Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria di Modena

Corso di Laurea in Ingegneria Informatica

ESTRAZIONE DI ENTITÀ NON NOTE DA CORPUS DI DOCUMENTI NON STRUTTURATI USANDO LA TECNOLOGIA COGITO

Relatore:
Chiar.mo Prof. Sonia Bergamaschi

Candidato:
Fabio Manganiello

Anno Accademico 2007/2008

Parole chiave:

corpus
semantica
information retrieval

INDICE

INDICE....3

1. INTRODUZIONE....5

2. L'AZIENDA E LA TECNOLOGIA COGITO....7

3. LINGUAGGI UTILIZZATI....8

3.1.C++....8
3.2.Perl....9
3.3.XML....10
3.4.SQL....11

4. GLOSSARIO....13

5. FLUSSO DEI DATI....15

5.1.Ottimizzazione e robustezza dell'applicazione....15
5.2.Architettura del sistema....18

6. ELEMENTI DELL'ARCHITETTURA....20

6.1.Client....20
6.2.Server GSL....25
6.3.Dispatcher....37
6.4.Database....40

7. CONSIDERAZIONI SUL CODICE....45

7.1.Parsing e salvataggio su database....45
7.2.Robustezza del codice....49
7.3.Gestione della memoria centrale....51
7.4.Ottimizzazione delle query....52

8. ANALISI DEI DATI....55

9. SVILUPPI DELL'ATTIVITÀ – PROGETTO OKKAM....60

9.1 Cos'è Okkam....60
9.2 Definizione di entità non note in base alla correlazione con i lemmi adiacenti....61
9.3 Misura del grado di correlazione per una coppia entità-lemma....63
9.4 Struttura del database usato per l'analisi....64
9.5 Analisi dei dati....66

10. CONCLUSIONI....70

11. SITOGRAFIA....72

Indice delle figure

Figura 1: Sequence diagram del sistema.....18
Figura 2: Activity diagram del client.....21
Figura 3: Activity diagram del server GSL.....27
Figura 4: Schema del database.....42
Figura 5: Schema generale del funzionamento del modulo per Okkam.....62
Figura 6: Correlazione fra le dimensioni del database e la quantità di rumore in esso presente e la precisione delle informazioni salvate.....63
Figura 7: Struttura del database usato per il test per Okkam.....65
Figura 8: Andamento degli score delle coppie entità-lemma all'interno della lista ordinata sullo score in senso decrescente.....69
Figura 9: Sviluppo delle tecnologie web e previsioni sulla rivoluzione del web semantico nel mondo web 3.0.....72

1.INTRODUZIONE

Il campo delle tecnologie semantiche, e in particolare dell'analisi di testi non strutturati, è un campo in continua crescita nell'informatica moderna. Una delle più grandi potenzialità del settore è quella di poter estrarre informazioni arbitrarie da testi non strutturati, trattandoli, attraverso database di forme grammaticali e sintattiche già note, e pattern per riconoscere queste entità all'interno del documento, alla stregua di testi strutturati per un calcolatore.

Una volta che si ha a disposizione un sistema esperto in grado di riconoscere entità logiche e grammaticali all'interno di un documento non strutturato, ed eventualmente anche i domini associati al documento stesso e ai termini usati, si può sviluppare su di esso una sovrastruttura in grado di estrapolare anche entità non note all'interno del documento, e classificarle in base al contesto in cui vengono usate. Tale catalogazione di parole non note a priori diventa ancora più interessante se è fatta non su un singolo documento, o comunque un numero esiguo di documenti, ma su grandi moli di testi. In tal modo la rilevazione diventa statisticamente più attendibile, ed eventuali errori che possono sorgere da termini usati in contesti diversi da quelli usuali all'interno di un dato testo, vengono esclusi in modo quasi naturale facendo statistiche sulle associazioni più ricorrenti, in favore di associazioni dotate di un peso statistico maggiore. Ad esempio, esaminando *N* documenti trovo 20 occorrenze dell'entità non nota *Apache Tomcat*. Di queste, 15 occorrenze sono state rilevate all'interno di documenti aventi come dominio riconosciuto quello di *informatica* o *internet*, e molte di queste occorrenze sono associate a verbi quali *installare*, *configurare* o a sostantivi quali *web server* o *applicazioni Java*. Potrei ritrovarmi anche con qualche occorrenza di tale entità associata al verbo *acquistare*, o al sostantivo *azienda*, o a documenti aventi come dominio principale *economia*, ma verosimilmente tali contesti saranno in minoranza, e sarà più probabile trovare l'entità, se vengono esaminate grandi moli di testi, all'interno di testi che rientrano nelle prime categorie esposte. Quando andrò quindi a fare rilevazioni statistiche sull'analisi compiuta, troverò molto più verosimili le 15 associazioni dell'entità *Apache Tomcat* con il dominio *internet* che le 2 associazioni con il dominio *economia*. Far lavorare il proprio sistema su grandi moli di informazioni, per cercare e catalogare entità non note,

consente quindi di ottenere dati, in base al numero di ricorrenze, statisticamente più attendibili, e fare una scrematura quasi naturale dei significati e dei contesti che una data entità può assumere in base al peso statistico dei risultati.

Lo scopo di questi 3 mesi di stage presso Expert System S.p.A. è stato proprio quello di sviluppare un software, sulla base delle librerie aziendali già esistenti, in grado di esaminare grandi moli di documenti non strutturati provenienti da una qualsiasi fonte, ed estrapolare da essi tutte le entità non note, effettuando poi una classificazione per ognuna di esse in base al contesto d'uso, il che include la classificazione:

- dei *domini* del documento con relativi *score*, ovvero probabilità di attinenza
- del "*parente virtuale*" assegnato eventualmente in modo automatico dal sistema
- del *verbo* associato all'entità (se esiste)
- del *complemento* associato all'entità (se l'entità svolge la funzione logica di soggetto) o del *soggetto* associato (se l'entità svolge la funzione logica di complemento). In entrambi i casi, se e solo se esistono
- dei *domini associati* alle funzioni logiche collegate all'entità (soggetto/verbo/complemento), se esistono

In tal modo si può avere un quadro abbastanza completo dei "casi d'uso", e quindi del potenziale significato, di una certa entità non nota al sistema, specie se a essere esaminate sono grandi moli di testi e se una certa entità è abbastanza ricorrente. Scopo dello stage è stato quello di sviluppare un software in grado di effettuare questo tipo di operazioni, sia in **locale** (*fetch* dei documenti, analisi semantica, memorizzazione su database), sia su un'**architettura distribuita** (il *fetcher* dei documenti, il gestore delle risorse, l'analizzatore e il database sono entità logiche e/o fisiche diverse). L'architettura distribuita ha l'obiettivo di distribuire le capacità di calcolo e memorizzazione del sistema senza sovraccaricare una sola macchina, e quindi ottimizzare le prestazioni complessive. L'oggetto del presente documento costituisce quindi un'estensione e una funzionalità aggiuntiva rispetto al software sviluppato in azienda, pur essendo perlopiù basato sulle librerie aziendali nella fase di disambiguazione dei documenti. Terminato il contesto

del tirocinio, ho poi modificato il software già scritto per l'integrazione nel progetto Okkam, di cui si parlerà in seguitp.

La tesi si articola nel seguente modo:

1. **Introduzione –** Descrizione generale del software sviluppato e del contesto in cui si va a collocare;
2. **L'azienda e la tecnologia Cogito** – Presentazione dell'azienda e delle librerie utilizzate;
3. **Glossario** – Elenco dei principali termini non comuni utilizzati nel documento con relativa descrizione;
4. **Linguaggi utilizzati** – Descrizione a grandi linee dei linguaggi a cui si è fatto ricorso per lo sviluppo del software;
5. **Flusso dei dati** – Descrizione del flusso informativo all'interno del sistema distribuito sviluppato;
6. **Elementi dell'architettura** – Descrizione dettagliata degli elementi che compongono l'architettura del software;
7. **Considerazioni sul codice** – Osservazioni sul codice e descrizione delle strategie intraprese per la sua ottimizzazione e robustezza;
8. **Analisi dei dati –** Osservazioni sull'affidabilità delle informazioni estratte dal software;
9. **Sviluppi dell'attività – Progetto Okkam** – Descrizione delle modifiche effettuate sul software dopo il tirocinio per l'integrazione dello stesso come modulo del progetto Okkam, con descrizione dei modelli matematici alla base delle scelte progettuali e delle analisi di testing;
10. **Conclusioni** – Analisi delle prospettive di sviluppo futuro delle tecnologie semantiche, con riferimento alla classificazione di entità non note;
11. **Sitografia** – Siti di riferimento per eventuali approfondimenti sull'argomento.

2. L'AZIENDA E LA TECNOLOGIA COGITO

Expert System S.p.A. è un'azienda operante da anni nel campo delle tecnologie semantiche, e specializzata nel trattamento di testi non strutturati. Nel corso di quindici anni di attività ha sviluppato *Cogito*, un software in grado di estrarre e analizzare le informazioni utili contenute in migliaia di documenti, email, pagine web, articoli, sms, e capire automaticamente il significato di ogni parola e di ogni testo scritto in linguaggio naturale, sfruttando appieno il patrimonio informativo di un'azienda, di un ente pubblico o dello stesso web. I software sviluppati da Expert System supportano le imprese nelle attività di gestione della conoscenza, e tramite l'analisi automatica dei testi mettono in evidenza il valore delle informazioni non strutturate.

Expert System è stata la prima azienda ad aver sviluppato, sulla base di una "vera" tecnologia semantica, software che possono essere sfruttati in molteplici applicazioni. Fra i servizi offerti e le applicazioni sviluppate dall'azienda, ci sono lo stesso *correttore ortografico* di Microsoft Word, e molti servizi per la gestione automatica delle richieste inviate via email o sms dalla clientela di aziende o enti pubblici. Tutti i servizi offerti da Expert System sono fortemente incentrati sulla tecnologia Cogito per l'analisi di testi non strutturati, e la tecnologia stessa è la base semantica del software sviluppato nel corso dei 3 mesi di stage presso l'azienda.

3. LINGUAGGI UTILIZZATI

Nel corso dello stage sono stati usati linguaggi diversi a seconda del contesto richiesto. In particolare, sono usati linguaggi differenti:

- per la scrittura del codice del client o del server (linguaggi di programmazione veri e propri, compilati o interpretati)
- per la strutturazione del flusso di dati elaborati a partire dai testi non strutturati e scambiati fra le diverse entità del sistema (linguaggi di markup)
- per l'interrogazione della base di dati e la memorizzazione permanente delle informazioni desiderate (linguaggi di query strutturati)

A seconda del caso sono stati usati:

3.1.C++

Linguaggio di programmazione compilato object-oriented che rappresenta un'estensione del C. Il C++ è stato sviluppato (in origine col nome di "C con classi") da Bjarne Stroustrup nel 1983, inizialmente come un miglioramento e un'estensione del C, con supporto per classi, funzioni virtuali, overloading degli operatori e delle funzioni, ereditarietà multipla, template e gestione delle eccezioni. L'applicazione *stand-alone* originale, e anche il server dell'architettura distribuita per l'elaborazione delle richieste, sono scritti interamente in C++, e fanno uso delle librerie aziendali per l'estrapolazione di informazioni dai testi di partenza e la disambiguazione degli stessi. Nella scrittura del codice C++ ho cercato di essere il più possibile aderente agli standard ISO del linguaggio e minimizzare le dipendenze esterne. Il codice C++ fa largo uso delle librerie aziendali per effettuare le operazioni di disambiguazione e classificazione di domini ed entità all'interno dei testi, delle librerie *boost* (ormai uno standard *de facto* del linguaggio, librerie che potrebbero essere presto integrate nel progetto ormai in stadio avanzato della standardizzazione 2.0 del linguaggio) per la lettura dei documenti dal filesystem e per la gestione delle espressioni regolari (*regex*) e delle librerie *mysqlclient* (almeno per la versione in *locale* dell'applicazione) per l'interfacciamento del codice con il DBMS *MySQL*, e quindi l'interrogazione e la memorizzazione delle informazioni sulla base di dati. Viene inoltre utilizzata la

STL (*Standard Template Library*) per la gestione in maniera estremamente versatile di array e iteratori tipizzati all'interno del codice.

3.2.Perl

Perl è un linguaggio di programmazione ad alto livello interpretato e open source sviluppato nel 1987 da Larry Wall, nato e sviluppato principalmente in ambiente Unix, nonostante esistano porting anche in ambiente Windows come *ActivePerl* e il codice scritto su un certo sistema generalmente funzioni senza problemi su qualsiasi sistema, a patto che sia installato l'interprete Perl o un suo porting su quel sistema. Le caratteristiche principali del linguaggio sono legate al suo essere *Unix-oriented*, e difatti Perl eredita sintassi e costrutti principalmente dal C e da linguaggi interpretati *Unix-side* quali *Awk*, *Sed* e *Bash*. Perl è un linguaggio estremamente versatile e modulare e dallo stile estremamente flessibile, caratteristiche che ne consentono un apprendimento relativamente rapido anche ai meno avvezzi all'arte della programmazione, e al giorno d'oggi è molto usato in contesti di *CGI* (*Common Gateway Interface*), quindi per applicazioni e pagine web dinamiche, il che ha portato molti a credere erroneamente che Perl sia un linguaggio dedicato alla creazione di pagine web dinamiche. In realtà, le applicazioni di Perl sono praticamente infinite, grazie anche ad un'enorme risorsa di moduli da cui attingere per ampliare i campi di applicazione del linguaggio, e in particolare è un linguaggio eccellente per la manipolazione di testi. Proprio quest'ultima caratteristica me l'ha fatto scegliere come linguaggio ideale per lo sviluppo del client nell'architettura distribuita. Il client dovrà infatti auto-configurarsi sulla base di un file .conf che dovrà ovviamente "parsare", prelevare da una directory i file di testo da esaminare, memorizzarli, inviarli uno per uno all'entità preposta all'elaborazione, ricevere l'output sotto forma di testo strutturato, parsare l'XML ricevuto, estrapolando da esso le informazioni, e inviare queste informazioni al database. Il client dovrà quindi effettuare una dose massiccia di analisi e parsing di testi, strutturati e non. Sulla base delle considerazioni preliminari, Perl si rivela un linguaggio perfetto per assolvere questo tipo di mansioni.

Nel client vengono in particolare usati i moduli

- *XML::Parser*, per il parsing del testo strutturato sotto forma di XML inviato dal server e quindi l'estrapolazione delle informazioni contenute al suo interno;
- *IO::Socket*, per la gestione dei socket per la comunicazione del client con le altre entità dell'architettura distribuita;
- *Net::MySQL*, per l'interfacciamento con il DBMS MySQL e quindi per la gestione di query per la richiesta delle informazioni già salvate e per l'inserimento di volta in volta di nuove informazioni sulla base dei documenti esaminati.

3.3.XML

XML (*eXtensible Markup Language*) è un metalinguaggio nato nel 1998, come estensione e adattamento dell'*SGML*, sviluppato, gestito e standardizzato dal *W3C* (*World Wide Web Consortium)*. Si tratta di un metalinguaggio e non di un linguaggio vero e proprio, anche se spesso viene erroneamente confuso con il cugino *HTML* come linguaggio di markup e viene spesso associato allo sviluppo di pagine web o in generale allo sviluppo di ipertesti. XML in realtà è un insieme di regole e standard per lo sviluppo di nuovi *pseudo-linguaggi* che consentono la gestione di testi strutturati. E' quindi possibile definire in XML tag in modo arbitrario a seconda dell'applicazione, mentre in HTML è possibile usare solo un insieme finito di tag per la gestione di ipertesti. E' molto utilizzato proprio per lo scambio di informazioni strutturate fra diverse entità logiche in un sistema, e in particolare per la gestione di dati provenienti o da salvare su una base di dati. XML verrà infatti utilizzato nel progetto come linguaggio per lo scambio di dati strutturati fra il client e il server. Quest'ultimo effettuerà infatti la disambiguazione del testo non strutturato, creando un file XML (quindi *semi-strutturato*) contenente le informazioni richieste dal client (domini del documento, entità non note con relativi parenti virtuali, soggetti/verbi/complementi associati e domini delle entità associate) e invierà queste informazioni al client. Questo esaminerà le informazioni strutturate inviate dal server, effettuerà il parsing dell'XML e memorizzerà le informazioni così estrapolate sul database.

La sintassi dell'XML è estremamente intuitiva. In accordo agli standard del linguaggio, ogni file XML in genere parte con un commento (i commenti in XML

sono racchiusi fra i tag <*?* e *?*>) contenente informazioni circa la versione utilizzata dello standard e la codifica. Esempio:

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

Di seguito, l'inizio di ogni tag è segnato dalla notazione

```
<nome_tag>
```

e la fine dalla notazione

```
</nome_tag>
```

E' anche possibile definire tag che iniziano e finiscono sulla stessa riga, attraverso la notazione

```
<nome_tag />
```

Ogni tag può inoltre avere un numero arbitrario di attributi e un contenuto arbitrario, sul modello

```
<nome_tag attributo1=”valore1” ... attributo_n=”valore_n”>
contenuto </nome_tag>
```

3.4.SQL

SQL (*Structured Query Language*) è un linguaggio strutturato sviluppato nel 1974 nei laboratori IBM per l'accesso alle informazioni memorizzate in una base di dati. Inizialmente chiamato *SEQUEL*, il nome fu poi cambiato in *SQL* nel 1977 per motivi legali, e nel 1983, con il rilascio da parte di IBM del DBMS relazione *DB2*, SQL divenne lo standard *de facto* (e *de iure* a partire dal 1986 con la standardizzazione ANSI, e in seguito ISO, con le varie standardizzazioni SQL86, SQL92 ed SQL2003) per la gestione dei dati memorizzati su una qualsiasi base di dati relazionale. L'obiettivo delle standardizzazioni ISO era quello di creare un linguaggio per la manipolazione di query che funzionasse su qualsiasi DBMS relazionale. Quest'obiettivo purtroppo non è mai stato raggiunto, in quanto, nonostante i vari produttori di DBMS abbiano adottato la base del linguaggio come *entry point* su cui basarsi, di fatto ogni produttore ha poi

elaborato un proprio *dialetto* di SQL a partire dalla base stessa del linguaggio. Senza addentrarci ulteriormente nell'ambito della strutturazione delle query per la gestione delle informazioni in un DBMS o nella teoria delle basi di dati relazionali, tematiche che esulano dallo scopo di questo documento, possiamo in maniera molto sintetica affermare che è possibile creare nuove entità in un database attraverso la parola chiave *CREATE* (*CREATE DATABASE, CREATE TABLE, CREATE TRIGGER, CREATE VIEW, CREATE PROCEDURE…*), alterare la struttura delle entità attraverso la parola chiave *ALTER*, inserire nuove entry attraverso la parola chiave *INSERT*, modificarle attraverso *UPDATE*, selezionarle attraverso *SELECT*. Questa velocissima (e un po' rozza) carrellata offre le basi per poter comprendere a grandi linee le query SQL usate all'interno del documento. Per qualsiasi approfondimento, rimando il lettore alla documentazione di riferimento di MySQL, DBMS usato per la gestione delle informazioni all'interno del progetto.

4. GLOSSARIO

In questa sezione verranno illustrati brevemente i termini poco comuni usati all'interno del documento, per una maggiore comprensione dello stesso e a scanso di equivoci.

- **SENSIGRAFO**. Risorsa primaria dell'azienda, è un'entità che offre il database di tutte le entità logiche e grammaticali riconosciute, dei lemmi, con relative parentele, dei possibili domini, e le funzioni di libreria per interfacciarsi con essi.
- **DISAMBIGUATORE**. Libreria basata sul sensigrafo in grado di effettuare la disambiguazione e l'analisi di testi non strutturati.
- **SYNCON**. Sinonimo del termine *synset*, è inteso come una chiave (generalmente un valore numerico intero) che identifica in modo univoco all'interno del sensigrafo l'insieme di termini (sinonimi) accomunati da una stessa definizione. Al lemma *cellulare* e al lemma *telefonino* ad esempio sarà verosimilmente associato nel sensigrafo lo stesso *syncon*. Il syncon si può quindi concettualmente vedere come una *chiave primaria* per accedere ad una determinata definizione all'interno del sensigrafo.
- **SUPERNOMEN**. Termine usato per identificare il *parente* di un certo syncon, è anche conosciuto come *iperonimo*. Ad esempio, verosimilmente il supernomen del lemma *alano*, così come del lemma *mastino*, sarà *cane*, così come il supernomen del lemma *transistor* sarà *dispositivo elettronico*. Per un dato syncon esiste sempre al massimo uno e un solo supernomen.
- **SUBNOMEN**. Il subnomen, anche conosciuto come *iponimo,* identifica un syncon di cui un altro syncon è supernomen. Il lemma *cane* avrà per esempio i subnomen *mastino, barboncino* o *alano*, così come il lemma *dispositivo elettronico* avrà come subnomen *transistor, diodo* o *amplificatore*.
- **GSL** (*Grammar, Semantics, Language*). Architettura di base usata in azienda per i server per l'elaborazione dei documenti non strutturati. Un server GSL è un'entità software estremamente estensibile. Accetta infatti un numero arbitrario di comandi, alcuni dei quali di base (richiesta della versione in uso, *echo* per accertarsi il corretto scambio di pacchetti ecc.) e altri a discrezione dello sviluppatore e in base al tipo di GSL che si vuole creare. Si possono aggiungere nuovi comandi semplicemente creando un nuovo insieme di funzioni che effettuano il compito desiderato, e

associando queste funzioni a un dato comando numerico richiamando la funzione principale in uno switch-case nel main, e lasciando inalterato il resto del codice. L'architettura del GSL di base è proprio quella su cui ho scritto il server dedicato all'analisi dei testi nell'architettura distribuita.

- **DISPATCHER**. L'architettura distribuita è pensata in modo estremamente ottimizzato e modulare, e per interfacciarsi con un numero sia di client che di server GSL arbitrario. Tutto ciò che saprà il client è l'ID della risorsa con cui mettersi in comunicazione, e tutto ciò che saprà il server GSL è il messaggio inviato dal client. Nel mezzo a fare da mediatore ci sarà una terza entità detta *dispatcher*, relativamente complessa, che ha il compito di registrare le risorse (server GSL) ad esso collegate e fornire un meccanismo di interfacciamento fra le risorse ed i client richiedenti.

Sulla base delle definizioni date sopra è quindi possibile costruire una sorta di *albero* di significati. Il sensigrafo in un certo senso può essere rappresentato proprio come un albero del genere, dove ad ogni elemento è associato un syncon e le parentele fra diversi syncon sono principalmente di tipo *supernomen/subnomen*, e ogni supernomen ha sotto di esso una certa gerarchia di subnomen.

5. FLUSSO DEI DATI

5.1.Ottimizzazione e robustezza dell'applicazione

L'ottimizzazione dei tempi è di cruciale importanza per la buona riuscita e la robustezza del progetto. L'applicazione dovrà infatti elaborare moli di dati anche di dimensioni elevate. E' stata ad esempio testata, in fase di stage, l'applicazione su corpus di dimensioni arbitrarie di documenti non strutturati proprio per verificarne la robustezza, l'attendibilità delle informazioni estrapolate, la minimizzazione dei tempi d'uso di memoria e CPU e l'ottimizzazione dei tempi. Per i fini di questa analisi, l'applicazione è stata testata su corpus quali tutti gli articoli tratti dalle ultime 5 annate di due mensili scientifici (*Newton* e *Le Scienze*), poi, aumentando le dimensioni, prima su tutte le voci dell'Enciclopedia Encarta in formato txt (circa 200 000 lemmi per altrettanti file di testo) e infine, per il test finale, su un corpus di dimensioni davvero notevoli quale tutti gli articoli in formato txt usciti negli ultimi 15 anni su un quotidiano a larga scala come il *Corriere della Sera* (circa un milione di articoli per altrettanti file di testo, per un corpus delle dimensioni totali di circa 2 GB). Con corpus di queste dimensioni diventa indispensabile l'analisi accurata del flusso di dati, con lo scopo di rendere l'applicazione abbastanza robusta da gestire simili quantità di informazioni senza dare problemi e di minimizzare i tempi di elaborazione. In particolare, l'ottimizzazione del codice deve agire su più fronti:

- **Struttura della directory dei documenti**. La memorizzazione di tale struttura e di conseguenza di tutti i file in essa contenuti è un'operazione che va effettuata una sola volta e all'inizio, per evitare tanti accessi al filesystem nella fase di elaborazione, che comprometterebbero le prestazioni complessive. Anche la struttura dei documenti va gestita in modo ottimizzato. La memorizzazione della struttura effettua chiamate di tipo ricorsivo ogni volta che viene incontrata una nuova directory all'interno del percorso di base specificato, e questo si traduce in tanti salvataggi sullo stack dello stato della memoria prima della chiamata. Se ho una struttura per i documenti male organizzata (ad esempio un gran numero di sottodirectory, ad esempio una per ogni numero del mio quotidiano, all'interno di una per tutti i numeri usciti la stessa settimana, all'interno di una per tutti i numeri usciti lo stesso mese, e così via) rischio di effettuare

un gran numero di chiamate ricorsive, che alle lunghe possono influire notevolmente, e negativamente, sui tempi di scansione della struttura, e nei casi peggiori portare anche a malfunzionamenti del codice per saturazione dello stack a causa del gran numero di record di attivazione generati per ogni chiamata ricorsiva. Cercare invece di minimizzare il numero di file e directory all'interno del corpus richiede sì tempi di lettura leggermente più elevati, dovendo essere i file e le directory da esaminare in genere di dimensioni maggiori, ma un numero minore di chiamate ricorsive. A ognuni chiamata ricorsiva viene passato come argomento un array di nomi di file le cui dimensioni possono anche essere nell'ordine dei MB, rischiando quindi di creare un'enorme massa critica di dati in memoria centrale per ogni chiamata ricorsiva.

●**Scambio dei dati**. In una simile architettura distribuita bisogna tenere in conto anche le capacità della rete. L'unico parametro su cui può agire il programmatore per cercare di non trasformare questo in un collo di bottiglia è minimizzare per quanto possibile il flusso di informazioni scambiato fra le varie entità. Le entità che fanno parte dell'architettura distribuita si dovranno scambiare informazioni nelle quantità minime indispensabili per il corretto funzionamento, senza elementi ridondanti.

●**Gestione della memoria dinamica**. Quando si ha a che fare con moli di dati tanto grandi va prestata molta attenzione all'allocazione della memoria dinamica. Ciò che non si fa sentire in maniera sensibile per moli di dati di dimensioni medio/piccole, ad esempio una mancata deallocazione di una zona di memoria, può avere effetti notevoli quando le dimensioni dei dati da trattare aumentano, e una mancata deallocazione può portare alla lunga al collasso dell'applicazione per *memory leak*. Questo anche nei casi che potranno sembrare più banali. Anche allocare 100 byte in memoria ad ogni ciclo di lettura e dimenticare di deallocarli alla fine può portare ad un'allocazione di 100 MB di memoria aggiuntivi e non utilizzati quando si trattano un milione di documenti, e memory leak come questo possono essere fatali per la robustezza e la stabilità del programma. E' quindi di importanza vitale ricordarsi di deallocare la memoria allocata quando non serve più.

●**Gestione del database**. Bisogna fare attenzione a non trasformare il database in un collo di bottiglia per il sistema. Il numero di query effettuate sul database può potenzialmente assumere valori molto elevati, se in media in ogni documento vengono trovate 5 entità non note, per ognuna 4

fra supernomen virtuali/verbi/soggetti/complementi associati, e per ogni supernomen/verbo/soggetto/complemento associato 2 domini, e per ogni documento 3 domini, si fa presto a notare che per ogni documento dovrò effettuare in media circa 40 istruzioni INSERT INTO sul database, e in un milione di documenti si fa presto a raggiungere la quota di 40 milioni di istruzioni INSERT INTO o più. Va quindi se possibile effettuata una piccola cache interna di dati da salvare sul database, e scrivere questi dati solo una volta che la cache si è riempita attraverso una INSERT INTO, in modo da minimizzare il numero di operazioni sul database. E' anche necessario cercare di ottimizzare i tempi di recupero delle informazioni dopo aver terminato l'inserimento di informazioni nel database. Ci ritroveremo infatti ad avere a che fare spesso con database contenenti tabelle con qualche milione di entry, e una SELECT su moli di dati di questo tipo, specie se associata a funzioni aggregate come GROUP BY o ORDER BY, necessarie per effettuare analisi statistiche sui dati memorizzati, può richiedere tempi inaccettabili. Dopo aver memorizzato sul database tutte le informazioni richieste per le ultime 15 annate di articoli del Corriere ho provato subito, prima dell'ottimizzazione, ad effettuare una query SELECT con GROUP BY per estrapolare le entità non note più ricorrenti, e sono stati necessari tempi molto elevati per portare a compimento la query

```
20 rows in set (1 hour 57 min 45.00 sec)
```

Tali tempi non sono in genere accettabili, e non è possibile effettuare rilevazioni statistiche sui dati memorizzati se ogni query richiede circa 2 ore per essere portata a termine. E' quindi necessario operare un'ottimizzazione anche su questo versante, indicizzando le colonne usate più spesso, creando eventualmente viste e usando tabelle di appoggio in grado di trasformare "attributi" dinamici generati da un GROUP BY in attributi statici attraverso delle INSERT INTO … SELECT.

5.2.Architettura del sistema

Il flusso delle informazioni e le entità principali del sistema si possono quindi schematizzare nel seguente modo:

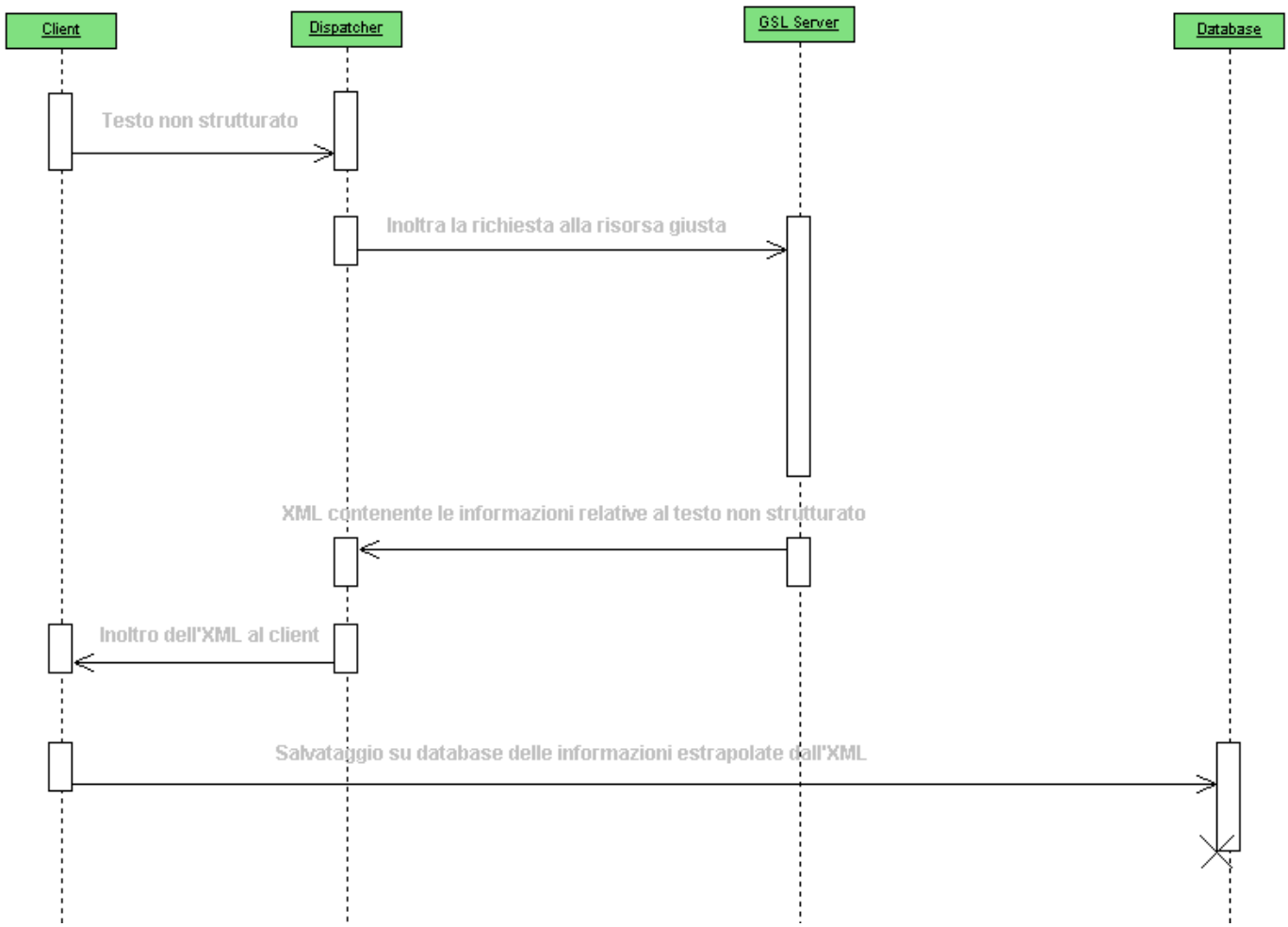

Figura 1: Sequence diagram del sistema

Concettualmente:

1.Il client ha sul proprio filesystem un insieme di documenti non strutturati e in formato txt da elaborare, tutti all'interno della stessa directory;
2.Legge la struttura della directory, la memorizza in un array, quindi legge i file uno per uno e ne invia il contenuto al dispatcher, specificando l'ID della risorsa a cui andranno inoltrate le richieste;

3.Il dispatcher ha memorizzato internamente le associazioni ID risorsa -> server GSL, e inoltra quindi la richiesta al server GSL giusto, ponendosi come intermediario fra client e server e consentendo quindi alle due entità di operare a un livello di astrazione superiore a quello dello scambio fisico di informazioni sul socket;
4.Il server GSL effettua la disambiguazione, classifica i domini di ogni documento, riconosce le entità non note in ogni documento e le classifica specificando per ognuna gli eventuali supernomen assegnati dal sensigrafo, complemento/soggetto o verbo associato, e per ognuno di questi syncon associati ne classifica anche il dominio.
5.L'output del server GSL è un documento XML contenente le informazioni richieste (domini, entità non note, entità associate) in formato strutturato. Tale output viene inviato dal server GSL al dispatcher.
6.Il dispatcher verifica la risposta del GSL a quale richiesta è associata, e rigira il testo XML al client che ha fatto la richiesta.
7.Il client effettua il parsing dell'XML, estrapolando da esso le informazioni memorizzate.
8.Le informazioni così estrapolate vengono inviate al DBMS, dove vengono memorizzate.

L'architettura è quindi distribuita nel modo più versatile e modulare possibile, ma nulla mi impedisce ovviamente di tenere due o più entità anche sulla stessa macchina fisica, nonostante questo non sia l'approccio più ottimale. Inoltre, grazie al modo in cui è strutturata l'architettura è possibile avere un grande insieme di documenti da elaborare, e ogni client può aver memorizzato una parte di essi. Ogni client può quindi inviare la sua parte da elaborare, e dall'altra parte del dispatcher ci può essere tanto un solo server GSL a effettuare l'elaborazione, quanto più di uno. Con questi mattoni è quindi possibile costruire architetture uno-a-uno, molti-a-uno, uno-a-molti o molti-a-molti.

6. ELEMENTI DELL'ARCHITETTURA

Esamineremo ora nel dettaglio ogni singolo elemento che compone l'architettura esposta sopra.

6.1.Client

Il client è l'elemento su cui sono memorizzati i documenti. Il suo compito è leggere il contenuto dei documenti da analizzare, inviarne uno per uno al dispatcher, leggere la risposta sotto forma di XML, effettuare il parsing dell'XML e inviare i dati così estrapolati al database. Sotto forma di activity diagram:

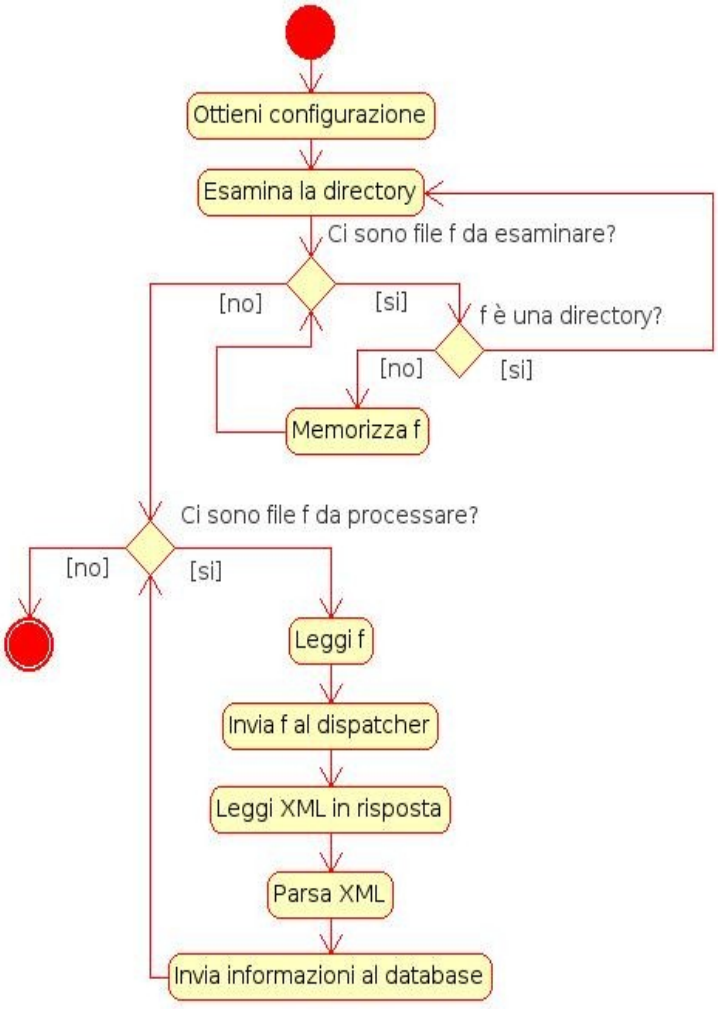

Figura 2: Activity diagram del client

La configurazione viene letta dal client da un file chiamato di default *client_gsl.conf*. La sua struttura di default è la seguente:

```
# Informazioni per la connessione al database
dbhost=<host_db>
dbname=<nome_db>
dbuser=<user_db>
dbpass=<pass_db>

# ID iniziale del client per il salvataggio dei documenti sul
database
start_doc_id=10000000

# ID iniziale del client per il salvataggio delle occorrenze sul
database
start_occ_id=10000000

# ID iniziale del client per il salvataggio delle entità associate
sul database
start_ent_id=10000000

# Informazioni per la connessione con il dispatcher
disp_server=192.168.196.1
disp_port=5500

# ID della risorsa
ris=3

# Comando di disambiguazione
cmd=123
```

Listato 1: Struttura di default del file .conf del client

Al suo interno sono contenute le informazioni per la connessione al database, informazioni sul dispatcher (unica informazione che è data conoscere al client sull'architettura della rete, indirizzo IP del dispatcher e porta sulla quale è in ascolto il servizio), l'ID della risorsa che effettuerà l'analisi del documento, il comando da inviare al server GSL e gli ID iniziali. Gli ID iniziali per ogni client vengono stabili dall'amministratore di sistema, e servono come punto di partenza dal quale il client può cominciare a memorizzare le chiavi primarie per le varie tabelle sul database. Sono scelte nell'ottica di un database sul quale possono scrivere anche più client: in tal caso, un client comincerà a scrivere gli ID a partire dall'indice 10000000, un altro dall'indice 20000000 e così via. Tale scelta è fatta essenzialmente per due motivi:

- Evitare di usare ID auto_increment per le tabelle. Usando quest'ultima strategia infatti il client non può sapere a priori che ID avrà l'entry che ha appena inserito nel database (dato che è indispensabile conoscere per

portarsi dietro delle foreign key fra le varie tabelle). Per conoscerlo dovrà ricorrere a una funzione specifica del DBMS o effettuare una SELECT max(id) o qualcosa di simile sulla tabella in questione, e questo, visto il numero di query che già vengono fatte, appesantirebbe notevolmente il carico di lavoro sul database.

•Orientamento verso un'architettura multi-client, ed eventualmente mono-database. Ogni client comincerà a scrivere dati sul database a partire da un certo valore, ad esempio da 10000000 a 19999999 (considerando improbabile che un dato client debba scrivere più di 10 milioni di entità in una tabella), e quindi sarà molto difficile che capitino violazioni di chiavi primarie per scritture su database dello stesso ID. Inoltre, con questa strategia l'amministratore di sistema può anche sapere quanti valori vengono memorizzati da un certo client all'interno del database, dato che di fatto ogni client avrà il suo range di ID.

Il parsing di questo file di configurazione viene effettuato da una semplice funzione in Perl:

```
sub getconf  {
  open IN,'< '. $conf or die "Impossibile leggere da $conf: $!\n";

  while (<IN>)  {
          if ($_ =~ /([^=]+)=(.*)$/)  {
                  $key=$1; $val=$2;

                  for ($key)  {
                          if (/^dbhost$/i)  { $dbhost=$val; }
                          elsif (/^dbuser$/i) { $dbuser=$val; }
                          elsif (/^dbpass$/i) { $dbpass=$val; }
                          elsif (/^dbname$/i) { $dbname=$val; }
                          elsif (/^start_doc_id$/i) { $doc_id=$val; }
                          elsif (/^start_ent_id$/i) { $ent_id=$val; }
                          elsif (/^start_occ_id$/i) { $occ_id=$val; }
                          elsif (/^disp_server$/i) { $disp=$val; }
                          elsif (/^disp_port$/i) { $port=$val; }
                          elsif (/^cmd$/i) { $cmd=$val; }
                          elsif (/^ris$/i) { $ris=$val; }
                          else { die "Opzione di configurazione
sconosciuta: $key=$val\n"; }
                  }
          }
  }

  close IN;
  return ($dbuser,$dbpass,$dbname,$dbhost,$doc_id,$occ_id,$ent_id,
$cmd,$ris,$disp,$port);
}
```

Listato 2: Funzione Perl per parsare il file .conf del client

e quindi il codice

```
($dbuser,$dbpass,$dbname,$dbhost,$doc_id,$occ_id,$ent_id,$cmd,$ris,
$disp,$port) = getconf;
```

non farà altro che ritornare nelle relative variabili la configurazione del client.

Una breve ma doverosa parentesi sulle **regular expressions** (*regex*). La riga

```
($_ =~ /([^=]+)=(.*)$/)
```

non fa altro che esaminare la riga appena letta dal file e splittarla in due parti, una con tutto ciò che c'è prima dell'uguale, una con tutto ciò che c'è dall'uguale alla fine della riga. In questo modo posso poi esaminare le due parti una per una ed effettuare il parsing del file di configurazione del client. Una delle ragioni per cui ho scelto Perl per questo compito è proprio l'estrema versatilità nella trattazione dei testi, anche grazie al meccanismo delle regex. Per qualsiasi approfondimento sull'argomento regex rimando all'ampia documentazione reperibile su internet (vedi sitografia).

A questo punto, bisogna esaminare la struttura della directory di base in cui sono contenuti i documenti e salvarla in un array, in modo da analizzare di volta in volta i vari file.

```
sub getdir  {
  my $dir=shift or die "Parametro richiesto\n";

  opendir DIR,$dir or die "opendir: $!\n";
  push my @objs,grep { !/^\./ } readdir DIR;

  foreach my $elem (@objs)  {
          push @files,"$dir/$elem" if ( -f "$dir/$elem" );
          getdir ($dir.'/'.$elem)  if ( -d "$dir/$elem" );
  }

  closedir DIR;
}
```

Listato 3: Funzione Perl per scansionare ricorsivamente il contenuto di una directory

Terminata la funzione, l'array @files conterrà l'elenco di tutti i file contenuti nella directory e nelle sue sotto gerarchie. Per ottenere il contenuto di ognuno di questi file basterà un

```
foreach my $elem (@files)  {
   open IN,$elem or next;
   push @content, join('',<IN>);
   close IN;
}
```

Attenzione: quest'ultimo approccio va bene finché si lavora con piccole moli di dati. Quando le dimensioni totali dei documenti sul client cominciano a crescere, quest'approccio rischia solo di saturare la memoria centrale del client con un array grande anche qualche GB, dato che i contenuti di tutti i documenti vengono salvati in un array in memoria, ed è preferibile leggere file per file, inviare il contenuto di ognuno al dispatcher, ricevere le informazioni, salvarle su database e quindi procedere con il successivo.

Si può ora procedere inviando il contenuto di ogni singolo file al dispatcher, che provvederà a sua volta a inoltrarlo al server GSL a cui è associata la risorsa. La richiesta da inoltrare è nella forma standard comprensibile a un server GSL:

```
<id_risorsa>|0|0|<numero_comando>|<contenuto_documento>\r\
n\0
```

Per ogni elemento dell'array @content va quindi eseguito il seguente ciclo:

```
for ($i=0; $i<scalar(@content); $i++)  {
  my $sock = new IO::Socket::INET(
          PeerAddr => $disp,
          PeerPort => $port,
          Proto => 'tcp'
  ) or die "Socket: $!\n";

  my $pack = "$ris|0|0|$cmd|".$dir.'/'.$files[$i]."|".$content[$i]."\
r\n\0";
  print "Analizzo ".$files[$i]."...\n";
  print $sock $pack or die "Errore in send: $!\n";

  my $xml='',$tmp='';

  while (<$sock>)  {
          $xml.=$_;
  }

  $xml =~ s/^(\d+\|){4}//;

  for ($j=0; $j<length($xml); $j++)  {
          if (ord(substr($xml,$j,1)))  {  $tmp.=substr($xml,$j,1);  }
```

```
    }

    chomp($tmp);
    $xml=$tmp;

    close $sock;

    if (length($xml)>1)  {
            $xp=new XML::Parser;
            $xp->setHandlers(
                    Start => \&parse_start,
                    End => \&parse_end,
                    Char => \&cdata
            );

            $xp->parse($xml);
    }
}
```

Listato 4: Ciclo in Perl che legge il contenuto di ogni singolo documento, lo invia al dispatcher via socket, legge la risposta in formato XML ed effettua il parsing

Quella che è probabilmente la funzione cardine del client è *parse_start*, funzione richiamata quando il parser XML incontra un nuovo tag, che effettua il parse vero e proprio del tag XML letto attraverso regex e invia le informazioni appena lette al database.

6.2.Server GSL

Il server GSL è l'entità deputata all'analisi vera e propria del documento non strutturato. Il suo compito è prelevare i testi non strutturati inviati dai client associati al dispatcher, effettuare l'analisi e ritornare al dispatcher l'XML contenente le informazioni richieste (domini del documento ed entità non note, con eventuali supernomen virtuali, predicati e soggetti/complementi associati, e i relativi ed eventuali domini ad essi associati). Per fare ciò il server GSL usa le librerie aziendali del disambiguatore e del sensigrafo, che consentono in modo estremamente comodo ed efficace di costruire un *tree* di disambiguazione e navigarlo in modo intuitivo tramite l'uso di iteratori in stile STL, e inoltre di accedere facilmente alle informazioni del sensigrafo (syncon, lemmi, navigazione delle relazioni di parentela fra diversi lemmi…).

Concettualmente, un server GSL si può schematizzare nel seguente modo:

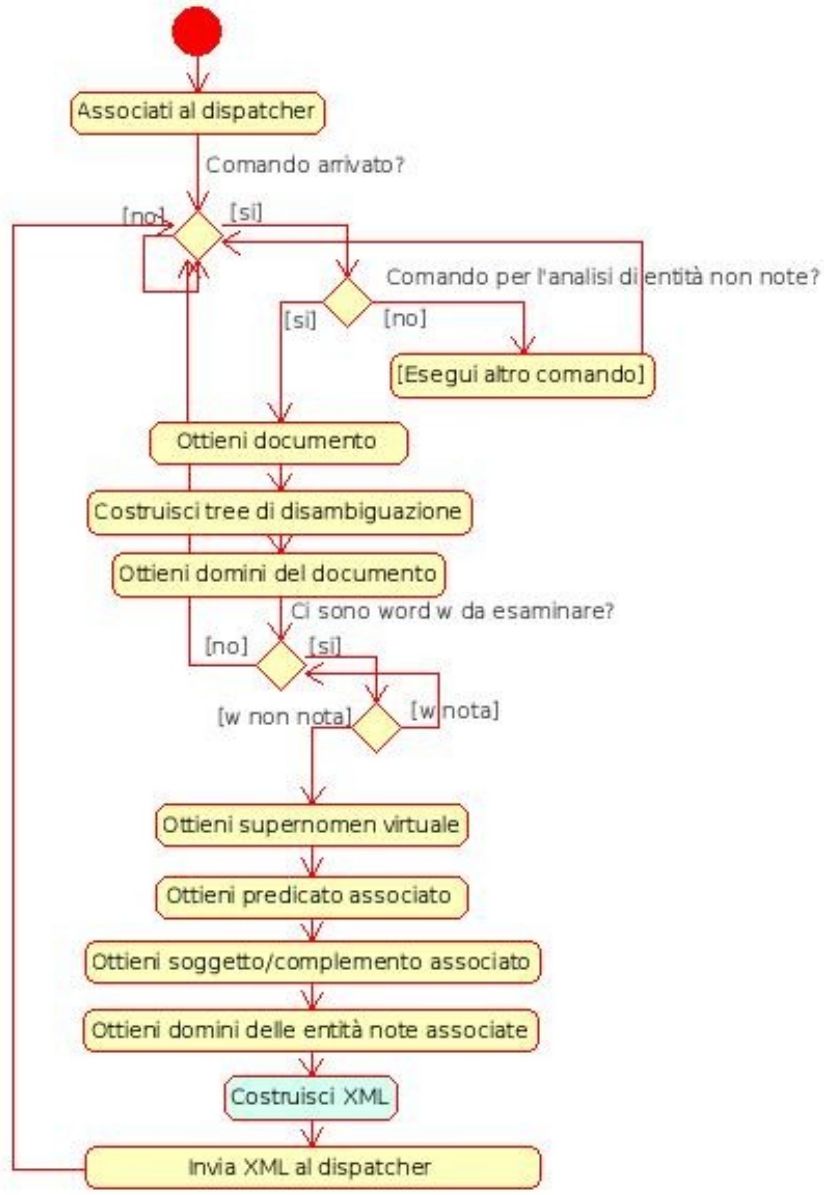

Figura 3: Activity diagram del server GSL

La gestione della sincronizzazione con il dispatcher e i comandi di base sono caratteristiche gestite già dallo scheletro di GSL fornito in principio, e questo ci rende possibile solo l'esame della parte interessante ai fini del presente documento, ovvero la disambiguazione e il riconoscimento di entità non note.

Il riconoscimento dei comandi inviati dal client viene fatto in uno *switch-case* all'interno del main. Possiamo quindi associare alla nostra operazione di riconoscimento delle entità non note un comando numerico all'interno dello switch e richiamare al suo interno la funzione vera e propria. Il messaggio inviato dal client sarà, come abbiamo già visto, del tipo

```
<id_risorsa>|0|0|<numero_comando>|<messaggio>
```

Quello che arriva fin qui è semplicemente il messaggio, dato che le informazioni sull'ID risorsa sono già state "scremate" dal dispatcher e il numero del comando è stato prelevato prima di arrivare allo switch. Il messaggio a sua volta l'ho strutturato come

```
<nome_documento>|<contenuto_documento>
```

in modo che le informazioni sul nome del file non vengano perse e possano poi essere memorizzate nel database. Nel comando quindi provvederò a separare il nome del file dal suo contenuto vero e proprio, per richiamare la funzione vera e propria di disambiguazione su entrambe le entità

```
string msg;   // Messaggio inoltrato dal dispatcher

switch(ss)  {

.. .. .. ..

// Numero di comando arbitrario
case 123:
 // Estrapolo dal messaggio ricevuto il nome del file e il contenuto
      int pos = (int) strstr(msg.c_str(),"|") - (int) msg.c_str() + 1;
      string content=string(msg.c_str()+pos);
      string fname=string(msg.c_str(),pos-1);

  // Invio come risposta l'XML elaborato dalla funzione dodis()
  GSL_SendResponse(dodis(fname,content,dis), STATUS_OK);
      break;
```

dove dodis() è una funzione così definita

```
string dodis (string fname, string msg, Dis &dis);
```

ed è la funzione principale del nostro server GSL, che prende come parametri il nome del documento, il suo contenuto e un riferimento al disambiguatore e ritorna un testo XML strutturato. Ad esempio, prendiamo un testo qualsiasi contenente una quantità arbitraria di termini non noti al sensigrafo, ad esempio un articolo di Wikipedia sul server web Apache Tomcat

Apache Tomcat (o semplicemente Tomcat) è un web container open source sviluppato dalla Apache Software Foundation. Implementa le specifiche JSP e Servlet di Sun Microsystems, fornendo quindi una piattaforma per l'esecuzione di applicazioni Web sviluppate nel

linguaggio Java. La sua distribuzione standard include anche le funzionalità di web server tradizionale, che corrispondono al prodotto Apache.

In passato, Tomcat era gestito nel contesto del progetto Jakarta, ed era pertanto identificato con il nome di Jakarta Tomcat; attualmente è oggetto di un progetto indipendente.

Tomcat è rilasciato sotto licenza Apache Software License, ed è scritto interamente in Java; può quindi essere eseguito su qualsiasi architettura su cui sia installata un JVM.

Listato 5: Documento non strutturato di esempio inviato al server GSL

Fornendo un simile documento al server GSL, l'output della funzione dodis(), e quindi ciò che verrà di fatto inoltrato al dispatcher e poi al client, sarà qualcosa del tipo

```
<?xml version="1.0" encoding="iso-8859-1"?>
<document name="Tomcat">
	<domain name="software" score="13.900000" />
	<domain name="programmazione" score="10.500000" />
	<domain name="internet" score="7.100000" />
	<domain name="hardware" score="1.400000" />
	<domain name="termini tecnici" score="0.300000" />
	<domain name="economia" score="0.200000" />
	<domain name="trasporti" score="0.100000" />
	<domain name="elettronica" score="0.100000" />
	<domain name="architettura" score="0.100000" />

	<entity name="Apache Tomcat" known="false" verb="230285">
	</entity>

	<entity name="Tomcat" known="false">
	</entity>

	<entity name="Apache Software Foundation" known="false" hyperonym="122945"
verb="88961" object="123141">
		<assoc syn="122945" dom1="diritto" score1="50" />
		<assoc syn="123141" dom1="internet" dom2="software" score1="60"
score2="30" />
	</entity>

	<entity name="JSP" known="false" hyperonym="53591" verb="95114" object="363652">
		<assoc syn="53591" dom1="termini burocratici" dom2="commercio"
score1="50" score2="50" />
		<assoc syn="95114" dom1="programmazione" dom2="elettronica" score1="40"
score2="40" />
		<assoc syn="363652" dom1="software" dom2="sistemi operativi" score1="30"
score2="10" />
```

```
      </entity>

      <entity name="Sun Microsystems" known="true" object="25410">
      </entity>

      <entity name="Tomcat" known="false" verb="230285">
      </entity>

      <entity name="Tomcat" known="false">
      </entity>

      <entity name="Tomcat" known="false" verb="230285">
      </entity>

      <entity name="Apache Software License" known="false">
      </entity>

</document>
```

Listato 6: Output XML generato dalla funzione dodis() del server GSL per il documento nel listato 5.2.1

L'XML generato è quindi nella forma

```
<?xml version="1.0" encoding="iso-8859-1"?>
<document name="nome_file">
      <domain name="dominio1" score="punteggio1" />
  <domain name="dominio2" score="punteggio2" />

  <entity name="nome entità" known="true/false" hyperonym="syncon del supernomen
associato" verb="syncon del predicato associato" object="syncon del
soggetto/complemento associato">
              <assoc syn="syncon associato 1" dom1="dominio1" score1="score1" />
              <assoc syn="syncon associato 2" dom1="dominio2" score1="score2" />
      </entity>
</document>
```

Listato 7: Forma base del documento XML generato dalla funzione *dodis()*

I tag principali:

- Tag **document**, attributo *name*: nome del file in questione
- Tag **domain** (un documento può avere uno o più domini associati)
  - Attributo *name*: nome del dominio associato
  - Attributo *score*: punteggio del dominio associato (indica l'attinenza del dominio ai contenuti del documento). Il punteggio di un dominio viene calcolato in base ai lemmi contenuti all'interno del documento

stesso. All'interno del sensigrafo esiste infatti una lista di domini associati a ogni lemma (ad ogni lemma vengono in particolare associati due domini con relativa attinenza). Lo *score* del dominio *i-esimo* del documento viene quindi calcolato come somma degli *score* dei lemmi attinenti a quel dominio sulla somma di tutti gli score "parziali" di tutti i domini riferiti ai lemmi nel documento:

$$s_i = \frac{\sum_j^N w_j}{\sum_k^M d_k | l_j}$$

Dove s_i è lo *score* del dominio *i-esimo* del documento, N è il numero di lemmi associati a quel dominio all'interno del documento, w_j lo *score* di attinenza del lemma *j-esimo* al dominio d_i, M il numero totale di domini presenti nel documento, e $d_k|l_j$ lo score del dominio d_k riferito al lemma l_j. Se ad esempio nel mio documento su un totale di 100 lemmi ho 20 di questi associati al dominio *informatica*, ognuno con uno *score* di attinenza al dominio del 50%, il documento ha uno *score* del 10% di attinenza al dominio *informatica* (in genere è difficile incontrare documenti con *score* di attinenza molto elevati).

- Tag **entity**
  - Attributo *name*: nome dell'entità non nota al sensigrafo
  - Attributo *known: true* o *false* a seconda che l'entità sia nota o meno al sensigrafo. Per finiti di debug sul sensigrafo infatti il GSL segnala sia le entità non note, sia le entità note composte da due o più parole. Ovviamente, le uniche entità ad essere considerate, e quindi incluse nell'XML, sono quelle considerate dal sensigrafo come nomi propri.
  - Attributo *hyperonym*: syncon dell'iperonimo (*supernomen*) virtuale associato all'entità. Quando possibile, il sensigrafo cerca di assegnare un genitore anche a entità non note, in base al contesto in cui queste vengono utilizzate. Se tale attributo esiste (e spesso può rivelarsi un'informazione preziosa per la classificazione dell'entità non nota), viene salvato nell'XML.
  - Attributo *verb*: syncon del predicato associato all'entità, se esiste.

- Attributo *object*: syncon del soggetto/complemento associato all'entità, se esiste.
- Tag **assoc**
  - Attributo *syn*: syncon dell'entità (soggetto/predicato/complemento/supernomen) associata all'entità non nota.
  - Attributo *dom1*: nome del dominio primario associato all'entità in questione, se esiste.
  - Attributo *score1*: punteggio del dominio primario dell'entità (se esiste), che ne indica il grado di pertinenza.
  - Attributo *dom2*: nome del dominio secondario associato all'entità in questione, se esiste.
  - Attributo *score2*: punteggio del dominio secondario dell'entità (se esiste), che ne indica il grado di pertinenza.

Possiamo ora cominciare a esaminare dall'interno la funzione dodis().

La prima cosa che fa la funzione è inizializzare il disambiguatore, quindi sfrutta quest'ultimo per costruire il *tree*. Il tree è una stringa pseudo-binaria costruita a partire dal testo da disambiguare, contenente al suo interno le proprietà e le relazioni logiche del testo stesso. E' poi possibile scorrere tutti questi oggetti all'interno del tree attraverso degli iteratori in stile *STL* (*Standard Template Library*). Le librerie mettono infatti a disposizione degli oggetti che ereditano le caratteristiche degli iteratori della *STL* del C++ e che consentono di navigare le proprietà logico-grammaticali del documento. Ad esempio, per navigare i domini basta un codice del genere:

```
string xml("");
Tree tree;
vector<string> domains;
vector<float> scores;

.. .. .. ..

getDocumentDomains(tree,domains,scores);

for (dom_iterator d=domains.begin(), float s=scores.begin();
       d!=domains.end(), s!=scores.end();
       d++, s++)  {
  char tmp[1000];
  sprintf_s (tmp,sizeof(tmp),
                 "\t<domain name=\"%s\" score=\"%f\" />\n",
                 d->c_str(),
```

```
                *s
            );
  xml.append(tmp);
}
```

Listato 8: Codice di base per ottenere i domini di un documento

dove `getDocumentDomains()` è una funzione utilizzata per estrapolare i domini del documento, dato il tree e il riferimento a un vettore di string ed uno di float su cui salvare, rispettivamente, i nomi dei domini del documento e i loro relativi *score* che ne indicano l'attinenza al contenuto effettivo. Il codice è relativamente semplice:

```
void getDocumentDomains(Tree tree, vector<string> &domains,
vector<float> &scores)  {
          for (dom_iterator i=tree.dom_begin(); i!
=tree.domain_end(); i++)  {
                domains.push_back(i.text());
                scores.push_back((float) i->GetFreq());
          }
     }
```

Listato 9: Funzione che effettua il salvataggio su due array dei domini del documento con relativi score

Ora si procede con uno scorrimento sulle singole *word* del tree. Una word può ovviamente essere composta anche da più parole: il sistema adotta come policy di default il raggruppamento massimo dell'insieme di sostantivi consecutivi appartenenti allo stesso gruppo logico, quindi navigando con le word sul tree in genere non si corre il rischio di trovarsi con entità "spezzettate".

```
// Itero su tutte le word del tree
for (w_iterator wi=tree.w_begin(); wi!=tree.w_end(); wi++)  {
                bool flag; bool is_ok=false;
                string ent;

                // Controllo se la word in questione è identificata
                // come nome proprio...
                if ((wi->Type() == Word::NPR))  {
                      // ...e non è conosciuta dal sensigrafo
                      // (synset/syncon < -1 e ricerca nel
sensigrafo
                      // dell'entità che ritorna NULL)
                      if ((wi->GetSynset()<-1) && (!
checkEntity(wi.text(),dis)))  {
                            // In questo caso, setto il flag
dell'entità
                            // come UNKNOWN_ENT. Il flag booleano
is_ok
                            // mi indica che la word in questione è
```

```
                              // candidata a essere esaminata come
entità
                              // non nota
                              ent=wi.text();
                              flag=UNKNOWN_ENT; is_ok=true;

                        // Per fini di debug del sensigrafo, considero
                        // anche le entità note, classificate come
nomi
                        // propri dal sensigrafo e composte da due o
più
                        // parole (in questo caso ovviamente
all'interno
                        // della word troverò almeno uno spazio)
                        } else if (wi.text().find(' ')!=string::npos)
{
                              // In questo caso, setto il flag
dell'entità
                              // come KNOWN_ENT, e is_ok sempre a true
                              ent=wi.text();
                              flag=KNOWN_ENT; is_ok=true;
                        }
                  }

.. .. .. ..
```

Listato 10: Codice per verificare se una word ha le "carte in regola" per essere considerata dall'applicazione

Il passo successivo è quello di includere l'entità nell'XML se il flag *is_ok* è settato a *true*.

```
if (is_ok)  {
                        char tmp[1024];
                        ent=sanitize_xml(ent.c_str());

                        sprintf_s (tmp,sizeof(tmp),
                              "\t<entity name=\"%s\" known=\"%s\"",
                              ent.c_str(),
                              (flag==UNKNOWN_ENT) ? "false" : "true"
                        );

                        xml.append(tmp);
                     memset (tmp,0x0,sizeof(tmp));
```

Listato 11: Codice per verificare aggiungere all'XML le informazioni di base su un'entità

Ora bisogna verificare se la word in questione possiede un iperonimo (supernomen) virtuale assegnatole dal sensigrafo in base al contesto:

```
string doment="";

if ((int hyp=getHyp(wi))!=-1 && hyp)  {
```

```
                              sprintf_s (tmp,sizeof(tmp),
                                    " hyperonym=\"%d\"",
                                    hyp
                              );

                              xml.append(tmp);
                              memset (tmp,0x0,sizeof(tmp));
                              doment.append(getEntDomains(hyp,dis));
}
```

Listato 12: Codice per ottenere l'eventuale syncon del supernomen virtuale dell'entità

La funzione `getEntDomains()`, utilizzata per tutte le entità associate ad un'entità non nota, serve a ottenere, interrogando il sensigrafo, i domini (al massimo due) di un dato syncon, e ritorna una riga XML già pronta come oggetto *string* con le relative informazioni (ritorna un tag di tipo *assoc* come quelli visti in precedenza). Prende come parametri un intero che indica il syncon dell'entità e un riferimento al disambiguatore.

```
string getEntDomains (int synset, Dis &dis)  {
            char ins[300];

            // Riferimento al sensigrafo
            QNet *net = dis.GetNet();
            SYNSET syn;

            // Ottengo, passando per il sensigrafo, le informazioni
            // sul synset passato come intero
            net->getsyncopy(synset,&syn);

            // Se esiste un dominio numerico associato al synset !=
0 ...
            if ((unsigned dom1=syn.iCategory))  {
                  // Artificio usato per ottenere il valore numerico
del
                  // dominio (dom1) e il suo relativo score (val1),
quindi
                  // via GetDom() ottengo la stringa associata a dom1
                  dom1--;
                  unsigned val1=syn.iCategoryVal/2;
                  string dom1_string=dis.GetDom(dom1);
                  unsigned dom2=syn.iSecondaryCategory;

                  // Stessa procedura anche con l'eventuale secondo
dominio
                  if (dom2)  {
                        dom2--;
                        unsigned val2=syn.iSecondaryCategoryVal/2;
                        string dom2_string=dis.GetDom(dom2);

                        sprintf (
                              ins,
                              "\t\t<assoc syn=\"%d\" dom1=\"%s\"
dom2=\"%s\" score1=\"%d\" score2=\"%d\" />\n",
                              synset,
```

```
                              dom1_string.c_str(),
                              dom2_string.c_str(),
                              val1,
                              val2
                        );
                  } else {
                        sprintf (
                              ins,
                              "\t\t<assoc syn=\"%d\" dom1=\"%s\"
score1=\"%d\" />\n",
                              synset,
                              dom1_string.c_str(),
                              val1
                        );
                  }

                  return string(ins);
            } else
                  return string("");
}
```

Listato 13: Funzione che consente di ottenere i domini di un'entità, ritornando una riga XML

Ora non rimane che ottenere le entità legate logicamente a quella non nota (soggetto/oggetto, predicato). Per fare ciò il codice è il seguente:

```
// Iteratore sul gruppo a cui appartiene l'iteratore sulle word wi.
// Si noti che è possibile in qualsiasi momento passare da un
iteratore
// con risoluzione maggiore sul tree ad uno con risoluzione minore,
// e il cast viene effettuato automaticamente, quindi è ad esempio
// possibile passare da un iteratore sulle word ad uno sui gruppi,
così
// come da un iteratore sui gruppi ad uno sui paragrafi.
g_iterator gi=wi;

// Uso i link iterator, iteratori sulle relazioni logiche. Con il
primo
// prelevo il gruppo a cui il gruppo di wi è logicamente collegato
// (link_to()), con il secondo quello a cui il gruppo si collega
// logicamente (link_from())
link_iterator_pair lit=gi.link_to();
linked_iterator_pair lif=gi.link_from();

bool has_verb=false,has_ref=false,has_obj=false;

// Navigo il link_iterator_pair esaminando i due gruppi che
// ne fanno parte
for (link_iterator li=lit.first; li!=lit.second; li++)  {
     w_iterator ref;
     w_iterator self=getWord(li.myself());

  // Se esiste una relazione logica a cui il gruppo si riferisce,
  // ovvero il gruppo logico è diverso da group_end()...
     if (li.to()!=tree.group_end())  {
       // ...ottengo il verbo all'interno del gruppo logico
```

```
            word_iterator verb=getVerb(li.to());

       // Se il verbo esiste nel gruppo...
            if (verb != li.to().word_end())  {
                  // ...lo scrivo nell'XML, associato all'entità a cui
                  // si riferisce
                  sprintf_s (tmp,sizeof(tmp),
                        " verb=\"%d\"",
                        verb->GetSynset()
                  );

                  xml.append(tmp);
                  memset (tmp,0x0,sizeof(tmp));
                  has_verb=true;
                  dоment.append(getEntDomains(verb->GetSynset(),dis));
            }
      }

      // Itero anche sul linked_iterator_pair
      if (lif.first.from()!=tree.group_end())  {
            // L'iteratore su gruppo "referer" contiene il gruppo
            // a cui l'entità si lega logicamente
            g_iterator referer=lif.first.from();

      // Itero sul gruppo
  for (word_iterator wi=referer.word_begin();
                  wi!=referer.word_end();
                  wi++)
            // Se la word in questione è un sostantivo o un
            // nome proprio (i casi che mi interessano...)
                  if (wi->GetType()==Word::SOS ||
                        wi->GetType()==Word::NPR)  {
                  // ...lo memorizzo come mittente o destinatario
                  // del legame logico, e setto il flag has_ref
                  // a true
                        ref=wi;
                        has_ref=true;
                  }
      }

      // Esamino la casistica ottenuta dall'analisi, e se l'oggetto
ref
      // è valido (è conosciuto dal sensigrafo, oppure ha supernomen
non
      // negativo), allora lo scrivo nell'XML
      if (has_ref)  {
            if (ref->GetSynset()<0)  {
                  if (getHyp(ref)<0)  {
                        obj=0;
                        has_ref=false;
                  } else
                        obj=getHyp(ref);
            } else obj=ref->GetSynset();
      } else  { obj=0; has_ref=false; }

      if (has_ref && obj && !has_obj)  {
            sprintf_s (tmp,sizeof(tmp),
                  " object=\"%d\"",
                  obj
            );
```

```
            xml.append(tmp);
            memset (tmp,0x0,sizeof(tmp));
            has_obj=true;
            doment.append(getEntDomains(obj,dis));
      }
}

xml.append (">\n" + doment + "\t</entity>\n\n");
```

Listato 14: Codice che consente di ottenere i legami logici (soggetto/predicato/complemento) di un'entità per poi ritornare l'informazione in formato XML

L'XML è ora pronto e contiene tutti i dati richiesti, e può essere ritornato al dispatcher in risposta al testo non strutturato.

6.3.Dispatcher

Il server GSL è un'entità software volutamente *stateless* e poco thread-oriented. L'entità server GSL semplicemente riceve richieste dal dispatcher, le elabora e invia le risposte al dispatcher: sia al client, sia al server GSL, ai due lati del dispatcher, è completamente celato il meccanismo di comunicazione fisica e scambio dei dati su socket. Semplicemente, il client identifica internamente una risorsa accessibile attraverso il dispatcher, corrispondente al server GSL, ignorando completamente il suo indirizzo di rete, la porta sulla quale è in ascolto, e anche il numero stesso di server a cui la sua richiesta verrà inoltrata (nulla mi impedisce di deputare più server GSL alla risoluzione di una data richiesta, associandogli lo stesso numero di risorsa). Allo stesso modo, un server GSL è completamente all'oscuro dei meccanismi della comunicazione fisica per inviare i dati al client. L'unica informazione di cui è in possesso è l'identificativo della risorsa mittente inoltrato dal dispatcher, e inoltrerà qualsiasi risposta al dispatcher indicando quello come destinatario. Il nodo dell'architettura è quindi il dispatcher, che offre all'architettura un elevato livello di astrazione, ma che è anche notevolmente complesso, thread-oriented, e si occupa anche di gestire lo stato e il tipo della comunicazione in corso fra le varie entità (è quindi il dispatcher a gestire eventuali comunicazioni *unicast* o *multicast*, in base al comando ad esso inviato o alle policy di configurazione, così come si occupa anche di "spegnere" una risorsa remota, ovvero renderla non più raggiungibile, in caso di timeout).

L'esame dell'entità software "dispatcher" quindi è particolarmente complesso ed esula dagli scopi del presente documento. Tuttavia, è doverosa una breve

infarinatura sul protocollo che ne è alla base, per una maggiore comprensione dell'architettura che stiamo esaminando.
Una volta avviato, il dispatcher entra in listen su 3 porte TCP. Di queste

- Due porte sono utilizzate dai client. Di default, viene usata la 5500 per le connessioni standard e la 5400 per le connessioni permanenti. La differenza si può assimilare alla differenza fra l'header "Connection: close" e "Connection: keep-alive" nell'HTTP.
- Una porta è utilizzata per le risorse connesse (di default la 5000)

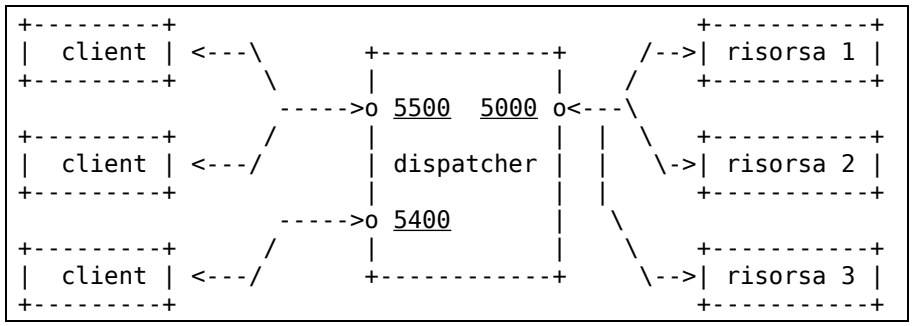

Figura 3a: Struttura di base di un'architettura client-dispatcher-risorse

Ogni messaggio scambiato fra le varie entità della rete è uno stream di byte terminante sempre con il carattere ASCII '\0'. Ovviamente, non è ammissibile la presenza del carattere ASCII '\0' se non alla fine del messaggio, in quanto la sua presenza indicherebbe la fine prematura del messaggio da processare. Quando un client vuole richiedere un determinato servizio ad una risorsa, inoltra al dispatcher il messaggio corrispondente, che garantisce l'inoltro alla risorsa giusta. Affinché il giro sia completo, il protocollo prevede che ad ogni messaggio del client il dispatcher attenda sempre almeno un messaggio di risposta da parte della risorsa in questione, che sia anche un semplice ACK. In caso di problemi il dispatcher inoltra al client un messaggio di errore, e anche il timeout della connessione è un parametro gestito dal dispatcher. Il valore di tale timeout, così come anche le porte su cui il dispatcher rimane in listen e le interfacce di rete su cui effettua il *bind*, sono tutti parametri specificati nel file .ini del dispatcher stesso.

Il protocollo dei messaggi scambiati nell'architettura è stato già brevemente accennato. Il client inoltrerà al dispatcher un messaggio del genere

```
IDR|0|0|NC|DatiIn
```

Dove *IDR* è l'ID della risorsa a cui richiedere il servizio e *NC* il numero del comando. A tale messaggio la risorsa, se raggiungibile, risponderà con un messaggio del tipo

```
IDI|0|E|0|DatiOut
```

dove *IDI* è l'ID interno con cui il dispatcher identifica la risorsa ed *E* è l'eventuale codice di errore (0 nel caso in cui non ci sia alcun errore).

Il client può anche effettuare richieste in *multicast* attraverso il dispatcher. In questo caso il pacchetto inviato dal client avrà una struttura del tipo

```
IDR|4|0|NC|DatiIn
```

e la risposta sarà

```
IDI|0|E|NUM|
```

dove *NUM* è il numero di risorse a cui è stata inoltrata la richiesta. E' anche possibile per il client effettuare una richiesta in multicast ed esaminare singolarmente le risposte dei GSL:

```
IDR|5|IDI|NC|DatiIn
```

La risposta sarà qualcosa del tipo

```
IDI|0|E|NUM|Len 1|Error 1|Data out 1|..| Len n | Error n | Data out n
```

Dove *Len i* identifica la lunghezza del troncone del GSL i-esimo, *Error i* identifica il suo codice di errore e *Data Out i* il suo messaggio di risposta.

La gestione del timeout, come accennato, è in genere fatta dal dispatcher, e il tempo di timeout è un parametro di configurazione del dispatcher stesso. Tuttavia, può anche essere lo stesso client a stabilire il timeout per la risoluzione di una data richiesta, inviando al dispatcher una richiesta così strutturata

```
IDR|6|TIMEOUT|NC|DatiIn
```

La risposta sarà nel formato canonico

```
IDI|0|E|0|DatiOut
```

Piccola nota: il timeout gestito dal dispatcher non è inteso come tempo in cui viene data una risposta da una risorsa ad una determinata richiesta, ma come tempo massimo entro il quale il GSL può prendere in carico la richiesta. Il dispatcher è dotato anche di una piccola cache interna per la ricezione dei messaggi, e se in un dato istante un client dovesse inoltrare una richiesta ad una risorsa non raggiungibile in quel determinato momento, tale richiesta verrà memorizzata nella cache e inoltrata al GSL una volta che esso tornerà ad associarsi al dispatcher.

6.4.Database

Il database è di importanza cruciale nella struttura, in quanto tutti i dati elaborati dal sistema client-dispatcher-GSL verranno poi salvati su di esso per effettuare elaborazioni e analisi statistiche a posteriori. Nello stage è stato utilizzato come DBMS MySQL, con cui ci si è interfacciati sia attraverso le API standard per il C fornite su http://dev.mysql.com (per i test in locale dell'applicazione), sia attraverso il modulo Net::MySQL di Perl, per la gestione della base dati lato client. Quindi tutte le query riportate di seguito, ed eventuali "scostamenti" dalla sintassi SQL standard, fanno riferimento alla sintassi MySQL.

Le informazioni notevoli da salvare sul database sono:

- Nomi dei documenti salvati e relativi ID
- Domini di ogni singolo documento e relativi score
- Entità richieste, in riferimento al documento in cui sono state trovate, con un flag booleano che indica se l'entità è non nota oppure è un'entità nota classificata come nome proprio e composta da due o più parole

- Entità logicamente associate ad ognuna delle entità non note, con relativo tipo logico di riferimento (supernomen virtuale/soggetto/predicato/complemento)
- Domini delle entità logicamente legate a quelle non note, se esistono

Si può quindi progettare la base di dati nel seguente modo:

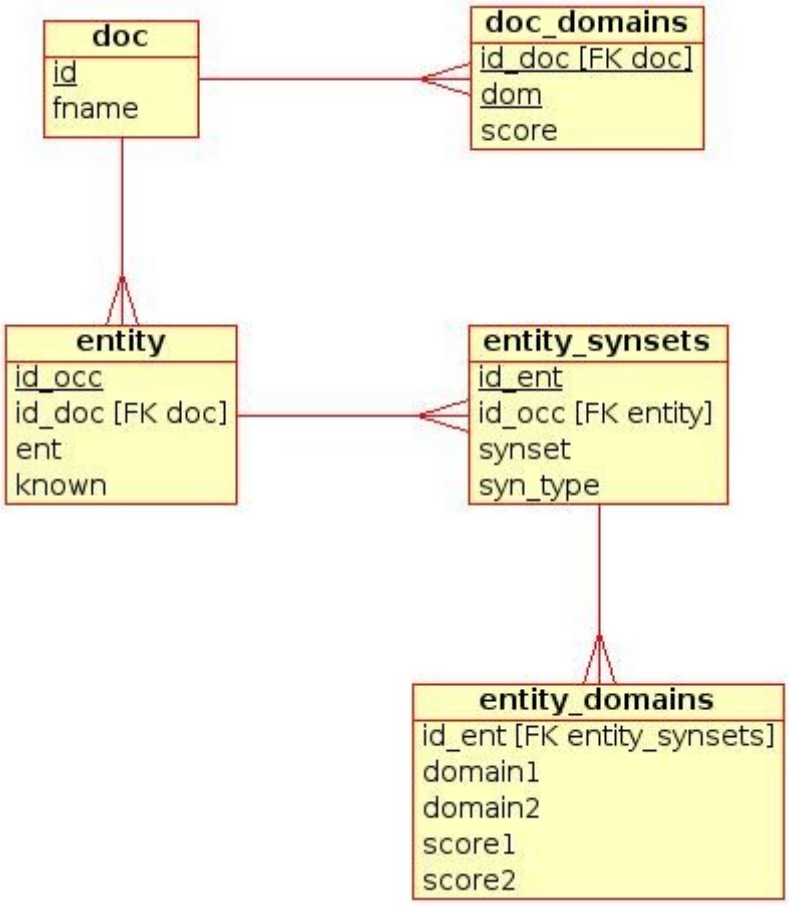

Figura 4: Schema del database

In SQL:

```
DROP DATABASE IF EXISTS synsets;
CREATE DATABASE synsets;
USE synsets;

CREATE TABLE doc(
   id     integer primary key,
   fname  varchar(255) unique not null  -- chiave alternativa
);
```

```
CREATE TABLE doc_domains(
   id_doc  integer,
   dom     varchar(40),
   score   real,
   foreign key(id_doc) references doc(id),
   primary key(id_doc,dom)  -- più domini per ogni documento
);

CREATE TABLE entity(
   id_occ  integer primary key,
   id_doc  integer,
   ent     varchar(255) binary,
   known   boolean,
   foreign key(id_doc) references doc(id)
);

CREATE TABLE entity_synsets(
   id_ent          integer primary key,
   id_occ          integer,
   synset          integer,
   syn_type        integer,
   foreign key(id_occ) references entity(id_occ) on delete cascade
);

CREATE TABLE entity_domains(
   id_ent          integer primary key,
   domain1         varchar(255),
   domain2         varchar(255),
   score1          integer,
   score2          integer,
   foreign key(id_ent) references entity_synsets(id_ent) on delete
cascade
);
```

Listato 15: File SQL per la generazione della base di dati

Per generare il database:

```
mysql –u<utente> -p<password> nome_db < database.sql
```

La distinzione fra le tabelle *doc* e *doc_domains* può apparire ridondante (dopotutto si potevano tranquillamente “joinare” e accorpare anche l’attributo *fname* nella stessa tabella), ma indispensabile per mantenere la base di dati in *seconda forma normale*. Se avessi infatti creato una sola tabella di questo tipo

```
CREATE TABLE doc(
   id      integer
   fname   varchar(255) unique not null
   dom     varchar(40),
   score   real,
   primary key(id,dom)
);
```

avrei avuto la tabella *entity* che avrebbe avuto come foreign key *id_doc* in riferimento a *doc*. Ciò è una chiara violazione della seconda forma normale, in quanto rappresenterebbe una dipendenza funzionale da parte della chiave primaria. *id_doc*, *id_occ* e *id_ent* vengono gestiti dal client, dove come visto in precedenza i valori iniziali per questi 3 ID sono memorizzati nel file di configurazione e incrementati ad ogni nuovo inserimento.

L'attributo *syn_type* nella tabella *entity_synsets* identifica il legame logico fra l'entità non nota e quella ad essa associata, ed è definito per convenzione così:

```
#define          HYP_T  0    // Supernomen/iperonimo virtuale
#define          VER_T  1    // Predicato associato
#define          OBJ_T  2    // Soggetto o complemento
associato
```

Infine, nelle tabelle *entity_synsets* ed *entity_domains*, legate direttamente o indirettamente alla tabella *entity* tramite vincoli di foreign key, i vincoli di foreign key hanno come policy quella di *on delete cascade*. Questo perché ci si ritrova spesso a fare rimozione di rumore a posteriori sul database, ovvero rimozione di entry nella tabella *entity* che sono state erroneamente considerate come entità valide. Mi è capitato ad esempio di notare il salvataggio su database di entità come "La", o "Il", classificati spesso dal sensigrafo come entità non note a se stanti. Ad esempio, nella frase "Il barbiere di Siviglia" l'articolo "Il" iniziale viene classificato come entità a se stante e non nota dal sensigrafo, in quanto è seguito da una parola minuscola che a sua volta viene classificata come un'altra entità. E questo è solo un esempio di inserimenti errati che possono avvenire nel database. Tali entry possono falsare i risultati delle statistiche, e quindi vanno rimossi a mano una volta terminato l'inserimento. Questa rimozione però non sarebbe concessa, in quanto ci sono due tabelle direttamente o indirettamente legate alla tabella *entity* da vincoli di foreign key. Diventa possibile solo specificando una policy di tipo *on delete cascade* per le foreign key in questione nelle due tabelle.

7. CONSIDERAZIONI SUL CODICE

Noto finalmente il modo in cui procedere per l'estrazione e l'estrapolazione di entità non note da corpus di documenti non strutturati, vediamo ora come rendere il codice più affidabile e robusto. Prima però riprendiamo in mano un argomento lasciato da parte in precedenza, ora che sono stati approfondite le nozioni necessarie per una buona comprensione: la funzione *parse_start* del client in Perl.

7.1.Parsing e salvataggio su database

La funzione *parse_start* del client è cruciale per il trattamento dei dati inviati dalla risorsa e la loro memorizzazione. La sua trattazione non è stata fatta in precedenza in quanto la comprensione del funzionamento è possibile solo dopo aver esaminato la struttura del documento XML e del database in cui verranno memorizzate le informazioni.

La funzione è richiamata attraverso il modulo XML::Parser dal codice Perl quando viene incontrato l'inizio di un nuovo tag nel codice XML.

```
$xp=new XML::Parser;
$xp->setHandlers(
      Start => \&parse_start,
      End => \&parse_end,
      Char => \&cdata
);

$xp->parse($xml);
```

Il suo compito è quello di effettuare il parsing del tag appena incontrato all'interno del testo strutturato inviato dal server GSL e salvare su database i dati esaminati al suo interno. Veniamo al codice:

```
sub parse_start  {
     my ($parser,$name,%attr) = @_;
      $tag=lc($name);

      foreach my $key (keys(%attr))  {
           $attr{$key} =~ tr/\'/`/;
```

```
            for ($tag)  {
                  my ($hyperonym,$verb,$object) = '';

                   if (/^document$/i)  {
                        if ($key =~ /^name$/i)  {
                              my $file=$attr{$key};
                                    $db->query(
                                          "INSERT INTO doc ".
                                          "VALUES(".++$doc_id.
                                          ",'$file')")
                                       or die $db->get_error_message;
                                 }
                         }

                         if (/^domain$/i)  {
                                 if ($key =~ /^name$/i)  {
                                    $domain=$attr{$key}; }
                                 elsif ($key =~ /^score$/i)  {
                                    $score=$attr{$key}; }

                                 if ($domain && $score)  {
                                       $db->query("INSERT INTO ".
                                        "doc_domains VALUES(".
                                        $doc_id,'$domain','$score')")
                                      or die $db->get_error_message;
                                        $domain=''; $score='';
                                 }
                         }

                         if (/^entity$/i)  {
                                 if ($key =~ /^hyperonym$/i)  {
                                    $hyperonym=$attr{$key}; }
                                 elsif ($key =~ /^object$/i)  {
                                    $object=$attr{$key}; }
                                 elsif ($key =~ /^verb$/i)  {
                                    $verb=$attr{$key}; }
                                 elsif ($key =~ /^known$/i)  {
                                     $known = ($attr{$key} =~
/^true$/i)
```

```
                                        ? 1 : 0;
                                    }

                                    elsif ($key =~ /^name$/i)  {
                                        $entity=$attr{$key}; }

                                if ($hyperonym)  {
                                    $db->query("INSERT INTO ".
                                        "entity_synsets(id_ent,id_occ,".
                                        "synset,syn_type) ".
                                        "VALUES(".++$ent_id.",".
                                        eval{$occ_id+1}.
                                        ",'$hyperonym',$hyp_t)")
                                    or die $db->get_error_message;
                                }

                                if ($object)  {
                                        $db->query("INSERT INTO ".
                                        "entity_synsets(id_ent,id_occ,".
                                        "synset,syn_type) ".
                                        "VALUES(".++$ent_id.",".
                                        eval{$occ_id+1}.
                                        ",'$object',$obj_t)")
                                    or die $db->get_error_message;
                                }

                                if ($verb)  {
                                        $db->query("INSERT INTO ".
                                            entity_synsets(id_ent,".
                                            "id_occ,synset,syn_type) ".
                                            "VALUES(".++$ent_id.
                                            ",".eval{$occ_id+1}.
                                            ",'$verb',$ver_t)")
                                        or die $db->get_error_message;
                                }

                                if ($entity)  {
                                        $db->query("INSERT INTO ".
                                            "entity(id_occ,id_doc,".
                                            "ent,known) ".
```

```
                                        "VALUES(".++$occ_id".
                                        ",$doc_id,'$entity',".
                                        '$known')")
                        or die $db->get_error_message;
                        ($entity,$known,$hyperonym,$verb,$object) =
'';
                            }
                    }
                }
                if (/^assoc$/)  {
                        if ($key =~ /^syn$/)  { $syn=$attr{$key}; }
                        if ($key =~ /^dom1$/)  { $dom1=$attr{$key};
                        if ($key =~ /^dom2$/)  { $dom2=$attr{$key}; }
                        if ($key =~ /^score1$/)  {
                                $score1=$attr{$key};}
                        if ($key =~ /^score2$/)  {
                                $score2=$attr{$key}; }
                        if (($dom1 && $score1) ||
                                ($dom2 && $score2))  {
                            $db->query("INSERT INTO ".
                                "entity_domains(id_ent,domain1".
                                ",domain2,score1,score2) ".
                                "VALUES($ent_id,'$dom1','$dom2',".
                                "'$score1','$score2')")
                            or die $db->get_error_message;
                           ($syn,$dom1,$dom2,$score1,$score2) = '';
                 }
            }
    }
}
```

Listato 16: Funzione parse_start, per effettuare il parsing di un tag dell'XML di risposta del server GSL e salvare i dati estrapolati al suo interno sul database.

La funzione prende come parametri, come standard del modulo XML::Parser, il riferimento all'oggetto parser chiamante, il nome del tag esaminato e un hash contenente le associazioni attributo → valore dell'XML associate al tag. I tag vengono esaminati uno per uno in un *foreach*. Come abbiamo già visto, i tipi di tag validi nel nostro XML sono

- **document**
- **domain**
- **entity**
- **assoc**

Tali tag vengono esaminati uno per uno, viene esaminato ed estrapolato ogni attributo con relativo valore, quindi tali valori vengono di volta in volta scritti su database attraverso delle INSERT INTO. Il software mira all'ottimizzazione delle prestazioni usando una cache interna per le INSERT, dato che tali istruzioni comportano una spesa notevole (aprire ogni volta il descrittore del database ed effettuare una scrittura su disco) e possono essere numerose per le grandi moli di dati gestite. Bisogna quindi evitare che la scrittura su disco diventi il collo di bottiglia del software. Volendo si può configurare il numero di tuple raggiunto il quale va eseguita una sola INSERT serializzata (es. 100), e quando viene raggiunta tale quota nella cache interna l'istruzione viene eseguita.

```
for i in (tuple trovate)
  salva tupla in cache

  if (tuple.size >= soglia)
       INSERT    INTO    tabella    VALUES(tupla1),(tupla2),
       (tupla3),...,(tuplan)
```

Altra strategia per l'ottimizzazione del client può essere quella di tenere il processo di lettura e invio dei documenti presenti sul filesystem, lettura delle risposte del GSL e scrittura dei dati sulla base di dati in 3 processi separati, in modo da incrementare in modo sostanziale le prestazioni grazie al parallelismo dei task.

7.2.Robustezza del codice

E' richiesto che il codice sia robusto e affidabile in ogni condizione. Dovendo agire su corpus di dimensioni notevoli è probabile che l'applicazione si debba ritrovare ad analizzare di tutto (file vuoti, non validi, entità con apici al loro interno o caratteri speciali...). Quindi i controlli da effettuare a monte sugli input non sono mai troppi. In particolare, bisogna assicurarsi la generazione di un

XML sempre valido. Questa è una condizione generalmente sempre soddisfatta: i domini sono stringhe memorizzate all'interno del sensigrafo e senza caratteri speciali al loro interno, e composti al massimo da due o tre parole di tipo solo alfabetico, quindi non è necessario fare controlli aggiuntivi su di essi. Gli attributi object, hyperonym, score ecc. sono invece di tipo numerico (intero o reale), e anche questi non danno problemi. Si potrebbero però avere problemi con i nomi dei file e con le entità, dato che questi parametri non sono controllati direttamente né dal sensigrafo né dalle varie entità dell'architettura. Il caso più noioso è quello in cui tali oggetti siano stringhe contenenti al loro interno i doppi apici ". In tal caso, il documento XML che ne verrebbe fuori non sarebbe valido. Se ad esempio in un certo documento dovessi estrapolare un'entità non nota del genere

Ernesto "Che" Guevara

l'XML che ne verrebbe fuori per tale entità sarebbe qualcosa del tipo

```
<entity name="Ernesto "Che" Guevara" known="false">
</entity>
```

Tale XML non è ovviamente valido, in quanto la chiusura prematura degli apici doppi dovuta alla particolare strutturazione dell'entità fa interpretare al parser *Che* come nome di attributo, e si avrà ovviamente un errore di sintassi XML non valida. Per evitare questo tipo di problemi si può ricorrere ad una soluzione nel codice C++ del server GSL che effettui il parsing di qualsiasi entità o nome di file per sostituire eventuali apici doppi e quindi renderli adatti all'inserimento in codice XML valido:

```
string sanitize_xml (const char *str)  {
      string s;

      for (size_t i=0; i<strlen(str); i++)  {
            if (str[i]=='"') s.append("`");
            else s.append(str+i,1);
      }

      return s;
}
```

Listato 17: Funzione sanitize_xml, per effettuare la sostituzione degli apici doppi in qualsiasi stringa che può essere scritta su un XML

La stessa cosa va effettuata lato client per il salvataggio delle entità su database. Se infatti al client durante il parsing dell'XML arriva un'entità del tipo

l'imbalsamazione del Faraone

la query per l'inserimento di tale entità nel database diventerà qualcosa del tipo

```
INSERT INTO entity(ent,known) VALUES('l'imbalsamazione del Faraone',0)
```

Si nota subito che l'apice singolo contenuto all'interno del nome dell'entità causa una terminazione prematura del valore da scrivere su database, portando nel migliore dei casi alla terminazione del codice del client per sintassi della query SQL non valida, nel peggiore a grandi rischi per la sicurezza dell'applicazione tramite SQL injection. Questa eventualità viene invece evitata nel codice del client, e in particolare nella funzione *parse_start*, con un

```
$attr{$key} =~ tr/\'/`/;
```

che trasforma qualsiasi apice singolo in un apice inverso attraverso regex, evitando quindi problemi come quello appena visto.

7.3.Gestione della memoria centrale

Altra cosa fondamentale quando si trattano grandi moli di dati è evitare anche il più piccolo *memory leak* (ovvero impossibilità di riutilizzare aree di memoria precedentemente utilizzate, spesso causata da una mancata deallocazione delle risorse quando non più necessarie), in quanto anche una sola variabile dinamica allocata e poi non deallocata, se ripetuta in un ciclo for per un milione di documenti, può portare occupazioni eccessive della memoria e anche il crash dell'applicazione. Per evitarlo si è operato su due filoni:

- *Lato server GSL*. Minimizzare l'uso di memoria dinamica, usare quando possibile solo variabili statiche, e ricordare *sempre* di deallocare la memoria allocata quando non serve più. In locale, per fare i test con il

DBMS all'interno del codice C++ stesso, ricordare sempre di richiamare la funzione *mysql_free_result* quando il risultato di una query non serve più. L'esperienza ha insegnato che anche la mancata deallocazione del risultato di una query che ritorna un solo oggetto alle lunghe porta a saturazioni inaccettabili della memoria centrale.

●*Lato client*. La scelta dell'ID di partenza stabilito a priori nel file .conf del client, in luogo di una serie di ID di tipo *auto_increment*, è stata fatta per due motivi. Pur essendo una strategia più pulita e versatile quest'ultima, non è stata adottata perché comporterebbe una query ausiliaria dopo un inserimento per conoscere l'ID dell'ultima entry inserita, attraverso una *SELECT MAX(id)*. Tale query ausiliaria dopo un inserimento comporta alle lunghe un leggero degrado delle prestazioni del sistema. Inoltre, quando si ha a che fare con tante *insert* seguite in genere da altrettante *select* per conoscere l'ID dell'entry appena inserita, è facile dimenticarsi di deallocare la memoria allocata con l'allocazione del risultato della query. E questo può, ancora una volta, portare a memory leak.

7.4.Ottimizzazione delle query

Il database su cui verranno salvati i dati potrà essere, a seconda della mole dei corpus esaminati, anche di dimensioni notevoli. L'analisi delle informazioni su una base di dati non ottimizzata che potenzialmente può contenere al suo interno centinaia di migliaia o milioni di entry è un'operazione che richiede tempi inaccettabili, anche nell'ordine delle ore per una *select*. Una volta terminato l'inserimento dei dati nel database si può ad esempio provare a selezionare le 20 entità più ricorrenti all'interno del corpus con una

```
SELECT ent,known,COUNT(*) AS num
FROM entity
GROUP BY 1
ORDER BY 3 DESC
LIMIT 20
```

Se l'analisi è stata fatta su corpus di grandi dimensioni, come può essere quello composto da intere annate di un mensile o di un quotidiano, o anche un'enciclopedia, non stupiamoci se tale query impiegherà tempi relativamente lunghi per essere portata a termine. Con riferimento a un database riempito

con i dati riferiti alle ultime 15 annate del *Corriere della Sera* e su un DBMS MySQL, i tempi per una simile query (che coinvolge circa un milione di tuple) possono aggirarsi anche intorno alle 2 ore:

```
20 rows in set (1 hour 57 min 45.00 sec)
```

E' allora il caso di studiare una strategia che ci consenta di fare l'analisi dei dati in tempi accettabili. In fase di tirocinio sono state valutate più alternative.

- *Uso di viste*. Associare una vista alle *select* effettuate più spesso, ad esempio

```
CREATE VIEW v AS
SELECT ent,known,COUNT(*) AS num
FROM entity
GROUP BY 1
ORDER BY 3 DESC
LIMIT 20
```

  consente di risparmiare qualcosa nei tempi di parsing della query, ma è scarsamente utile per i tempi di *fetch* dei dati. I tempi di risoluzione della query rimangono praticamente identici.

- *Uso di indici*. Indicizzare le colonne usate maggiormente nella query, sia nell'elenco dei campi da visualizzare, sia nelle condizioni della *select*, porta vantaggi nei tempi di risoluzione della query notevoli, ma solo se la query opera su campi *statici*. Se nella query ci sono funzioni aggregate come *group by*, i dati nella tabella dovranno in ogni caso essere esaminati tutti, e i tempi sostanzialmente non cambiano.
- *Uso di tabelle di appoggio*. Questa si è rivelata la soluzione migliore. Per fini statistici funzioni aggregate come *group by* vengono usate in modo massiccio per selezionare le entry, ad esempio, più ricorrenti, meno ricorrenti, i documenti con più entità ecc. E per questo tipo di query l'indicizzazione sulla tabella si rivela scarsamente utile. Si può però creare una tabella di appoggio nella quale salvare i dati che si desidera analizzare e poi effettuare le rilevazioni solo su questa tabella attraverso query

statiche. Tornando all'esempio di prima, si può creare una tabella così strutturata.

```
CREATE TABLE appoggio (
     id_occ     integer          primary key,
     ent        varchar(255),
     known      boolean,
     num        integer
);
```

e inserire i dati su cui effettuare l'analisi al suo interno con una semplice

```
INSERT INTO appoggio(id_occ,ent,known,num)
SELECT id_occ,ent,known,COUNT(*) AS num
FROM entity
GROUP BY 1
ORDER BY 3 DESC
```

Ora possiamo tranquillamente indicizzare i campi

```
CREATE INDEX on_ent ON appoggio(ent)
CREATE INDEX on_num ON appoggio(num)
```

e vedremo subito che i tempi di risoluzione delle query vengono ridotti drasticamente:

```
SELECT ent,known,num
FROM appoggio
ORDER BY num DESC
LIMIT 20
.......

20 rows in set (0.02 sec)
```

I risultati della query saranno gli stessi avuti prima, ma si è passati da un tempo di circa 2 ore a uno di circa 2 centesimi di secondo.

8. ANALISI DEI DATI

Effettuata l'analisi del corpus, è interessante fare un'analisi dei dati memorizzati sul database per verificare l'attendibilità del metodo implementato. Nel corso dello stage aziendale si è testato l'applicazione su vari corpus, fra cui le ultime annate di mensili scientifici come *Newton* e *Science*, l'enciclopedia *Microsoft Encarta*, l'enciclopedia della storia del cinema e le ultime 15 annate del *Corriere della Sera*.

Cominciamo ad esaminare i risultati dell'analisi su un corpus di dimensioni medie, quale può essere quello rappresentato dalle ultime 5 annate del mensile scientifico Newton (ci riferiamo a un totale di circa 2000 articoli, memorizzati su altrettanti file di testo). Ecco le entità non note al sensigrafo più ricorrenti in tali file:

```
mysql> select ent,known,count(*)
       from entity
       group by 1
       order by 3 desc
       limit 20;
```

ent	known	num
Stati Uniti	1	25
Gran Bretagna	1	25
Science	0	18
XRT	0	15
Hubble	0	13
Philae	0	13
Agenzia Spaziale Europea	0	12
Airbus	0	12
Regno Unito	1	11
A380	0	11
Agenzia Spaziale Italiana	0	10
MetOp	1	10
Cassini	0	10
Infn	1	9
San Francisco	1	9

```
| New York                  |     1 |        9 |
| Huygens                   |     0 |        9 |
| H5N1                      |     0 |        9 |
| INGV                      |     1 |        9 |
| BAX                       |     0 |        8 |
+---------------------------+-------+----------+
20 rows in set (0.02 sec)
```

Potrebbe ad esempio essere interessante sapere in che tipo di documenti all'interno del corpus considerato ricorre maggiormente l'entità *Hubble*. Noteremo risultati abbastanza plausibili:

```
mysql> select dom,count(*)
       from doc_domains d join entity e
       on d.id_doc=e.id_doc and e.ent='Hubble'
       group by 1
       order by 2 desc
       limit 12;

+-------------------------+----------+
| dom                     | count(*) |
+-------------------------+----------+
| aeronautica             |       15 |
| commercio               |       15 |
| astronautica            |       15 |
| economia                |       15 |
| astronomia              |       15 |
| spettacolo              |       11 |
| ingegneria aerospaziale |       11 |
| termini tecnici         |       11 |
| architettura            |       11 |
| diplomazia              |       11 |
| missilistica            |       11 |
| astrofisica             |       11 |
+-------------------------+----------+
12 rows in set (0.00 sec)
```

Discorso simile, ad esempio, per l'entità *Agenzia Spaziale Europea*:

```
mysql> select dom,count(*) as num
       from doc_domains d join entity e
       on d.id_doc=e.id_doc
       and e.ent='Agenzia Spaziale Europea'
       group by 1
       order by 2 desc
       limit 14;

+-------------------------+-----+
| dom                     | num |
+-------------------------+-----+
| fisica                  |  12 |
| astronautica            |  12 |
| ingegneria aerospaziale |  11 |
| astronomia              |  11 |
| economia                |   9 |
| telecomunicazioni       |   8 |
| termini scientifici     |   8 |
| sport                   |   7 |
| termini tecnici         |   6 |
| mitologia               |   6 |
| astrofisica             |   6 |
| diplomazia              |   6 |
| meteorologia            |   6 |
| aeronautica             |   5 |
+-------------------------+-----+
14 rows in set (0.00 sec)
```

Per verificare la correttezza del metodo, possiamo analizzare quali sono i domini delle entità più ricorrenti logicamente associate all'entità non nota *Hubble*:

```
mysql> select domain1,count(*) as num
       from entity_domains d join entity_synsets s join entity e
       on e.id_occ=s.id_occ
       and s.id_ent=d.id_ent
       and ent='Hubble'
       group by 1
```

```
       order by 2 desc
       limit 20;

+--------------+-----+
| domain1      | num |
+--------------+-----+
| astronautica |  14 |
| fotografia   |   1 |
| diplomazia   |   1 |
+--------------+-----+
3 rows in set (0.00 sec)
```

Notiamo quindi la relativa attendibilità dei dati salvati sul database, attendibilità che ovviamente diventa tanto più forte quanto maggiore è la mole di dati trattati. Sulla base di questi presupposti il sensigrafo potrebbe già effettuare automaticamente, partendo dai dati esaminati, e ovviamente con un livello di attendibilità proporzionale al numero di documenti analizzati, ipotesi relativamente verosimili riguardo alla categorizzazione di una qualsiasi entità incontrata all'interno di un corpus di documenti.

Per completezza, questo risulta essere invece il risultato dell'esame sulle ultime 15 annate del *Corriere della Sera* per quanto riguarda le entità più ricorrenti (qui è stata usata la tabella di appoggio indicizzata vista prima per migliorare le prestazioni):

```
mysql> select * from appoggio order by num desc limit 20;

+-------------------+-------+-------+
| ent               | known | num   |
+-------------------+-------+-------+
| Berlusconi        |     0 | 33758 |
| Forza Italia      |     1 | 15867 |
| Di Pietro         |     0 | 15545 |
| Stati Uniti       |     1 | 13412 |
| Scalfaro          |     0 | 12927 |
| Bossi             |     0 | 12634 |
| D'Alema           |     0 | 11763 |
| Dini              |     0 | 11338 |
| New York          |     1 | 10368 |
```

```
| Craxi             |     0 |  9563 |
| Prodi             |     0 |  9399 |
| Andreotti         |     0 |  7645 |
| Ciampi            |     0 |  7222 |
| Cossiga           |     0 |  6935 |
| Eltsin            |     0 |  6822 |
| Bertinotti        |     0 |  6253 |
| Occhetto          |     0 |  5947 |
| Silvio Berlusconi |     0 |  5863 |
| Casa Bianca       |     1 |  5861 |
| Corriere Lavoro   |     0 |  5743 |
+-------------------+-------+-------+
20 rows in set (0.02 sec)
```

Per comprendere la mole di dati con cui abbiamo a che fare in quest'ultimo caso:

```
mysql> select count(*) as numero_entita from appoggio;
+---------------+
| numero_entita |
+---------------+
|        716782 |
+---------------+
1 row in set (0.56 sec)
```

9. SVILUPPI DELL'ATTIVITÀ – PROGETTO OKKAM

9.1 Cos'è Okkam

Il software sviluppato nel contesto del tirocinio è stato, alla fine di quest'ultimo, il punto di partenza per un modulo da integrare nell'ambito del progetto **Okkam**. Okkam è un progetto finanziato dalla Comunità Europea e al quale partecipano i seguenti enti/aziende:

- University of Trento, *Italia* (coordinatore)
- L3S Research Center, *Germania*
- SAP Research, *Germania*
- Expert System, *Italia*
- Elsevier B.V., *Olanda*
- **Europe Unlimited** SA, *Belgio*
- National Microelectronics Application Center (**MAC**), *Irlanda*
- Ecole Polytechnique Fédérale de Lausanne (**EPFL**), *Svizzera*
- DERI Galway, *Irlanda*
- University of Malaga, *Spagna*
- **INMARK**, *Spagna*
- Agenzia Nazionale Stampa Associata (**ANSA**), *Italia*

Obiettivo di Okkam è la realizzazione di un'immensa rete di entità (persone, organizzazioni, luoghi, eventi, prodotti...) decentralizzata, open source e basata su un'immensa knowledge base dalla quale attingere attraverso internet, entro il giugno 2010. Il mio lavoro si è collocato all'interno di questo contesto come modulo per la ricerca e la classificazione di entità di ogni tipo provenienti da qualsiasi tipo di sorgente non strutturata. L'output può essere poi salvato su un database per effettuare il debug o analisi dei dati estrapolati, oppure inviato come XML al server preposto alla memorizzazione.

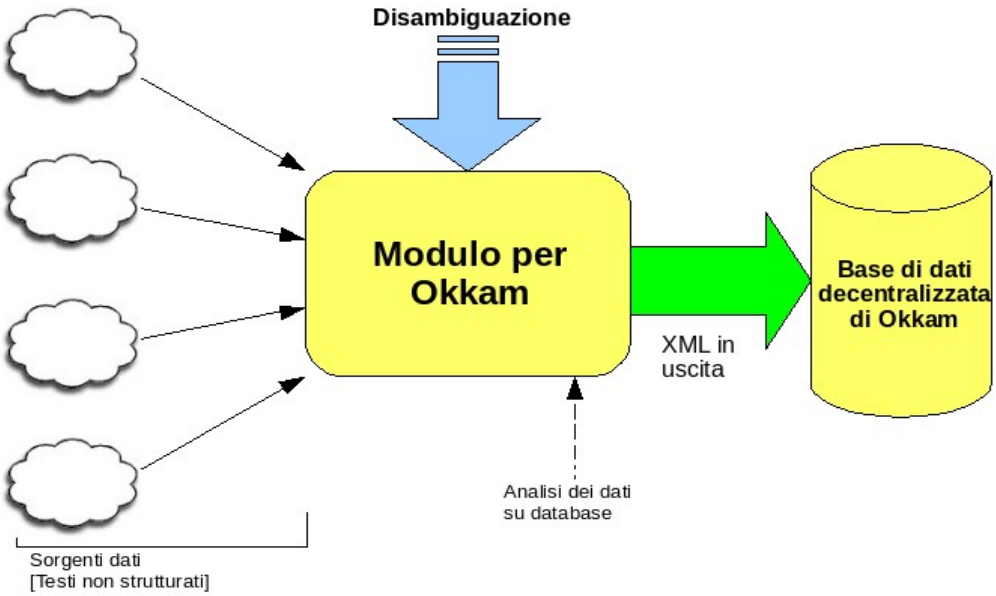

Figura 5: Schema generale del funzionamento del modulo per Okkam

9.2 Definizione di entità non note in base alla correlazione con i lemmi adiacenti

Cambia leggermente anche l'impostazione dell'attività sotto quest'ottica. L'obiettivo non è più estrapolare, trovata un'entità non nota, tutti e solo i lemmi collegati logicamente ad essa, ma, per avere in mano il maggior numero di informazioni possibili vengono salvati tutti i lemmi presenti:

- nella stessa *sentence* dell'entità *e*
- nelle due *sentences* immediatamente adiacenti alla *sentence* di *e*
- nelle due *sentences* a loro volta adiacenti
- ...
- fino a un livello arbitrario *n*

Il legame fra un'entità *e* e un lemma *l* viene pesato sul numero di ricorrenze della coppia *(e,l)* e sul peso di ogni ricorrenza. Il peso della i-esima ricorrenza $(e,l)_i$ viene calcolato in funzione della distanza (misurata in *sentences*) fra *e* ed *l*. Se *e* ed *l* sono nella stessa frase, quindi, avranno distanza nulla, se sono in due frasi adiacenti avranno distanza unitaria, se sono in due frasi a loro volta adiacenti avranno distanza 2, e così via. Da un certo punto in poi non ha più senso considerare la correlazione fra *e* ed *l* se questi due oggetti sono

contenuti in due frasi molto distanti. Anzi, così facendo si rischia solo di introdurre una notevole quantità di *rumore* nel database, sia perché si vogliono inserire con il relativo peso due lemmi scarsamente correlati fra loro, sia perché con l'aumentare delle *sentences* adiacenti da considerare aumenta esponenzialmente il numero di tuple da salvare nel sistema.

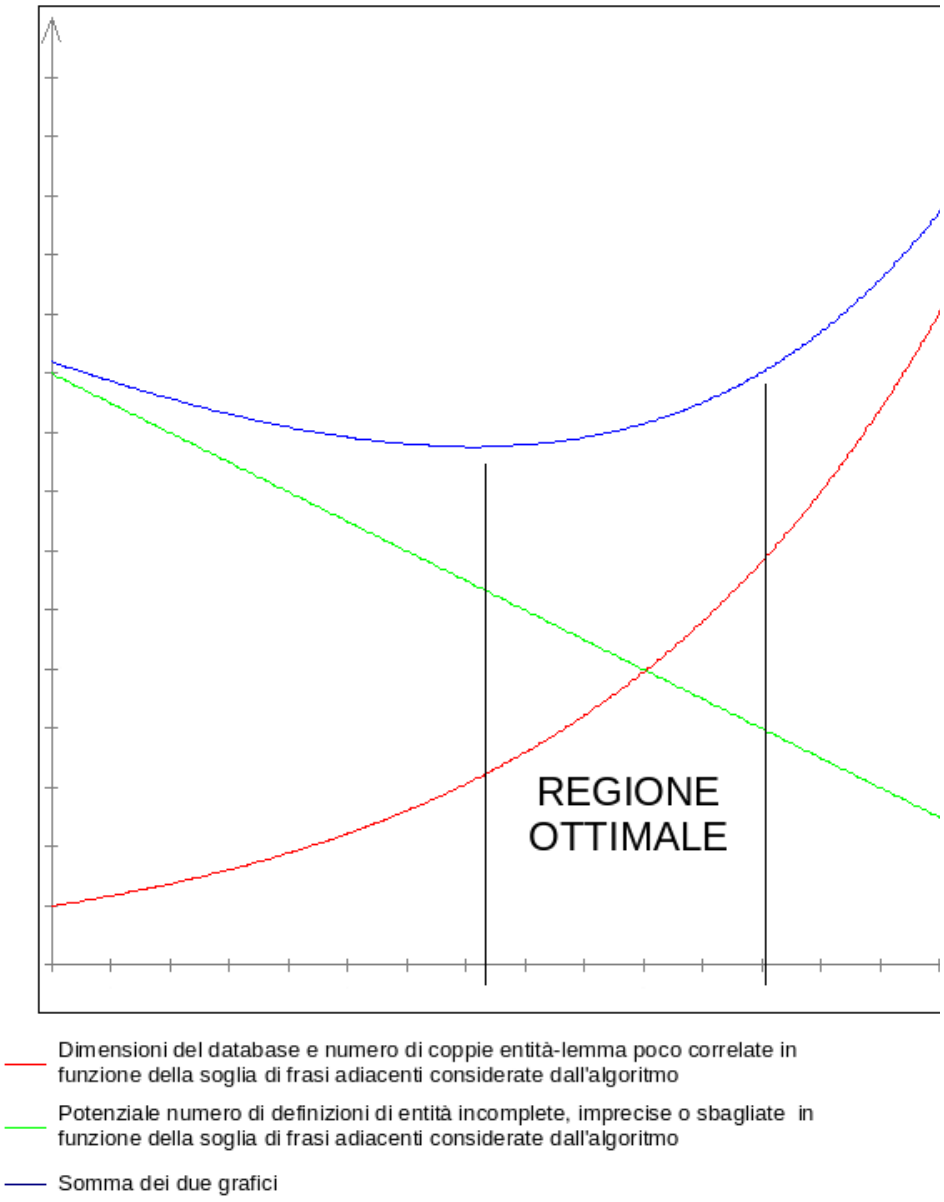

Figura 6: Correlazione fra le dimensioni del database e la quantità di rumore in esso presente e la precisione delle informazioni salvate

In figura è illustrata qualitativamente la correlazione fra due grandezze:

- dimensioni del database, e conseguente numero di coppie poco correlate presenti in esso;

- precisione delle potenziali definizioni date di un'entità sulla base dei lemmi ad essa correlati, misurata come numero di definizioni incomplete, imprecise o erronee.

Si nota anche intuitivamente che considerando un numero maggiore di *sentences* adiacenti per ogni entità, e quindi una potenziale quantità maggiore di informazioni messe da parte per ogni entità non nota, le dimensioni del database crescono in modo notevole (se eseguo l'analisi con una soglia di *sentences* adiacenti da considerare anche solo superiore di un'unità rispetto a quella considerata, il numero di lemmi salvati per ogni entità crescerà in modo notevole, e quindi il legame fra la soglia in questione e le dimensioni della base di dati si può considerare di tipo esponenziale), e allo stesso tempo con le dimensioni del database cresce sia il numero di coppie entità-lemma poco correlate, e quindi il *rumore* del database, che il tempo che l'algoritmo impiegherà prima di terminare. Tuttavia, con una soglia relativamente elevata il numero di potenziali definizioni date di un'entità (le definizioni sono date sulla base dei lemmi ad essa associati) che si rivelano incomplete, imprecise o erronee decresce notevolmente, in quanto aumenta la quantità di informazioni in mano per ogni entità. Un compromesso utile si può trovare minimizzando la somma di questi due grafici, e trovando quindi un range di soglia ottimale. Se volessimo esprimere qualitativamente la correlazione fra il livello di profondità dell'algoritmo e le dimensioni del database (e di conseguenza anche i tempi di esecuzione richiesti) come $D(n)$, la correlazione fra il livello di profondità e il numero di definizioni incomplete o erronee come $E(n)$, e la loro somma per un dato valore di n come $S(n) = D(n) + E(n)$, avremmo che il livello ottimale di profondità da considerare è dato dalla seguente relazione:

$$S'(n) = D'(n) + E'(n) = 0$$

È stato constatato con dei test che si ottiene un buon compromesso fra dimensione/tempi/rumore e quantità di informazioni/basso tasso di errori è dato da un livello di profondità pari a 2 o 3 (ovvero, considerando i lemmi nella stessa *sentence* dell'entità, quelli nelle due *sentences* adiacenti, e quelli nelle due *sentences* a loro volta adiacenti).

9.3 Misura del grado di correlazione per una coppia entità-lemma

Come accennato, il grado di correlazione per una specifica occorrenza di una coppia entità-lemma si misura in funzione della distanza in *sentences* fra questi due oggetti. Il legame è di tipo esponenziale:

$$w_i(e,l)=2^{-d_i}$$

dove $w_i(e,l)$rappresenta il peso dell'i-esima occorrenza della coppia *(e,l)* e d_i la loro distanza in sentence, ed è un numero compreso fra 0 e 1 (se *e* ed *l* hanno distanza 0, ovvero sono nella stessa *sentence*, *w=1*, se hanno distanza 1, *w=0.5*, se hanno distanza 2, *w=0.25*, e così via).

Il grado complessivo di correlazione fra l'entità *e* e il lemma *l* si ottiene quindi come somma dei pesi di tutte le occorrenze:

$$w(e,l)=\sum_{i=0}^{N} w_i(e,l)=\sum_{i=0}^{N} 2^{-d_i}$$

dove *N* è il numero di occorrenze della coppia *(e,l)*.

9.4 Struttura del database usato per l'analisi

Per la fase di analisi dei dati estrapolati mi sono appoggiato a un database MySQL, usando come corpus di test sempre quello del Corriere. Il database è stato strutturato nel seguente modo:

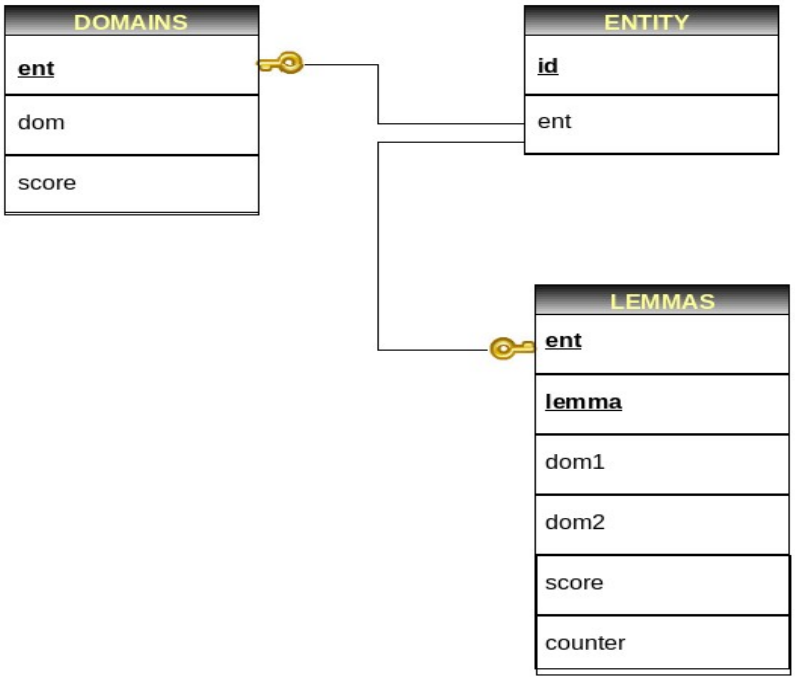

Figura 7: Struttura del database usato per il test per Okkam

DOMAINS contiene i domini associati alle entità, estrapolati dai documenti a cui si riferiscono, dove

- *ent* è l'entità (identificata dalla stringa)
- *dom* è il dominio associato (identificato come stringa)
- *score* è il grado di pertinenza di quel dominio all'entità. Tale valore è calcolato come

$$w(d,e)=\sum_{i=0}^{N} w_i(d \wedge e)=\sum_{i=0}^{N} \frac{\sum_{j=0}^{N} w_j(d,l_j)}{\sum_{k=0}^{M} d_k | l_j}$$

dove *w(d,e)* è appunto lo *score* (peso) complessivo della coppia entità-dominio in questione, calcolato come somma degli *score* dei domini dei documenti in cui ricorre anche l'entità *e* (*N* è il numero di documenti in cui ricorre l'entità *e* e il dominio *d*). Lo score del dominio *d* all'interno del documento i-esimo viene calcolato nel modo visto in precedenza (somma dei gradi di attinenza fra il

generico lemma j-esimo del documento i-esimo e il dominio *d* diviso la somma di tutti gli “score parziali” del documento).

ENTITY contiene le entità estrapolate dall'analisi con relativo *id* (che potrebbe tornare utile per l'inserimento in Okkam come “numero di synset” per quell'entità).

LEMMAS contiene tutti i lemmi l associati all'entità *e* nel modo descritto prima, dove

- **ent** è l'entità in questione
- **lemma** è il lemma associato
- **dom1** è il dominio primario del lemma *l*
- **dom2** è l'eventuale dominio secondario del lemma *l*
- **score** è il grado di pertinenza fra *e* ed *l*, misurato nel modo descritto in precedenza
- **counter** conta quante volte ricorre la coppia *(e,l)*

9.5 Analisi dei dati

Ecco le 20 coppie entità-lemma più ricorrenti nel corpus:

```
mysql> select ent,lemma,score
from lemmas
order by score desc
limit 20;

+-----------------+-----------+--------+
| ent             | lemma     | score  |
+-----------------+-----------+--------+
| Moratti         | sindaco   |  686.5 |
| Prodi           | governo   |    568 |
| Letizia Moratti | sindaco   |  530.5 |
| Berlusconi      | governo   |    356 |
| Moratti         | Milano    | 284.75 |
| Letizia Moratti | Milano    |    280 |
```

```
| Berlusconi      | Prodi     |    237 |
| Palazzo Marino  | Comune    |  230.5 |
| Romano Prodi    | governo   | 208.75 |
| Vittorio Sgarbi | assessore | 193.25 |
| Moratti         | città     |    177 |
| Romano Prodi    | premier   |    177 |
| Prodi           | premier   | 159.25 |
| Sforzesco       | Castello  | 157.25 |
| Veltroni        | sindaco   |  156.5 |
| Lambro          | Parco     | 140.75 |
| Palazzo Marino  | assessore |    133 |
| Pirelli         | Telecom   | 130.25 |
| Palazzo Marino  | sindaco   | 127.25 |
| Vittorio Sgarbi | Cultura   |  124.5 |
+-----------------+-----------+--------+
20 rows in set (0.08 sec)
```

La prima “pulizia” da effettuare sul database consiste nell'eliminare tutte le coppie entità-lemma aventi *score <= 0.5* e *counter == 1* (ovvero sia tutte le coppie occorrenti una sola volta, sia quelle ricorrenti o una sola volta, e in due frasi adiacenti, o al massimo due volte e in due frasi a distanza 2), in quanto poco determinanti a fini statistici. Tuttavia, anche dopo questa fase di rimozione del rumore (che fra l'altro porta via oltre la metà delle coppie occorrenti), lo score medio delle coppie entità-lemma salvate rimane sorprendentemente basso:

```
mysql> select avg(score)
       as "score medio"
       from lemmas;
+-----------------+
| score medio     |
+-----------------+
| 3.2243184912657 |
+-----------------+
```

```
1 row in set (0.02 sec)
```

a fronte invece della presenza di coppie, nella vista esaminata in precedenza, con uno score molto elevato (nell'ordine delle centinaia) ed effettivamente anche molto pertinenti. Notiamo quindi che, in una lista delle coppie entità-lemma ordinata in senso decrescente sullo score, la pertinenza fra l'entità e il lemma nella coppia ha un comportamento decrescente e fortemente asintotico, situazione rappresentata qualitativamente in figura:

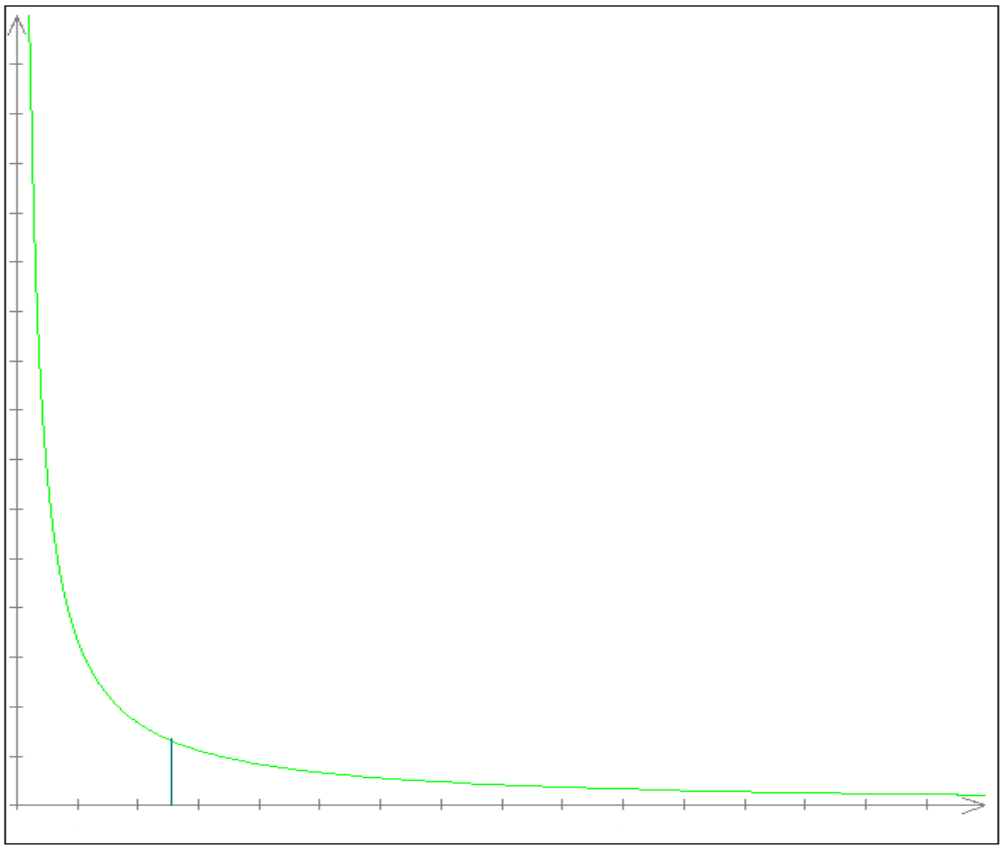

Figura 8: Andamento degli score delle coppie entità-lemma all'interno della lista ordinata sullo score in senso decrescente

Sull'asse *x* è riportato l'ID che sarebbe associato alla coppia nel caso in cui la lista venisse ordinata in senso decrescente in base allo score, e sull'asse *y* lo score corrispondente. E il numero di coppie a destra del valore medio (la cui ascissa è rappresentata dalla linea azzurra) è molto superiore rispetto a quelle alla sua sinistra. Il totale di coppie presenti nella tabella (riferite al biennio 2006-2007 e con il database già pulito dalle coppie scarsamente rilevanti, ovvero con *score* <= *0.5* o *counter* == *1*) è infatti di

```
mysql> select count(*)
       from lemmas;
+---------------+
| coppie totali |
+---------------+
|         34519 |
+---------------+
1 row in set (0.30 sec)
```

e quelle che hanno score minore della media sono

```
mysql> select count(*)
       as “score inferiori alla media”
       from lemmas
       where score < (select avg(score)
                      from lemmas);
+----------------------------+
| score inferiori alla media |
+----------------------------+
|                      27410 |
+----------------------------+
1 row in set (0.05 sec)
```

ovvero quasi l'80% delle coppie totali. Tuttavia, uno score di 3.2 (ovvero il valore medio dello score) è davvero basso per poter rappresentare un risultato di una certa rilevanza. Quindi, tutte le coppie con score inferiore a tale valore (ovvero a destra della linea azzurra sul grafico visto sopra) possono essere eliminate dal database senza avere grosse perdite a livello logico o di informazione gestita, e riducendo quindi drasticamente anche le dimensioni della base di dati. Si può anche adottare, a seguito di qualche analisi empirica sulla base, una soglia diversa dal valore medio dello score, e magari anche più restrittiva, in base alle dimensioni del database che si vogliono ottenere, al numero di informazioni che si vuole salvare, al livello di precisione delle stesse, e al livello massimo di rumore nei dati tollerabile.

10. CONCLUSIONI

Uno degli scenari di maggior interesse nel futuro dell'informatica è quello della progettazione su larga scala di sistemi esperti in grado di comprendere ed analizzare il linguaggio comune, non solo testi strutturati in modo da essere comprensibili alle specifiche del sistema stesso. Lo sviluppo di un sistema in grado di classificare in modo empirico non solo le entità note all'interno di un documento, ma anche quelle non note, sulla base del contesto, rappresenta una rivoluzione enorme nel campo dei sistemi esperti, e un grande passo avanti verso un'informatica che diventa sempre più flessibile in fatto di trattamento dell'informazione. Gli scenari che si aprono sono infiniti. Aziende o enti pubblici che possono fornire servizi automatici alla clientela, accessibili tramite sms o email ad esempio, basati su sistemi esperti in grado di elaborare automaticamente le richieste fatte in linguaggio naturale. Gestione e classificazione automatica di grandi moli di documenti in formato elettronico, operazioni che potrebbero non essere più svolte a mano da una persona fisica ma da un sistema esperto. E, infine, queste tecnologie potrebbero avere un ruolo notevole nella rivoluzione alle porte del *web 3.0*, nel mondo del web semantico, che consentirà di avvicinare sempre di più l'informatica alla logica e al modo di esprimersi umano, sfatando il mito dell'informatica rigida, paradigmatica e fortemente strutturata ancora presente presso molti non addetti ai lavori e avvicinando ulteriormente la tecnologia alle masse.

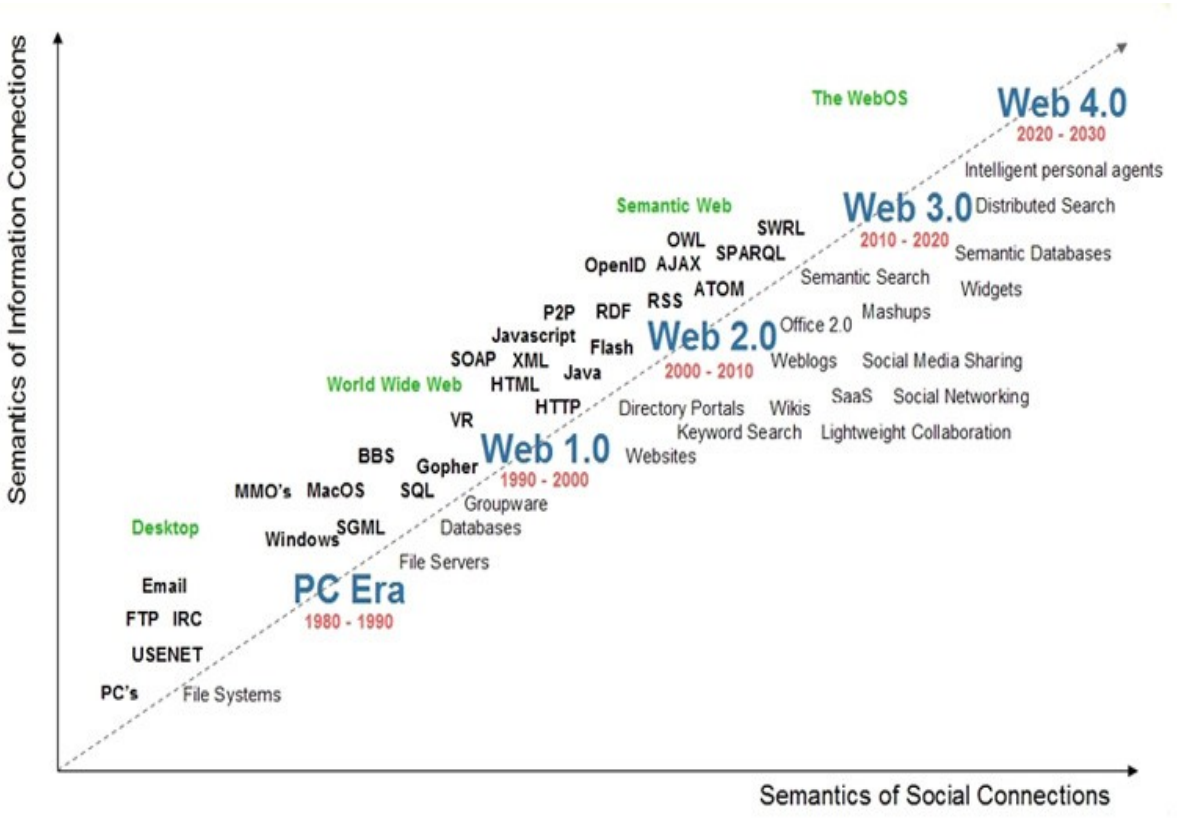

Figura 9: Sviluppo delle tecnologie web e previsioni sulla rivoluzione del web semantico nel mondo web 3.0

Fonte: Radar Networks & Nova Spivack, 2007 – www.radarnetworks.com

11. SITOGRAFIA

- http://en.wikipedia.org/wiki/Web_3.0
- http://en.wikipedia.org/wiki/Hypernym
- http://en.wikipedia.org/wiki/Information_retrieval
- http://www.expertsystem.it/
- http://www.dcs.gla.ac.uk/Keith/Preface.html
- http://wordnet.princeton.edu/
- http://www.regular-expressions.info/
- http://www.w3.org/XML/
- http://www.boost.org/
- http://www.okkam.org/