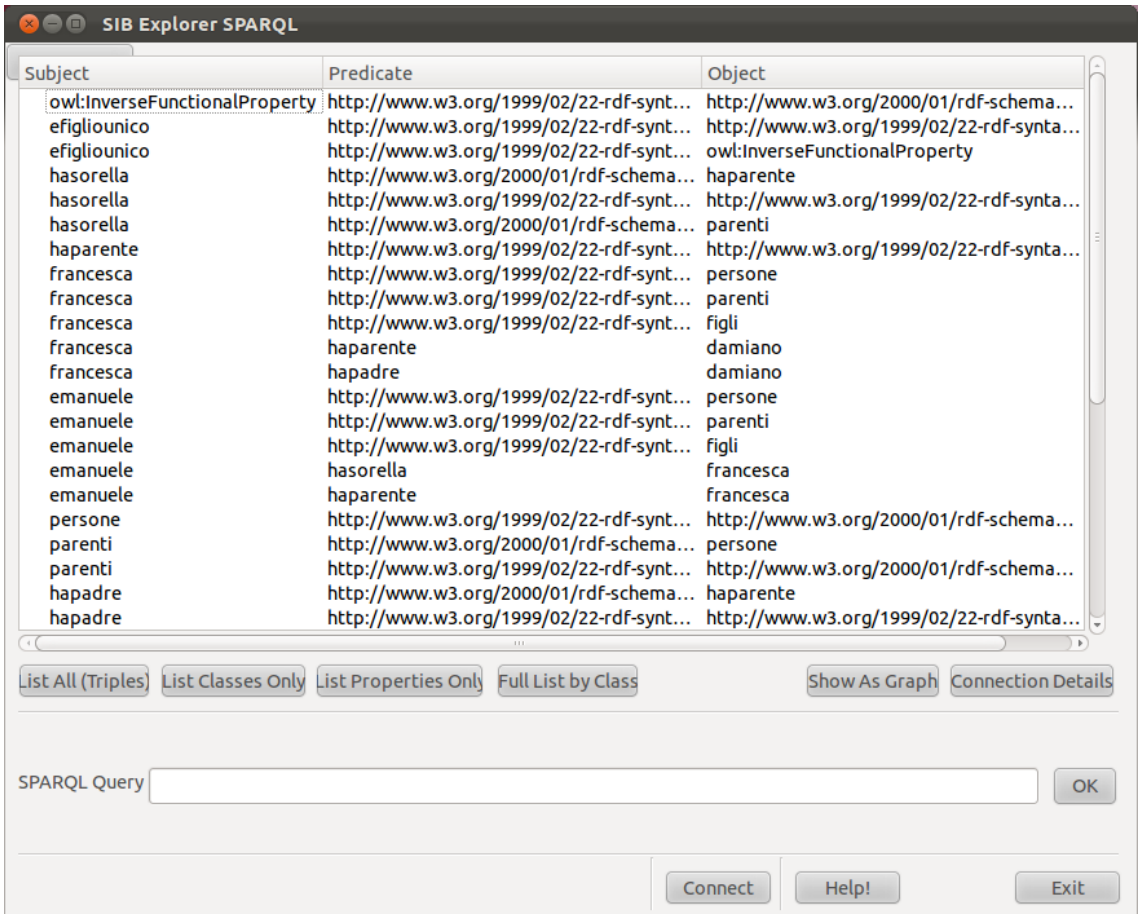

Illustrazione 19: Modello RDF++ ottenuto (prima parte)

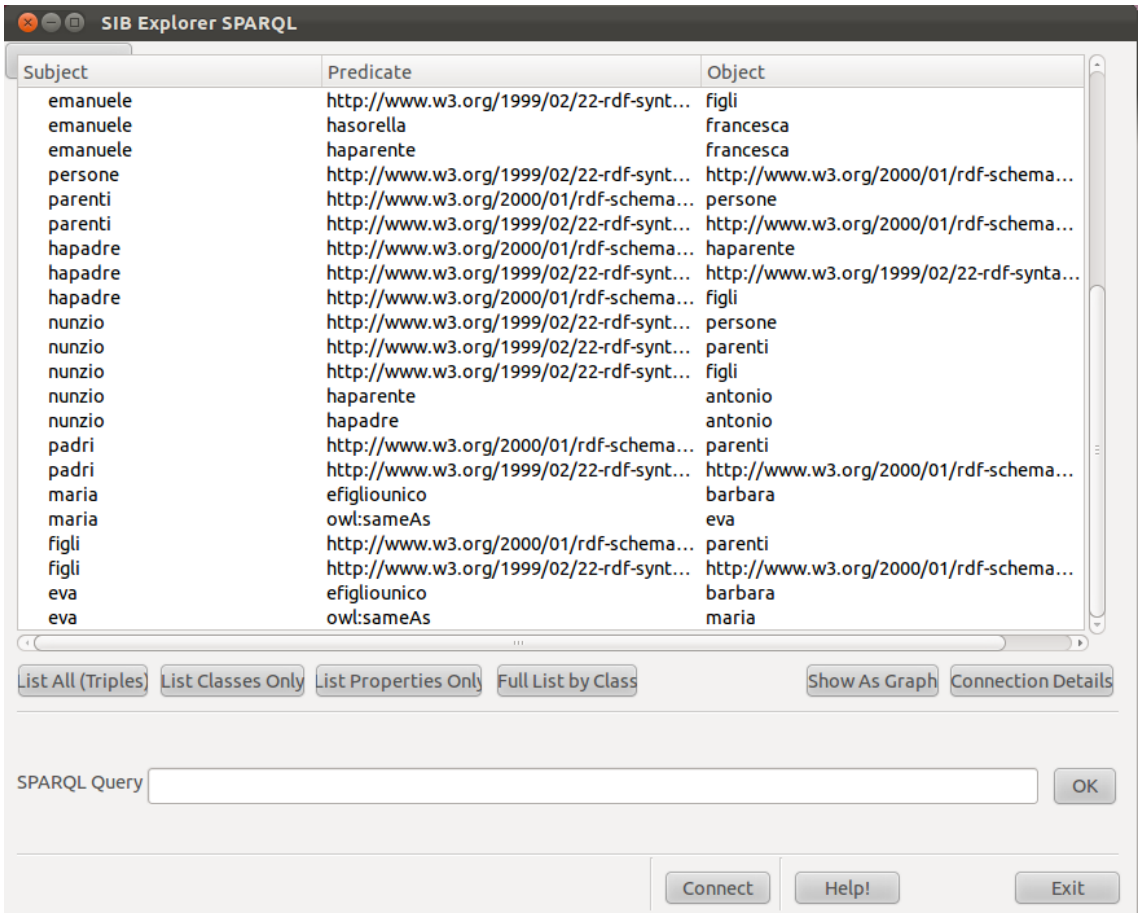

Illustrazione 20: Modello RDF++ ottenuto (seconda parte)

Come si vede dalle figure precedenti, dall'inserimento della tripla <figli,rdfs:subClassOf,parenti> sono state aggiunte le due triple che definiscono le classi "figli" (<figli,rdf:type,rdfs:Class>) e "parenti" (<parenti,rdf:type,rdfs:Class>). Dall'inserimento della tripla <padri,rdfs:subClassOf,parenti> è stata aggiunta la tripla che definisce "padri" come classe (la tripla che definisce parenti come classe era già presente). Dopo l'inserimento della tripla <parenti,rdfs:subClassOf,persone>, viene aggiunta la tripla che definisce "persone" come classe. Di conseguenza, appena viene inserita la tripla che asserisce che "emanuele" è un'istanza di "figli", vengono inserite a seguire le triple che asseriscono che "emanuele" è anche un'istanza di "parenti" e

di “persone” (<emanuele,rdf:type,parenti> e <emanuele,rdf:type,persone>).

Dall'inserimento delle triple che definiscono le proprietà “hapadre” e “hasorella” come sottoproprietà della proprietà “haparente”, vengono inserite che definiscono questi tre elementi come delle proprietà (<hapadre,rdf:type,rdf:Property>, <hasorella,rdf:type,rdf:Property> e <haparente,rdf:type,rdf:Property>). Poi con l'inserimento della tripla <nunzio,hapadre,antonio>, è stata inserita la tripla <nunzio,haparente,antonio>. Inoltre, dall'inserimento consecutivo delle triple <hapadre,rdfs:domain,figli> e <francesca,hapadre,damiano>, vengono aggiunte le triple <francesca,rdf:type,figli> e <nunzio,rdf:type,figli>, che, poiché sono due istanze della classe “figli”, sottoclasse della classe parenti, sono anche istanze delle classi “parenti” e “persone”. Inoltre viene aggiunta la tripla <francesca,haparente,damiano> perchè la proprietà “hapadre” è sottoproprietà della classe “haparente”. In seguito, l'inserimento sequenziale delle triple <hasorella,rdfs:range,parenti> e <emanuele,hasorella,francesca> implica l'aggiunta della tripla <emanuele,haparente,francesca>, perchè “hasorella” è sottoproprietà di “haparente”. Infine dall'inserimento della proprietà funzionale inversa “efigliounico” e delle triple <maria,efigliounico,barbara> e <eva,efigliounico,barbara>, vengono aggiunte le triple che definiscono la proprietà funzionale inversa come una classe, “efigliounico” come proprietà e le triple <maria,owl:sameAs,eva> e <eva,owl:sameAs,maria> che asseriscono che “maria” ed “eva” sono lo stesso nodo del grafo RDF.

Questo semplice esempio dimostra come in questo modello RDF sia stata implementata l'inferenza secondo le regole dettate da Lassila, per ottenere un modello RDF++.

Capitolo 7

Conclusioni

Nel corso di questa tesi, è stata implementata la primitiva di *query* SPARQL del protocollo SSAP di Smart-M3.

Come si è detto più volte, lo SPARQL è il linguaggio di *query* ufficiale della W3C.

Questo linguaggio, grazie alla sua sintassi, permette di eseguire *query* singole abbastanza complesse che, se non fosse supportato, sarebbero implementate tramite diverse *query* di tipo RDF-M3. Quindi, avere uno strumento di *query*, che consente con una sola *query* di cercare effettivamente quello che richiede l'utente (o agente software), è utile in quanto riduce lo scambio di messaggi; ciò riduce il carico computazionale dei KP, poiché non devono in continuazione eseguire delle *query* di tipo RDF-M3 ma basta una sola *query* di tipo SPARQL, per ottenere i risultati cercati. In questo modo viene ridotto lo spreco di banda derivante dalle tante operazioni di *query* che il KP esegue sulla SIB.

Tutto ciò è alla base dello sviluppo del web semantico e degli *smart environments*.

Lo step successivo nello sviluppo della piattaforma software Smart-M3 sarà quello di implementare la primitiva di *query* SPARQL con la sua nuova versione, SPARQL 1.1 (poiché per il lavoro svolto si è utilizzata la versione 1.0 di SPARQL).

Inoltre in questa tesi, è stato realizzato un modello di *reasoning* che permette di estendere il normale funzionamento di RDF aggiungendo informazione implicita. Questo modello RDF++ è stato ottenuto applicando una serie di regole, dettate dal ricercatore Ora Lassila, per ogni inserimento di una nuova asserzione nella SIB.

Bibliografia

[1] "Handbook of Ambient Intelligence and Smart Environments", Hideyuki Nakashima, Hamid Aghajan, Juan Carlos Augusto

[2] http://www.sofia-project.eu/

[3] http://w3c.it/papers "Introduzione al Semantic Web", Oreste Signore

[4] http://en.wikipedia.org/wiki/XML

[5] http://en.wikipedia.org/wiki/Smart-M3

[6] http://en.wikipedia.org/wiki/Tbox

[7] "Smart-M3 Information Sharing Platform", Jukka Honkola, Hannu Laine, Ronald Brown, Olli Tyrkkö

[8] http://www10.org/cdrom/papers/490/

[9] http://www.w3.org/TR/rdf-sparql-query/

[10] http://www.python.it

[11] "Programming Semantic Web Applications:A Synthesis of Knowledge Representation and Semi-Structured Data", Ora Lassila