

UNIVERSITÀ DEGLI STUDI DI PISA



FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI  
CORSO DI LAUREA SPECIALISTICA IN TECNOLOGIE  
INFORMATICHE

TESI DI LAUREA

# Un sistema Peer-to-Peer per l'interrogazione distribuita di dati XML

CANDIDATO

Nicola Gioia

RELATORE

Prof. Giorgio Ghelli

CONTRORELATORE

Prof.ssa Susanna Pelagatti

Anno Accademico 2003/2004



*Ai miei genitori,  
ai miei nonni  
e a tutti coloro  
che mi hanno aiutato  
a crescere...*



# Riassunto

Nel corso degli ultimi anni, abbiamo assistito al diffondersi dell'architettura Peer-to-Peer, la quale promette di riuscire a sfruttare meglio la rete nell'utilizzo di sistemi di condivisione di risorse.

L'uso di architetture Peer-to-Peer anche nel mondo delle basi di dati, può permettere di superare alcuni limiti propri dei DBMS distribuiti, come la staticità dei nodi e l'intenso lavoro di amministrazione dell'intero sistema, e di riuscire a sfruttare le potenzialità offerte da Internet per condividere dati agevolmente.

Il lavoro svolto per la tesi è consistito nella partecipazione alla progettazione e nell'implementazione di un prototipo funzionante di un sistema Peer-to-Peer. Il sistema studiato permette di interrogare dati semistrutturati, distribuiti su rete e memorizzati con XML, e ne permette la manipolazione con XQuery.



# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Obiettivi del progetto <i>XPeer</i> . . . . .	2
1.1.1	Scelta del modello di comunicazione . . . . .	2
1.2	Scelta della tipologia dei dati . . . . .	3
1.3	Organizzazione della relazione . . . . .	4
<b>2</b>	<b>Il modello dei dati XML</b>	<b>5</b>
2.1	La nascita dell'XML e le sue caratteristiche . . . . .	5
2.1.1	Struttura di un documento XML . . . . .	7
2.1.2	Rappresentazione logica di un documento XML . . . . .	10
2.1.3	Interpretazione di un documento XML . . . . .	14
2.1.4	Tecnologie correlate all'XML . . . . .	14
2.2	I linguaggi di interrogazione dei dati XML . . . . .	15
2.2.1	XPath . . . . .	15
2.2.2	XQuery . . . . .	19
<b>3</b>	<b>Uno sguardo ai sistemi P2P esistenti</b>	<b>25</b>
3.1	Le topologie di reti P2P . . . . .	26
3.1.1	Topologia centralizzata . . . . .	27
3.1.2	Topologia gerarchica . . . . .	27
3.1.3	Topologia ad anello . . . . .	28
3.1.4	Topologia decentralizzata . . . . .	29
3.1.5	Alcune topologie composte . . . . .	29
3.2	I principali modelli esistenti . . . . .	30
3.2.1	Piazza . . . . .	30
3.2.2	Gnutella . . . . .	31
3.2.3	P-Grid . . . . .	32

3.2.4	FastTrack/OpenFT . . . . .	34
3.2.5	Altri . . . . .	35
<b>4</b>	<b>Progettazione del sistema XPeer</b>	<b>37</b>
4.1	Architettura del sistema . . . . .	38
4.1.1	Struttura della rete . . . . .	40
4.1.2	Gestione della robustezza . . . . .	40
4.1.3	Caratteristiche supplementari . . . . .	40
4.2	Architettura del singolo nodo . . . . .	41
4.2.1	Il livello Peer . . . . .	41
4.2.2	Il livello SuperPeer . . . . .	43
4.2.3	Il sottosistema di comunicazione . . . . .	44
4.3	Descrizione dei protocolli . . . . .	44
4.3.1	Protocolli di sistema . . . . .	45
4.3.2	Protocolli di innesco . . . . .	49
4.3.3	Protocolli di evoluzione della rete . . . . .	50
4.3.4	Protocolli di elaborazione delle query . . . . .	54
4.3.5	Protocolli di comunicazione . . . . .	56
<b>5</b>	<b>Implementazione di XPeer</b>	<b>61</b>
5.1	Sistema di comunicazione . . . . .	62
5.1.1	Stack del sistema di comunicazione . . . . .	62
5.1.2	Scelte implementative . . . . .	68
5.2	Sistema di gestione dei Peer . . . . .	71
5.2.1	Scelte implementative . . . . .	72
5.3	Interfacce grafiche . . . . .	73
5.4	Errori nell'implementazione di Java della Sun . . . . .	74
<b>6</b>	<b>Conclusioni</b>	<b>77</b>
	<b>Bibliografia</b>	<b>81</b>

# Capitolo 1

## Introduzione

Nel corso degli ultimi anni si è osservato un aumento dei sistemi peer-to-peer (P2P) per la condivisione di informazioni tramite rete. Una fiorente area di ricerca si è sviluppata intorno alla creazione di sistemi che gestiscono dati semistrutturati e che utilizzano una piattaforma di condivisione basata su piattaforma P2P. Un gruppo di ricerca al Dipartimento di Informatica dell'Università di Pisa si è formato con l'obiettivo di studiare un nuovo sistema P2P, denominato *XPeer*, che possieda le seguenti caratteristiche:

1. gestione di dati XML con un qualsiasi schema che possa variare nel tempo;
2. capacità di funzionare in presenza di numerose operazioni di connessione/disconnessione di un Peer dal sistema;
3. semplicità del sistema, utilizzabile senza bisogno di amministratori locali o globali.

La creazione di un sistema nuovo è giustificata dal fatto che quelli attualmente esistenti posseggono solo alcune di queste caratteristiche contemporaneamente:

- Piazza [Tat04], è un esempio di database corporato P2P per dati XML, ma è utilizzabile solo per costruire sistemi statici, perché per potersi connettere ad una rete Piazza è indispensabile la presenza di un amministratore che renda possibile ad un peer la connessione alla rete, e ogni volta che uno dei peer deve cambiare il proprio schema di rappresentazione dei dati occorre nuovamente l'intervento dell'amministratore del sistema per mantenere la funzionalità del sistema stesso.
- KaZaA [Kaz], uno degli strumenti commerciali per la condivisione di file su rete, che lavora su rete FastTrack, è uno degli strumenti più evoluti per condivisione di file, ma è limitato da uno schema fisso su cui è possibile eseguire le interrogazioni.

## 1.1 Obiettivi del progetto *XPeer*

Il progetto *XPeer*, sviluppato dal Dipartimento di Informatica dell'Università degli studi di Pisa, mira alla realizzazione di un sistema P2P senza amministratore, per condividere in rete dati XML [XML04] interrogabili con XQuery [XQu04].

La maggior parte degli utenti di un sistema di scambio informazioni, sono utenti disposti al più ad installare un software ed essere subito pronti per poter fare delle ricerche distribuite su scala globale avendo come unico onere quello di rendere disponibili agli altri alcune delle proprie informazioni locali. Per questo è necessario creare un sistema che funzioni e che sia semplice da gestire per l'utente finale.

L'esigenza di un sistema semplice da gestire, con la possibilità di crescere in dimensioni molto rapidamente e improvvisamente, senza richiedere la presenza di un server da configurare, gestire ed eventualmente potenziare per il suo corretto e continuo funzionamento, è alla base della scelta dello sviluppo su piattaforma P2P piuttosto che Client-Server

### 1.1.1 Scelta del modello di comunicazione

#### Client-Server

In un sistema Client-Server è necessario distinguere fra due classi di entità distinte che appartengono al sistema: I Client e il Server.

I Client sono entità praticamente identiche tra loro, ciascuna delle quali, per funzionare appieno, ha bisogno di alcuni servizi che debbono necessariamente essere forniti da un'altra entità esterna diversa da se: il Server. Il Server dall'altro verso è un'entità che ha come unico motivo di esistenza quello di fornire risposte alle richieste di servizi da parte di un Client.

In molti casi questo meccanismo è utile e ben funzionante, basti pensare a tutti i sistemi che funzionano con questa filosofia: il web (con il protocollo http), il controllo della posta elettronica (con i protocolli POP e IMAP), lo scambio di files remoto (con FTP) ecc.

#### Peer-to-Peer

Il sistema Peer-to-Peer è un'evoluzione del sistema Client-Server in quanto ogni Peer richiede i servizi di cui ha bisogno ad un altro Peer praticamente identico a se stesso (suo pari appunto) che gli risponde fornendogli il servizio richiesto, o, in sistemi più evoluti, indicandogli un altro Peer che è in grado di farlo.

Ciascuna entità in questo nuovo sistema agisce quindi sia da Client, perché è in grado di richiedere servizi, sia da Server, perché è in grado di fornirne (in alcuni casi un Peer è chiamato Servant per questa sua duplice attitudine). Ogni Peer è perciò più complesso di un singolo Client o di un singolo Server dovendo in realtà espletare entrambi i compiti. Grazie a questa caratteristica è quindi possibile creare applicazioni più flessibili che non sono legate al funzionamento di una singola unità. Il sistema risulta così più dinamico, ed è in grado di funzionare anche se dei peer si scollegano dal sistema, purché almeno un gruppo di essi rimanga sempre collegato per mantenere il sistema attivo, non importa quali, le configurazioni successive del sistema possono anche non avere mai nessun peer in comune, pur essendosi evolute senza nessun fallimento.

## 1.2 Scelta della tipologia dei dati

L'informazione in una raccolta di dati, può essere classificata, in dipendenza della regolarità della sua struttura, come strutturata, non strutturata e semistrutturata.

L'informazione strutturata prevede l'esistenza di una struttura (definita dallo schema) molto precisa che descrive i dati che appartengono alla raccolta di dati, e se alcuni di essi non rispettano fedelmente la struttura imposta dallo schema non possono entrare a far parte della raccolta stessa.

L'informazione non strutturata non impone nessun vincolo all'organizzazione dei dati, che quindi possono essere inseriti nella raccolta in qualunque forma. Per utilizzare i dati memorizzati bisogna riuscire a interpretarli, in quanto non è garantita la presenza di nessuna informazione sull'organizzazione e sul significato dei dati nel sistema, e se esiste una minima informazione strutturale sui dati, essa è memorizzata insieme ai dati stessi.

L'informazione semistrutturata (vedi [Bun97]) si pone in mezzo ai due estremi. Come nel caso precedente la struttura va specificata insieme ai dati stessi. Si ottiene così che, come nel caso della rappresentazione non strutturata, non esiste un vincolo strutturale ai dati che devono essere immessi, e, come nel caso della rappresentazione strutturata, i dati presenti nella base di dati posseggono una struttura abbastanza regolare che li descrive. I dati semistrutturati devono quindi essere in grado di descrivere autonomamente la propria struttura.

Il modello di dati relazionale si presta molto bene alla rappresentazione di dati strutturati, ma è inutilizzabile sia in contesto semistrutturato che in un contesto strutturato.

Il modello di dati XML, invece, si presta molto bene a rappresentare dati in tutti

i formati, in modo autodescrittivo, riunendo cioè i dati alla loro struttura in modo inscindibile. Per creare un sistema di interrogazione dati occorre usare un linguaggio di interrogazione che riesca ad manipolare dati memorizzati con XML. XQuery è un linguaggio di interrogazione ed elaborazione di dati XML che riesce a trattare bene sia rappresentazioni strutturate che rappresentazioni semistrutturate di dati, anche se mal si presta a gestire dati non strutturati.

Nell'ambiente dinamico in cui ci si trova ad operare per l'utilizzo di un sistema con le caratteristiche suddette è impensabile utilizzare una rappresentazione di dati strutturata. Questa, infatti, richiederebbe un accordo preventivo da parte di tutti gli utenti sulla struttura che si vuole rappresentare, e la conseguente presenza di un amministratore che ne possa decidere una comune, presenza che si è esclusa a priori.

Viceversa utilizzare una rappresentazione di dati non strutturata è inutile, poiché l'assenza di una struttura complica la comprensione di dati rendendola interpretabile solo da persone (che conoscono bene l'ambito di interesse dei dati stessi), e non elaborabili semi-automaticamente da macchine.

Una base di dati che memorizza la raccolta di dati in formato XML e che utilizza XQuery per l'interrogazione e la manipolazione dei dati stessi costituisce un sistema in grado di gestire sia dati strutturati che dati semistrutturati, e quindi si presenta come la soluzione ideale per la creazione del sistema XPeer.

### **1.3 Organizzazione della relazione**

La presente Tesi è organizzata come segue. Nel capitolo 2 sono illustrate le nozioni base sull'XML e le tecnologie correlate che sono state usate per la realizzazione del DBMS locale. Nel capitolo 3 sono enunciati e classificati alcuni esempi di protocolli P2P esistenti per lo scambio di informazioni sulla rete. Nel capitolo 4 è descritta la progettazione del sistema. Nel capitolo 5 sono spiegate le scelte implementative adottate per la realizzazione di un prototipo del sistema. Nel capitolo 6 infine vengono tratte le conclusioni al lavoro di Tesi.

# Capitolo 2

## Il modello dei dati XML

### 2.1 La nascita dell'XML e le sue caratteristiche

L'eXtensible Markup Language (XML, linguaggio estensibile a marcatori) è un metalinguaggio che permette di creare dei linguaggi personalizzati di markup; nasce dall'esigenza di un linguaggio più evoluto dell'HTML e che potesse essere utilizzato per strutturare meglio i dati, non solo per formattarli. Nasce così un metalinguaggio a marcatori che permette di esprimere dati anche con struttura complessa e ne rende possibile una facile distribuzione nel web.

XML fu sviluppato da XML Working Group (originariamente noto come SGML Editorial Review Board) costituitosi sotto gli auspici del World Wide Web Consortium (W3C) nel 1996. Esso era presieduto da Jon Bosak della Sun Microsystems con la partecipazione attiva dell'XML Special Interest Group (precedentemente noto come SGML Working Group) anch'esso organizzato dal W3C. Obiettivo di questo gruppo di lavoro era modificare il linguaggio SGML al fine di renderlo più accessibile al Web. L'SGML è un linguaggio per la specifica dei linguaggi di markup ed è il genitore del ben noto HTML. Nel 1998 nasce la prima specifica del linguaggio XML.

Gli obiettivi progettuali di XML sono:

1. **XML deve essere utilizzabile in modo semplice su Internet:** in primo luogo, l'XML deve operare in maniera efficiente su Internet e soddisfare le esigenze delle applicazioni eseguite in un ambiente di rete distribuito.
2. **XML deve supportare un gran numero di applicazioni:** deve essere possibile utilizzare l'XML con un'ampia gamma di applicazioni, tra cui strumenti di creazione, motori per la visualizzazione di contenuti, strumenti di traduzione e applicazioni di database.
3. **XML deve essere compatibile con SGML:** questo obiettivo è stato definito sulla base del presupposto che un documento XML valido debba anche essere

un documento SGML valido, in modo tale che gli strumenti SGML esistenti possano essere utilizzati con l'XML e siano in grado di analizzare il codice XML.

4. **Deve essere facile lo sviluppo di programmi che elaborino documenti XML:** l'adozione del linguaggio è proporzionale alla disponibilità di strumenti e la proliferazione di questi è la dimostrazione che questo obiettivo è stato raggiunto.
5. **Il numero di caratteristiche opzionali deve essere mantenuto al minimo possibile:** al contrario dell'SGML, l'XML riduce al minimo le caratteristiche opzionali, in tal modo qualsiasi elaboratore potrà pertanto analizzare qualunque documento XML, indipendentemente dai dati e dalla struttura contenuti nel documento.
6. **I documenti XML dovrebbero essere leggibili da un utente e ragionevolmente chiari:** poiché utilizza il formato testuale per descrivere i dati e le loro relazioni, l'XML è più semplice da utilizzare ed è leggibile anche da umani, diversamente da un formato binario che esegue la stessa operazione.
7. **La progettazione di XML dovrebbe essere rapida:** l'XML è stato sviluppato per soddisfare l'esigenza di un linguaggio estensibile per il Web. Questo obiettivo è stato definito dopo aver considerato l'eventualità che se l'XML non fosse stato reso disponibile rapidamente come metodo per estendere l'HTML, altre organizzazioni avrebbero potuto provvedere a fornire una soluzione proprietaria, binaria o entrambe.
8. **La progettazione di XML deve essere formale e concisa:** questo obiettivo deriva dall'esigenza di rendere il linguaggio il più possibile conciso, formalizzando la formulazione della specifica.
9. **I documenti XML devono essere facili da creare:** i documenti XML possono essere creati facendo ricorso a strumenti di semplice utilizzo, quali editor di testo normale.
10. **Non è di nessuna importanza l'economicità nel markup XML:** nell'SGML e nell'HTML la presenza di un tag di apertura è sufficiente per segnalare che l'elemento precedente deve essere chiuso. Benché così sia possibile ridurre il lavoro degli autori, questa soluzione potrebbe essere fonte di confusione per i lettori, nell'XML la chiarezza ha in ogni caso la precedenza sulla concisione.

### 2.1.1 Struttura di un documento XML

Per descrivere un documento XML occorre distinguere tra la rappresentazione sintattica, e la rappresentazione logica del documento stesso. La rappresentazione sintattica indica la sintassi che deve essere posseduta da un file di testo per poter essere considerato un file XML. La rappresentazione logica indica l'interpretazione che va data ad un documento XML.

#### La rappresentazione sintattica dell'XML

Prima di iniziare a descrivere la struttura di un documento XML dobbiamo definire che cosa è un tag. Un tag è una qualunque stringa racchiusa tra parentesi angolate. Un tag si dice di chiusura se inizia con il carattere “/”, ed è un tag di chiusura per il nome che è riportato subito dopo il carattere “/”. Un tag vuoto è un tag che termina con il carattere “/”.

Un documento XML è costituito da dichiarazioni, elementi, istruzioni di elaborazione e commenti. Alcune componenti sono opzionali, altre sono necessarie:

- le dichiarazioni sono costituite da tag che iniziano con il carattere “!”, e sono opzionali;
- gli elementi sono costituiti da un tag vuoto oppure da tutto ciò che è racchiuso tra il tag di apertura, formato da una stringa in cui la prima parte detta nome del tag è composta da un sequenza di numeri e lettere più i caratteri “\_”, “:” e “.” iniziante per lettera, e il corrispondente tag di chiusura; è obbligatoria la presenza di un elemento;
- le istruzioni di elaborazione sono tag che iniziano e finiscono per il carattere “?”, possono essere presenti in un qualunque punto del documento<sup>1</sup>, e sono opzionali.

Le istruzioni di elaborazione sono tag utilizzati da opportuni programmi per eseguire dei comandi speciali prima dell'elaborazione del documento, o in un qualche punto durante l'elaborazione del corpo del documento;

- i commenti sono tag che iniziano con la sequenza di caratteri “!--” e terminano con la sequenza “--”, possono essere presenti in un qualunque punto del documento<sup>1</sup>, e sono opzionali.

I commenti sono tag che vengono ignorati dall'elaboratore XML, essi servono per annotare qualche informazione che potrebbe essere utile per un eventuale controllo manuale del file.

---

<sup>1</sup>tranne che all'interno di un tag

In un documento XML possiamo riconoscere due parti: il prologo e il corpo.

### Prologo

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

**Dichiarazione XML.** La dichiarazione XML identifica la versione delle specifiche XML a cui è conforme il documento. Sebbene la dichiarazione XML sia un elemento opzionale, se presente deve essere inserita in cima al documento XML. Nell'esempio di figura 2.1 è visibile una dichiarazione XML di base.

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

Figura 2.1: Esempio di prologo minimo XML.

Una dichiarazione XML può contenere diversi attributi: una *version* per indicare la versione dell'XML con cui è costruito il documento; un *encoding* per specificare la codifica del carattere usato nel documento; e una *standalone* (dichiarazione di documento autonomo) per specificare se sono presenti elementi del documento XML esterni al file in lettura.

**Dichiarazione del tipo di documento.** La dichiarazione del tipo di documento è costituita da codice di markup che indica le regole grammaticali o la "*definizione del tipo di documento*" (DTD) per una particolare classe di documenti. Questa dichiarazione può fare riferimento ad un file esterno che contiene tutta o parte della DTD e deve essere visualizzata dopo la dichiarazione XML e prima dell'elemento Document. Queste stringhe di codice aggiungono una dichiarazione del tipo di documento all'esempio di figura 2.2.

```
<?xml version="1.0"?>  
<!DOCTYPE Ortaggi SYSTEM "Ortt.dtd">
```

Figura 2.2: Esempio di prologo con DTD XML.

Per maggiori chiarimenti sull'utilizzo e la sintassi di un DTD vedere la sez. 2.1.2 a pag. 12.

## Corpo

Il secondo elemento strutturale di un documento XML è il corpo, obbligatorio costituito e da un elemento.

Un elemento può contenere al suo interno uno o più sottoelementi, che a loro volta contengono altri sottoelementi, e così via. Tutti i sottoelementi di un elemento devono essere correttamente annidati. Ogni elemento non vuoto, tra il tag di apertura e quello di chiusura deve possedere solo tag che si aprono e chiudono totalmente dentro il tag stesso. Per meglio chiarire il concetto dell'annidamento vediamo un esempio in figura 2.3 di corpo di documento e descriviamone gli errori nell'annidamento.

```
<a>
  <b>
    <c>
      <d>
    </c>
    </d>
    <e></e>
  </b>
</a>
```

Figura 2.3: Esempio di annidamento errato.

L'elemento "a" ha come unico sottoelemento "b", ed è un sottoelemento ben annidato .

"b" ha due sottoelementi ben annidati ("e" e "c") e un sottoelemento non annidato correttamente: "d". L'annidamento non corretto si può vedere dal fatto che tra il tag di apertura e quello di chiusura di "b" è presente un tag di chiusura che non ha nessun tag di apertura corrispondente. per lo stesso motivo anche "c" ha problemi di annidamento, perché tra il suo tag di apertura e quello di chiusura è presente un altro tag di apertura che non ha il corrispondente tag di chiusura.

Ogni elemento di un documento XML può infine possedere degli attributi e del testo. Gli attributi vanno espressi nella forma chiave="valore" all'interno del tag che ne definisce il nome, prima della chiusura della parentesi angolata, e dell'eventuale carattere "/" di chiusura del tag, separati tra di loro da uno spazio. Il testo presente facoltativamente tra l'apertura e la chiusura di un elemento può contenere qualunque carattere diverso da:

< > & ' "

caratteri che invece possono essere rappresentati rispettivamente con

&lt; &gt; &amp; &apos; &quot;

Per un esempio del corpo di un documento completo vedi figura 2.4.

```
<ROOT>
  <PARENT1 attrib1="val 1" attrib2="val 2">
    affermo che 2 &lt; 3
    <CHILD1>altro testo</CHILD1>
    <CHILD2 attrib3="val 3"/>
  </PARENT1>
</ROOT>
```

Figura 2.4: Esempio di elementi con attributi e testo.

Ogni elemento di un documento XML può essere qualificato da un namespace. Il namespace è un prefisso posto davanti al nome dell'elemento che termina con il carattere ":".

### 2.1.2 Rappresentazione logica di un documento XML

La rappresentazione logica di un documento XML è un albero etichettato che ha per radice l'elemento speciale *document*, e in cui, per ogni elemento del documento, è presente un nodo corrispondente nell'albero. In particolare l'elemento *document* ha sempre come unico figlio un nodo che corrisponde all'elemento obbligatorio del corpo del documento XML. Se un elemento del documento XML ha dei sottoelementi, per ognuno di essi il nodo corrispondente dell'albero avrà un figlio: il nodo corrispondente al sottoelemento. Se l'elemento è legato ad un namespace o ha degli attributi, allora al nodo corrispondente nell'albero sono assegnate un'etichetta per il namespace, e un'etichetta per gli attributi<sup>2</sup>.

L'elemento Document contiene tutti i dati di un documento XML inclusi tutti i sottoelementi annidati e le entità esterne. Nella figura 2.5 vediamo la rappresentazione logica del documento di figura 2.4.

<sup>2</sup>l'etichetta è un nodo speciale che non può avere figli

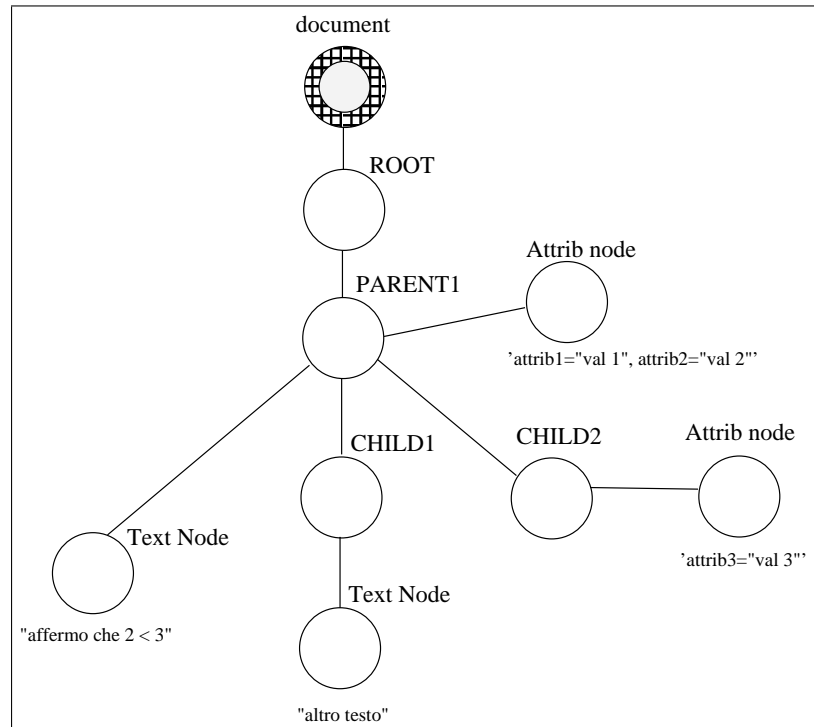


Figura 2.5: Esempio di albero XML.

L'XML possiede altre caratteristiche, come le entità, che gli permettono di fare riferimenti a file esterni, e gli permettono di definire alcune funzioni, ma per questo utilizzo più specifico rimandiamo alla letteratura [XML04, W3C, Cor].

Di un documento XML è possibile controllare se esso è valido e se è ben formato.

Più specificatamente, un documento XML si dice ben formato se:

1. tutti i tag di apertura e di chiusura corrispondono e sono ben annidati;
2. esiste un elemento radice che contiene tutti gli altri;
3. tutti i valori degli attributi sono racchiusi tra virgolette;
4. tutte le entità sono dichiarate;
5. il prologo, se presente, è all'inizio del file XML.

Un documento XML si dice valido se:

1. è ben formato;
2. segue le regole specificate nel DTD.

## Il DTD di un documento XML

Il DTD (Document Type Definitions) è una delle sezioni opzionali del documento, che definisce in modo rigoroso lo schema dei dati rappresentabili in un documento XML. Il DTD serve quindi a validare un documento XML.

Il DTD, se presente, va dichiarato all’inizio del file XML subito prima della sezione document, e può essere dichiarato o come insieme di dichiarazioni nel file, oppure come un’unica dichiarazione che fa riferimento ad un file esterno, in cui sono poi presenti tutte le dichiarazioni sulla struttura.

Nel caso della dichiarazione esterna abbiamo una sintassi simile a quella della figura 2.2, in cui la parola “Ortaggi” indica il nome dello schema che si sta rappresentando, la parola “SYSTEM” è una parola riservata che indica che il file da cercare è nel filesystem locale (un’alternativa a SYSTEM è PUBLIC che indica che il file è da ricercare in una URI pubblica), ed infine c’è il nome del file contenente il DTD.

Nel caso di DTD contestuale nel documento XML avremo una sintassi sul modello della figura 2.6.

```
<?xml version="1.0"?>
<!DOCTYPE Ortaggi [ DTD_BODY ]>
<ROOT>
  ...
</ROOT>
```

Figura 2.6: Esempio di documento XML con DTD interno al file.

La parte di testo che nella figura è chiamata “DTD\_BODY” è la stessa che è contenuta nell’eventuale file esterno del DTD ed è costituita da un elenco di dichiarazioni. Per spiegare la sintassi del corpo del DTD facciamo un esempio in figura 2.7. L’esempio indica che la struttura del file XML sarà costituita da una radice con nome “PIANTE” che avrà almeno un figlio, ma anche più, di tipo “PIANTA”. Ogni elemento di tipo pianta è costituito da:

1. una lista di sottoelementi possibilmente anche vuota di tipo “NOME COMUNE”;
2. un sottoelemento obbligatorio di tipo “NOME SCIENTIFICO”;
3. un sottoelemento facoltativo di tipo “CONSISTENZA”.

```

<!ELEMENT PIANTE (PIANTA+)>
<!ELEMENT PIANTA (NOME COMUNE*, NOME SCIENTIFICO, CONSISTENZA?)>
<!ATTLIST PIANTA famiglia CDATA #REQUIRED >
<!ELEMENT NOME COMUNE (#PCDATA)>
<!ELEMENT NOME SCIENTIFICO (#PCDATA)>
<!ELEMENT CONSISTENZA EMPTY>
<!ATTLIST CONSISTENZA tipo(Erbacea|Legnosa) "Erbacea">

```

Figura 2.7: Esempio di DTD.

L'elemento `pianta` inoltre ha anche un attributo obbligatorio di nome `famiglia` che è di tipo stringa. Gli elementi `NOME SCIENTIFICO` e `NOME COMUNE` sono entrambi elementi privi di sottoelementi e contenenti solo dati testuali (perché c'è la parola chiave `#PCDATA`), mentre l'elemento `CONSISTENZA` è un tag vuoto con un attributo di nome `tipo`, con valori possibili `Erbacea` o `Legnosa`, e con valore di default, in caso di assenza attributo, `Erbacea`.

La cardinalità di ogni elemento si riconosce dalla presenza o meno di un simbolo di fianco all'elemento:

- `+` indica una presenza obbligatoria e possibilmente ripetuta del sottoelemento (come in `PIANTE` l'elemento `PIANTA`);
- `*` indica una presenza opzionale e possibilmente ripetuta del sottoelemento (come in `PIANTA` l'elemento `NOME COMUNE`);
- `?` indica la presenza opzionale di un'istanza del sottoelemento (come in `PIANTA` l'elemento `CONSISTENZA`);
- nessuno dei simboli precedenti indica cardinalità uno (come in `PIANTA` l'elemento `NOME SCIENTIFICO`).

La presenza di sottoelementi può essere espressa in due modi diversi: con parole chiave, o con una lista. Le parole chiave consentite sono: `ANY`, `EMPTY`, o `#PCDATA`. `ANY` indica la possibile presenza di un qualunque sottoelemento; `EMPTY` indica che il tag è un tag speciale vuoto; e `#PCDATA` indica la presenza di solo testo. La lista può contenere elementi separati dal carattere `,` o dal carattere `|`: la presenza della virgola indica che sono presenti entrambi i sottoelementi; il carattere `|` indica la presenza alternativa dei sottoelementi.

La dichiarazione degli attributi presenti, effettuata con l'attributo `ATTLIST`, indica per ogni elemento tutti gli attributi che possiede. Come primo parametro

troviamo il nome dell'elemento cui fa riferimento la lista di attributi, e poi una lista di terne che indicano le caratteristiche di ciascun attributo. Il primo elemento della terna indica il nome dell'attributo, il secondo il tipo ed il terzo il valore di default. I tipi possibili sono presenti in un elenco delle specifiche dell'XML, qui citiamo solo i più importanti che sono:

- “CDATA” indica che il valore dell'attributo è una stringa qualunque;
- “ID” il valore dell'attributo è una chiave per i dati del documento XML;
- “IDREF” il valore dell'attributo è una chiave esterna;
- “enumerated” elenca una lista di valori possibili per attributi separati dal carattere “|”.

Il valore di default può essere una stringa racchiusa tra doppi apici, oppure una parola chiave che ne indica il comportamento in caso di assenza dell'attributo:

- “#REQUIRED” attributo richiesto, la sua assenza causa errore;
- “#IMPLIED” attributo opzionale, la sua assenza è ignorata;
- “#FIXED *fixvalue*” l'attributo deve avere il valore *fixvalue* in ogni caso.

### 2.1.3 Interpretazione di un documento XML

Esistono due tecniche principali per “interpretare” un documento XML: DOM e SAX. Un parser DOM (Document Object Model) prevede un parsing totale del file con successiva creazione dell'albero etichettato (il DOM appunto) corrispondente al file, mentre un parser SAX (Simple Api for Xml) è un parser che attiva degli handler durante la lettura del file XML, quando incontra dei tag specifici. Il DOM ha il vantaggio di essere molto più semplice da usare, ma, per contro, ha una grossa occupazione di memoria causata dal dover mantenere in memoria l'intero albero XML che rappresenta il file. Il SAX è più complesso da usare perché richiede di avere inizializzati tutti gli handler necessari all'esecuzione delle funzioni di interpretazione del file, ma richiede una minore occupazione di memoria durante l'operazione di parsing, che però può risultare molto a causa delle continue chiamate a funzioni di gestione degli eventi.

### 2.1.4 Tecnologie correlate all'XML

Di fianco all'XML sono nate poi numerose altre tecnologie basate su di esso: WML, XHTML, XSL, XSD, XPath, XQuery e altri. Enuncierò brevemente le prime,

perché non legate al nostro progetto, e descriverò più in dettaglio le ultime due che sono due linguaggi di interrogazione di dati XML.

WML (Wireless Markup Language) è lo standard attuale per pubblicare le pagine WAP, si colloca cioè nel gruppo dei linguaggi per la pubblicazione di informazione strutturata cui appartiene l'antenato HTML specializzato invece per il WEB.

XHTML è il successore dell'HTML e si propone come sua evoluzione per rendere più rigorosa la sintassi dell'HTML pur rimanendo retro-compatibile con essa.

XSL (eXtensible Stylesheet Language, linguaggio estensibile per i fogli di stile) è un linguaggio per i fogli di stile, e per trasformare l'XML e gli altri linguaggi a marcatori. L'obiettivo principale del XSL è quello di creare un linguaggio per trasformare e presentare una informazione strutturata (come quella rappresentata da un documento XML). XSD (XML Schema Definition o più semplicemente XML Schema) è un linguaggio (basato su XML) per definire il formato degli schemi contenuti in documenti XML. L'XML Schema serve cioè a validare altri documenti XML sulla base dello schema che si aspetta di trovarvi dentro, come per il DTD. Il motivo dell'introduzione di questa nuova tecnologia è dovuta all'eliminazione dei limiti del DTD sulla possibilità di vincolare i tipi delle espressioni.

## 2.2 I linguaggi di interrogazione dei dati XML

Sono stati proposti numerosi linguaggi per interrogare dati XML, a partire da SQL fino a XQueryX, ma i più importanti per espressività, e semplicità sono XPath, per la ricerca di elementi, e XQuery per ricercare elementi ed eventualmente ricostruire altri documenti XML come viste di documenti già esistenti.

Abbiamo scelto di utilizzare XQuery perché è un linguaggio standard per l'interrogazione di dati, e perché tra i più semplici da utilizzare.

### 2.2.1 XPath

Xpath [XPa99], attualmente giunto alla versione 2.0 [XPa03], è nato con lo scopo di dare una sintassi e una semantica comune a tutte le tecnologie che hanno la necessità di indirizzare delle porzioni di XML (come XSLT, o XPointer). Lo scopo principale di XPath è quello di selezionare un frammento di un documento XML. XPath usa una sintassi compatta non basata su XML per semplificarne l'uso, e rendere più facilmente leggibili le espressioni. XPath opera sulla struttura logica astratta di un documento XML, piuttosto che su quella sintattica.

XPath considera un documento XML come se fosse un albero di nodi, ma con nodi tipizzati. I tipi di nodo considerati da XPath sono sette, tra cui: nodi elemento,

attributo, namespace e testo. Ciascuno dei quattro tipi di nodo ha un nome: il nome dell'elemento nel caso di un nodo elemento, il nome dell'attributo, nel caso di un nodo attributo, il nome del namespace in caso di nodo namespace, e dall'intera stringa di testo nel caso di un nodo testo.

Il costrutto sintattico principale di XPath è l'espressione che è valutata per ottenere un valore, che può avere uno dei quattro tipi seguenti:

1. **insieme di nodi**: una raccolta ordinata di nodi senza duplicati;
2. **booleano**: un valore vero o falso;
3. **numero**: un numero in virgola mobile;
4. **stringa**: una sequenza di caratteri UCS.

Ogni espressione è valutata nell'ambito di un contesto, costituito da un nodo di contesto, che indica una posizione all'interno di una visita all'albero logico di un documento XML, più altre informazioni per l'esecuzione delle espressioni (vedi [XPa99]).

La principale espressione di XPath è la cosiddetta "*location path*". Una "*location path*" è una espressione che consiste in una serie di step (passi) separati dal carattere "/", che, valutati in un particolare nodo di contesto, permettono di navigare l'albero logico dell'XML. Uno degli step principali è "l'axis step" che in un'espressione "*location path*" può essere espresso con due diverse sintassi: la sintassi estesa e la sintassi abbreviata. La sintassi estesa inizialmente ci permetterà di capire meglio il significato dell'axis step, ma la sintassi abbreviata, una volta compresa la semantica dei vari passi, permette una lettura più immediata dell'espressione. Inizieremo ora a spiegare l'utilizzo delle espressioni con sintassi estesa, per poi spiegare come si esprimono con la sintassi abbreviata.

Un axis step permette di navigare un albero partendo da un nodo di contesto e spostandosi di passo in passo in una qualunque direzione (axis), ad esempio se da un nodo di contesto vogliamo muoverci verso i figli di nome "pianta" l'axis step sarà: `child::pianta`, oppure se ci vogliamo muovere verso i discendenti del nodo di contesto che ha nome "consistenza" l'axis step sarà `descendant::consistenza`.

Come si può vedere dai due esempi, un axis step è composto da due parti separate dalla stringa "::". La prima parte indica la direzione (axis) verso cui ci si vuole muovere, mentre la seconda serve a selezionare i nodi nella destinazione scelta, e può essere un nome, una funzione o il carattere "\*".

La presenza di un nome nella seconda parte dell'axis step indica che la selezione è vincolata al nome degli elementi presenti nella direzione specificata, la presenza

di una funzione indica una selezione per tipo di nodo destinazione piuttosto che per nome, il carattere “\*” seleziona tutti i nodi nella direzione designata.

La prima parte dell’axis step indica una delle 13 direzioni disponibili in XPath : `child`, `descendant`, `attribute`, `self`, `descendant-or-self`, `following-sibling`, `following`, `parent`, `ancestor`, `preceding-sibling`, `preceding`, `ancestor-or-self`, e `namespace` (la direzione `namespace` non è supportata da XQuery).

Il significato delle direzioni è:

- `child` indica i figli del nodo di contesto.
- `descendant` indica i discendenti del nodo di contesto: un discendente è un figlio, o un figlio del figlio, e così via fino al raggiungimento delle foglie; un discendente non contiene mai nodi attributo, e nodi namespace.
- `parent` indica il nodo genitore del nodo di contesto, se esiste.
- `ancestor` indica gli antenati del nodo di contesto; gli antenati di un nodo sono il padre, il padre del padre e così via fino alla radice; l’insieme dei risultati contiene sempre la radice, a meno che il nodo di contesto non sia la radice.
- `following-sibling` indica tutti i fratelli a destra del nodo di contesto ; se il nodo di contesto è un nodo attributo o un nodo namespace allora il risultato dell’axis step sarà vuoto.
- `preceding-sibling` come il precedente salvo che indica i fratelli a sinistra del nodo di contesto.
- `following` indica tutti i nodi che sono nello stesso documento del nodo di contesto che sono successivi al nodo di contesto secondo l’ordine del documento, esclusi i discendenti, e esclusi i nodi attributo e namespace.
- `preceding` indica tutti i nodi nello stesso documento del nodo di contesto e che precedono il nodo di contesto nell’ordine del documento, escluso gli antenati e esclusi i nodi attributo e namespace.
- `attribute` indica gli attributi del nodo di contesto ; l’insieme dei risultati sarà vuoto se il nodo di contesto non è un nodo elemento.
- `namespace` indica il nodo namespace del nodo di contesto; il risultato sarà vuoto se il nodo di contesto non è un nodo elemento.
- `self` indica solo il nodo di contesto stesso.

- `descendant-or-self` indica il nodo di contesto stesso più i suoi discendenti.
- `ancestor-or-self` indica il nodo di contesto stesso più i suoi antenati.

Ogni axis step può essere seguito da una lista di espressioni, racchiusi ognuno tra parentesi quadre, per selezionare i nodi che appartengono al risultato.

La query

```
document("piante.xml")/child::PIANTE/
  child::PIANTA[child::NOMESCIENTIFICO="Solanum Melogena"]
```

eseguita sul documento “piante.xml” che rispetta il DTD di figura 2.7 restituisce tutti i nodi “PIANTA”, figli della radice “PIANTE” del documento, che hanno un figlio “NOMESCIENTIFICO” con testo uguale alla stringa “Solanum Melogena”.

La query

```
document("piante.xml")/child::piante/
  child::pianta[child::CONSISTENZA]/child::NOMECOMUNE[1]
```

eseguita sullo stesso tipo di documento dell’esempio precedente seleziona tutti i nodi “PIANTA” che hanno almeno un figlio di tipo “CONSISTENZA”, e per ognuno di loro restituisce il primo figlio “NOMECOMUNE”.

In entrambi gli esempi precedenti si può notare l’uso della funzione `document(string)` che restituisce il nodo document del file XML il cui nome è identificato dal parametro passato per argomento, l’uso dell’axis step `child` e l’uso di tre diversi tipi di espressione per filtrare i nodi selezionati. Nell’ordine riscontrato troviamo: l’espressione predicato, l’espressione esistenziale, e l’espressione posizione.

La sintassi abbreviata prevede che alcuni nomi di direzioni siano semplificate:

- `child::` viene rimosso (è considerato come la direzione di default);
- `attribute::` diventa “@”;
- `descendant-or-self::node()` diventa “//”;
- `self::node()` diventa “.”;
- `parent::node()` diventa “..”.

Per le altre caratteristiche di XPath come le funzioni e i namespace, rimandiamo alla letteratura [XPa99, XPa03].

### 2.2.2 XQuery

Il W3C [W3C] ha riunito un gruppo di lavoro per progettare un linguaggio di interrogazione per dati XML. Il nuovo linguaggio chiamato XQuery è ancora in evoluzione e le sue caratteristiche sono definite in una serie di documenti pubblicati dal gruppo di lavoro [XQu04]. XQuery è un linguaggio funzionale contenente numerosi tipi di espressione che possono essere composte con completa generalità. Il linguaggio è basato sui sistemi di tipi di XML Schema, ed è progettato per essere compatibile con gli altri standard XML correlati (vedi [Cha02]). Uno degli standard esistenti che più ha influenzato la progettazione di XQuery è XPath, infatti XQuery include una parte di XPath.

Il progetto globale di XQuery è basata su una precedente proposta di linguaggio di interrogazione chiamato Quilt [CRF00], il quale a sua volta si ispira ad alcuni aspetti presenti in altri linguaggi di interrogazione già esistenti: OQL [OQL96] per le sue caratteristiche di linguaggio funzionale, SQL [SQL99] per la sintassi basata sulle parole-chiave, e alle precedenti proposte per linguaggi di interrogazione per dati XML come XQL [RLS], XML-QL [DFF<sup>+</sup>] e Lorel [AQM<sup>+</sup>97].

XQuery è un linguaggio funzionale Turing equivalente, quindi ogni espressione XQuery una volta eseguita restituisce un valore senza che si abbiano altri effetti laterali. In XQuery ci sono molti tipi di espressioni, composte da espressioni base composte con operatori e parole chiave.

La più semplice espressione di XQuery è l'espressione letterale, cioè la rappresentazione di un valore atomico, e sono presenti letterali di tipo intero (12 a esempio), di tipo decimale (3.5 ad esempio), di tipo decimale a doppia precisione (1.3E-3 ad esempio, distinguibile dal precedente per la presenza dell'esponente), e di tipo stringa ("35" oppure '44' ad esempio, le stringhe possono essere delimitate sia dal carattere " che dal carattere '). Le espressioni letterali possono essere usate solo per creare i 4 tipi predefiniti di XQuery, gli altri tipi derivati possono essere creati grazie ai costruttori, di cui parleremo in seguito.

Le espressioni letterali si possono usare con gli opportuni operatori per creare le prime espressioni complesse. Sono disponibili tutti gli operatori che ci si potrebbe aspettare per un linguaggio di programmazione standard che debba operare anche su alberi, per cui troveremo gli operatori aritmetici, booleani, quelli di comparazione (generica di valore e di ordine), e quelli di creazione delle sequenze (, to).

In XQuery è possibile definire e chiamare funzioni. Per la sintassi per la definizione di funzioni, rimandiamo alla letteratura [XQu04, BCF<sup>+</sup>04], mentre per l'utilizzo di funzioni, basta scrivere il nome della funzione seguita dall'elenco dei suoi parametri,

se presenti, separati da virgole, e racchiusi tra parentesi tonde. Un esempio di chiamata di funzione (predefinita) è: `substring("stringa di esempio",9, 10)` che restituisce la stringa “di”.

Una variabile in XQuery è un nome che inizia con il simbolo “\$” (`$i $var $item` sono tre esempi di variabili). Una variabile può essere legata ad un valore ed usata in una espressione per rappresentare il valore a cui è legata. Il costrutto base di XQuery è l’espressione **FLWR** (si legge come flower ed è l’acrostico di `for let where return`). L’esempio di figura 2.8 visualizza una semplice espressione **FLWR** che costruisce la sequenza “1 2 3”, perché, dopo aver assegnato i valori di 1 e 2 rispettivamente a `$a` e `$b` assegna alla variabile `$c` il valore di 3 (risultato della somma dei valori delle due variabili) e poi restituisce la sequenza formata dai numeri compresi tra 1 e 3.

```
let $a := 1, $b := 2, $c := $a + $b
return ($a to $c)
```

Figura 2.8: Esempio di semplice espressione **FLWR** con soli `let` e `return`.

Dall’esempio di figura 2.8 si vede l’utilizzo della clausola `let`, per la definizione di variabili, e l’utilizzo della clausola `return` che determina il valore da restituire.

Un altro modo di assegnare un valore ad una variabile è costituito dalla clausola “`for`”, che assegna iterativamente tutti gli elementi di in una sequenza alla variabile, come vediamo da esempio di figura 2.9 in cui vengono restituiti i primi 3 numeri pari.

```
for $a in (1, 2, 3)
return $a * 2
```

Figura 2.9: Esempio di semplice espressione **FLWR** con soli `for` e `return`.

Dall’esempio di figura 2.9 si vede come utilizzare la clausola “`for`” con una variabile di iterazione con valori appartenenti ad un sequenza definita. È possibile usare anche più variabili contemporaneamente, ed in questo caso vengono eseguiti più livelli di iterazione con il loop più interno rappresentato dallo scorrimento della variabile più a destra con il loop più esterno rappresentato dalla variabile più a sinistra.

L’ultima clausola **FLWR** è la clausola `where`, che serve a filtrare il risultato delle operazioni precedenti in base ad una condizione. Essa, a differenza delle clausole `for`

e `let`, può essere presente al più una volta in una singola espressione, e viene valutata per ogni tupla del risultato. Vediamo adesso un esempio completo di utilizzo delle quattro clausole insieme in figura 2.10 in cui vengono calcolati i primi cinque numeri pari, ma vengono restituiti solo quelli maggiori di 5.

```
for $a in (1 to 5)
let $b := $a * 2
where $b > 5
return $b
```

Figura 2.10: Esempio di semplice espressione **FLWR** con tutte le clausole.

Naturalmente le espressioni **FLWR**, come tutte le espressioni di XQuery, possono far riferimento non solo ad espressioni semplici, ma anche ad espressioni che denotano dei nodi di un albero XML. Per poter scegliere delle porzioni di un albero XML da analizzare con i costrutti specifici di XQuery si utilizza un sottoinsieme della sintassi di XPath. Attualmente XQuery supporta solo sei dei tredici axis definiti per XPath, e sono gli stessi descritti nella sezione precedente su XPath (vedi sez. 2.2.1).

Introduciamo ora il concetto di costruttori per la creazione di elementi. Il più semplice costruttore di elementi ha una sintassi analoga a quella per descrivere l'elemento in XML (vedi fig. 2.11).

```
<PIANTA famiglia="Solanacee">
  <NOME COMUNE>Melanzana</NOME COMUNE>
  <NOME SCIENTIFICO>Solanum Melogena</NOME SCIENTIFICO>
</PIANTA>
```

Figura 2.11: Esempio di un costruttore per un elemento “PIANTA”.

Più interessanti sono i costruttori che permettono di calcolare alcune parti dell'elemento. A tale scopo occorre che i campi che si vuole lasciare parametrici siano racchiusi tra parentesi graffe come in figura 2.12.

È anche possibile creare delle entità semplicemente combinando espressioni che denotano nodi, come in figura 2.12. Il risultato del costruttore di figura 2.12 è uguale a quello di figura 2.13 salvo che per il secondo caso le tre variabili non sono semplici stringhe, ma sono legate a dei nodi entità (come nel caso delle variabili `$nc` e `$ns`) o a nodi attributo (come nel caso della variabile `$fam`) specifici.

```

<PIANTA famiglia="{ $fam }">
  <NOME COMUNE>{ $nc }</NOME COMUNE>
  <NOME SCIENTIFICO>{ $ns }</NOME SCIENTIFICO>
</PIANTA>

```

Figura 2.12: Esempio di un costruttore parametrico.

```

<PIANTA>
{
  $fam,
  $nc,
  $ns
}
</PIANTA>

```

Figura 2.13: Esempio di un costruttore parametrico con sottoalberi.

Esiste infine un costruttore che permette di costruire un elemento parametrizzando anche il nome: il “costruttore di elementi calcolati”. Esso è costituito dalla parola chiave `element` seguita da due espressioni racchiuse in parentesi graffe. La prima espressione denota il nome del nuovo elemento, mentre la seconda restituisce i dati e gli attributi dell’elemento. Come per il “costruttore di elementi calcolati” esiste anche il “costruttore di attributi calcolati”: in esso la parola chiave è `attribute` e le due espressioni contengono nell’ordine il nome ed il valore dell’attributo da costruire. Entrambi i casi sono visibili nella figura 2.14.

```

element
{ $elemName }
{
  attribute{ $attName }{ $fam },
  $nc,
  $ns
}

```

Figura 2.14: Esempio di un “costruttore di elementi calcolati”.

Anche questo esempio può produrre lo stesso risultato degli altri esempi precedenti purché la variabile `$elemName` sia legato ad una stringa con valore "PIANTA",

`$attName` ad una stringa di valore "famiglia", e le altre variabili abbiano lo stesso valore che hanno nell'esempio precedente di figura 2.13.



# Capitolo 3

## Uno sguardo ai sistemi P2P esistenti

L'Intel P2P Working Group ha definito il P2P come “La condivisione di risorse e servizi tra computer tramite scambio diretto tra i sistemi interessati”. Data questa definizione si possono ricavare due caratteristiche fondamentali:

- **Scalabilità:** non c'è limite teorico o tecnico alla dimensione del sistema<sup>1</sup>;
- **Affidabilità:** il malfunzionamento di ogni nodo non ha importanti effetti sull'intero sistema<sup>2</sup>.

Le reti di file sharing come Gnutella [Lim, Kan01] sono un buon esempio di scalabilità e affidabilità. In Gnutella i Peer sono connessi ad una rete piatta in cui ogni Peer è perfettamente uguale agli altri. I Peer sono connessi direttamente tra di loro senza la necessità di un server principale, ed il malfunzionamento di uno qualunque di essi non influenza il corretto funzionamento degli altri.

Le reti P2P possono essere classificate in due modi a seconda di come è strutturata la rete: P2P puri, e P2P ibridi.

I P2P puri, come Gnutella e Freenet [Cla99, Fre03] costituiscono reti totalmente piatte, in cui non c'è nessuna differenza tra i Peer.

I modelli P2P ibridi come Napster, OpenNap [Shi01, Nap00] impiegano un server centrale per ottenere meta-informazioni sull'identità dei Peer che posseggono le informazioni richieste. Nei modelli ibridi di questo tipo prima di poter scambiare le informazioni direttamente tra i Peer occorre contattare un nodo che opera da server.

I modelli ibridi come FastTrack [Sha02] o OpenFT [GiF02], in cui alcuni Peer sono eletti al ruolo di SuperPeer<sup>3</sup>, sfruttano l'esistenza di Peer che sono più “presenti” di

---

<sup>1</sup>la complessità del sistema dovrebbe essere circa costante rispetto al numero di nodi del sistema

<sup>2</sup>né su ciascun altro nodo del sistema

<sup>3</sup>o UltraPeer, o Group-Leader-Peer in altre architetture

altri all'interno della rete, per leggerli a svolgere ruoli più importanti come quello di gestire i Peer cui fa capo<sup>4</sup>. Le politiche di elezione dei SuperPeer e la gestione della caduta di uno di essi può essere decisa a seconda delle altre caratteristiche della rete in modo da bilanciare diversamente fattori come: la dinamicità, la capacità di "autostrutturazione" o la velocità della rete che si vuol ottenere.

Il vantaggio di una struttura pura rispetto ad una ibrida sta nella maggiore resistenza ai fallimenti dei singoli nodi, in quanto il fallimento ravvicinato di più nodi con compiti da SuperPeer potrebbe, in alcuni casi, sconnettere alcuni Peer dalla rete.

Un'altra possibile classificazione dei sistemi P2P, ortogonale alla precedente, può essere fatta sulla organizzazione della ricerca delle informazioni. I sistemi si possono dividere in: sistemi strutturati e sistemi non strutturati. Nei sistemi non strutturati non c'è nessuna conoscenza sulla distribuzione delle risorse tra i diversi Peer sparsi sulla rete, quindi una ricerca di una risorsa non si può che fare inondando, in maniera più o meno controllata, la rete di messaggi di ricerca, così come avviene per Gnutella. Nei sistemi strutturati, invece, ogni Peer che si connette alla rete aggiorna lo stato della rete notificando le risorse da lui possedute alla rete, di modo che alla richiesta di una risorsa, che si sa essere posseduta da un Peer, non occorre inondare la rete di messaggi, ma si fa richiesta di quella risorsa solo (o quasi) ai Peer che ne hanno una, come avviene in Pastry [RD01] e CAN [RFH<sup>+</sup>01].

Un aspetto negativo della strutturazione è quello di rendere più complessi i protocolli di rete, che, oltre a gestire la distribuzione delle informazioni provenienti dai Peer, devono tenere al minimo il traffico di messaggi ad ogni comparsa e scomparsa di un Peer dalla rete, caratteristica da tenere in considerazione in reti con alto tasso di connessioni e disconnessioni, o con forte mobilità di un Peer all'interno della rete stessa.

### 3.1 Le topologie di reti P2P

Come definito in [Min01, Min02] una caratteristica che accomuna tutti i sistemi P2P è il fatto che le informazioni utili ad un Peer, e possedute da un altro, vengono scambiate direttamente tra due Peer, senza nessun intermediario. Invece ciò che contraddistingue i diversi sistemi P2P è il modo in cui un Peer interessato ad una informazione scopre l'identità del Peer che la possiede. Come classificato in [Min01, Min02] le reti P2P possono essere classificate in 4 topologie base: centralizzata,

---

<sup>4</sup>tenere l'indice dei file posseduti, e indirizzare così in modo più specifico le query nella rete, e mettersi in contatto con gli altri SuperPeer esistenti

decentralizzata, gerarchica e ad anello. È difficile però trovare un sistema complesso che utilizzi direttamente una di queste architetture, è più probabile che il sistema utilizzi una particolare combinazione di queste strutture allo scopo di formare un sistema che, pur essendo più complesso, riesce a sopperire ai difetti intrinseci di ciascuna delle strutture semplici. Darò ora una breve introduzione alle topologie base e alle più semplici combinazioni di esse.

### 3.1.1 Topologia centralizzata

La struttura di una topologia centralizzata è raffigurata in figura 3.1 ed è fortemente basata sul concetto di sistema Client/Server. Il server centralizzato serve per mantenere le informazioni sulla distribuzione delle informazioni sui Peer sparsi sulla rete, cosicché ogni Peer che si registra al sistema invia al server le informazioni e/o i servizi messi a disposizione degli altri, oltre al proprio indirizzo. Ogni volta che qualche Peer ha bisogno di una informazione/servizio chiede al server di inviargli la lista degli ip delle macchine che lo mettono a disposizione di modo da poterlo contattare direttamente per lo scambio.

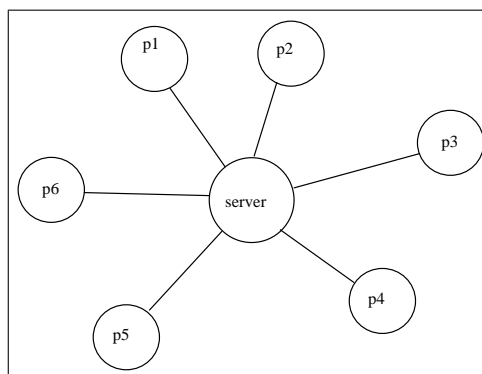


Figura 3.1: Esempio di topologia centralizzata.

Questa filosofia è adottata da Napster/OpenNap, seti@Home e folding@home.

### 3.1.2 Topologia gerarchica

I sistemi gerarchici funzionano delegando alcuni servizi più semplici, o meno importanti, a membri che si trovano ad una categoria più bassa, riducendo così il carico di lavoro ai livelli della gerarchia superiori, ma se un nodo di un livello inferiore non riesce a svolgere il compito richiesto allora passa il controllo al suo immediato superiore, e così via fino al raggiungimento del più alto nodo della gerarchia. Un esempio di struttura gerarchica è visibile in figura 3.2. In una topologia gerarchica è possibile tenere fisso, o permettere delle variazioni sia nella cardinalità degli elementi

di ordine inferiore per ogni elemento superiore, sia il numero di livelli della gerarchia. Un esempio di sistema che usa una gerarchia a numero di livelli fisso è OpenFT (con una gerarchia a tre livelli).

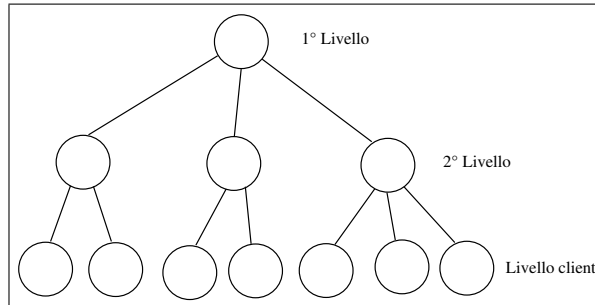


Figura 3.2: Esempio di topologia gerarchica.

### 3.1.3 Topologia ad anello

La topologia ad anello è costituita da un insieme di entità connesse tra di loro con una struttura circolare in cui ogni entità è connessa esattamente ad altre due. L'anello è una topologia che si presta bene alla distribuzione del carico di lavoro su più macchine in modo sufficientemente omogeneo a patto che si possa sopperire facilmente al fallimento di una di esse. Essa ha come forti vantaggi quelli di: distribuire bene il carico di lavoro tra i Peer, e avere una grande disponibilità, visto il basso carico di lavoro di un singolo nodo.

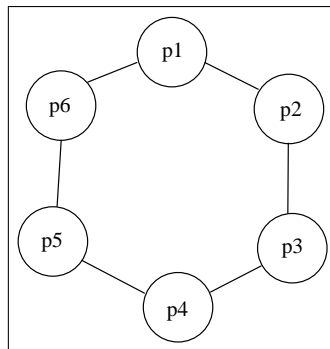


Figura 3.3: Esempio di topologia ad anello.

Questa topologia è indicata se si vuole aumentare la disponibilità di un server centrale quando da solo non è in grado di soddisfare tutte le richieste. È la topologia ideale quando si vuole trasformare un singolo server in cluster di più macchine che si distribuiscono il lavoro. Nella topologia ad anello ogni macchina può comunicare solo con i propri vicini<sup>5</sup> per scambiarsi informazioni, quindi per spedire un messaggio

<sup>5</sup>in alcuni casi il la percorrenza dell'anello è a senso unico come nelle reti ad anello

in un altro nodo potrebbero occorrere più propagazioni di uno stesso messaggio. Un esempio di organizzazione ad anello è visibile in figura 3.3

### 3.1.4 Topologia decentralizzata

Nel P2P puro non esiste nessuna forma di centralizzazione, tutti i Peer sono eguali, perciò creano una topologia piatta, apparentemente non strutturata (vedi fig. 3.4). Per collegarsi alla rete un nuovo Peer deve contattare un Peer che sa già esserci connesso, o perché sempre on-line, o perché si trova in una lista pubblica di Peer connessi alla rete. In questa topologia in realtà non c'è mai una conoscenza globale della rete, ma ogni Peer conosce solo gli altri Peer cui è collegato in un dato istante. In questa topologia per effettuare una ricerca è obbligatorio inondare la rete di messaggi per la ricerca, e poi la risposta può essere data direttamente tra il Peer richiedente e il Peer possessore (come imporrebbe la filosofia del P2P) o può percorrere la strada inversa del messaggio di richiesta (per mantenere l'anonimato assoluto tra i Peer). Un esempio di rete che fa uso della topologia decentralizzata è

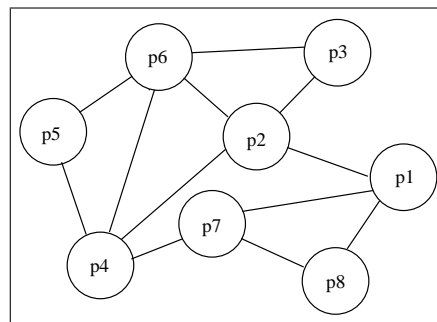


Figura 3.4: Esempio di topologia decentralizzata.

Gnutella<sup>6</sup>.

### 3.1.5 Alcune topologie composte

Esistono molti modi per comporre le topologie allo scopo di formarne una nuova più efficiente, e a seconda del sistema che si vuole creare è possibile combinarle in modo diverso.

Nella figura 3.5 è mostrata una topologia molto simile a quella usata dalla rete FastTrack, in cui ogni Peer fa riferimento ad un padre cui chiedere i servizi, come in una rete centralizzata, ed i padri (i SuperPeer in FastTrack) sono connessi tra loro con una rete decentralizzata.

<sup>6</sup>anche Gnutella nelle sue ultime versioni sta abbandonando la topologia piatta per passare ad una topologia più strutturata che le garantisce una minore fetta di banda occupata dai messaggi di controllo

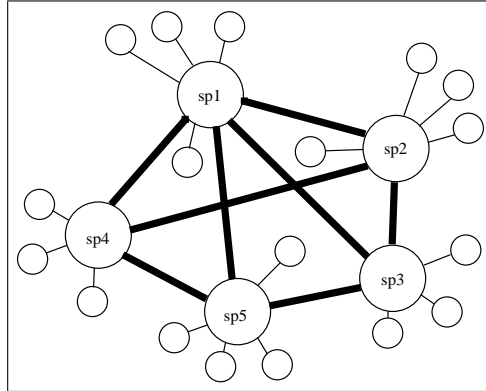


Figura 3.5: Esempio di topologia composta decentralizzata-centralizzata.

Nella figura 3.6 è mostrato un altro esempio di composizione di reti semplici per formare un sistema Client-Server in cui il server non è un'unica entità, ma un anello di Peer che servono i clienti. Questa topologia ibrida si può considerare molto simile a quella utilizzata dalla rete Napster/OpenNap.

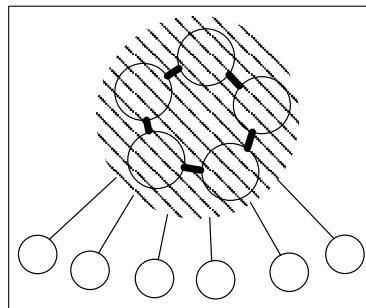


Figura 3.6: Esempio di topologia composta anello-centralizzata.

## 3.2 I principali modelli esistenti

### 3.2.1 Piazza

Piazza [Tat04] è un PDBMS (Peer Data Management System, Sistema per la Gestione di Dati su Peer), costruito come un P2P puro non strutturato, che ha come obiettivo quello di permettere l'integrazione tra dati, rappresentati in formato XML, che hanno schemi simili tra loro, o comunque traducibili da un formato all'altro. Piazza si può considerare il sistema più sviluppato per creare DBMS corporati che gestiscono dati XML distribuiti su rete. In Piazza chiunque può aggregarsi alla rete di Peer esistenti a patto di poter fare un mapping di tipo GLAV (vedi [FLM99]) con almeno uno dei Peer già presenti nella rete. Dopo la connessione alla rete si può effettuare una qualunque query formulata con un sottoinsieme delle query di

XQuery (vedi [XQu04]) e si avranno i risultati che provengono sia dalla propria base di dati locale, sia quelli provenienti dalle altre basi di dati locali a ciascun Peer della rete.

Il sistema di funzionamento di Piazza prevede l'uso di algoritmi che, avendo a disposizione le traduzioni da uno schema ad un altro dei Peer direttamente connessi tra loro, traduce la query riformulandola in una equivalente, ma su uno schema differente.

### 3.2.2 Gnutella

Gnutella è una rete P2P pura non strutturata ideata da J. Frankael e T. Pepper nel marzo del 2000 e rapidamente diffusa, nonostante sia stata subito ritirata dalla AOL, azienda per cui lavoravano i due ideatori subito dopo la pubblicazione.

Gnutella è strutturata come una rete piatta decentralizzata in cui ogni Peer è perfettamente uguale agli altri. In Gnutella ogni Peer per richiedere un servizio invia una query ai suoi vicini, i quali la elaborano e propagano il messaggio di query arrivato agli altri vicini i quali a loro volta ripetono l'operazione. L'elaborazione di un messaggio consiste nel verificare se le risorse locali al Peer soddisfano la richiesta arrivata dalla query, e, in caso affermativo, inviare una risposta al Peer che ha inviato il messaggio. C'è da notare che in una rete Gnutella i messaggi vengono propagati solo tra vicini, anche per la risposta alle query<sup>7</sup>. Questo meccanismo particolare rende più inefficiente la gestione dei messaggi, ma garantisce un maggior anonimato tra i Peer.

Per potersi connettere alla rete un nuovo Peer deve prima conoscere l'indirizzo di un Peer già connesso, e per fare ciò di solito si utilizza una cache di macchine (di solito la GnuCache) che sono praticamente sempre connesse. Dopo aver contattato una delle macchine presenti nella cache attende risposta positiva da esse per potersi considerare connesso alla rete. Un Peer già connesso che riceve la richiesta di connessione da un nuovo Peer può anche rifiutare la richiesta di connessione per diversi motivi come il raggiungimento del numero massimo di connessioni supportate, una differente versione del protocollo ecc. Una volta connesso alla rete un Peer periodicamente effettua dei ping ai suoi vicini per scoprire se ci sono dei nuovi Peer nella rete cui connettersi. Tipicamente un Peer è connesso contemporaneamente a più di un Peer e i Peer cui è connesso possono cambiare sia in identità che in numero, a causa di fallimenti o disconnessioni volontarie.

---

<sup>7</sup>a causa di questa politica c'è la possibilità di perdita di messaggi di risposta se tra il passaggio del messaggio di query, e quello di risposta una delle connessioni percorse dal primo messaggio è saltata

Per ridurre il traffico di messaggi in una rete è stato introdotto il meccanismo degli identificatori di messaggio che permettono di attuare meccanismi di cache temporale presso ogni Peer che non propaga mai lo stesso messaggio più di una volta<sup>8</sup>. Un'altra tecnica adottata è l'utilizzo di messaggi con "Time-To-Live" per limitare il diametro di propagazione dei messaggi nella rete.

L'evoluzione della rete Gnutella però tende ad abbandonare la struttura tipica a rete decentralizzata pura per convergere verso una rete ibrida gerarchica-decentralizzata per ridurre il traffico di messaggi di controllo necessari al corretto funzionamento del protocollo [CCR04].

### 3.2.3 P-Grid

P-Grid è un sistema strutturato con topologia decentralizzata per la gestione di dati distribuiti. La gestione della strutturazione è affidata ad un sistema di gestione di un "albero di ricerca distribuito virtuale" (VDST), che è strutturato in maniera molto simile ad una tabella Hash distribuita (DHT) standard.

Nell'esempio è possibile vedere un albero di ricerca binaria virtuale in cui ai nodi "00" e "10" sono assegnati due Peer, mentre agli altri due nodi è assegnato un solo Peer. In questo sistema, come in tutti i sistemi basati su tabelle Hash, i dati gestiti da un Peer non sono i dati di cui il Peer dispone, ma sono dati che provengono da altri Peer nel sistema che per risultato di una funzione gli sono stati dati in gestione, mentre i suoi dati sono dati in gestione ad altri. Ogni Peer quindi ha dei dati locali, che per essere pubblicati devono essere distribuiti ad altri per tutto il tempo di connessione del Peer alla rete.

Nel sistema P-Grid in particolare ogni Peer, oltre ad avere un identificatore univoco per l'identificazione fisica del Peer (nell'esempio il numero scritto nel cerchio nero) possiede anche un identificatore dei dati gestiti (il numero che etichetta la foglia dell'albero in cui è il nodo) che identifica quale gruppo di dati gestisce, e una tabella di routing per conoscere qual è il nodo da contattare per accedere a dati che hanno un identificatore diverso dal proprio. Gli identificatori dei dati possono assumere tutti i valori che appartengono al codominio della funzione Hash utilizzata per la ricerca dei dati. La tabella di routing invece contiene una rotta per ciascun prefisso di identificatore logico diverso dal proprio che riferisce a uno dei Peer che gestiscono il prefisso in questione.

Per chiarire meglio il significato delle strutture possedute da un nodo vediamo un esempio pratico di come viene effettuata una ricerca dei dati sull'esempio in

---

<sup>8</sup>tecnica sufficiente a prevenire i cammini circolari in un grafo quale è quello dei nodi di Gnutella

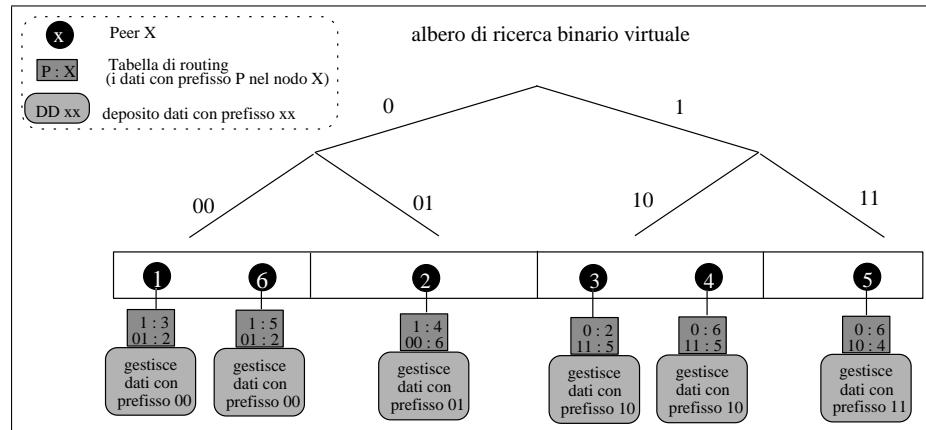


Figura 3.7: Esempio di VDSST in P-Grid.

figura 3.7. supponiamo che il Peer con identificatore “6” abbia bisogno di dati la cui funzione Hash dia come risultato l’identificatore logico “10”. Allora il Peer “6” non essendo il gestore dei dati con identificatore “10” controlla nella sua tabella di routing qual è il nodo con il prefisso comune a “10” e trova il nodo “5” che gestisce il prefisso “1”, invia quindi la richiesta del dato al nodo 5, il quale, a sua volta non essendo il gestore del dato con l’identificatore desiderato, controlla nella sua tabella di routing e rispedisce la query al nodo “4” il quale infine essendo uno dei gestori del dato desiderato può rispondere al nodo che ha iniziato la query.

La separazione tra identificatori logici e identificatori fisici ideata per la rete P-Grid ha lo scopo di permettere protocolli di bilanciamento del carico di lavoro. In P-Grid infatti sono previsti algoritmi per il riasegnamento dei dati da gestire a seconda delle richieste di particolari dati. Infatti se una qualunque delle fogli dell’albero risulta sovraccarica rispetto alle altre, è prevista la possibilità di scegliere uno dei nodi che ha un basso carico di lavoro e spostarlo per affiancare gli altri Peer già presenti a gestire quel particolare nodo, cambiando solo la tabella di routing del nodo da spostare, e aggiornando parte delle tabelle di routing dei nodi che puntano al Peer che risultava sovraccarico per farlo puntare al nuovo entrato.

Esistono poi meccanismi per gestire l’aggiornamento dei dati e l’entrata e l’uscita di nuovi Peer all’interno del sistema, che basandosi su algoritmi randomizzati assicurano con buona probabilità la consistenza delle informazioni distribuite con la tecnica Hash.

Come in tutti i sistemi P2P strutturati che fanno ricorso a tecniche DHT o derivate, il sistema risulta molto efficiente per query con ricerca esatta, mentre risulta inutile per tutti gli altri tipi di query.

### 3.2.4 FastTrack/OpenFT

FastTrack è una rete proprietaria di cui non è stata pubblicata nessuna specifica, ma di cui si conoscono alcune caratteristiche fondamentali grazie ad uno studio di reverse engineering compiuto da un gruppo di lavoro riunito sotto il nome di giFT (“Generic Interface to FastTrack”, interfaccia generica a FastTrack) presso SourceForge [GiF02]. Le caratteristiche del protocollo OpenFT sono per questo molto simili a quelle del protocollo di FastTrack. Nel 2003 è stata aggiunto un livello di criptazione al protocollo di FastTrack che ne ha reso impossibile uno studio più approfondito. Da quel momento in poi i due protocolli sono diventati indipendenti, ma molte caratteristiche di FastTrack sono state scoperte e documentate. D’ora in poi descriveremo la parte dell’architettura di FastTrack che è stata ricostruita, aggiungendo in fondo le caratteristiche che sono state aggiunte ad OpenFT dai membri del progetto giFT.

Il protocollo FastTrack nasce dall’esigenza di superare i problemi che si erano riscontrati in Napster (limitato dalla presenza di un server centrale) e in Gnutella (limitato dal traffico di controllo che occupa molta banda e/o dal raggio di ricerca limitato dal TTL). Come già detto precedentemente, FastTrack è basato su di un’architettura ibrida centralizzata-decentralizzata in cui la rete di SuperPeer è connessa con topologia decentralizzata, e i Peer semplici hanno più di un SuperPeer assegnato come server (di solito tre) per resistere ai fallimenti di un SuperPeer.

Il compito di un SuperPeer è quello di tenere un indice delle informazioni possedute dai Peer di cui è padre e di propagare i messaggi di ricerca nella rete decentralizzata di SuperPeer in maniera praticamente identica a quella usata da Gnutella. L’utilizzo di questa tecnica, nel contesto del protocollo FastTrack, risente meno del problema della perdita di messaggi per caduta di connessioni tra nodi, in quanto i SuperPeer sono tali anche in quanto più affidabili, e connessi tra di loro da una rete a più larga banda.

I Peer per potersi connettere alla rete hanno bisogno di conoscere l’indirizzo di uno dei SuperPeer già presenti nella rete. Esistono dei nodi che sono sempre connessi alla rete e che monitorano continuamente la rete per tenere traccia dei SuperPeer presenti. I Peer prendono la lista di SuperPeer dai nodi speciali di monitoraggio e provano a connettersi ad alcuni di essi fino a quando non si è raggiunto il numero desiderato di padri. Durante la vita, un Peer viene a conoscenza dell’esistenza di nuovi SuperPeer, e ne tiene gli indirizzi memorizzati in cache per poterli utilizzare per la ricerca. Infatti un Peer quando deve effettuare un’operazione di ricerca invia il messaggio di query non solo ai propri genitori, ma a tutti i SuperPeer che conosce, per avere un risultato più rapido e completo.

OpenFT possiede una struttura a tre livelli invece che i due di FastTrack: clienti, ricerca e indice. Il livello clienti è costituito dai Peer semplici che effettuano le ricerche e pubblicano i dati che posseggono a diversi padri che appartengono al livello ricerca. Il secondo livello costituito dai nodi ricerca che hanno caratteristiche del tutto equivalenti a quelle dei SuperPeer di FastTrack, e possono gestire fino ad un massimo di 500 figli. Il terzo livello è costituito da un numero molto esiguo di macchine (visti i requisiti molto stringenti per entrare a far parte del gruppo, tra cui l'affidabilità e la permanenza nella rete) e svolgono il compito di tenere l'indice dei nodi ricerca presenti, di effettuare statistiche sul sistema, e controllare la struttura della rete. Questa struttura solo lievemente diversa dalla precedente è stata creata con lo scopo di creare un sistema che possa permettere di connettere numerose reti esistenti per creare un'unica rete di scambio informazioni tra le reti già esistenti. Il significato della sigla giFT è infatti ora considerato "Generic Internet File Transfer".

### 3.2.5 Altri

Esistono moltissimi altri sistemi studiati sotto diversi aspetti, per avere un'idea basta leggere [RM04] che parla solo dei meccanismi di ricerca di risorse su moltissimi sistemi P2P, oppure controllare la letteratura su [Cit, Sch]. Un'attenzione particolare va posta su Freenet che ha una struttura molto simile a quella di Gnutella, salvo il fatto che possiede algoritmi di ristrutturazione della rete. In Freenet infatti ogni volta che un Peer esegue delle query ed ottiene delle risposte da un Peer che non è suo vicino, allora la rete cambia struttura per tendere ad avvicinare Peer che più frequentemente si scambiano dati.



## Capitolo 4

# Progettazione del sistema XPeer

La progettazione del sistema è stata influenzata dall'idea iniziale di creare un sistema P2P strutturato, con topologia ibrida (derivata da una topologia centralizzata-gerarchica come in figura 4.1), per avere un sistema che sia in grado di gestire diverse tipologie di interrogazioni su una rete dinamica in modo semplice.

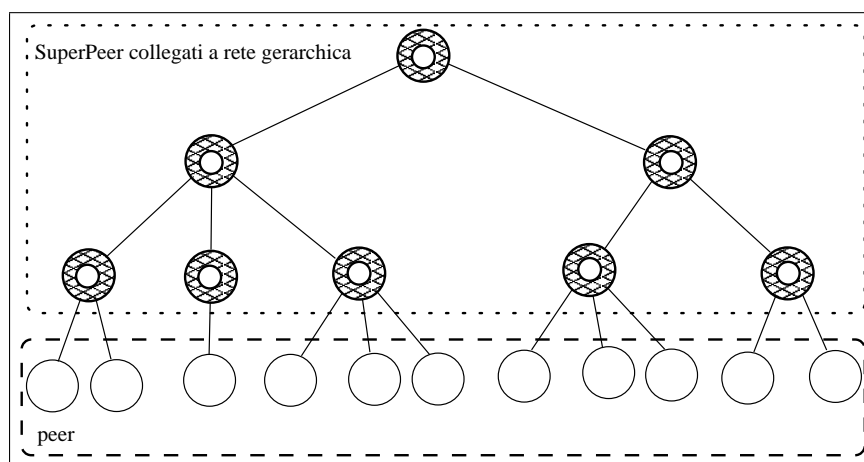


Figura 4.1: Topologia del sistema *XPeer*.

Il sistema *XPeer* è creato con l'obiettivo di avere un unico sistema globale per la condivisione di informazioni, utilizzato da tutti. Non è necessario cioè avere più istanze del sistema, una per ogni settore di informativo, ma è sufficiente creare un unico sistema che si organizzerà automaticamente per creare dei sottogruppi che hanno interessi comuni e che non appesantiscono di richieste la parte di sistema che non possiede le informazioni desiderate. Il sistema sarà cioè in grado da solo di creare dei gruppi che sono abbastanza autonomi tra loro e che grazie a questa autonomia sfruttano la struttura gerarchica della rete per tenere al minimo il traffico di messaggi di controllo. Un piccolo esempio del funzionamento del raggruppamento è visibile in figura 4.2 dove si può vedere come tre gruppi distinti di ricercatori sono connessi al sistema, il quale è organizzato autonomamente in una struttura organizzata con

gruppi simili vicini tra loro avendoli riconosciuti tramite gli schemi dei dati che pubblicavano al momento della loro connessione.

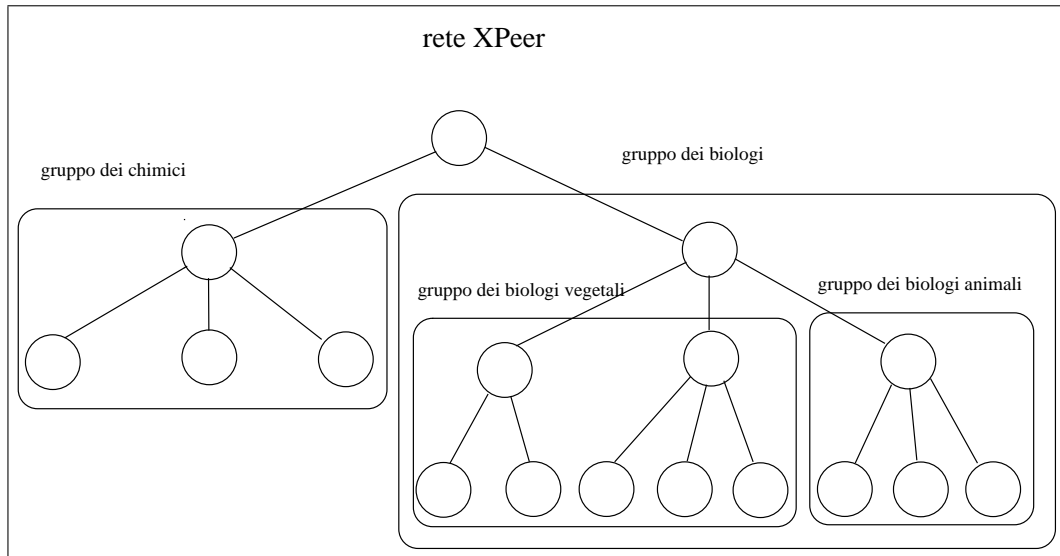


Figura 4.2: Esempio di raggruppamento.

## 4.1 Architettura del sistema

La scelta dell'architettura della rete è stata influenzata dall'idea di come eseguire una interrogazione sui dati distribuiti.

L'esecuzione di una interrogazione è divisa in due parti: la compilazione dell'interrogazione, per creare un piano di accesso ai dati, e l'esecuzione del piano di accesso, per eseguire il piano di accesso creato con l'operazione precedente, e produrre il risultato dell'interrogazione distribuita.

La fase di compilazione serve ad individuare l'insieme dei Peer sulla rete che posseggono dati con una struttura "compatibile" con quella della query, e i SuperPeer che effettuano la ricerca sanno instradare la query attraverso la rete per diffondere il messaggio di query solo ai SuperPeer vicini che hanno uno schema della base di dati "compatibile" con l'interrogazione. La fase di compilazione termina con la raccolta dei messaggi di risposta che contengono l'indirizzo dei Peer che partecipano all'interrogazione, con il proprio schema, e la costruzione di un piano di accesso ai dati (locali e remoti) per costruire la risposta finale. Un esempio della compilazione è visibile in figura 4.3.

La fase di esecuzione del piano di accesso prevede la visita dell'albero del piano di accesso precedentemente creato eseguendo localmente le sottoquery per i dati locali ed inviando le richieste di esecuzione di sottoquery direttamente ai Peer re-

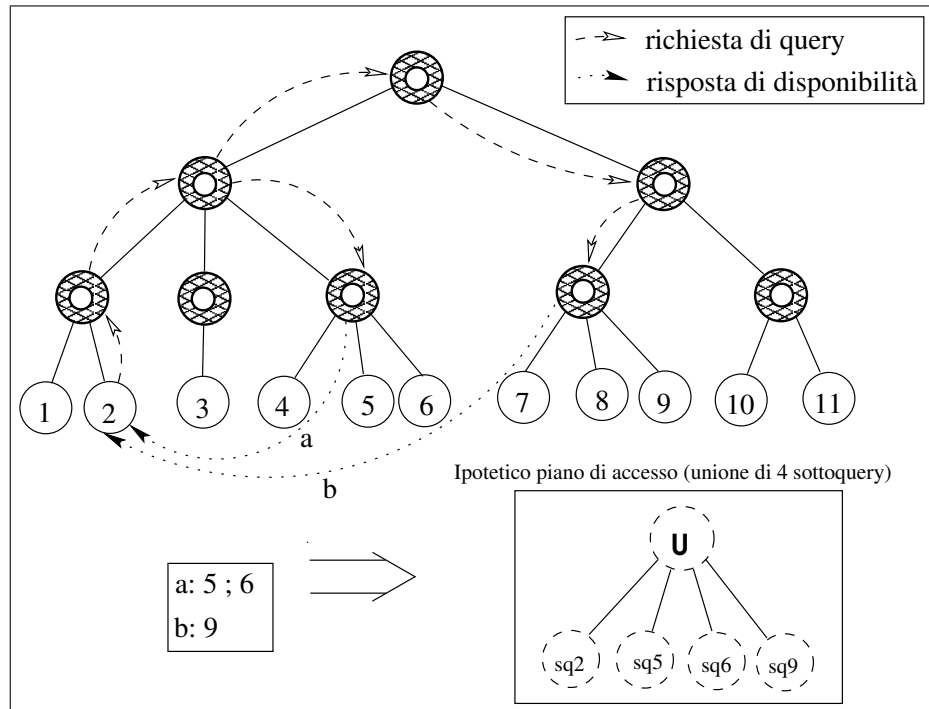


Figura 4.3: Compilazione di una query.

moti nel caso di dati remoti, e componendo i risultati così ottenuti. Un esempio dell'esecuzione del piano di accesso è visibile in figura 4.4.

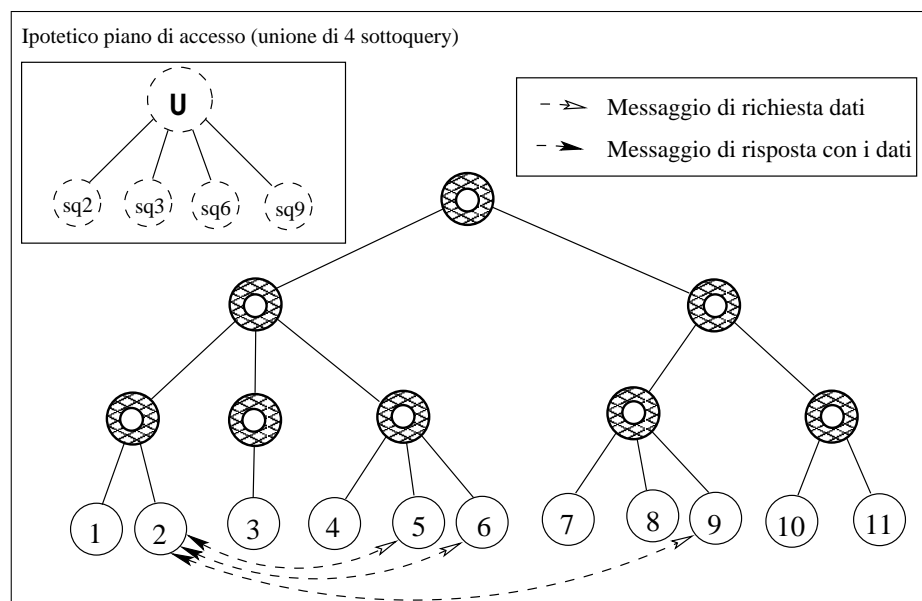


Figura 4.4: Esecuzione di una query.

### 4.1.1 Struttura della rete

Per evitare di inondare la rete di messaggi si è adottata una rete strutturata, in cui l'informazione distribuita sulla rete è costituita dalle informazioni relative allo schema posseduto da ogni singolo Peer, in modo da indirizzare i messaggi di query solo verso i Peer che hanno uno schema che è compatibile con la query da eseguire.

La strutturazione segue una filosofia diversa da quella dei tradizionali algoritmi DHT (“Distributed Hash Table”, tabella hash distribuita) che pur essendo estremamente efficienti per query di selezione per valore, non danno nessun vantaggio nel caso di query associative o query su intervallo. La struttura distribuita sulla rete è relativa agli schemi posseduti dai Peer che pur potendo essere molto simili tra loro<sup>1</sup> non sono totalmente identiche, più altre informazioni su delle costanti presenti all'interno dello schema che rendono più selettive le propagazioni di interrogazioni.

Grazie a questo tipo di strutturazione siamo in grado di propagare le query solo verso quei peer che posseggono uno schema che contiene le informazioni richieste dalla query. Per poter utilizzare questo tipo di strutturazione della rete è stato sviluppato un algoritmo in grado di estrarre lo schema da una base di dati XML, e sono state create strutture dati in grado di manipolare efficientemente tali schemi.

### 4.1.2 Gestione della robustezza

Il sistema *XPeer* possiede un meccanismo di prevenzione dei fallimenti del sistema causati dal fallimento (o dalla semplice disconnessione dal sistema) di un nodo.

Il meccanismo di prevenzione dei fallimenti che permette di risolvere in modo semplice e con grado di robustezza definibile il problema della dinamicità della rete (cioè della scomparsa improvvisa di nodi) è la clonazione. Ogni SuperPeer è clonato in un numero di cloni maggiore di uno e sufficiente a garantire un'adeguata resistenza ai fallimenti, e conseguentemente un carico di lavoro adeguato e bilanciato tra i cloni. Ogni SuperPeer è presente come un insieme di cloni collegati l'uno all'altro da un grafo completo con topologia decentralizzata (vedi fig. 4.5), in modo da garantire una sincronizzazione in un tempo che è lineare nel numero di cloni (salvo presenza di fallimenti di cloni).

### 4.1.3 Caratteristiche supplementari

Per semplificare la scrittura dei protocolli abbiamo deciso di creare due livelli principali in cui dividere il sistema:

---

<sup>1</sup>altrimenti sarebbe impossibile effettuare delle interrogazioni sensate

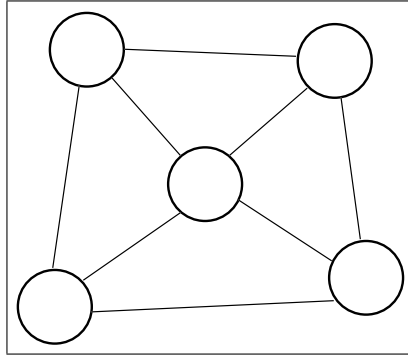


Figura 4.5: Topologia dei cloni di un SuperPeer.

- un sottosistema di comunicazione, che ci permette di spedire i messaggi attraverso la gerarchia di nodi, e che gestisce automaticamente le comunicazioni all'interno dei cloni di un SuperPeer;
- un sistema di gestione dei Peer che, sfruttando il sottosistema di comunicazione, si occupa della gestione della gerarchizzazione di Peer e della compilazione delle interrogazioni.

Nel nostro sistema il ruolo di SuperPeer è svolto dalle stesse macchine che sono anche Peer, cioè ogni Peer connesso al sistema può, se ce n'è bisogno, ricevere una richiesta di attivazione del processo che gestisce i SuperPeer, e vedersi presente nel sistema in due posizioni diverse. L'elezione a SuperPeer non provoca infatti l'abbandono del ruolo di Peer, ma comporta la comparsa di un nuovo nodo di tipo SuperPeer che serve a far funzionare meglio il sistema.

## 4.2 Architettura del singolo nodo

In ogni nodo sono distinguibili tre componenti principali (vedi fig. 4.6): il livello Peer, il livello sistema di comunicazione, e il livello SuperPeer (che come detto precedentemente può non essere attivo).

### 4.2.1 Il livello Peer

Nel livello Peer si riscontrano due entità principali: il "Sistema di Gestione della Base di Dati" (SGBD) locale, e il gestore del Peer.

Il gestore della base di dati locale è un normale SGBD centralizzato per gestire dati XML esteso con la possibilità di propagare interrogazioni verso altri SGBD analoghi, e di ricevere interrogazioni da altri sistemi omologhi esterni. Infatti se si eseguono interrogazioni sui dati XML in un momento in cui il nodo non è connesso

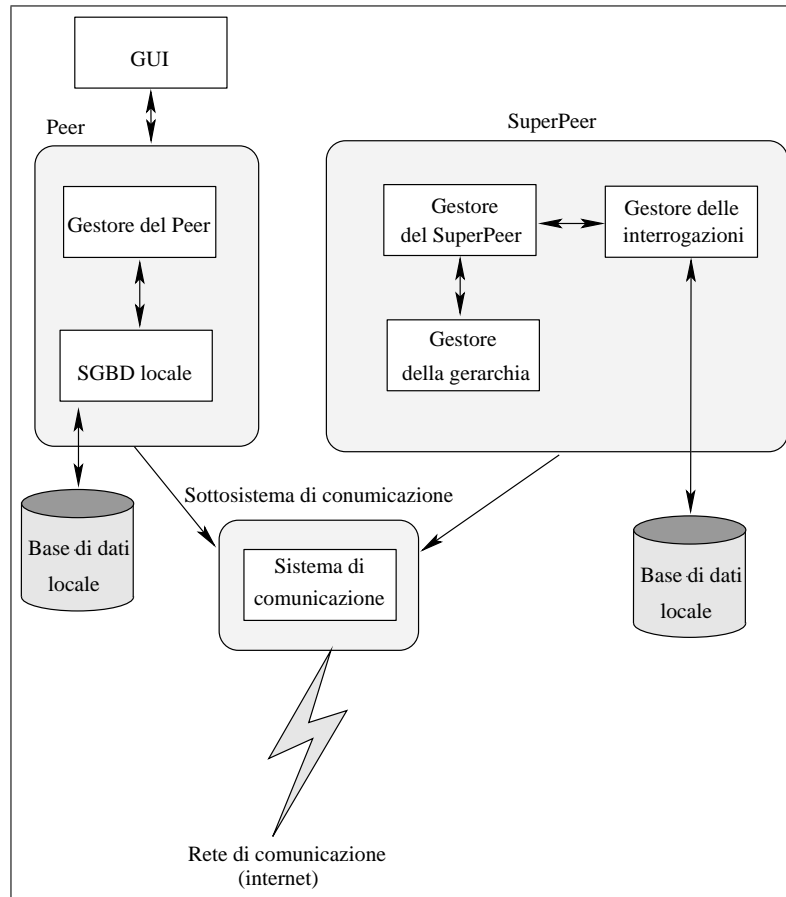


Figura 4.6: Architettura di un nodo

alla rete si hanno comunque delle risposte, ma si avranno solo i risultati ottenibili come se il SGBD gestisse una base di centralizzata in locale. Il SGBD utilizzato è Xtasy [Sar03] (recentemente esteso con la grammatica del W3C di XQuery del 2004 in [Mon04]).

Il gestore del Peer è l'entità basilare del sistema, esso per poter funzionare all'interno dell'intera rete deve potersi connettere ad essa. Il gestore del Peer deve possedere delle strutture dati che gli servono per poter far parte della rete *XPeer*, e sono:

- l'indirizzo del SuperPeer padre con cui interagire per le principali operazioni;
- l'identità con cui riconoscersi univocamente nella rete<sup>2</sup>;
- lo schema della base di dati che si possiede, ricavato dalla base di dati sottostante e aggiornato costantemente.

La connessione di un Peer al sistema è regolata da un insieme di protocolli detti protocolli di innesco nella rete, e può essere iniziata solo conoscendo l'identità di

<sup>2</sup>l'identità del Peer e del SuperPeer che si trovano sullo stesso nodo sono differenti

un Peer (o di un SuperPeer) che fanno parte del sistema. Se si deve avviare per la prima volta la rete il Peer che vuole dare il via al sistema deve creare l'entità Peer, creare l'entità SuperPeer, e avviare i protocolli di innesco con l'entità SuperPeer posseduta nel nodo locale, si ha così un primo abbozzo di rete con un solo Peer e un solo SuperPeer<sup>3</sup>.

### 4.2.2 Il livello SuperPeer

Il livello SuperPeer, attivo su un nodo solo se richiesto dal sistema, rappresenta un clone di un SuperPeer che insieme ad altri con i costituiscono l'entità SuperPeer virtuale.

Un SuperPeer virtuale è un'entità astratta che è presente ad un qualunque livello della gerarchia ed ha un padre (salvo se è il SuperPeer radice) e un insieme di figli (che possono essere tutti SuperPeer virtuali o tutti Peer) ed è costituito da un insieme di cloni collegati tra loro con un grafo completo. Un clone è un processo attivo su di un nodo (il processo SuperPeer visibile nella figura 4.6) che fornisce le funzionalità necessarie alla costituzione di un SuperPeer virtuale insieme ad altri processi analoghi presenti su altri nodi.

Il SuperPeer virtuale concorre a costituire il nucleo del sistema di instradamento delle query e di funzionamento complessivo del sistema. Per poter funzionare un SuperPeer deve possedere le seguenti strutture:

- il padre, che indica l'immediato superiore nella gerarchia dei Peer;
- la tabella dei figli, contenente l'elenco dei figli del SuperPeer con i metadati di ciascuno di loro;
- l'identità con cui identificarsi univocamente all'interno del sistema<sup>4</sup>;
- il "Superschema" che contiene l'insieme dei metadati di tutti i figli del SuperPeer.

La gestione della gerarchia dei SuperPeer è affidata ad un insieme di protocolli a livello di sistema di gestione dei peer (i protocolli di evoluzione della rete), mentre la gestione della clonazione è affidata a protocolli a livello di sottosistema di comunicazione. La gestione dell'instradamento delle query è affidata al gestore delle interrogazioni che, grazie alle informazioni sui metadati dei figli è in grado

---

<sup>3</sup>in questo caso il SuperPeer è costituito da un solo clone, per mancanza di nodi, ma aumenterà il grado di clonazione al primo innesco di un altro nodo

<sup>4</sup>l'identità del SuperPeer virtuale è univoca rispetto sia agli altri SuperPeer virtuali sia agli altri Peer

di instradare opportunamente i messaggi di query attraverso la gerarchia (con i protocolli di elaborazione delle query).

Nel corso della descrizione del sistema si farà riferimento al SuperPeer senza qualificare se esso è un SuperPeer virtuale o un processo SuperPeer che costituisce un clone di un SuperPeer virtuale, ma la distinzione sarà riconoscibile dal contesto. In ogni caso solo il sistema di comunicazione è a conoscenza della presenza dei cloni, mentre il sistema di gestione dei Peer conosce solo i SuperPeer virtuali.

### 4.2.3 Il sottosistema di comunicazione

Il Sistema di Comunicazione è un'entità che permette di inviare messaggi strutturati su code di messaggi presenti su nodi remoti, astruendo dai protocolli di rete usati effettivamente per l'implementazione. Esso si occupa inoltre della gestione automatica del riconoscimento dell'indirizzo della macchina fisica su cui è presente il nodo, e di tutte le operazioni legate alla funzionalità della clonazione, rendendo così il livello Peer e quello SuperPeer inconsapevoli dell'esistenza dei cloni di un SuperPeer.

Per poter svolgere il suo compito il sottosistema di comunicazione deve possedere le seguenti strutture:

- la mappa di corrispondenza tra SuperPeer virtuali e i cloni che lo costituiscono;
- una rete di comunicazione con cui costruire il sistema di code di messaggi.

## 4.3 Descrizione dei protocolli

I protocolli necessari alla gestione del sistema sono stati divisi in gruppi identificati per il compito dei protocolli. Troviamo così 6 gruppi di protocolli da cui possiamo distinguere:

1. i protocolli di sistema che non sono veri e propri protocolli, ma sono piuttosto dei metaprotocolli che servono a semplificare la creazione degli altri protocolli;
2. i protocolli di innesco per permettere la connessione di un peer al sistema, o di un SuperPeer ad un nuovo padre;
3. i protocolli di evoluzione della rete che permettono la ristrutturazione della rete in caso di sovraccarico o di sottocarico di un SuperPeer;
4. i protocolli di elaborazione delle query che eseguono l'instradamento delle query;

5. i protocolli di comunicazione che gestiscono i meccanismi di spedizione dei messaggi e mantengono sincronizzate le informazioni tra i cloni di uno stesso SuperPeer.

Nella descrizione dei protocolli si farà riferimento a spedizioni di messaggi a Peer. Il significato vero di questa frase è in realtà quello di spedire messaggi verso identificatori che possono identificare Peer o SuperPeer. Nei casi in cui il significato è diverso verrà specificato di volta in volta.

### 4.3.1 Protocolli di sistema

I protocolli di sistema sono costituiti da due meta-protocolli e due protocolli veri e propri a livello di sistema di gestione dei Peer (che quindi sono inconsapevoli della clonazione). I due meta-protocolli che abbiamo chiamato “MetaSearch” e “MetaUpdate” servono a propagare i messaggi di richiesta lungo la rete di SuperPeer secondo due meccanismi che servono rispettivamente a cercare delle risorse nella rete, o ad aggiornare delle strutture dati. I due protocolli: “ModifyChildrenTable” e “SchemaUpdate” servono invece rispettivamente a modificare la tabella dei figli di un SuperPeer, e ad aggiornare il padre con una nuova versione dello schema posseduto.

Entrambi i meta-protocolli si compongono di due parti: la parte di gestione dell'instradamento dei messaggi, e la parte di calcolo delle risposte locali. La parte di gestione dell'instradamento dei messaggi è unica e ben definita per ciascuno dei due meta-protocolli, mentre la parte relativa al calcolo delle risposte locali va definita per ogni protocollo che fa uso della “MetaSearch” o della “MetaUpdate” per eseguire le operazioni a lui necessarie.

#### MetaSearch

La “MetaSearch”, è eseguita sia sui Peer che sui SuperPeer, e serve a trovare risorse nel sistema. La “MetaSearch” utilizza messaggi di ricerca che sono propagati attraverso la rete ed elaborati localmente su ogni Peer su cui passano (vedi fig. 4.7).

Il risultato dell'elaborazione locale, oltre a contenere delle risposte locali alla richiesta pervenuta, contiene l'elenco dei Peer a cui propagare il messaggio. L'insieme dei Peer cui propagare il messaggio di ricerca può essere un qualunque sottoinsieme di quello formato da tutti i Peer figli più il Peer padre meno il Peer da cui si è ricevuto il messaggio. Dopo aver calcolato il risultato locale, la “MetaSearch” si occupa di propagare il messaggio di meta-ricerca, se l'insieme dei destinatari non è vuoto o non è stato raggiunto un time-to-live indicato nel messaggio di meta-ricerca. L'ultima

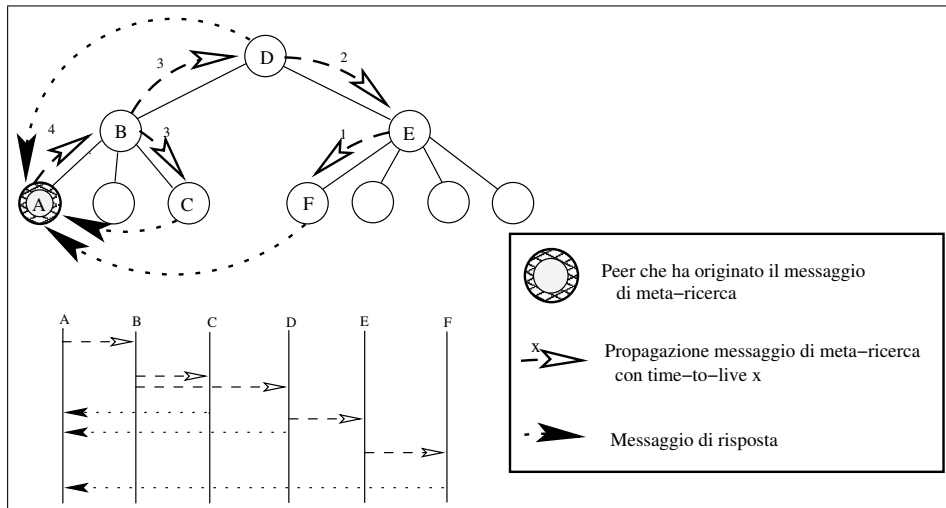


Figura 4.7: Esempio di propagazione di un messaggio di MetaSearch.

azione effettuata dalla “MetaSearch” è quella di spedire il messaggio di risposta al Peer che ha originato il processo di meta-ricerca, se è stato prodotto un valido risultato locale. Un esempio di propagazione del messaggio è visibile in figura 4.7 dove si possono notare i time-to-live dei messaggi in propagazione, e le risposte che vengono dirette a colui che ha originato il messaggio. Lo pseudocodice che costituisce la MetaSearch è visibile in figura 4.8.

```

MetaSearch(MetaMessage[action,responseNode,invokerNode,msgId,params] msg){
  //Il valore di res.localResult dipende dall'azione eseguita
  MetaResult[responseHeader, localResult, params, nextNodes] res = executeAction(msg);
  for node n ∈ res.nextNodes ≠ sender do{
    //Notiamo che res.params è sempre uguale al corrispondente valore in msg
    send(n, ['metaSearch', msg.action, msg.responseNode,msg.msgId, res.params])
  }
  if (localResult ≠ 0 ;)then{
    //Il risultato viene spedito indietro all'originatore del messaggio solo se ha un valore utile
    sendAnswer(msg.responseNode, [res.responseHeader, msg.msgId, res.localResult, this.Identity])
  }
}

```

Figura 4.8: Pseudocodice della MetaSearch.

## MetaUpdate

La “MetaUpdate” è eseguita solo sui SuperPeer e serve a chiedere degli aggiornamenti su dei SuperPeer adiacenti tramite messaggi di meta-aggiornamento che vengono propagati nella rete e che stavolta chiedono conferme esplicite di aggiornamento effettuato (vedi fig. 4.9).

Il funzionamento della “MetaUpdate” prevede l'esistenza di tre fasi locali ad ogni

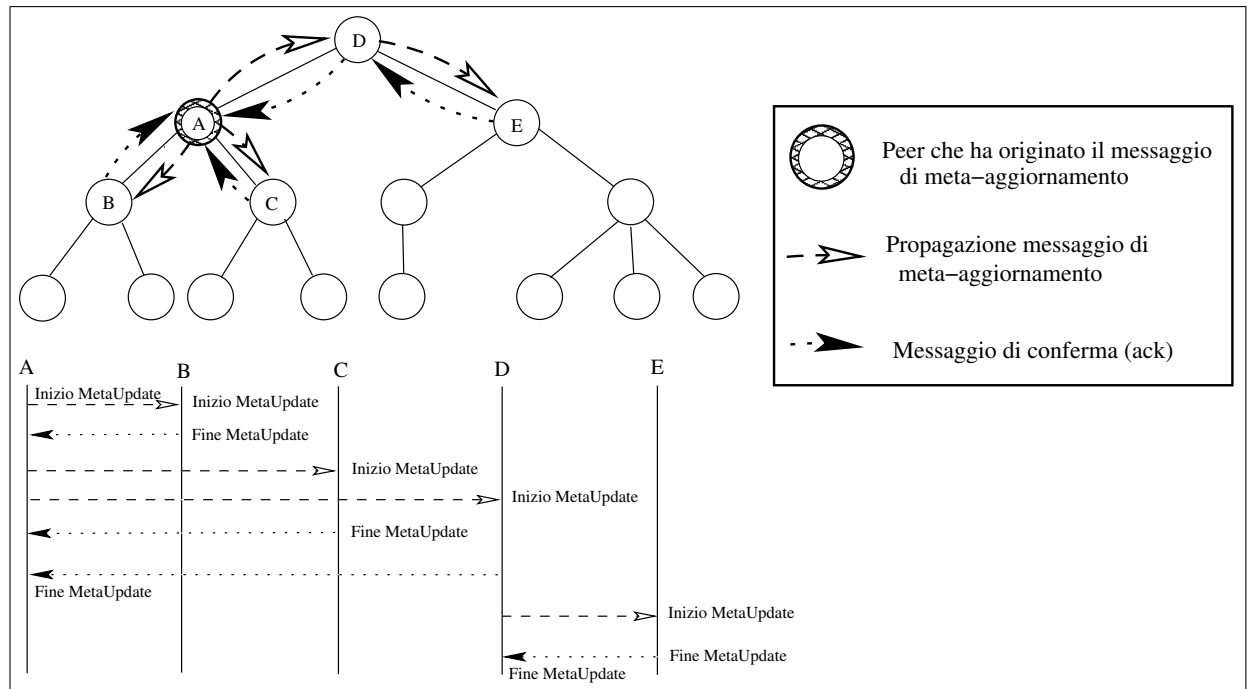


Figura 4.9: Esempio di propagazione di un messaggio di MetaUpdate.

SuperPeer che attraversa: la prima prevede l'esecuzione degli aggiornamenti locali; la seconda fase prevede la spedizione di un messaggio di conferma (il messaggio di "ack"); e la terza fase prevede la spedizione di un messaggio di propagazione ad ogni SuperPeer che appartiene all'insieme dei SuperPeer a cui propagare il messaggio, ed una attesa per il messaggio di conferma da ognuno dei destinatari.

L'insieme dei destinatari di una propagazione di un messaggio può essere un qualunque sottoinsieme dell'insieme formato da tutti i figli del SuperPeer (solo se sono anch'essi SuperPeer) più il nodo padre (se esiste), meno il nodo che ha spedito il messaggio. Un esempio di propagazione dei messaggi è visibile in figura 4.9, dove si può notare come il messaggio non raggiunga mai i Peer alle foglie dell'albero, e i messaggi di conferma tornino al Peer che ha spedito la richiesta di aggiornamento, e non a quella che lo ha originato. Lo pseudocodice che costituisce la MetaUpdate è visibile in figura 4.10

### ModifyChildrenTable

Il protocollo di "ModifyChildrenTable" si occupa di tenere sincronizzata una struttura distribuita tra più nodi: la tabella dei figli di un SuperPeer presente su tutti i cloni di un SuperPeer virtuale.

Il protocollo può essere attivato, o mediante chiamata diretta della funzione, o mediante ricezione di un messaggio di richiesta.

```

MetaUpdate(MetaUpdateMessage[action, responseNode, msgId, params] msg){
  //Il valore di res.localResult dipende dall'azione eseguita
  MetaUpRes[localResult, params, nextNodes] res = executeUpdate(action, params)
  //Viene sempre spedito prima il messaggio di ack, e poi si propaga l'aggiornamento
  sendAnswer(msg.responseNode, ['updateDone', msgId, this.Identity()], localResult)
  ID myMsgId = newMsgID()
  for (n  $\in$  res.nextNodes  $\neq$  sender) do{
    //Per ogni messaggio di propagazione dell'aggiornamento si aspetta un ack
    send(n, ['metaUpdate', 'updateSchema', this.Identity, myMsgId, res.params])
    receiveAnswer(n, ['updateDone'],myMsgId)
    exception :
      fail after timeout and redo;
  }
}

```

Figura 4.10: Pseudocodice della MetaUpdate.

Se un clone vuole effettuare un aggiornamento della propria struttura, allora chiama la funzione di aggiornamento locale, la quale effettua un aggiornamento dei dati locali al clone, e invia un messaggio di aggiornamento a tutti i cloni del proprio SuperPeer virtuale con la primitiva “sendToAll” (che definiremo meglio nella sezione 4.3.5 a pagina 56).

Se un clone di un SuperPeer riceve un messaggio di aggiornamento della propria tabella dei figli effettua un semplice aggiornamento dei dati locali.

## SchemaUpdate

Il protocollo di “SchemaUpdate” serve ad aggiornare i livelli superiori della gerarchia dell'avvenuto cambiamento dello schema locale (o del SuperSchema nel caso di un SuperPeer) ed è eseguito in maniera molto semplice con l'invio di un messaggio di meta-aggiornamento al padre, se presente, in cui si comunica, oltre alla propria identità, il nuovo schema posseduto, e delle informazioni di controllo per verificare l'autenticità dello schema inviato. Il SuperPeer che riceve il messaggio di aggiornamento esegue le modifiche alla tabella dei figli con il protocollo di “ModifyChildrenTable”, e calcola la lista dei destinatari della propagazione del messaggio come una lista costituita al più da un elemento: il padre, se l'aggiornamento ha modificato il proprio SuperSchema, o dalla lista vuota altrimenti. Come previsto dal protocollo viene poi rispedito il messaggio di conferma all'originatore del protocollo di “SchemaUpdate” e viene propagato il messaggio di aggiornamento.

### 4.3.2 Protocolli di innesco

La classe dei protocolli di innesco comprendono dei protocolli che sono eseguiti principalmente durante l'inserimento di un Peer all'interno della rete. I protocolli che appartengono a tale classe sono:

- connessione;
- registrazione;
- deregistrazione;
- cambio padre.

#### Connessione

Il protocollo di connessione è il protocollo che viene eseguito al momento della connessione alla rete da parte di un Peer. Esso serve ad assegnare un padre al Peer che vuole connettersi alla rete. Per poter attivare il protocollo di connessione, un Peer, ha bisogno di conoscere una lista di Peer a cui chiedere di essere ammesso all'interno della rete.

Il protocollo inizia effettuando dei Ping ai Peer nella lista, e non appena trova il primo Peer che risponde al Ping allora gli invia il messaggio di connessione. Il messaggio di connessione serve a chiedere al sistema l'identificatore di un SuperPeer a cui potersi registrare come figlio. Se il Peer che riceve il messaggio di connessione è un SuperPeer, allora gli risponde inviandogli il suo identificatore, altrimenti gli invia quello del padre. Una volta ricevuto il messaggio di risposta alla richiesta di connessione il protocollo prevede di attivare il protocollo di registrazione presso il SuperPeer che gli è stato indicato dal messaggio di risposta.

#### Registrazione

Il protocollo di registrazione è il protocollo che permette ad un peer (o ad un SuperPeer) di registrarsi con un nuovo SuperPeer che gli è stato assegnato come padre. Il protocollo di registrazione è attivato con l'identificatore di un SuperPeer che è stato proposto come padre. Il protocollo di registrazione consiste nell'inviare un messaggio di meta-aggiornamento al SuperPeer passato come parametro in cui si deve specificare, oltre alla propria identità, anche lo schema da pubblicare. Lo schema da pubblicare per un Peer è lo schema della propria base di dati locale, mentre per un SuperPeer lo schema da pubblicare è il SuperSchema, cioè l'unione di tutti gli schemi dei figli del SuperPeer.

Il SuperPeer che riceve il messaggio di meta-aggiornamento controlla se può avere un nuovo figlio, e se lo schema che ha ricevuto è sufficientemente simile agli schemi che pubblica, e in caso contrario richiede agli altri SuperPeer se ce n'è uno disposto ad accogliere il Peer come figlio. A questo punto il Peer designato come padre aggiorna le proprie strutture locali e accetta il Peer come nuovo figlio inviandogli il messaggio di conferma previsto dal protocollo di MetaUpdate. Per aggiornare le strutture dati locali il SuperPeer che riceve la richiesta di registrazione deve eseguire il protocollo di "ModifyChildrenTable" e se questo comporta l'aggiornamento del suo SuperSchema deve eseguire anche il protocollo di "SchemaUpdate".

Il Peer che riceve il messaggi di conferma a quel punto può assegnare il SuperPeer indicato nel suddetto messaggio come suo padre, e comunicare al suo sistema di comunicazione l'avvenuta operazione.

### **Deregistrazione**

Il protocollo di deregistrazione è il protocollo da eseguire per disconnettersi dalla rete in modo corretto. Il protocollo di deregistrazione consiste nell'inviare un messaggio di meta-aggiornamento al padre con la richiesta di uscita dal sistema. Il padre che riceve il messaggio di deregistrazione aggiorna la propria tabella dei figli ed il proprio SuperSchema, e risponde con il messaggio di conferma al figlio. Se il SuperSchema è stato modificato allora, come nel caso della registrazione, il messaggio va propagato verso le gerarchie superiori per comunicare l'aggiornamento di schema. Il figlio che a questo punto riceve il messaggio di conferma può ritenersi disconnesso dal padre.

### **Cambio padre**

Il protocollo di cambio padre è il protocollo che permette ad un Peer di cambiare il padre presso cui è registrato. Il protocollo di cambio padre serve ad effettuare delle ristrutturazioni della rete per ribilanciamenti di carico o per la creazione di gruppi di interesse comune come specificato precedentemente per la descrizione della figura 4.2. Il protocollo di cambio padre consiste semplicemente nell'eseguire una deregistrazione dal padre cui si è attualmente registrati, seguita da una registrazione presso il nuovo padre indicato nel parametro di attivazione del protocollo.

### **4.3.3 Protocolli di evoluzione della rete**

I protocolli di evoluzione della rete sono un insieme di protocolli che servono a tenere bilanciata la rete dei SuperPeer e sono composti da:

- un protocollo di ricerca di padri adottivi per figli che si vogliono eliminare;
- un protocollo per l'avvio della riduzione del numero di figli di un SuperPeer;
- un protocollo per la divisione di un SuperPeer in due SuperPeer fratelli;
- un protocollo per l'abbandono dei figli;
- un protocollo per fondere insieme due fratelli sottocaricati;

Per poter eseguire i protocolli di ristrutturazione della rete è indispensabile la presenza di un monitor, all'interno di ogni singolo nodo (su cui sia attivo un SuperPeer) per controllare il carico di lavoro del SuperPeer stesso. Il monitor è in grado di valutare le condizioni di sovraccarico, o di sottocarico, e di attivare i giusti protocolli per risolvere il problema.

La gestione di un sovraccarico, può essere gestita in due modi: aumentando il numero di SuperPeer del sistema, ed in particolare creando un SuperPeer fratello per dare in adozione i propri figli, o aumentando il numero di cloni che costituiscono il SuperPeer virtuale cui si appartiene.

La gestione di un sottocarico, può essere gestita simmetricamente al caso precedente, in due modi: diminuendo il numero di SuperPeer del sistema, ed in particolare fondendosi con un SuperPeer fratello, sufficientemente scarico, per adottarne i figli, o diminuendo il numero di cloni che costituiscono il SuperPeer virtuale cui si appartiene.

Per tenere l'albero il più possibile bilanciato, nel caso di un ramo particolarmente carico di lavoro, si è cercato di limitare la creazione di fratelli, introducendo un meccanismo alternativo per diminuire il numero di figli: l'adozione. Infatti quando un SuperPeer si accorge di essere troppo carico, e che il modo migliore per ridurre il proprio carico è diminuire il numero di figli, allora, prima di cercare di dividersi, cerca di dare in adozione una parte dei propri figli.

### **Monitor di carico**

Il monitor di carico è un processo che tiene sotto controllo il numero di messaggi che un SuperPeer sta elaborando per controllare che non esca al di fuori di un intervallo di giusto carico.

Se il carico in un dato momento risulta essere troppo alto allora il monitor è in grado di stabilire se è necessario aumentare il grado di clonazione per risolvere il problema, o se è necessario dividere il SuperPeer in due SuperPeer fratelli. Il fattore discriminante per la scelta tra queste due possibilità è dato dalla diversità di

schemi tra i figli del SuperPeer sovraccarico. Se infatti i figli del SuperPeer hanno schemi molto dissimili tra loro è conveniente dividere i figli in due gruppi che abbiano degli schemi più omogenei tra di loro, ripartendo così in modo più equo possibile i messaggi di interrogazione tra i due fratelli. Se gli schemi dei figli invece sono molto simili tra di loro risulta più conveniente invece aumentare il grado di clonazione di un SuperPeer che si troverebbe così ad avere un numero di risorse aumentato e in grado di gestire meglio un numero anche grande di messaggi.

Anche in caso di rilevamento di sottocarico il monitor è in grado di scegliere quale delle due strategie scegliere, se la riduzione del grado di clonazione, o l'accorpamento di due fratelli sottocaricati. La scelta questa volta è influenzata dal numero di cloni presenti, che non può essere troppo basso e dalla presenza di un altro SuperPeer sottocaricato in sua prossimità.

Ogni volta che il monitor rileva una delle 4 condizioni suddette attiva il protocollo corrispondente per cercare di risolvere il problema. I due protocolli che agiscono sul grado di clonazione sono protocolli al livello di sistema di comunicazione, mentre gli altri due sono quello di riduzione del numero di figli e quello di fusione di due SuperPeer.

## **Adozione**

Il protocollo di adozione serve a ridurre il carico di un nodo diminuendo il numero dei suoi figli assegnandoli ad altri. Il protocollo di adozione funziona usando la MetaSearch per ricercare dei SuperPeer disposti ad adottare un sottoinsieme dei figli. Tutti i SuperPeer che ricevono il messaggio di meta-ricerca per l'adozione esaminano l'elenco dei figli dati in adozione e, in base al proprio carico di lavoro, ed alla similitudine degli schemi con i loro, possono scegliere di adottare alcuni dei Peer nella lista. Se scelgono di adottare dei figli, allora inviano un messaggio di risposta al SuperPeer che ha originato la richiesta in cui sono specificati i Peer che intendono adottare, e propagano sulla restante parte della rete i figli rimanenti. Il nodo che riceve i messaggi di risposta esegue, per ogni figlio che viene dato in adozione, il protocollo di cambio padre con il primo Peer che si è offerto di adottarlo.

Come in tutti i casi di MetaSearch la durata dell'operazione di attesa delle risposte è limitata da un timeout che interrompe il protocollo anche se non sono stati raggiunti i risultati sperati.

Il protocollo di adozione termina restituendo la lista dei figli che si volevano dare in adozione, ma che non hanno trovato un padre.

### **Riduzione del numero di figli**

Ogni clone di un SuperPeer che si accorge di essere sovraccarico può decidere di eseguire uno dei protocolli per la riduzione del carico, ma non può avviarlo localmente, in quanto potrebbe esserci un altro clone che contemporaneamente verifica la stessa situazione e avvia lo stesso protocollo, causando così inconsistenze. Allora è stato creato un meccanismo di sincronizzazione a livello di sistema di comunicazione che elegge un clone come responsabile momentaneo dell'esecuzione dei protocolli di evoluzione della rete, di modo che qualunque clone di un SuperPeer decida di effettuare una ristrutturazione della rete invia un messaggio di richiesta allo stesso clone.

Il clone che riceve il messaggio di richiesta di ristrutturazione, dà il via al protocollo di riduzione del numero di figli e per tutta la durata del protocollo accoglie gli altri messaggi analoghi eliminandoli dalla propria coda.

Il protocollo di riduzione del numero di figli divide i figli del SuperPeer in due gruppi cercando di creare i due gruppi con gli schemi più possibile simili tra di loro, poi sceglie il gruppo da tenersi, e cerca di dare in adozione l'altro gruppo con il protocollo di adozione. Se al termine del protocollo sono rimasti ancora troppi figli che non hanno trovato un padre, allora dà il via al protocollo di divisione del SuperPeer.

### **Divisione di un SuperPeer**

Il protocollo di divisione del SuperPeer cerca di trovare dei nodi in cui non è ancora attivo il processo SuperPeer per poterlo attivare con la MetaSearch.

Se il SuperPeer che si vuole dividere in due è anche la radice dell'albero allora bisognerà creare due nuovi SuperPeer, uno che diventerà il nuovo padre, e uno che diventerà il nuovo fratello.

Se la MetaSearch non riesce a creare un numero sufficiente di cloni allora annulla la creazione del/dei SuperPeer e attiva il protocollo per l'abbandono dei figli di troppo.

Se invece le risorse per la divisione sono state trovate allora il protocollo prevede l'esecuzione del protocollo di cambio padre per il nuovo fratello creato, e se c'è stata l'elezione di una nuova radice allora anche il SuperPeer che si sta dividendo si registra al nuovo SuperPeer eletto a radice. Dopo la stabilizzazione del nuovo fratello viene eseguito il protocollo di cambio padre a tutti i figli che si vuole eliminare. Termina così il protocollo di Divisione.

### Abbandono dei figli

Il protocollo di abbandono dei figli prevede l'invio di un messaggio di abbandono a ciascuno dei figli che si vuole abbandonare, e la loro eliminazione dalla tabella dei figli. I figli che ricevono il messaggio allora si considerano abbandonati, e danno il via al protocollo di connessione con la lista di Peer che hanno incontrato recentemente nella loro presenza all'interno della rete, escludendo quello che li ha appena abbandonati.

### Fusione SuperPeer

Il protocollo di fusione dei SuperPeer prevede la ricerca, tramite il protocollo di MetaSearch, del SuperPeer che è più scarico degli altri con il quale fondersi. Allora viene scelto il SuperPeer che deve chiudersi e questo avvia il protocollo di cambio padre a tutti i suoi figli, viene eseguita la deregistrazione dal padre attuale, e vengono disattivati tutti i cloni del SuperPeer che sarà distrutto.

#### 4.3.4 Protocolli di elaborazione delle query

La classe di protocolli di elaborazione delle query sono un insieme di protocolli che servono ad eseguire un qualunque tipo di query in maniera distribuita sulla rete.

L'esecuzione di una query nel sistema *XPeer* è composta da due parti. La prima parte detta compilazione serve a creare un piano di esecuzione della query distribuita sulla rete di modo da poter avere un albero di esecuzione di una query con, per ogni nodo, la sottoquery che può essere eseguita su un determinato Peer, e viene ottenuta chiedendo ai SuperPeer di trovare i Peer che possono partecipare alla query. Dopo aver ottenuto l'albero di esecuzione si può iniziare l'esecuzione vera e propria della query, inviando, quando necessario, richieste di esecuzione remota di una sottoquery direttamente al Peer che è indicato nell'albero di esecuzione (senza passare quindi per nessun SuperPeer). La combinazione di porzioni di query presenti su nodi remoti distinti tra loro vengono eseguite tutte localmente al nodo che ha originato la query, e che alla fine otterrà i risultati.

I protocolli di questa classe comprendono protocolli per:

- servire richieste di esecuzioni di sottoquery;
- chiedere la compilazione di una query sulla rete;
- coordinare l'esecuzione distribuita della query.

### **Esecuzione di sottoquery**

Il protocollo di esecuzione locale delle sottoquery, attivo su tutti i Peer, è la parte di sistema che si occupa di inviare alla base di dati le richieste di esecuzione di query che provengono dalla rete.

Il protocollo è molto semplice e prevede, per ogni messaggio di richiesta di una query, di inviare la richiesta al gestore locale della base di dati compilando ed eseguendo la query localmente, la quale restituirà un risultato che sarà reinviato indietro al mittente della richiesta.

### **Compilazione query**

Il protocollo di compilazione della query è un protocollo distribuito che fa uso della MetaSearch per eseguire le compilazioni distribuite.

Il protocollo prevede semplicemente l'implementazione di un sistema distribuito di ricerca di schemi compatibili con la query usando la MetaSearch come base per la creazione dell'algoritmo distribuito.

Ogni SuperPeer che riceve una richiesta di compilazione, controlla che tra i suoi figli ci siano degli elementi che posseggono uno schema che sia compatibile con la query di cui è richiesta la compilazione. Per ogni figlio che rispetta il vincolo di schema viene controllato se esso è un Peer o un SuperPeer, e se è un Peer allora viene inserito nella risposta locale da inviare indietro al Peer che ha originato la richiesta, se è un SuperPeer invece viene inserito nella lista dei SuperPeer a cui propagare il messaggio di compilazione. Infine viene aggiunto il padre del SuperPeer e viene eliminato il nodo che ha spedito la richiesta e continua il protocollo standard di MetaSearch.

Il nodo che ha originato la richiesta di MetaSearch rimane in attesa di una risposta per un periodo di tempo limitato e quando ritiene di aver avuto un sufficiente numero di risposte costruisce l'albero di esecuzione della query con le risposte ottenute.

### **Esecuzione query**

Il protocollo di esecuzione delle query esegue una visita dell'albero di esecuzione creato con la compilazione. La produzione del risultato della query si ottiene visitando l'albero di esecuzione, come se rappresentasse un'espressione algebrica con dati locali, salvo che quando si incontra un nodo dell'albero che rappresenta un insieme di dati remoti, si effettua una richiesta al Peer remoto, piuttosto che alla base di dati locale.

Durante la visita dell'albero vengono anche eseguite le giunzioni e tutte le altre operazioni presenti nell'albero di esecuzione per produrre alla fine del protocollo un albero XML con il risultato dell'esecuzione distribuita della query.

### 4.3.5 Protocolli di comunicazione

I protocolli di comunicazione sono una classe di protocolli che servono ad implementare le primitive di comunicazione usate dal Peer e dal SuperPeer sovrastante. I protocolli appartenenti a tale classe possono a loro volta essere divisi in due sottoclassi: le primitive di comunicazione e i protocolli di gestione della clonazione.

#### Protocolli di comunicazione

Le primitive di comunicazione realizzano un'astrazione sulla rete di comunicazione usata per gestire le comunicazioni effettive tra i Peer fornendo un semplice meccanismo di spedizione e ricezione di messaggi basato su delle code che hanno associato un nome.

L'interazione del sistema di comunicazione con il Peer e con il SuperPeer per quanto riguarda la gestione dei messaggi è garantita dalla presenza di tre primitive per la spedizione di messaggi, e di altre due primitive per la ricezione:

- **send(dest, queueName, msg, sender)** per spedire il messaggio “msg” di richiesta sulla coda “queueName” di uno dei cloni del Peer “dest” specificando come mittente “sender”;
- **sendAnswer(dest, queueName, msg, sender)** per spedire il messaggio “msg” di risposta ad una richiesta sulla coda “queueName” del clone del Peer che ha spedito il messaggio di richiesta “dest” specificando come mittente “sender”;
- **sendToAll(dest, queueName, msg, sender)** per spedire il messaggio “msg” sulla coda “queueName” di tutti i cloni del Peer “dest” specificando come mittente “sender”;
- **receive(listQueueName, dest)** per mettersi in attesa indefinita sulla ricezione di messaggi che hanno come destinatario designato “dest” che arrivano su una delle code specificate in “listQueueName”;
- **receive(queueName, dest, timeout)** per mettersi in attesa, limitata a “timeout” millisecondi, per la ricezione di un messaggio sulla coda “queueName” che abbia come destinatario designato “dest”.

La distinzione tra “**send**” e “**sendAnswer**” è dovuta alla presenza dei cloni. Infatti la gestione effettiva di ciascun protocollo su di un SuperPeer è tenuta da uno dei cloni del SuperPeer, e così quando si invia un messaggio di richiesta ad un altro SuperPeer si attende la risposta sullo stesso clone che ha effettuato la richiesta. La “**send**” infatti è progettata per spedire la richiesta ad clone scelto a caso nella lista dei cloni corrispondenti all’identificatore del SuperPeer destinatario presente nella mappa tra identificatori di Peer e identificatori di cloni nel sistema di comunicazione. La “**sendAnswer**” invece è stata creata per inviare un messaggio di risposta allo stesso clone del SuperPeer indicato come mittente che ha inviato il messaggio di richiesta.

La presenza della primitiva di “**sendToAll**” è dovuta alla necessità per alcuni protocolli di spedire messaggi uguali a tutti i cloni di un SuperPeer (per sincronizzare strutture dati) i messaggi di risposta che si potranno attendere saranno naturalmente inviati con la “**sendAnswer**” per gli stessi motivi del caso precedente.

### Gestione della clonazione

I protocolli di gestione della clonazione sono un insieme di protocolli che servono principalmente a tenere delle strutture dati distribuite aggiornate, e che quindi vanno eseguiti ogni volta che un clone di un SuperPeer deve notificare il cambiamento di struttura di un dell’entità SuperPeer che rappresenta agli altri cloni, o agli altri SuperPeer che necessitano della notifica.

La struttura dati fondamentale da tenere aggiornata è la mappa di corrispondenza tra SuperPeer virtuali e cloni che lo costituiscono. Non esiste nessun nodo che possiede la mappa completa di tutte le corrispondenze, ma ogni nodo, nel suo sistema di comunicazione possiede solo una informazione limitata alla corrispondenza per il SuperPeer che rappresenta (se il processo SuperPeer è attivo) e per i SuperPeer figli e i SuperPeer padri (anch’essi non sempre presenti). Un esempio di distribuzione delle mappe di corrispondenza è visibile in figura 4.11. Nell’esempio è stato usato per semplicità un unico identificatore per identificare un Peer, un clone di un SuperPeer, e il sistema di comunicazione che sono presenti sullo stesso nodo. Nell’esempio, che utilizza un albero con soli due livelli di SuperPeer, è già ben visibile come in ogni nodo, il sistema di comunicazione, possiede le entrate del padre del Peer, e se presente l’entità SuperPeer anche del SuperPeer e del relativo padre e figli, e questo per ogni nodo causa una replicazione notevole dell’informazione. La composizione (mappatura) del SuperPeer “sp3” è posseduta da ben 9 dei 13 nodi che costituiscono il sistema, nonostante il nodo sia costituito da soli 2 cloni (i cloni “10” e “12”).

Per tenere sincronizzata una struttura dati distribuita e con il grado di repli-

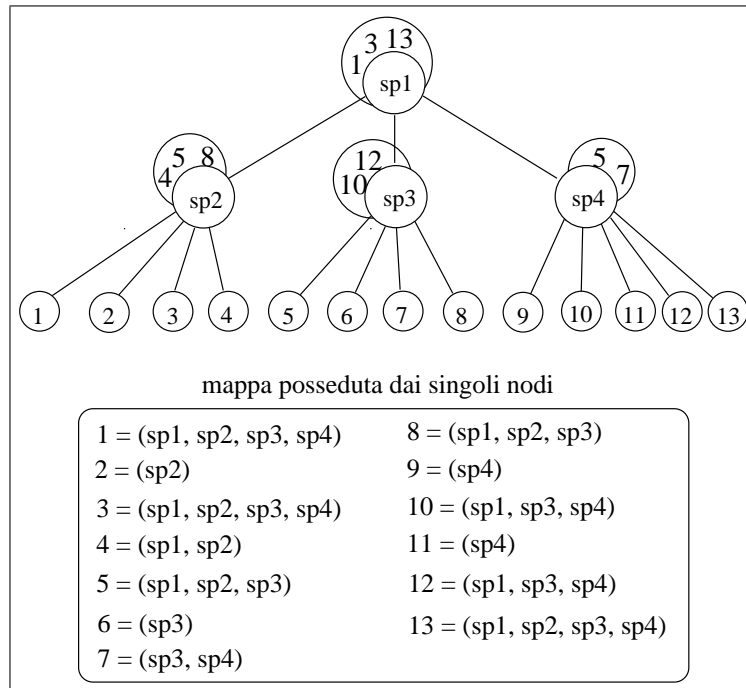


Figura 4.11: Esempio di distribuzione delle mappe.

cazione suddetto occorrono molti protocolli per poter ridurre il numero di messaggi necessari alla sincronizzazione.

I protocolli di gestione della clonazione presenti nel sistema di comunicazione consistono in:

- modifica della tabella di mapping;
- richiesta di aggiornamento a causa della registrazione del nodo ad un nuovo padre;
- richiesta aggiornamento a causa della deregistrazione del nodo dal padre;
- estensione del grado di clonazione del SuperPeer;
- riduzione del grado di clonazione del SuperPeer

### Modifica della tabella di mapping

La modifica della tabella di mapping è una operazione che può essere richiesta localmente da un sistema di comunicazione, o che può essere invocata da remoto da un altro sistema di comunicazione per operazioni di sincronizzazione.

Il protocollo prevede semplicemente di eseguire tre tipi di operazioni su una tabella di corrispondenza tra identificatori di SuperPeer e identificatori di cloni:

l'inserimento di una nuova entrata, la cancellazione di un'entrata, e la modifica dei dati di un'entrata.

Se l'attivazione del protocollo è iniziata localmente allora il protocollo prevede la propagazione della mappatura a tutti i cloni dello stesso SuperPeer; se la richiesta invece proviene da un altro sistema di comunicazione allora le modifiche sono solo locali.

### **Registrazione**

Il protocollo di registrazione al livello del sottosistema di comunicazione serve ad aggiungere una entrata nella mappa di corrispondenza sia sul Peer (o SuperPeer) che si è registrato, sia sul SuperPeer che ha appena ricevuto un nuovo figlio.

Il protocollo quindi prevede l'invio di un messaggio di registrazione al sistema di comunicazione del nuovo padre che aggiungerà localmente un'entrata con il protocollo di modifica della tabella di mapping sul SuperPeer locale, e rispedirà al figlio indietro come risposta la mappa dei suoi cloni per permettergli di effettuare l'operazione analoga sul lato del figlio.

### **Deregistrazione**

Il protocollo di deregistrazione eseguito interamente il locale su entrambi i nodi esegue una semplice rimozione dell'entrata dalla mappa di corrispondenza.

### **Sincronizzazione**

Il protocollo di sincronizzazione serve ad eleggere un clone di un SuperPeer responsabile di operazioni dovranno essere eseguite sul SuperPeer stesso e che quindi richiedono l'elezione di un responsabile. il protocollo di elezione è costituito semplicemente da un funzione che eseguita su un insieme di identificatori restituisce sempre lo stesso identificatore su qualunque nodo sia eseguita.

### **Estensione dei cloni**

Il protocollo di estensione dei cloni è eseguito ricercando con la MetaSearch un nuovo nodo che non abbia ancora attivato il processo di SuperPeer per poterlo attivare. La MetaSearch viene eseguita con la stessa modalità usata per l'elezione di un nuovo SuperPeer. Se viene trovato un nodo disposto a diventare SuperPeer allora gli comunica il suo identificatore di SuperPeer e tutti i dati che condivide con gli altri cloni, ed infine esegue una operazione di aggiornamento della mappa locale con il protocollo di modifica della tabella di mapping, e richiede la stessa operazione al padre e a tutti i figli.

**Riduzione dei cloni**

Il protocollo di riduzione dei cloni serve a ridurre il grado di clonazione di un SuperPeer ed è eseguito scegliendo un clone da disattivare, e dopo avergli mandato il messaggio di disattivazione esegue tutti gli aggiornamenti necessari come nel protocollo di estensione.

# Capitolo 5

## Implementazione di XPeer

Per l'implementazione del sistema abbiamo deciso di adottare come linguaggio di programmazione Java<sup>TM</sup> con le librerie standard disponibili nella versione 1.4.2.

L'implementazione riguarda un sottoinsieme del sistema totale, che, opportunamente integrato con le componenti mancanti, potrebbe arrivare a costituire il sistema complessivo. Il sistema creato comprende un sistema di comunicazione in cui la clonazione non è stata totalmente implementata (ogni SuperPeer è costituito da esattamente un clone), ed in cui non è stata attivata la gestione della compilazione distribuita delle interrogazioni.

Il sistema creato è in grado di creare una struttura gerarchica di Peer con SuperPeer ad albero eseguendo tutti i protocolli di innesco, ed i principali protocolli di evoluzione della rete, oltre che naturalmente tutti i protocolli di sistema.

Il sistema gestisce un log, utilizzando il supporto fornito dal gestore dei log della libreria standard di Java presente nel package “**java.util.logging**” per tenere traccia dei messaggi in transito sulla rete, e l'esecuzione dei metodi chiave del sistema.

Il sistema contiene poi un'interfaccia grafica (**LocalPeerGUI**) che permette di richiamare i comandi principali eseguibili su di un Peer o su di un SuperPeer. Per eseguire dei test, è stata creata un'interfaccia grafica che permette di lanciare comandi su **LocalPeerGUI** remote (**GlobalPeersManagerGUI**). È infine possibile programmare dei test scrivendo dei file di esecuzione che vengono parsati ed eseguiti dai **LocalPeerGUI**.

Il sistema consiste di un insieme di packages che costituiscono le varie componenti di *XPeer*.

I packages principali sono: i packages per la gestione del SGBD locale; quelli per la gestione degli schemi; quelli per la gestione delle comunicazioni; e il package per la gestione dei Peer, dei SuperPeer e delle loro strutture dati. I packages che sono stati realizzati per lo svolgimento della tesi sono quelli relativi alla gestione delle comunicazioni, dei Peer e SuperPeer, e delle loro strutture dati. Gli altri packages

riguardanti la gestione della base di dati e degli schemi sono quelli che costituiscono il sistema Xtasy [Sar03] esteso per la gestione degli schemi della base di dati.

## 5.1 Sistema di comunicazione

Il sistema di comunicazione è costituito da due packages principali, oltre ad altre classi di utilità presenti in un package separato: “**xPeer.acl**” e “**xPeer.messages**”.

Il package “**xPeer.acl**” contiene tutte le classi che implementano un sistema protocollare a stack che costruisce il sistema di comunicazione desiderato secondo le specifiche progettuali, usando come supporto di comunicazione un sistema a socket creato sopra i protocolli TCP/IP.

Il package “**xPeer.messages**” racchiude un sistema di tipi di messaggi, tutti derivati da una classe astratta denominata “**AbstractMessage**” (anch’essa presente nello stesso package) che fornisce lo scheletro generale dei messaggi che saranno scambiati per l’esecuzione dei protocolli.

### 5.1.1 Stack del sistema di comunicazione

Allo scopo di rendere più modulare il sistema di comunicazione, sul supporto da usare per la spedizione dei messaggi è stato creato un sistema a più livelli di astrazione che costituisce uno stack protocollare completo per fornire tutte le funzionalità richieste dal sistema. La strutturazione dei livelli è visibile in figura 5.1.



Figura 5.1: Stack usato per il sistema di comunicazione.

Ad ogni livello sono associate delle funzionalità aggiunte rispetto a quelle fornite dal livello superiore.

### Gestore messaggi

Il gestore dei messaggi fornisce funzionalità di spedizione di messaggi in formato stringa da un socket ad un altro. La ricezione di un messaggio è gestita mediante l'invocazione di una callback che invia a un processo registrato a livello superiore il messaggio arrivato dalla rete.

Ogni volta che il server-socket in ascolto per messaggi in arrivo riceve una richiesta di connessione, lancia un nuovo “**Thread**” che si occupa di leggere il messaggio proveniente dal socket appena creato, e di effettuare tutte le operazioni di notifica alle classi registrate al livello superiore.

Il lancio di un nuovo “**Thread**” per ogni messaggio arrivato è necessario a causa delle possibili lunghe callback al livello superiore che potrebbero tenere occupato per troppo tempo il sistema di comunicazione per un unico messaggio, bloccando l'arrivo di altri.

La struttura del gestore messaggi è visibile in figura 5.2.

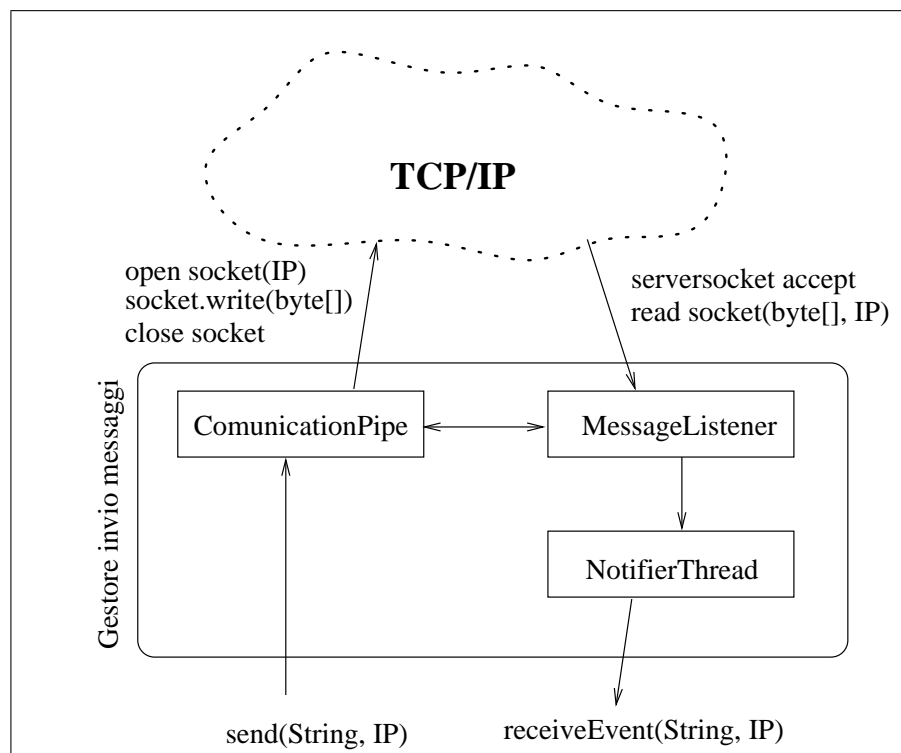


Figura 5.2: Gestore messaggi.

## Gestore code

Il gestore delle code si occupa di realizzare un sistema di trasmissione messaggi in cui è possibile inviare messaggi su una specifica coda presente su un altro nodo. Ogni coda di messaggi su un nodo è identificata da un nome, e può essere attivata ad accettare messaggi, o può essere chiusa, rifiutando così tutti i messaggi che vorrebbero accodarsi in quella particolare coda. La spedizione di un messaggio causa la materializzazione del messaggio sulla coda opportuna del ricevente. Dal lato del ricevente, il messaggio viene memorizzato in strutture apposite, e viene notificato l'arrivo di un messaggio sulla coda in questione ad un insieme di processi che si sono registrati per ricevere la notifica.

Per garantire un corretto funzionamento del sistema in presenza del multi-threading del livello sottostante è stato necessario utilizzare dei lock opportuni per la gestione della materializzazione dei messaggi sulle singole code, in modo da assicurare una corretta esecuzione delle procedure in presenza di arrivo a breve distanza di più messaggi sulla stessa coda. La propagazione delle notifiche ai livelli superiori è invece immune alla collisione delle richieste, trattandosi di una semplice notifica priva di informazioni delicate.

La gestione di ogni coda è gestita secondo la politica FIFO a coda circolare con capacità fissa e con cancellazione dei messaggi più vecchi in caso di riempimento della coda. È possibile sostituire la politica di gestione delle code abbandonando la politica FIFO e adottando un'altra politica, per fare ciò, basta creare una nuova classe che estende l'interfaccia "**QueueManager**" e che utilizza la politica alternativa. Dopo aver creato la nuova classe, sarà sufficiente dire al gestore degli identificatori che utilizzerà il gestore delle code di usare la nuova classe, e il protocollo continuerà a funzionare con la nuova politica.

La ricezione dei messaggi è effettuata secondo un meccanismo di ricezione non bloccante, cioè, se la coda di messaggi richiesta contiene almeno un messaggio, allora esso viene restituito al richiedente, altrimenti viene restituita la stringa nulla.

Per ovviare al problema della possibile attesa attiva per la ricezione di un messaggio, è possibile registrarsi con il gestore delle code per ricevere una notifica ogni volta che arriva un messaggio su di una particolare coda.

Per permettere l'identificazione della coda di destinazione al nodo ricevente, viene spedito insieme al messaggio, come intestazione, il nome della coda verso la quale si vuole destinare il messaggio.

La struttura del gestore code è visibile in figura 5.3.

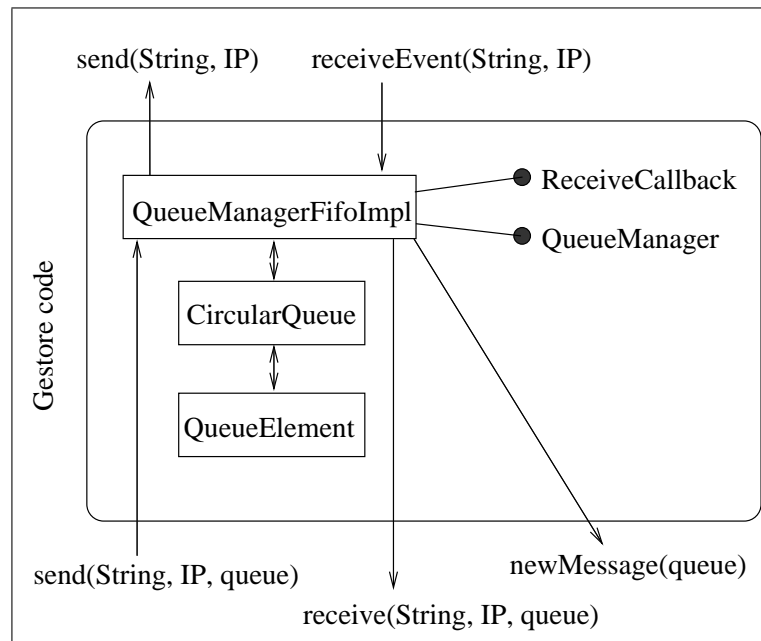


Figura 5.3: Gestore code.

### Gestore degli identificatori

Il gestore degli identificatori fornisce un livello di astrazione maggiore sui riceventi e sui destinatari di un messaggio. Infatti a questo livello si considerano solo identificatori di Peer (o SuperPeer) come possibili mittenti o destinatari di un messaggio.

Quando si spedisce un messaggio dal livello di gestione degli identificatori bisogna quindi inserire come destinatario non più l'indirizzo IP del destinatario e la porta su cui si spedirà il messaggio, ma andrà semplicemente inserito l'identificatore del Peer (o del clone del SuperPeer) che dovrà ricevere il messaggio, sarà il gestore degli identificatori a tradurre tra i due formati. Il destinatario del messaggio, per contro, vedrà il mittente del messaggio come un Peer (o SuperPeer), piuttosto che come un indirizzo IP ed una porta. Tutto il passaggio di queste nuove informazioni avviene inserendo le informazioni supplementari all'interno della stringa che sarà inviata dal livello sottostante.

La spedizione e la ricezione dei messaggi continuano ad essere eseguite con le stesse modalità del livello sottostante, e quindi continua la possibilità di registrazione per le notifiche di arrivo di un messaggio.

La struttura del gestore degli identificatori è visibile in figura 5.4.

### Gestore dei messaggi strutturati

Il livello di gestione dei messaggi strutturati fornisce un livello ulteriore di astrazione sulla forma dei messaggi che si possono spedire. Sopra il livello di gestione

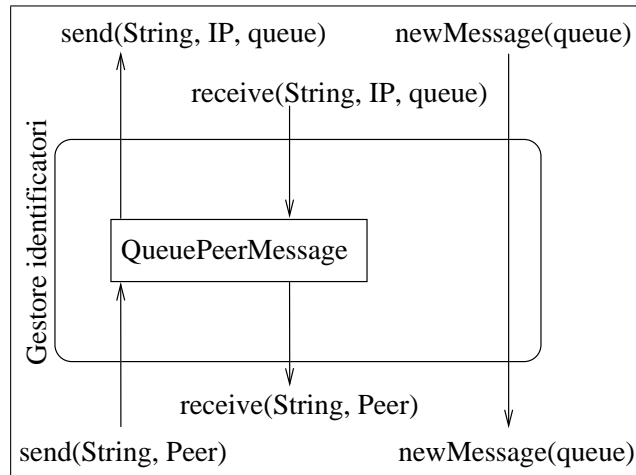


Figura 5.4: Gestore identificatori.

dei messaggi strutturati si vedranno solo transitare messaggi di tipo “**AbstractMessage**”, e quindi oggetti anche complessi, che vengono serializzati/deserializzati in stringhe e spediti con i meccanismi messi a disposizione dai livelli sottostanti.

Il livello utilizza le classi messa a disposizione dal package “**xPeer.messages**” per gestire la serializzazione dei messaggi. Il package “**xPeer.messages**” contiene una gerarchia di classi, tutte derivate dalla classe “**AbstractMessage**”, che costituiscono l’insieme dei messaggi che si possono spedire con il gestore dei messaggi strutturati. La classe “**AbstractMessage**” possiede due campi che devono essere presenti in tutti i messaggi: il possessore del messaggio, ed una stringa contenente i parametri del messaggio, e estende un’interfaccia di serializzazione (vedi sez. 5.1.2 a pag. 69).

La classe astratta “**AbstractMessage**” possiede due metodi statici per eseguire la trasformazione da oggetto e stringa.

Per convertire una qualunque istanza di “**AbstractMessage**” in stringa, si invoca il metodo “**message2String(Obj)**” il quale, invocando il metodo “**serialize**” posseduto dall’oggetto “**Obj**”, lo converte in una stringa, e restituisce la stringa serializzata ottenuta appesa in fondo al nome della classe che è stata serializzata.

Per ricostruire un’istanza di “**AbstractMessage**” a partire da una stringa si invoca invece il metodo “**string2Message(str)**”, il quale, dopo aver letto nella prima parte della stringa “**str**” il nome della classe che è stata serializzata, crea un’istanza della giusta classe, e invoca su di essa il metodo “**deserialize**” con la parte di stringa successiva, riottenendo così la copia della classe spedita.

La struttura del gestore dei messaggi strutturati è visibile in figura 5.5.

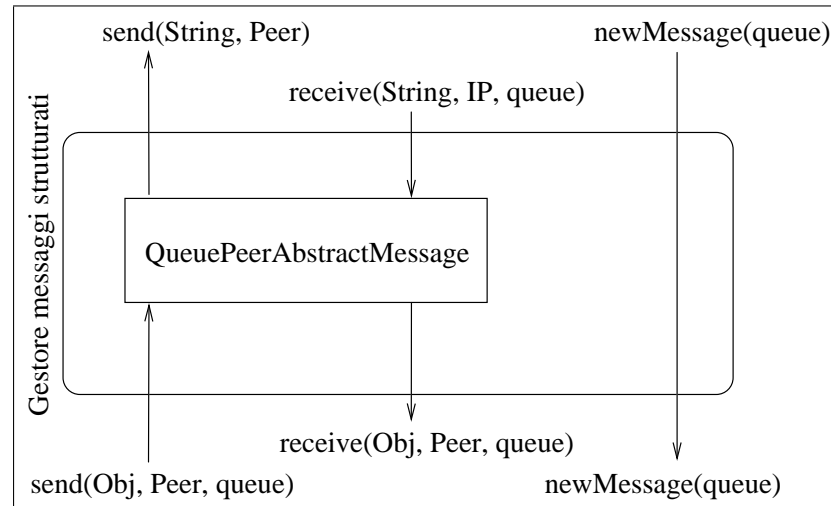


Figura 5.5: Gestore messaggi strutturati.

### Interfaccia del sistema di comunicazione

Il livello di interfaccia del sistema di comunicazione, realizza la struttura necessaria per la gestione della clonazione, e le primitive che sono state richieste dalle specifiche progettuali per la spedizione e la ricezione di messaggi.

La primitiva di “**send**” che spedisce il messaggio ad uno dei cloni del SuperPeer destinatario è stata implementata eseguendo una scelta casuale di un clone all’interno della lista dei cloni costituenti il SuperPeer destinatario (anche se allo stadio attuale di implementazione la lista è lunga esattamente 1).

La primitiva di “**sendAnswer**” che risponde esattamente al clone del SuperPeer che ha spedito un messaggio di richiesta si avvale della presenza dell’identificatore fisico di un clone associato con l’identificatore di SuperPeer virtuale. Ogni volta che viene spedito un messaggio infatti, viene inserito, insieme all’identificatore virtuale, l’identificatore fisico del clone, che è invisibile al di fuori del sistema di comunicazione, e viene utilizzato solo per rispedire indietro risposte con la primitiva “**sendAnswer**”. La struttura di un identificatore di Peer è spiegata più in dettaglio nella sezione 5.1.2 a pagina 71.

La primitiva di “**sendToAll**” che spedisce il messaggio a tutti i cono del SuperPeer indicato come destinatario, è realizzata eseguendo tante “**send**” quanti sono i cloni del SuperPeer destinatario, escludendo se stesso.

Per implementare la “**receive**” bloccante su una o più code con timeout sono state utilizzate due classi. La classe “**CommunicationLayer**” riceve le richieste di “**receive**” su una o più code. Per ogni richiesta di ricezione lancia un’istanza di “**ReceiverThread**”, e rimane in attesa della terminazione del “**Thread**” con una “**join**” con timeout. Il “**ReceiverThread**” lanciato si registra per la ricezione di

messaggi sulla (sulle) code per cui ha ricevuto richiesta, e rimane in attesa dell'arrivo di notifiche dai livelli sottostanti. Se una notifica arriva prima dello scadere del timeout il **“ReceiverThread”** comunica il nome della coda su cui è arrivato il messaggio e il messaggio al **“CommunicationLayer”** e termina. Quando il **“CommunicationLayer”** si accorge della terminazione del **“ReceiverThread”** restituisce i messaggi al processo che ha invocato la receive. Se invece la **“join”** è stata sbloccata da un timeout e non dalla terminazione del **“ReceiverThread”** restituisce all'invocante il valore **“null”**.

L'implementazione delle primitive di comunicazione ha richiesto l'aggiunta di un parametro a tutte le primitive di comunicazione. Siccome il sistema di comunicazione deve gestire la ricezione e la spedizione di messaggi per conto di due entità con diverse identità allora il sistema di comunicazione deve conoscere quale identità sarà utilizzata per ricevere o spedire il messaggio. Il parametro aggiunto è un flag booleano che indica se l'identità da usare per la spedizione/ricezione del messaggio è quella del Peer o del SuperPeer, ed il sistema di comunicazione che conosce entrambe le identità userà quella opportuna per l'operazione che deve eseguire. I messaggi che vengono scambiati direttamente tra i sistemi di comunicazione utilizzano l'identità del Peer come default.

I protocolli di gestione della clonazione che sono stati implementati sono: il protocollo per l'aggiornamento della tabella di corrispondenza tra SuperPeer virtuali e cloni (utilizzato dai due protocolli precedenti), e i protocolli per aggiornarsi a seguito di registrazioni e deregistrazioni dal padre.

La struttura dell'interfaccia del sistema di comunicazione è visibile in figura 5.6 (il ProtocolManager presente in figura gestisce i protocolli al livello di sistema di comunicazione, e sarà spiegato più avanti nella sez. 5.2.1 a pag. 72).

### 5.1.2 Scelte implementative

Per l'implementazione del sistema si è scelto di usare solo tecniche di elaborazione che non vincolassero all'uso di un linguaggio di programmazione particolare (Java in questo caso), ma si è scelto di creare un prototipo del sistema che fosse compatibile anche con prototipi realizzati con altri linguaggi.

In particolare la serializzazione dei messaggi non viene eseguita con i meccanismi di serializzazione standard presenti in Java, ma viene adottato un diverso principio di serializzazione creato appositamente.

Gli identificatori di Peer o SuperPeer virtuale sono costituiti da una coppia: un numero intero a 64 bit per l'identificazione dell'entità virtuale, con associato un identificatore di un Peer reale associato. L'utilizzo del doppio identificatore si è

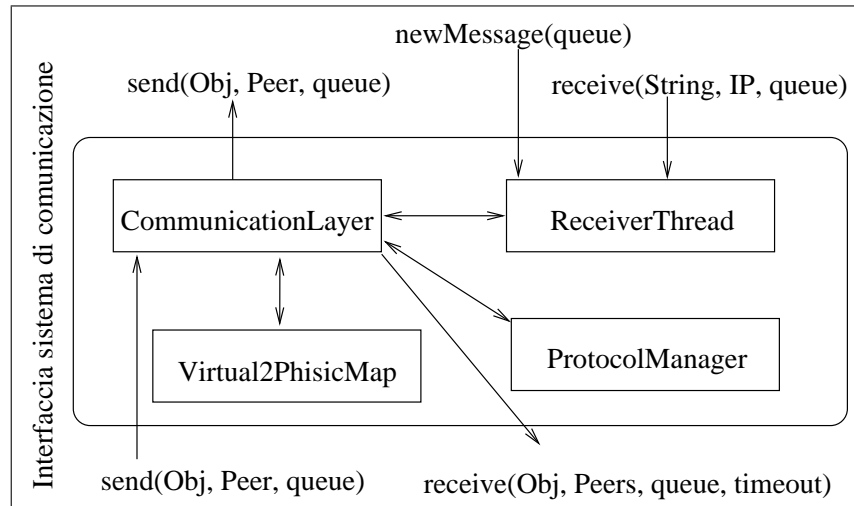


Figura 5.6: Interfaccia sistema di comunicazione.

reso necessario per assicurare un corretto funzionamento del sistema al momento dell'entrata in funzione della clonazione. I messaggi di risposta ad una richiesta che devono essere spediti con la primitiva “**sendAnswer**” devono giungere allo stesso clone che ha inviato la richiesta, per questo insieme all'identificatore del Peer viene fornito l'identificatore del peer fisico che lo rappresenta e che può essere utilizzato, se serve solo a livello del sistema di comunicazione.

### Gestione della serializzazione

La serializzazione è gestita mediante due classi ed un'interfaccia.

L'interfaccia “**xPeer.util.Serializable**” è un'interfaccia che devono implementare tutte le classi che devono essere serializzabili secondo il protocollo previsto. È un'interfaccia che contiene due soli metodi da implementare: “**void serialize(String)**” e “**String deserialize()**”.

La classe “**SerializableObject**”, che implementa l'interfaccia “**xPeer.util.Serializable**”, è una classe di utilità che permette di serializzare anche delle classi che non implementano l'interfaccia precedentemente descritta, ed è stata usata per serializzare valori con tipo: stringa, booleani, intero ed intero lungo (a 64 bit).

La classe “**Serializator**” è un'altra classe di utilità che è in grado di serializzare array o collection di oggetti di tipo “**Serializable**”, e di deserializzare stringhe in array o Liste di oggetti di tipo “**Serializable**”.

La serializzazione di un oggetto viene eseguita inserendo in un array di oggetti “**Serializable**” tutti i campi che si vogliono serializzare. La classe “**Serializator**” si occupa poi di creare la stringa contenente la versione serializzata dell'oggetto,

utilizzando l'array di oggetti appena creati. Un esempio di classe serializzabile è visibile in figura 5.7.

```

class OBJ implements xPeer.util.Serializable{
    int value = 25;
    String name = "Un nome";

    ...
    public String serialize(){
        Serializable[ ] objs = new Serializable[2];
        objs[0] = new SerializableObject(value);
        objs[1] = new SerializableObject(name);
        return Serializator.Serialize(objs);
    }
    public Object deserialize(String str){
        Object[ ] objs = Serializator.deserialize(str);
        this.value = ((SerializableObject) objs[0]).getInt();
        this.name = ((SerializableObject) objs[1]).getString();
        return this;
    }
}

```

Figura 5.7: Esempio di classe serializzabile.

La stringa serializzata è composta da due parti un preambolo ed un corpo vero e proprio. il preambolo è la prima porzione dei stringa racchiusa tra due parentesi quadre, e contiene una lista di campi separati dal carattere “;” una per ogni oggetto nell’array da serializzare. Ogni elemento nella lista è formato da un nome ed un valore separati dal carattere “=”, in cui il nome rappresenta la classe effettiva cui appartiene l’oggetto, e il valore rappresenta la lunghezza della sua versione serializzata contenuta nel corpo. Un esempio è visibile in figura 5.8.

```
[xPeer.util.SerializableObject=2;xPeer.util.SerializableObject=7]25Un nome
```

Figura 5.8: Versione serializzata della classe **OBJ**.

Per poter deserializzare una stringa così composta è stata utilizzata la reflection di Java, che ha reso il codice necessario alla deserializzazione delle stringhe molto breve, ed estendibile.

La forma di serializzazione scelta occupa molto spazio, ma risulta molto robusta, ed assicura una ricostruzione completa dei campi di un messaggio senza introdurre le tute le informazioni sulla classe e sulle sue superclassi, come fatto dalla serializzazione standard di Java.

## Gestione degli identificatori

Gli identificatori di Peer o SuperPeer virtuale sono costituiti da un numero intero a 64 bit per l'identificazione dell'entità virtuale. Per semplificare la gestione delle risposte ai messaggi in presenza di clonazione, all'identificatore logico è sempre associato un identificatore di un Peer reale, che indica uno dei cloni del Peer virtuale.

L'identificatore dell'entità virtuale è un numero a 64 bit costruito in modo da essere univoco all'interno della rete con alta probabilità.

La costruzione dell'identificatore è eseguita assegnando ai 32 bit più significativi l'IPv4 della macchina, ed ai 32 bit meno significativi i 32 bit meno significativi dell'orologio di sistema. Grazie a questa costruzione è altamente improbabile che due Peer diversi possano avere lo stesso identificatore, a meno che non vengano creati nell'arco dello stesso millisecondo (o a esattamente  $2^{32}$  millisecondi di distanza l'uno dall'altro) sulla stessa macchina, o su due macchine diverse, ma che hanno lo stesso indirizzo IP e nello stesso istante.

Il Peer reale è a sua volta un identificatore probabilmente univoco, costituito da diversi campi, assegnati al momento della creazione del Peer. I campi del Peer reale contengono, oltre all'indirizzo e la porta di accesso al Peer, anche un nome (in formato stringa) ed una versione (un intero) che viene modificata ad ogni fallimento del Peer.

Questi 4 campi rendono praticamente univoco anche l'identificatore del Peer reale non essendo possibile aprire per due applicazioni diverse la stessa porta sulla stessa macchina, ed in caso di caduta del Peer una ripartenza con numero di versione differente assicura la risoluzione di problemi di perdita di messaggi causata dal fallimento.

## 5.2 Sistema di gestione dei Peer

Il Sistema di gestione dei Peer è costituito da un unico package in cui sono presenti le principali classi che costituiscono il supporto di gestione dei Peer, escludendo la parte relativa al modulo per la gestione del database locale.

Il package è costituito da due classi principali "**Peer**" e "**SuperPeer**", più altre classi di utilità come la "**ChildrenTable**", per la gestione della tabella dei figli da parte di un SuperPeer, la classe "**SchemaInfo**", per la memorizzazione di informazioni su di uno schema all'interno della rete, e la classe "**InfoContainer**", per tenere memorizzati insieme uno schema con le relative informazioni.

### 5.2.1 Scelte implementative

La strutturazione delle classi prevede una grande similitudine tra le classi Peer e SuperPeer. La similitudine strutturale non rende le due classi totalmente uguali tra loro perché la semantica delle strutture è molto diversa nelle due classi. Basti pensare alla MetaSearch che, pur se presente su entrambe le classi, deve eseguire operazioni molto diverse per il calcolo del risultato locale della computazione. Per queste similitudini strutturali si è scelto di far ereditare la classe SuperPeer dalla classe Peer, sovrascrivendo i metodi comuni, ma con implementazione sostanzialmente diversa.

La classe Peer è una classe che, oltre alle strutture previste dalla progettazione del sistema, possiede un insieme di strutture aggiuntive utili per l'esecuzione del sistema:

- lista dei Peer conosciuti, con cui effettuare delle eventuali riconessioni in caso di abbandono da parte del padre;
- due gestori di protocolli, che sono due thread che hanno lo scopo di rimanere in attesa di messaggi provenienti dalle rete per l'attivazione dei protocolli corrispondenti;
- un puntatore al clone del SuperPeer presente eventualmente sullo stesso nodo;
- un gestore del TimeToLive per gestire il time-to-live dei messaggi di MetaSearch che si propagano nella rete;
- un campo “**willing\_to\_become\_superPeer**” per valutare la possibilità di risposta positiva alla richiesta da parte della rete di diventare SuperPeer.

Un'attenzione particolare è da prestarsi ai gestori dei protocolli.

#### Il gestore dei protocolli

Un gestore dei protocolli è un classe (appartenente al package “**xPeer.util**”) che estende la classe “**Thread**”, e che è fatta per rimanere in attesa di messaggi dalla rete, ed alla loro ricezione, attivare il protocollo corrispondente al messaggio arrivato.

La classe “**ProtocolManager**” è costituita come un “**Thread**” che rimane in attesa della ricezione di messaggi su di una lista di code. Non appena arriva un messaggio su di una delle code gestite, si rimuove il messaggio dalla coda, si controlla che la gestione dei messaggi sulla coda specifica non sia stata disattivata, ed in caso

negativo si controlla in una mappa locale qual è il metodo da invocare alla ricezione di un messaggio sulla coda in questione, allora grazie alla reflection di Java si invoca il metodo presente nella mappa sull'oggetto che possiede tali metodi.

Un “**ProtocolManager**” possiede quindi una tabella di corrispondenza tra nomi di code e metodi da invocare, una lista dei nomi delle code che gestisce in ogni momento (inizialmente coincidente con le chiavi presenti nella tabella di corrispondenza) per permettere la disabilitazione della gestione di alcune code di messaggi, e quindi di alcuni protocolli (e quindi simulare la perdita di messaggi, o la mancanza di un protocollo particolare), un oggetto su cui invocare i metodi presenti nella mappa, ed un nome, utile per identificare i vari “**ProtocolManager**” in esecuzione su di una macchina all'interno del file di log degli eventi, oltre che naturalmente al “**CommunicationLayer**” usato per la ricezione dei messaggi e a un flag per sapere se ricevere i messaggi dal sottosistema di comunicazione come protocollo di Peer o di SuperPeer.

## 5.3 Interfacce grafiche

Le interfacce grafiche comprendono un'interfaccia per gestire un nodo localmente (“**LocalPeerGUI**”), permettendo la creazione di un Peer, l'attivazione del lato SuperPeer, e la connessione/disconnessione dalla rete del Peer; ed un'interfaccia per la gestione complessiva del sistema (“**GlobalPeersManagerGUI**”), che è in grado di inviare comandi ad interfacce locali da remoto, gestendo così un nodo a distanza.

Il package “**xPeer.GUI**” possiede anche altre interfacce grafiche, ma sono tutte interfacce secondarie delle due principali.

### LocalPeerGui

L'interfaccia locale permette di eseguire le operazioni basilari sui Peer, come la creazione, e la chiusura del Peer, l'aggiornamento del campo versione del Peer, la connessione e la disconnessione dalla rete. Tutte le operazioni eseguibili dall'interfaccia hanno accesso dalla finestra principale, o da uno dei menu a tendina presenti nella barra dei menu. Per un esempio vedere figura 5.9

La parte inferiore dell'interfaccia contiene due tab, che permettono di visualizzare rispettivamente l'output del sistema, e permettono di caricare il file di log scritto fino a quel momento sul file.

La redirezione dell'output sulla finestra di output contiene tutti i messaggi che erano destinati allo standard output e tutti i messaggi destinati allo standard error,

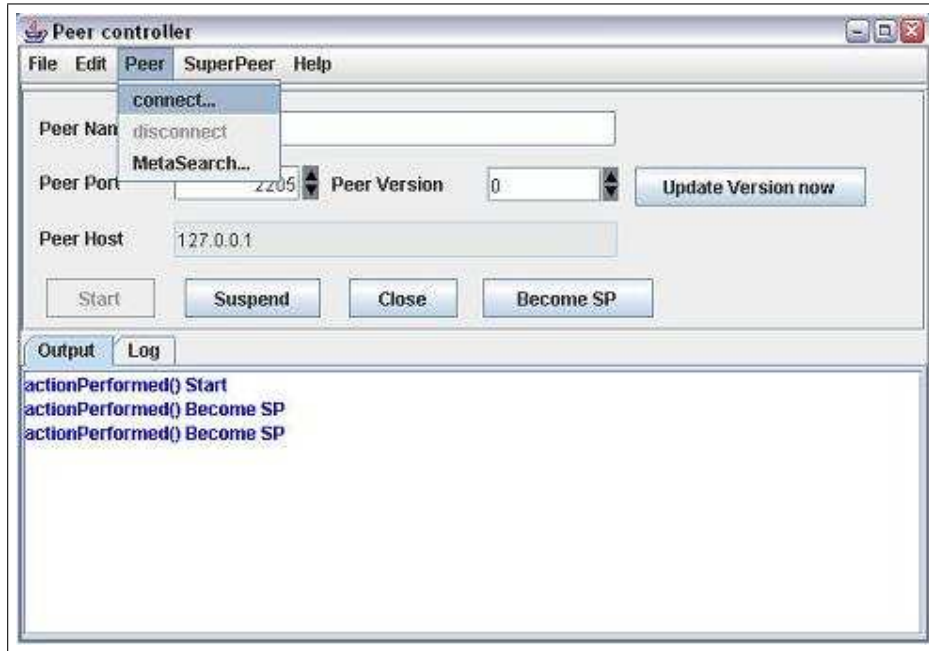


Figura 5.9: Anteprima di LocalPeerGUI

ed è stata eseguita utilizzando la classe “`xPeer.util.OutputStream`” che è in grado di duplicare uno stream inserendo un nuovo output in una “`JTextArea`”.

Per il caricamento del file di log, invece, non si può fare affidamento sul tempo reale di aggiornamento del file, in quanto il file non viene aggiornato immediatamente al verificarsi di un evento, ma vengono effettuati aggiornamenti periodici al file allo scopo di ottimizzare le prestazioni del sistema evitando di rallentare l’esecuzione a causa di un blocco dovuto ad una scrittura su disco. Il file di log, non essendo poi completo, non contiene il tag di chiusura della radice, quindi non potrebbe neanche considerarsi un documento XML valido, e quindi non è parsabile con uno strumento che ne controlla la validità, ma è leggibile solo in formato testuale. Per leggere meglio il log è consigliabile usare uno strumento apposito per leggere documenti XML, strumento che possibilmente supporti una formattazione con XSLT, per consentirne una formattazione adeguata ed una più semplice interpretazione.

## 5.4 Errori nell’implementazione di Java della Sun

Nel corso dell’implementazione del sistema sono stati individuati due banchi nell’implementazione delle API Java da parte della Sun [Sun]. Entrambi i banchi sono stati individuati nel corso dell’utilizzo della libreria standard per la gestione del file di log, ed entrambi sono causati da errori presenti in classi esterne al package “`java.util.logging`” che contiene la libreria stessa.

Il primo bug riguarda il parsing del file di configurazione del gestore del log (la classe “**java.util.LogManager**”). Il parsing del file di configurazione eseguito dalla classe “**java.util.Properties**” con il metodo “**load(InputStream)**” può causare una lettura incorretta dei parametri di configurazione. La classe “**Properties**” esegue il parsing di tutti i file di configurazione di una qualunque classe che lo richieda, e prevede che il file di configurazione sia costituito principalmente da un elenco di righe contenenti ognuna una chiave associata ad un elenco di valori. Per chiave si intende tutta la stringa presente dall’inizio di una riga fino al carattere “=”, esclusi i caratteri considerati di spazio all’estremità della chiave stessa. I valori associati ad una chiave dovrebbero essere tutte le stringhe presenti a destra del carattere “=” separati tra loro dal carattere “,” e dovrebbero essere esclusi anche in questo caso gli spazi presenti alle estremità dei valori, ma in questo caso vengono eliminati solo i caratteri di spazio presenti prima del valore, e vengono lasciati i caratteri spazio dopo il valore. Questo tipo di baco causa un errore nel gestore del log, in quanto il file di configurazione del logging contiene dei parametri di configurazione che come valore posseggono dei nomi di classi, e se un nome di classe non è correttamente letto, si verificano delle eccezioni al “**ClassLoader**” di Java che cerca di caricare le classi configurate, e che per cattura dell’eccezione utilizza le classi di default, piuttosto che quelle presenti nel file di configurazione.

Il primo bug è stato risolto prestando attenzione alla scrittura dei file di configurazione del logger, facendo in modo che non contengano spazi a destra dei valori, e prima del carattere di ritorno a capo.

Il secondo bug riguarda la gestione del file di lock per la scrittura del file di log. Il gestore della scrittura del file di log su disco, infatti, si preoccupa di prendere l’accesso esclusivo in scrittura del file, anche senza bloccare la lettura, utilizzando un file ausiliare di lock su disco. Il file di lock però non viene gestito correttamente dal sistema, in quanto non viene cancellato al momento della chiusura del sistema che fa uso del logging di Java. Il file di lock quindi non viene mai cancellato dal disco fisso. Il bug è stato già riconosciuto dagli sviluppatori della Sun (vedere la gestione dei bug su [Jav]) ed è stato corretto nella versione 1.5 della libreria. Gli sviluppatori di Java della Sun, per risolvere l’inconveniente anche nelle versioni precedenti consigliano di inserire un metodo “**doExit()**” da eseguire alla chiusura del sistema che fa uso del logging che ha come scopo quello di cancellare il file di lock presente sul disco.

Il secondo baco è stato lasciato in sospeso visto che non si presenta durante l’esecuzione del sistema utilizzando la libreria fornita dalla Sun nella versione 1.5, in quanto l’ambiente a tempo di esecuzione nella nuova versione è in grado di eliminare automaticamente il file di lock.



# Capitolo 6

## Conclusioni

Nella tesi sono stati presentati i principali aspetti progettuali e implementativi sul lavoro svolto durante la Tesi.

Il sistema progettato costituisce un sistema completo in grado di effettuare interrogazioni distribuite su larga scala utilizzando XQuery per interrogare dati XML. Il sistema progettato è in grado di creare una struttura di rete che autonomamente si evolve nel tempo per adattarsi alle richieste di distribuzione del carico di lavoro.

Il sistema implementato è in grado di creare le strutture di rete necessarie alla costituzione del sistema complessivo, anche se non è stato realizzato il supporto per la clonazione che rende il sistema resistente ai fallimenti, e non è stata implementata la compilazione distribuita delle interrogazioni.

Il lavoro svolto è consistito nella partecipazione alla progettazione dell'intero sistema di gestione della rete, ed in particolare modo alla sua implementazione in Java<sup>TM</sup>. L'implementazione del sistema non comprende l'intero sistema progettato, in quanto l'intero sistema risulta troppo esteso per essere realizzato nell'arco di un unico lavoro di Tesi, viste soprattutto le continue modifiche architetturali scaturite dallo studio sulle prestazioni, sulla scalabilità e sulla modularità del sistema stesso. Per questo il sistema implementato consiste nella realizzazione dello scheletro completo del sistema in cui sono stati inseriti i moduli per la gestione delle comunicazioni, e la gestione dell'evoluzione della rete, per l'inizializzazione, e i protocolli di sistema, e sono stati lasciati solo abbozzati i moduli per la gestione della clonazione, e per l'esecuzione distribuita delle query.

### Sviluppi futuri

Il sistema può essere esteso con le componenti mancanti aggiungendo le componenti che gestiscono i protocolli per la clonazione nel “**CommunicationLayer**”, ed aggiungendo la gestione di un'azione specifica nell'esecuzione delle operazioni locali: l'operazione di “**queryMatch**” che permette di eseguire la compilazione distribuita

delle interrogazioni, ed aggiungendo infine i tre piccoli protocolli di elaborazione delle query per ottenere il sistema completo.

Il progetto del sistema è in continua evoluzione. Infatti sono state trovate delle soluzioni alternative per la strutturazione della rete che risultano più efficienti e più scalabili di quelle adottate finora. Il sistema XPeer verrà esteso in futuro per adottare una struttura doppia di rete per gestire l'interrogazione dei dati e l'aggiornamento degli schemi posseduti in modo da ottimizzare i tempi di interrogazione e di aggiornamento del sistema e da garantire una separazione tra le due fasi che assicura una maggiore flessibilità per la gestione del sistema.

La nuova architettura prevede una topologia decentralizzata-centralizzata per la gestione delle interrogazioni, in cui ogni Peer è connesso all'unico SuperPeer virtuale che costituisce questo livello di gestione, ed in cui il SuperPeer virtuale è costituito da un insieme di cloni che possiedono dati uguali tra di loro.

La gestione degli aggiornamenti degli schemi (e quindi anche delle registrazioni di nuovi Peer nel sistema) è invece affidata ad una struttura ad albero simile a quella usata nel sistema implementato attualmente che garantisce una maggiore gestione della scalabilità del sistema, permettendo di propagare gli aggiornamenti verso le gerarchie superiori del sistema, e verso la struttura di gestione degli aggiornamenti solo quando il sistema ha un carico di lavoro tale da poter accettare gli aggiornamenti pervenuti.

Per facilitare questi tipi di evoluzioni del sistema, il sistema implementato è stato dotato di una struttura delle classi che permettesse di effettuare un agevole passaggio tra queste due strutture architetture, senza dover sostituire troppe classi, ma solo dividendo la classe **“SuperPeer”** in due: una che si occupa di gestire le interrogazioni, e una che si occupa di gestire gli aggiornamenti; e assegnando alla classe **“Peer”** due padri: uno per la gestione delle interrogazioni e uno per gli aggiornamenti. La parte rimanente del sistema rimarrà pressoché invariata, salvo alcuni ovvi aggiornamenti per distinguere i messaggi diretti verso il padre da parte del Peer che dovranno essere smistati verso il giusto padre.

### **Esperienze acquisite**

La cooperazione con un gruppo di ricerca dotato di una certa esperienza nello studio di nuovi sistemi ha messo in evidenza molti degli aspetti tipici della progettazione cooperativa di un sistema software, e ha messo in luce praticamente il ciclo di vita di un prodotto software, almeno nelle fasi iniziali della sua vita.

Ulteriori esperienze sono state maturate dallo studio di sistemi Peer-to-Peer già esistenti per capirne gli ambiti di utilizzo ed eventualmente sfruttarne le caratter-

istiche interessanti, e dallo studio sull'utilizzo dell'XML e delle tecnologie ad esso collegate, come XQuery, XPath, XSL.

L'implementazione del sistema ha fatto maturare esperienza nell'ambito della programmazione con utilizzo della reflection fornita da Java<sup>TM</sup>, per la semplificazione del codice e la sua semplificazione se accoppiata opportunamente al polimorfismo dei sottotipi costituito dalla gerarchia dei tipi creata grazie all'ereditarietà.



# Bibliografia

- [AQM<sup>+</sup>97] S. Abiteboul, D. Quass, J. McHugh, J. Windom, and J. Wiener. The Lorel query language for semistructured data, aprile 1997. [www-db.stanford.edu/windom/pubs.html](http://www-db.stanford.edu/windom/pubs.html).
- [BCF<sup>+</sup>04] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0:an XML Query Language. Technical report, World Wide Web Consortium, ottobre 2004. Working Draft.
- [Bun97] P. Buneman. Semistructured data. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.*, 1997. Tucson, Arizona: ACM Press.
- [CCR04] M. Castro, M. Costa, and A. Rowstron. Should we build gnutella on a structured overlay? *ACM SIGCOMM Computer Communication Review*, 34(1):131–136, 2004.
- [Cha02] Don Chamberlin. Xquery: An XML query language. *IBM System Journal*, 41(4):597–615, 2002.
- [Cit] Motore di ricerca per pubblicazioni. [www.citeseer.com](http://www.citeseer.com).
- [Cla99] I. Clarke. A distributed decentralized information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.
- [Cor] [www.html.it/xml/guida/index.html](http://www.html.it/xml/guida/index.html). Corso per imparare le caratteristiche dell'XML.
- [CRF00] D. Chamberlin, J. Robie, and D. Florescu. [www.almaden.ibm.com/cs/people/chamberlin/quilt.html](http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html), dicembre 2000.
- [DFF<sup>+</sup>] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. [www.research.att.com/mff/files/final.html](http://www.research.att.com/mff/files/final.html).

- [FLM99] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational plans for data itegration. In *16<sup>a</sup> conferenza nazionale sull'Intelligenza Artificiale*, pages 67–73, 1999. Orlando, Florida.
- [Fre03] Freenet's next generation routing protocol, luglio 2003. [freenetproject.org/index.php?page=ngrouting](http://freenetproject.org/index.php?page=ngrouting).
- [GiF02] The giFT project, 2002. [gift.sourceforge.net](http://gift.sourceforge.net).
- [Jav] Sito ufficiale di java<sup>TM</sup>. [java.sun.com](http://java.sun.com).
- [Kan01] G. Kan. *Gnutella Peer-to-Peer: Harnessing the power of disruptive technologies*. O'Reilly Press USA, 2001.
- [Kaz] [www.kazaa.com](http://www.kazaa.com). Sito di KaZaa Media Desktop.
- [Lim] The Gnutella protocol specification v0.4. [www.clip2.com](http://www.clip2.com).
- [Min01] N. Minar. Distributed systems topologies: Part 1. O'Reilly Network, dicembre 2001. [www.openp2p.com/pub/a/p2p/2001/12/14/topologies\\_one.html](http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html).
- [Min02] N. Minar. Distributed systems topologies: Part 2. O'Reilly Network, gennaio 2002. [www.openp2p.com/pub/a/p2p/2002/01/08/p2p\\_topologies\\_pt2.html](http://www.openp2p.com/pub/a/p2p/2002/01/08/p2p_topologies_pt2.html).
- [Mon04] Giacomina Monreale. Completamento della realizzazione di un DBMS XML nativo. Master's thesis, Dipartimento di Informatica Università di Pisa, dicembre 2004.
- [Nap00] How Napster worked, 2000. [www.howstuffworks.com/napster1.htm](http://www.howstuffworks.com/napster1.htm).
- [OQL96] *The Object Database Standard: ODMG-93*. Morgan Kaufman Publisher, 1.2 edition, 1996.
- [RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale Peer-to-Peer system. Technical report, IFIP/ACM Middleware, 2001.
- [RFH<sup>+</sup>01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. Technical report, ACM SIGCOMM, 2001.
- [RLS] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). [www.w3.org/TandS/QL/QL98/pp/xql.html](http://www.w3.org/TandS/QL/QL98/pp/xql.html).

- [RM04] John Risson and Tim Moors. Survey of research towards robust Peer-to-Peer networks: Search methods. Technical report, University of New South Wales, settembre 2004.
- [Sar03] Carlo Sartiani. *Efficient Management of Semistructured XML Data*. PhD thesis, Dipartimento di Informatica Università di Pisa, 2003.
- [Sch] Motore di ricerca per pubblicazioni. [scholar.google.com](http://scholar.google.com).
- [Sha02] A. Sharma. The FastTrack network. PC Quest Magazine, India, settembre 2002. [www.pcquest.com/content/p2p/102091205.asp](http://www.pcquest.com/content/p2p/102091205.asp).
- [Shi01] C. Shirky. *Listening to Napster, Peer-to-Peer: Harnessing the power of disruptive technologies*. O'Reilly Press USA, 2001.
- [SQL99] Information technology-database language sql. Technical report, International Organization for Standard (ISO), 1999.
- [Sun] Sito ufficiale di sun microsystems. [sun.com](http://sun.com).
- [Tat04] Igor Tatarinov. Semantic data sharing with a peer data management system. Master's thesis, University of Washington, 2004.
- [W3C] [www.w3.org](http://www.w3.org). Sito ufficiale del World Wide Web Consortium (W3C).
- [XML04] Specifica dell'xml 1.0, febbraio 2004. [www.w3.org/TR/2000/REC-xml-20040204](http://www.w3.org/TR/2000/REC-xml-20040204).
- [XPa99] Specifica di XPath 1.0, aprile 1999. [www.w3.org/TR/xpath](http://www.w3.org/TR/xpath).
- [XPa03] Specifica di XPath 2.0, maggio 2003. [www.w3.org/TR/xpath20](http://www.w3.org/TR/xpath20).
- [XQu04] Specifica di XQuery, febbraio 2004. [www.w3.org/TR/xquery](http://www.w3.org/TR/xquery).
- [Zer] Raccolta di programmi di filesharing. [www.zeropaid.com](http://www.zeropaid.com).



# Elenco delle figure

2.1	Esempio di prologo minimo XML. . . . .	8
2.2	Esempio di prologo con DTD XML. . . . .	8
2.3	Esempio di annidamento errato. . . . .	9
2.4	Esempio di elementi con attributi e testo. . . . .	10
2.5	Esempio di albero XML. . . . .	11
2.6	Esempio di documento XML con DTD interno al file. . . . .	12
2.7	Esempio di DTD. . . . .	13
2.8	Esempio di semplice espressione <b>FLWR</b> con soli let e return. . . . .	20
2.9	Esempio di semplice espressione <b>FLWR</b> con soli for e return. . . . .	20
2.10	Esempio di semplice espressione <b>FLWR</b> con tutte le clausole. . . . .	21
2.11	Esempio di un costruttore per un elemento “PIANTA”. . . . .	21
2.12	Esempio di un costruttore parametrico. . . . .	22
2.13	Esempio di un costruttore parametrico con sottoalberi. . . . .	22
2.14	Esempio di un “costruttore di elementi calcolati”. . . . .	22
3.1	Esempio di topologia centralizzata. . . . .	27
3.2	Esempio di topologia gerarchica. . . . .	28
3.3	Esempio di topologia ad anello. . . . .	28
3.4	Esempio di topologia decentralizzata. . . . .	29
3.5	Esempio di topologia composta decentralizzata-centralizzata. . . . .	30
3.6	Esempio di topologia composta anello-centralizzata. . . . .	30
3.7	Esempio di VDST in P-Grid. . . . .	33
4.1	Topologia del sistema <i>XPeer</i> . . . . .	37
4.2	Esempio di raggruppamento. . . . .	38
4.3	Compilazione di una query. . . . .	39
4.4	Esecuzione di una query. . . . .	39
4.5	Topologia dei cloni di un SuperPeer. . . . .	41

---

4.6	Architettura di un nodo . . . . .	42
4.7	Esempio di propagazione di un messaggio di MetaSearch. . . . .	46
4.8	Pseudocodice della MetaSearch. . . . .	46
4.9	Esempio di propagazione di un messaggio di MetaUpdate. . . . .	47
4.10	Pseudocodice della MetaUpdate. . . . .	48
4.11	Esempio di distribuzione delle mappe. . . . .	58
5.1	Stack usato per il sistema di comunicazione. . . . .	62
5.2	Gestore messaggi. . . . .	63
5.3	Gestore code. . . . .	65
5.4	Gestore identificatori. . . . .	66
5.5	Gestore messaggi strutturati. . . . .	67
5.6	Interfaccia sistema di comunicazione. . . . .	69
5.7	Esempio di classe serializzabile. . . . .	70
5.8	Versione serializzata della classe <b>OBJ</b> . . . . .	70
5.9	Anteprima di LocalPeerGUI . . . . .	74