UNIVERSITÀ DEGLI STUDI DI TRENTO
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Tesi

Il Federation System
Service Brokering su base peer-to-peer

The Federation System
Service Brokering the peer-to-peer way

Relatore
Prof. Marco Aiello

Laureando
Michele Mancioppi

Correlatore
Prof. Mike P. Papazoglou

Anno accademico 2005/2006

Alle tutte le persone
che mi sono state vicine
nel mio periglioso percorso
attraverso la giungla universitaria

Contents

Introduzione

I Web Service sono sistemi di middleware pensati per supportare interazioni tra computer su una rete, astraendo dai dettagli delle piattaforme utilizzate, quali sistemi operativi e linguaggi di programmazione. I Web Service fanno uso di linguaggi aperti, basati su XML, e di protocolli di trasporto universalmente disponibili, per scambiare dati con i rispettivi client. I Web Service e i loro client possono essere realizzati con molteplici linguaggi di programmazione, e possono essere eseguiti su massima parte delle piattaforme esistenti. La necessitá di descrivere i protocolli di comunicazione utilizzati in maniera strutturata, é soddisfatta attraverso l'utilizzo del linguaggio per la descrizione di servizi chiamato *Web Service Description Language* (WSDL) [16]. Un documenti WSDL descrive un Web Service in termini delle sue caratteristiche tecniche, quali le operazioni disponibili, raggruppate in *interfaces* (in precedenti versioni dello standard erano note come *portTypes*), dichiarazione di tipi complessi, URL ai quali il servizio é disponibile, e cosí via. Un documento WSDL che descrive un servizio, nella gran parte dei casi viene generato automaticamente a partire dal codice sorgente del servizio stesso con l'utilizzo di strumenti che ne esaminano l'interfaccia di programmazione.

I documenti WSDL non riportano alcuna informazione concernente la semantica delle operazioni ivi descritte. Questa mancanza di semantica viene spesso colmata con informazioni aggiuntive che descrivano il servizio e chi lo offre in termini di catalogazioni industriali (tassonomie), contatti, o simili. Dato che le interfacce dei Web Service sono correlate soltanto con le loro caratteristiche tecniche, fornitori che offrono servizi affini, molto facilmente sviluppano interfacce diverse. Se un consumatore vuole utilizzare servizi affini offerti da fornitori differenti, deve provvedere a sviluppare ed adoperare molteplici client per Web Service. E anche quando i client per Web Service sono stati sviluppati, devono essere integrati all'interno delle applicazioni dei consumatori. Il compito di integrare client di Web Service all'interno delle applicazioni degli usufruitori dei servizi é completamente a carico di questi ultimi. Per ridurre i costi che derivano da questi processi, autori scientifici e vendor hanno proposto soluzioni basate su due principi differenti: integrazione dinamica o statica di Web Service. L'approccio statico riguarda lo sviluppo e l'integrazione del client per Web Service all'interno del codice dell'applicazione di chi deve usufruire del servizio. Il codice sorgente del client puó essere generato automaticamente a partire dalla descrizione dell'interfaccia del Web Service, per esempio in WSDL, tramite l'utilizzo di strumenti come *Java2WSDL*, che fá

parte del progetto *Apache Axis*[1]. L'approccio dinamico, di cui un esempio é il *Web Services Invocation Framework* (WSIF)[2], rende disponibile un ambiente di esecuzione nel quale l'interazione tra client e server sia realizzata sulla base di una descrizione dell'interfaccia del servizio che é gestita a tempo di esecuzione. Comunque, entrambi questi approcci per l'integrazione di Web Service si fondano sulla possibilitá di reperire le interfacce dei servizi con i quali si desidera interagire. Questo é reso possibile dalla disponibilitá di repository pubblici, chiamati Service Broker, attraverso quali i fornitori di servizi rendono disponibili le interfacce dei loro servizi, congiuntamente ad informazioni addizionali, quali, ad esempio, una descrizione in linguaggio naturale o in parte formale di come il servizio elabora i dati di cui dispone.

I Service Broker attualmente disponibili, tra i quali spiccano le realizzazioni dello standard *Universal Description, Discovery and Integration* (UDDI) ed *Electronic Business using eXtensible Markup Language* (ebXML), si basano di norma su un approccio client/server. Per quanto un'architettura centralizzata goda di benefici, quali una minima ridondanza dei dati e una maggior consistenza degli stessi, ha anche degli svantaggi, come la mancanza di scalabilitá e l'impossibilitá da parte degli utenti di cercare Web Service quando il server ha malfunzionamenti o non é disponibile. Inoltre, i Service Broker centralizzati non forniscono supporto per interazioni dirette anteriori all'invocazione del servizio tra chi lo offre un servizio e chi lo usa. Riteniamo che la possibilitá di cercare e pubblicare Web Service con un meccanismo affine alla condivisione di file su una rete peer-to-peer, in scenari nei quali in gran numero di fornitori e consumatori di servizi sono interessati a servizi che offrano funzionalitá affini o correlate si adatterebbe meglio di un'architettura centralizzata.

Noi abbiamo realizzato un Service Broker distribuito, chiamato *Federation System* (FS), che offre funzionalitá di pubblicazione e ricerca di servizi, facendo uso della gestione decentralizzata dei dati che é propria delle reti peer-to-peer. Questo lavoro é basato su una nostra precedente proposta presentata in [68]. Il FS rende possibile a fornitori e consumatori di servizi simili, o correlati come business, di formare un proprio e-marketplace verticale, chiamato *federazione*, o entrare a far parte di uno giá esistente. I partecipanti alla federazione, chiamati *Peer*, possono essere promotori e consumatori di servizi allo stesso tempo.

Il Federation System offre ai Peer tre differenti meccanismi di ricerca di Web Services:

1. *ricerca di servizi centralizzata*: i Peer interrogano una base di dati centralizzata che contiene informazioni riguardo ogni pubblicazione presente all'intero della federazione. Le richieste sono espresse in termini di descrizioni tecniche o caratteristiche di servizio che i risultanti Web Service devono soddisfare;

2. *ricerca di servizi distribuita*: i Peer interrogano i loro paritetici equivalenti richiedendo i servizi che soddisfano un insieme di criteri. Similmente alla ricerca di servizi centralizzata, le richieste possono contenere requisiti

[1] Apache Axis Homepage: `http://ws.apache.org/axis/`
[2] Apache WSIF Homepage: `http://ws.apache.org/wsif/`

espressi come descrizioni tecniche o caratteristiche di servizio.

3. *sottoscrizioni*: i Peer specificano un inseme di criteri sotto forma di descrizioni tecniche o caratteristiche di servizio, che fanno a formare una sottoscrizione. Dopo che la sottoscrizione e' sottomessa alla federazione, il Peer a cui appartiene viene informato attraverso un meccanismo di notifca di tutte le publicazioni, rimozioni e modifiche che coinvolgono i servizi disponibili nella federazione che soddisfano i criteri espressi nella sottoscrizione.

La qui presente Tesi Specialistica é organizzata come segue. Il capitolo 2 riassume lo stato dell'arte nel campo del Web Services Brokering. Il Capitolo 3 presenta il Federation System. Il Capitolo 4 viene descritta l'implementazione del Federation System che é stata realizzata, mentre il capitolo 5 riporta le considerazioni conclusive e possibili future direzioni riguardanti lo sviluppo del sistema.

Chapter 1

Introduction

Web Services are software systems designed to support interoperable machine-to-machine interaction over a network, which use open, XML-based standards and ubiquitous transport protocols to exchange data with calling clients. The Web Services and their corresponding clients can be implemented in a number of different programming languages and run on different platforms. The need for describing the communications in some structured way, abstracting from details involved in the communications between the applications and implementation programming language's peculiarities, is usually addressed by describing the *interfaces* in a standard service-describing language called *Web Service Description Language* (WSDL) [16]. WSDL documents describe Web Services in terms of technical characteristics such as lists of invokable operations grouped in interfaces (formerly called *portTypes*), declared complex types, endpoint addresses (i.e., the URL the Web Services addressed) and so on. A WSDL document describing a service is usually produced automatically by tools which examine the programming interface of the service itself.

WSDL documents neither contain information about the semantics of the operations exposed by the described Web Services, nor they express possible Web Services' usages. In the case of Web Services used in e-business and enterprise oriented environment, this lack of semantics is often filled with additional service-descriptive and business-descriptive data. Typical service-descriptive data are the service's name, a reference to its publisher and one or more taxonomies' entries identifying the kind of commercial services provided by the Web Services (i.e. car rental, hotel booking, and so on) as well as product or geographic codes. Business-descriptive data usually provide information about the services' providers, such as contacts or industry codes.

Since Web Services' interfaces are related only with the implementations' technical characteristics, similar services provided by different providers are likely to have different interfaces. If a consumer wants to interact with equivalent Web Services exposing different interfaces, he is in charge of developing and operating two different Web Service clients. Once the Web Services' clients have been deployed and tested, they still have to be integrated into the service consumers' business applications. The tasks of integrating Web Services' clients into business applications are entirely in charge of the service requesters.

In order to reduce the costs in terms of application development complexity and time, authors and vendors have being proposing solutions based on two different approaches: static and dynamic Web Services integration. The static approach is the developing and the integration of dedicated Web Services' clients in the code of the consumers' application; the clients' sources can be mechanically generated starting from the target Web Service's WSDL description by tools, such as the *Java2WSDL* comprised in the *Apache Axis*[1] project. On the other hand, the dynamic approach, examples of which is the *Web Services Invocation Framework* (WSIF)[2], aims at providing runtime interactions with abstract representations of Web Services through the provided WSDL descriptions. Anyway, both the Web Services integration approaches rely on retrieving Web Services' interfaces descriptions. The task is accomplished by querying publicly available repositories, named Service Brokers, in which the service providers submit the public interfaces, as well as the executional behaviors, of their services.

Current Service Brokers, most notably the Universal Description, Discovery and Integration (UDDI) and *Electronic Business using eXtensible Markup Language* (ebXML) implementations, are typically designed according to a Client / Server architecture. Although a centralized approach has many benefits, such as minimized information redundancy and aids in keeping data consistent, it also has drawbacks, such as lack of scalability and uncapability of discovering Web Services upon centralized Service Broker failure. Moreover, current Service Brokers do not support direct interaction among service requesters and service providers prior to the service invocation phase. In particular, the possibility of discovering and publishing Web Services in a manner similar to filesharing with Peer-to-Peer systems will fit better than the centralized approach in scenarios in which a very large number of service providers and requesters are involved into services that provide similar functionalities.

We propose a peer-to-peer distributed system named *Federation System* (FS), which offers service publication and service discovering capabilities, involving decentralized data management which is distinctive of peer-to-peer networks. This work is based on our initial proposal presented in [68]. The FS allows providers and consumers of similar and business-related Web Services to form their own vertical e-marketplaces, named *Federations*, or join already-existent ones. The Federation's participants, named *Peers*, can be both services' publisher and consumer at the same time.

The Federation System provides Peers with three different types of Web Services discovering facilities:

1. *centralized service discovering*: Peers inquires for published services a centralized base of data containing the whole Federation's knowledge. The queries are expressed in terms of technical and service-descriptive data that the resulting published services have to match;

2. *distributed service discovering*: Peers inquire in a distributed way their equivalents for the services matching a set of criteria they provide. Like-

[1] Apache Axis Homepage: `http://ws.apache.org/axis/`
[2] Apache WSIF Homepage: `http://ws.apache.org/wsif/`

wise the centralized service discovering facility, the queries may contain technical and service-descriptive requirements;

3. *subscriptions*: Peers specify a set of technical and service-descriptive requirements, thus composing a *subscription*. Afterwhat the subscription is signed on the Federation, the Peer owning the latter is promptly informed through a *notification* of all the publications, removal and updating of all the services matching the defined criteria occurring within the Federation.

The present Master Thesis is organized as follows. Chapter 2 is devoted to the presentation of the state of the art in Web Service brokering. In Chapter 3, the Federation System is presented. In Chapter 4, we describe the related implementation, while Chapter 5 presents final remarks and open issues.

Chapter 2

State of the Art in Web Services Brokering

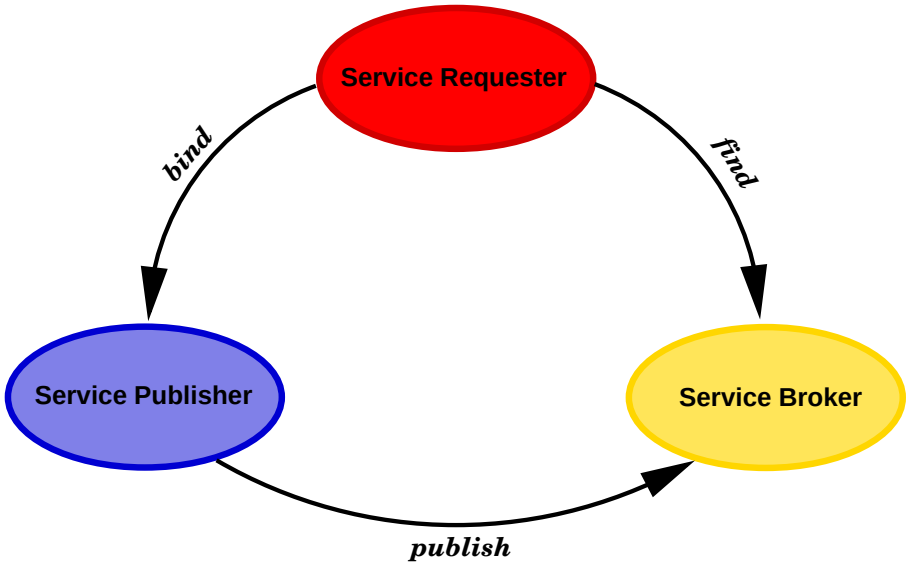
During the past few years, many consortia and vendors, such as the UDDI and the ebXML ones, have offered their own vision of scalable, extensible architectures, focused on providing high levels of interoperability across different platforms, programming languages and applications, thus enabling their customers to solve integration problems among different IT infrastructures and increasingly supporting sophisticated business processes.

The results of the efforts made by consortia and vendors are exemplified by the *Service Oriented Architectures* (SOAs) [67, 66]. A SOA is a distributed networked system made of interacting and autonomous software applications which offer and request functionalities in a loosely couple manner. The SOA design is structured around the *Service Oriented Computing* (SOC) paradigm, the latter applying the *REpresentational State Transfer* (REST) architectural design style. A REST application's state and functionality is divided into resources, which are uniquely addressable using a universal syntax, i.e., Unique Resource Identifiers (URIs) for HTTP resources. The resources are transfered between client and resource through a uniform interface, which is based on a well-defined operations' set, such as the HTTP's methods, and a stateless client/server protocol.

A SOC environment comprises connected applications, named *Services*, which communicate with each other by passing data or coordinating activities. As shown in Figure 2.1, SOC describes the three roles played by Services: *Service Providers*, *Service Requesters* and *Service Brokers*. Every Service can play one or more roles. *Services* perform three basic operations: *publish*, *find* and *bind*. Service Providers publish information on repositories, named Service Brokers, by submitting their public interfaces as well as their executional behaviors; the latter are typically described as work-flow processes. Service Requesters find Service Providers by querying Service Brokers, thus providing matching-criteria such as provided functionalities, performances, costs, availability, security and scalability. Service Requesters bind the located Service Providers to access the desired functionalities.

The SOC paradigm states the *composability* of Services: two or more Ser-

Figure 2.1: The roles and the action played within a SOA.



vices may be combined, even recursive, to obtain a new Service. Service compositional features should include:

1. *Service Coordination*: provides coordination mechanisms which may be based upon specific coordination protocols. The latter should be aimed at creating contexts needed to propagate activities to other services and to provide service-registering facilities.

2. *Service Transaction*: describes coordination types which are intended to be used with the Service Coordination framework. Examples of coordination types could be *durable two-phase commit*, *volatile two-phase commit* and *completion*.

Global availability, reliability and security characterize SOAs: the applications to be deployed according to the new architectures are usually planned to be business end-to-end solutions, which are required to be scalable, reliable and fault-tolerant. Flexibility and modularity are very important: defining large, monolithic specifications that offer end-to-end functionality is not effective. On the other hand, committees often require agile architectures made by modular components which can be composed into more complex solutions: this pattern of application design is sometimes called *building-block approach*. The latter may be deployed in different scenarios such as *Business-to-Business* (B2B) solutions, *Business-to-Consumer* (B2C) services, P2P applications, thus avoiding expensive and time-consuming re-engineering processes.

2.1 Current Service Brokering Architectures

Service discovery plays a prominent role in the SOA design, it is the basis for realizing automatic services' composition. Nowadays, the most widely adopted

service brokering architectures are the UDDI and ebXML ones, that are hereby briefly introduced.

2.1.1 UDDI

When, at the beginning of the year 2000, the *Universal Description Discovery & Integration* (UDDI) Consortium was launched, it was aimed at developing an architecture to ensure automatic discovering and locating of trading interlocutors, allowing businesses to describe the type of services they offer and executions of e-commerce transactions by integrating business services using the Internet. According to this requirement, the first set of UDDI specification described an architecture which was meant to serve as a sort of directory of publicly available e-commerce services.

Through time, the technological requirements for Web Service integration-oriented platforms had a shift to establishment and maintenance of vertical e-marketplaces and local or private Web Service-based integrated systems in a Business-to-Business oriented environment. During the development, which has been lasting for six years, UDDI specifications have evolved, reflecting the changed vision of the scenario where they settled. The main steps of the development process are summarized in Table 2.1. At first, UDDI was focused on the foundation of public, huge, global registries, aimed at making easier business-systems integration. At present, UDDI's implementations are still focused on providing such a middleware connectivity, but UDDI registries usually tend to be private or semi-private, serving on internal-environment intra-lan or extra-net connections shared with trusted business-partners. The most recent release of the specifications is the version 3.0.2 [60], released on October 2004, addresses increased trust mechanisms and more robust categorization and security capabilities. Moreover, according to the established service-policies, applications may apply different schemes, which are summed up in Table 2.2.

Table 2.1: A brief chronological history of UDDI specifications [81].

Version	**Date**	**Key Objective**
1.0 [87]	September 2000	Create foundation for registry of Internet-based business services
2.0 [57]	June 2001	Align specification with emerging web services architectures and provide more flexible taxonomy
3.0 [59]	July 2002	Support wide interaction of private and public implementations (URI-based keys, promoting keys across multiple registries)
3.0.2 [60]	October 2004	Support for digital signatures, multistep queries, pointers to actual services moved from Globally Unique Identifiers to Universal Resource Identifier

Table 2.2: The three deployment schemas UDDI applies to.

Registry Accessibility	**Network Type**	**Example Application**
Public	Internet	Universal Business Registry (UBR)
Private	Intranet	Internal test environment, Application Server
Semi-Private	Extranet	Trading Partner, Vertical e-Marketplace

UDDI specifications contain the semantics of data structures which are specified through the UDDI API Schema. UDDI messages structure relies on XML to spread among the clients the required Web Service interfaces. Data contained within UDDI Registries may be categorized into two main groups: XML-based and key-type elements. XML-based data are also known as *datum*: datum can be individual facts about a business, as well as data about its services, related technical information, information about specifications for services or assertions within a business relationship; key-type data are meant to uniquely identify datum as well as other resources within the UDDI infrastructure.

In the UDDI architecture, each datum is kept separate. UDDI nodes associate each datumwith a single *Universal Unique IDentifier* (UUID). UUIDs are used to access the associated datum on demand. From UDDI specifications 3.0 on, it is suggested UUID to be based on mnemonic *DNS* address.

UDDI registries store and propagate data structuring them into four types of XML-based data structures, which contain nearly every information that can be retrieved about Web Services, Web Services providers and how to interact with those Web Services. Since these structures are based on the XML language (which are usually WSDL documents), their representations follow predefined schemes. Each data structure represents a level of the hierarchy which stands among informations describing Web Services.

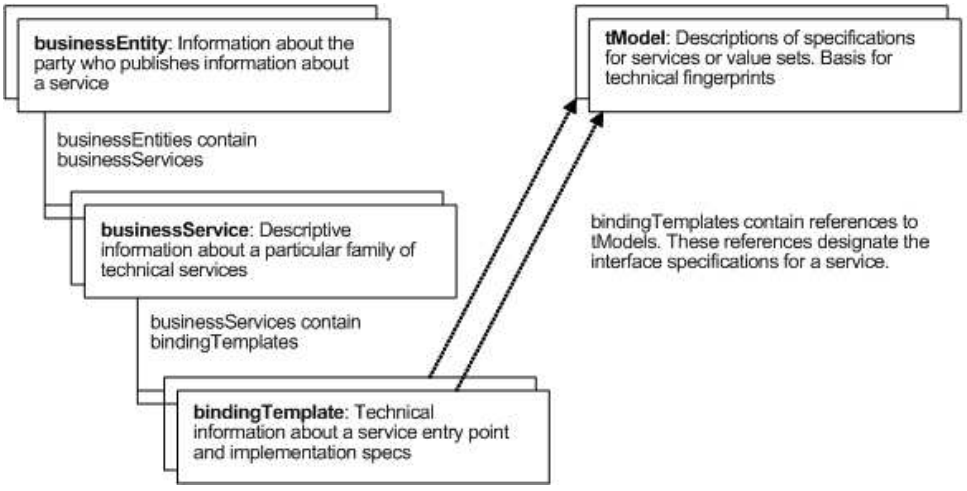
XML-based data are classified as follows:

- *businessEntity*: a *businessEntity* instance represents a business or a provider within the UDDI by containing descriptive information about subscribers and the services they offer. Those information may include names and descriptions in multiple languages, contact and classification information. Usually, data contained in the *businessEntity* structures is referred to as "white pages".

- *businessService*: data regarding the description of every service contained in a *businessEntity* are held in a *businessService* instance. *businessService* structures (also known as "yellow pages") keep the description of an industrial categorization or a particular family of technical services, based on standard taxonomies.

- *bindingTemplate*: this structure (also named "green pages") contains technical data regarding a particular service, such as how to interoperate

Figure 2.2: A representation of the hierarchy among core structures [59].



with that service, its entry point location and its construction specifications.

- *tModel*: *tModels* describe specifications for taxonomies or services, just like a technical fingerprint. Each *tModel* should contain an *overviewURL*, which references a document that describes the *tModel* and its use in detail. *tModels* are referred by Web Service descriptions (held by a *bindingTemplate* instance) in order to indicate their compliance with the service type definition.

Often in the literature, *bindingTemplate*, *businessService* and *businessEntity* are also known as *core structures*. As shown in the table presented in Figure 2.2, core structures instances are nested in a 4-tier hierarchical XML scheme. The whole composed structure may refer to an external *tModel*, which stands as a separate entity. No single core structure, identified by its own unique identifiers, is ever contained in more than one parent structure: though a strict intercorrelation stands among various instances of the three types of core structures. The same "at the most one parent" property is shared by *tModels* structures.

The Replication API [58] introduced the possibility of having multiple UDDI implementations cooperating and sharing information. The Replication API has then been included into the specifications 3.0, and relies on the notion of UDDI Registry. A UDDI Registry can be made of a cloud of applications, named UDDI nodes, providing interaction with UDDI core data structures through one or more UDDI API sets (Inquire, Publication, Subscription). By choosing appropriate policies, multiple registries may form a group, known as an "affiliation", whose purpose is to permit controlled copying of core data structures among them. A UDDI registry affiliation has the registries sharing a common namespace for entity keys, having compatible policies for assigning keys to entities and allowing publishers to assign keys according to policies. A node conceptually has access to and manipulates a complete logical copy of the

UDDI data managed by the registry of which it is a part. Typically, UDDI replication occurs between UDDI nodes which reside on different systems in order to manifest this logical copy in the node. The replication mechanism makes use of the *Update Sequence Number* (USN) to keep track of the number of modifications applied to resource databases. USN are incremented whenever data changes occur whenever an operation which involves changes in any datum occurs, thus USN operate as always-increasing high mark vectors. Gaps in an Operator node's USN sequence are explicitly allowed and are likely to occur upon system crashes or restarts.

From the user's perspective, each UDDI Registry appears as a single system holding all the knowledge its nodes own. When publishing information in a UDDI registry the information becomes part of the published content of the registry. During publication of an item of UDDI information, a relationship is established between the publisher, the item published and the node at which the publish operation takes place.

2.1.2 ebXML

Electronic Business using eXtensible Markup Language (ebXML) Consortium[1] is a suite of specifications aiming at enabling enterprises of any size and in any geographical location to conduct business over the Internet. The ebXML started in 1999, stemming from earlier work on *Object Oriented EDI* (ooEDI), as an initiative of the *Organization for the Advancement of Structured Information Standards* (OASIS) and the United Nations/ECE agency *Centre for Facilitation of Practices and Procedures for Administration, Commerce and Transport* CEFACT.

Similarly to UDDI, the ebXML architecture is built on top of XML, SOAP, and WSDL. Similarly to UDDI, it provides support for discovering and integration of third parties' Web Services. Though, it is not a UDDI equivalent. Even if it builds atop the same building blocks UDDI relies on, the ebXML architecture provides a wider set of functionalities, including, among the others, extensive support to business processes and messagging capabilities. That is, the ebXML architecture provides end-to-end for electronic business, whereas the UDDI architecture is focused on Web Services discovery only, leaving all the other aspects of Web Services integration to the other layers of the Web Services Stack.

The original project envisioned and delivered five layers of substantive data specification, including XML standards for:

- Business Processes Specification [27]
- Core data components [30, 29]
- Collaboration protocol agreements [28]
- Messaging [31]
- Registries and repositories [33]

[1] `http://www.ebxml.org`

The specifications of the above listed components have been approved by the *International Organization for Standardization* (ISO) as the ISO 15000 standard, under the general title, "Electronic business eXtensible markup language". Since ebXML is made up of a set of different specifications, is more a is a self-contained set of specifications that is internally consistent and doesn't rely on emerging standards than a standard itself.

The *ebXML Message Service* (ebMS) extends the SOAP specifications to provide features related to either security and reliability:

- message persistence
- packaging payloads
- retries on delivery error
- error notification
- receipt acknowledgment
- management and verification of digital signatures, authentication and authorization
- error handling

Similarly to SOAP, the ebMS specifies protocol bindings to HTTP and SMTP, and allows implementations to support other protocols. The ebMS specification imposes no restrictions on the content messages carry, may it be XML, EDI, or binary data. Moreover, message's content can also be referenced by hyperlinks only and not transported itself.

Typically, ebXML implementations are built around the ebXML registries and repositories. The registries provide the "yellow page" services, indexing and a large range of data objects, and the repositories hold them. The types of data that may be included into a repository are, among the others, XML schemas, business process descriptions, ebXML Core Components, UML models, generic trading partner information, and software components. All the content of a registry is organized accordingly to the *Registry Information Model* (RIM) [34], which provides a blueprint for high-level schema and information on the type of metadata that are stored in the registry. From the point of view of the registry's client, information may be accessed through two interfaces: the *Lifecycle Management Interface*, used to manage the lifecycle of the objects in the registry and the repository, and the *Query Management Interface*, which is used to make queries against a registry.

In the ebXML framework two companies run business to business based on an ebXML *Collaboration Protocol Agreement* (CPA) on the basis of a previously shared *Collaboration Protocol Profile* (CPP). A CPP provides the information describing how a party intends to do electronic business:

- general information about the party: contacts, industry classification, identifiers, websites' addresses, etc ...
- supported business processes

- the roles in the business: for instance, "buyer" or "seller"
- adopted transport protocol: e.g. HTTP, SMTP, HTTPS
- security related info: is a secure transport protocol is used and which, if encryption is adopted, which algorithm and, eventually, related public keys or certificates
- how business documents are packaged: for instance, in case XML documents are to be exchanged, which XML schema they comply to

Business parties store their CPPs into their repositories, and make registries index them. Trading partners discover their peers inquiring the registries, and when suitable parties are found, they start negotiating how the electronic business has to be carried out. The CPA results as the optional negotiated intersection of two CPPs to be used by all the parties who play roles in a business process, and may contain technical details and functional overrides to the perties' CPPs. The CPAs are therefore registered to the parties' ebXML Messaging Services prior the conduction of businesses.

Due to the similarities between the ebXML's design based on registries and repositories, and the UDDI's one, there is a certain overlapping between the two different architectures. This makes possible having UDDI and ebXML implementations interoperating, for instance having UDDI registries containing ebXML information [88], or using UDDI's facilities to discover ebXML's registries and repositories [35].

2.1.3 Comparing UDDI and ebXML

Both the ebXML and UDDI standards have a prominent role in the current Web Services scenario. They are often considered "alternatives", but this is not correct. It is true that both the sets of specifications provide support for service discovery. However, they have very different scope and extent. On the one hand, the UDDI standard specifically focuses on enabling publication and discovery of services. On the other hand, ebXML is an end-to-end framework supporting business-to-business scenarios. It is made of several standards providing functionalities ranging from service discovery to messagging, security, support for business processes, collaboration protocol agreements, and more. That is, UDDI is one of the building blocks of a SOA, whereas ebXML is a complete SOA itself. In fact, the ebXML standards cover all the requirements typically listed for SOAs: service description is assessed through the adoption of WSDL, service composition is addresses by the ebXML Business Process Specification Schema, reliability, messagging, security and transactions are supported by the ebXML Messaging Services specification.

Differently, UDDI Registries can be compared with ebXML ones, for they provide similar functionalities and have similar aims. A schematic comparison of UDDI and ebXML Registries is proposed in Table 2.3. UDDI Registries are very specific: they focus on storing and providing inquire support for business/service descriptions. Differently, ebXML registries are general purpose distributed content and metadata management system. This different approach

Table 2.3: A comparison between UDDI Registries and ebXML ones[8].

	UDDI Registries	**ebXML Registries**
Arbitrary Content	No	Yes
Extensible information model	No	Yes
Collection support (ability to group data together)	No	Yes
Registries' Coupling	Tight Coupled	Loosely Coupled

causes different registries' characteristics: ebXML registries provide persistence for any type of content, the possibility of creating arbitrary relations among different data (the *association* specified in the Registry Information Model), and the capability of extending the information model of the registry. Otherwise, UDDI Registries can not contain arbitrary content: non-UDDI data are stored elsewhere and referenced through their URLs (e.g., *overviewDocumentURL*), data are related only through the relationships established among the *core* datastructures, and the information model of the registries can not be modified.

Another relevant difference between UDDI and ebXML registries is how different collaborating instances of the same kind of registries couple. UDDI Registries are tight coupled: the adopted replication mechanism requires contracts between registry operators prior to interactions among the registries (e.g., a shared initial set of information in the registries). Conversely, ebXML Registries can cooperate in a loosely coupled manner creating *federations*: federated registries return to inquiring clients a unified result set after applying the query to each member of the federation in a parallel [56].

2.2 Extending Current Service Brokers

Several architectural extensions have been proposed to both the UDDI and ebXML architectures. Some extensions aim at adding functionalities to the existing registries, whereas others strive at enhancing the expressiveness of service descriptions, for instance by adding semantics information.

The UDDI Extension (UDDIe) [77] adds to existing UDDI implementations the concept of *blue pages*, that is, publication and discovery support for user defined properties associated with a service. The user-defined properties are stored in a data structure named *propertyBag*, which is comprised within an extended version of the UDDI's *businessService* data type. Moreover, UDDIe extends the search capabilities with numeric and logical (AND/OR) ranges, and it enables "leasing" for service publications. The leasing functionality consists in allowing service providers to register with UDDI for a limited time period, which can also be start at a future time. After the lease's expiration, the service description is no longer available for discovery in the registry. Leasing is suitable for services whom descriptions change often, or that will be available only for

a finite, known *a priori*, span of time.

Quality of Service (QoS) awareness to service discovery can be built on top of UDDIe [3] by implementing the *Grid QoS Management* (G-QoSM) framework described in [7]. G-QoSM aims at providing support for resource and service discovery and management based on QoS properties. QoS can be guaranteed either at application, middleware or network level. Enforcing of QoS parameters is based on *Service Level Agreements* (SLAs).

QoS awareness to UDDI service discovery is also provided by the UDDI eXtension (UX) [92, 91]. The QoS metrics provided are response time, cost and reliability, thou other metrics can be added at will. Metrics are measured both through predictions based on analysis of the network model of service requester's connection at publication time, and feedbacks posted by service requesters. Service requesters' feedback is used to create *summaries* for invoked services, which are themselves used to predict services' future performances. Moreover, UX supports federated service discovery, that is, having multiple registries cooperating to serve requesters' queries. When a UX server does not have enough information to carry out a request, it starts propagating the processed query to the other servers. However, it may be the case that the same service, registered to different UX servers, accounts different QoS characteristics on the latter. The differences between the same service's performances by requesters in different domains are measured through the similarity of QoS between two domains. Similarity measurement is based on a dynamic learning approach that analyzes the QoS summaries of the involved UX servers. When two domains are not "similar", predictions are corrected through the translation on received summaries according to the recorded differences. Queries are propagated among the UX servers according to the *Cooperating Server Graph* model (CSG). The CSG model and propagation tree adapts dynamically to the change of cooperating servers and the underlying network topology. The CSG approach uses a minimum-weight spanning tree to optimize links between UX servers automatically. In case of failure of one of the nodes in the graph, which may be due to a network failure as well as a service denial, information are propagated to all neighbors of the failed node in the tree. Nodes' addition and removal, long time failures and the relative recovery, are handled through local reconfigurations of the propagation trees that are local to the nodes. Local reconfigurations are then spread to all the nodes when a given amount of changes in the tree is triggered.

Federated UDDI nodes have been introduced in the UDDI standard with the Replication API specification [58], and then included in the UDDI specification version 3.0 [59] (see Section 2.1.1). However, other replication approaches have been investigated. The lazy replication scheme presented in [85]. Relying on *Read-Only Transactions* (ROTs), it is basically a two phase locking protocol integrating based on *Monotonic reads guarantee*, which means ensuring that "read operations are made only on database copies containing all writes whose effects were seen by previous reads within the session" [86]. An eager middleware based replication scheme is proposed in [84]. Eager replication foresees all the nodes updating the modified information at the same time on the basis of concurrency control to guarantee serializability across the system. Still preserv-

ing the notion of "custody of data", which is proper of the UDDI specifications, this approach removes the gap between the data modification and the relative propagation, at the cost of having an increased complexity of the replication mechanism.

Query federations of UDDI Registries are proposed in [73]. Each registry acts as a node in a graph. The nodes route the received queries to their peers they are connected with. The idea is borrowed from the query federation originally defined by the trading object specification of CORBA, and enhanced with authentication and authorization control for access to UDDI entries within the nodes. Since the nodes can be connected by a graph, in order to avoid processing the same query multiple times, nodes blacklist the queries' identifiers that have already been processed.

2.3 Distributed Service Brokers

Service Registries have traditionally had a centralized architecture, e.g. UDDI, and, if distributed on multiple machines, they are consisting of multiple repositories that synchronize periodically. Replication may solve issues typical of centralized architectures, such as *single points of failure*. However, as the number of Web Services grows, and they change more often, centralized approaches with replication quickly become impractical. When adopting a peer-to-peer approach, the quality of information is more up-to-date because the publishers are also the owners and managers of the data, though hindering the reachability of the information. However, the reliability provided by the high connectivity of peer-to-peer systems comes with performance costs and lack of guarantees of predicting the path of propagation; moreover, the reachability of the information may be hindered by network partitions or peers leaving the system.

As a matter of fact, peer-to-peer systems and centralized registries strike different tradeoffs, and this makes them fit different situations. On the one hand, centralized registries are more appropriate in static environments where information is persistent. On the other hand distributed/peer-to-peer discovery architectures fit in dynamic environments such as ubiquitous computing, or in situations in which the service descriptions are likely to change often.

Several Peer-to-Peer systems have been proposed, providing service brokering facilities, some relying on well-known and widely adopted pure peer-to-peer protocols, others adopting hybrid solutions. In [76] a peer-to-peer system is proposed, which makes use of a distributed hash table (DHT) to index service descriptions. The system makes use of the Chord [82] overlay network topology, routing algorithms and management mechanism. Service descriptions are indexed by keywords. Queries can containing partial keywords and wildcards. The index space is created through an approach based on Hilbert space-filling curves [54], which guarantees that all existing service descriptions matching a given query are to be found with bounded costs in terms of number of messages and number of nodes involved.

The *Web Service Discovery Architecture* (WSDA) [45] faces the problem of distributed service discovery with an approach based on grid computing, peer-

to-peer networks, and distributed databases. The WSDA makes use of a small set of orthogonal primitives covering service identification, service description retrieval, data publication, and query support, that are combined together to provide high-level functionalities. Querying can be made via XQuery [12] expressions and MinQuery, which is a very simple SQL-like query language created ad-hoc. The WSDA's registries may support caching, and are notified by service publishers upon data changes. The WSDA's peer-to-peer capabilities are based on the *Peer Database Protocol* (PDP) [44], which supports centralized and peer-to-peer databases, is message-oriented (that is, loosely coupled, operating on structured data), stateful at the protocol level and supports transactions for one or more discrete message exchanges.

The pure peer-to-peer system presented in [42] makes use of the UDDI *businessEntity*, *businessService*, *bindingTemplate*, and *tModel* datastructures to describe publications that are made available in a pure peer-to-peer network which resembles the structure of Gnutella's ones [71]. Service publishers make available their services by joining the peer-to-peer network, thus becoming *WS-Peers*, whereas service consumers join the network as *User-Peers*. The join of a WS-Peer is carried out by contacting another WS-Peer, which provides the newcomer with a mixed set of addresses of WS-Peers and User-Peers. Queries' routing is based on *neighborhood*: for WS-Peers, all the other WS-Peers of which the address is known are neighbors, whereas the User-Peers consider the other User-Peers neighbors as well as the WS-Peers. All the peer collaborate in spreading queries, routing the ones they receive to all their neighbors. Timestamps are associated with the neighbors, keeping track of the last time there have been communications between the peers. Peers with the oldest timestamp are dropped when a threshold of known neighbors is reached.

A large number of the peer-to-peer service discovery architectures that have been proposed in the past few years make use of semantics element to describe published services. They are described in Section 2.4.2.

2.4 Semantic Web Services

Web Service descriptions encoded in WSDL provide syntactic-level descriptions of Web Services' functionalities, lacking of any formal definition concerning what the syntactic definitions might mean in terms of provided capabilities. Automatic composition of services heavily depends on the precision that the underpinning discovery mechanism addresses. In particular, the discovery mechanism should be based on required/offered service capabilities, rather than the keywords-based approach that is used, for instance, in search engines. Technologies developed within the Semantic Web have been applied to Web Services, aiming at creating intelligent services supporting automatic discovery, composition, invocation and interoperation. Such enhanced Web Services are called *Semantic Web Services* (SWSs) [41, 53, 83].

One of the goals of Semantic Web Services research is to improve the services' discovery process. This is assessed by increasing the expressiveness of service descriptions. This is typically done by introducing semantics in ser-

vices' descriptions. One of the languages adopted for such a task is OWL-S, is described in Section 2.4.1.

The *Semantic Web Services Initiative* (SWSI)[2], the *Data, Information, and Process Integration with Semantic Web Services* (DIP)[3], and other initiatives, take place in industry and academia, investigating solutions for the main issues regarding the infrastructure for SWSs. Semantic Web Service infrastructures can be characterized along three orthogonal dimensions [15]:

- *usage activities*: functional requirements, such as publishing, discovery, deployment and invocation that a framework for SWSs is expected to support
- *architecture*: the components needed to accomplish the usage activities, such as reasoners, matchmakers, decomposers, registries and invokers
- *service ontologies*: aggregations of all the concept models related to the description of a Semantic Web Service, thus being a knowledge-level model of the information describing and supporting the usage of the service; service ontologies can optionally contain or link to domain knowledge

Service providers publish SWSs to and discover SWSs from semantics-enabled registries, named *Semantic Registries*. Services are discovered through a semantic matching between the description of available services and the description of the service request, which may involve the service name, required input, provided output, preconditions, and tasks or goals to be achieved. Service ontologies define SWSs representing the capabilities of a service itself and the restrictions applied to its use. Service ontologies integrate at the knowledge-level the information encoded in Web services standards, e.g., UDDI and WSDL, with related domain knowledge. Related domain knowledge may include:

- inputs, output, pre-conditions and post-conditions
- category, cost and quality of service
- provider related information (contacts, company name and similar)
- task or goal-related information

In the end, service ontologies used for describing SWS rely on the expressiveness and inference power of the adopted underpinning ontology language.

2.4.1 Describing Web Services with Semantics

The semantic description of Web Services is a goal pursued by several Working Groups, which have proposed different languages that share similarities. Hereby are introduced the three languages that seem to be most relevant and likely to be widely adopted in the next few years: OWL-S, WSMO and WSDL-S. Moreover, a brief comparison among the languages is proposed, together with an analysis focusing on how the languages may interoperate one with each other.

[2] `http://www.swsi.org`
[3] `http://dip.semanticweb.org`

OWL-S

OWL-S [51] is a Web Service ontology framework based on the *Web Ontology Language* (OWL) [9, 79, 52], which enables users and software agents to automatically discover, invoke, compose, and interact with web resources offering a service that adheres to requested constraints [70]. OWL-S evolved from DAML-S [5], which relied on the *DARPA Agent Markup Language and Ontology Inference Language* (DAML+OIL)[4], a description logic based Web ontology language which describes the structure of a domain in terms of classes, named *concepts* in description logic, and properties, which in description logic are called *roles*.

OWL-S provides means of expressing semantic information enabling service descriptions to be made and shared. This additional information consists of a set of markup constructs (i.e., basic classes and properties), that can be used to declare and describe services. In OWL-S, complex services are referred to as services which are composed of multiple services, which typically require users to perform multiple interactions with the service to gather the desired results. Service Descriptions written in OWL-S are made out of the following components:

- *Service Profile*: offers support automatic service discovery by reporting information like the organization providing the service, the service's name and functional aspects of the service, non-functional aspects of the service such as contact information of the individuals or organization responsible for the service and additional features used to specify the characteristics of the service (industrial categorization of the service, qos characteristics, geographic availability and such;

- *Service Model*: describes how a service works, by defining relations between inputs and outpouts by the means of pre-conditions and effects;

- *Service Grounding*: specifies the details of how a service can be accessed, such as a communication protocol, message formats, and other service-specific details such as port numbers used in contacting the service;

- *Resource Ontology*: contains classes of attributes to express pre-conditions of processes and sub-classes for resource composition.

WSDL-S

Web Service Semantics (WSDL-S) [2] is an extension of WSDL with semantic enhancements. WSDL-S provides an approach for creating semantic Web service descriptions that has a lot of similarities with OWL-S, but is slightly less expressive. WSDL-S's structure is simplified with respect to OWL-S's one in the sense that the former does not separate the service grounding from the service profile. WSDL-S allows integration of semantic and non semantic descriptions of Web services, having as users specifying defined XSD types translating them

[4] `http://www.daml.org/2001/03/daml+oil-index.html`

into OWL. Moreover, WSDL-S supports definitions of types according to the *XML Metadata Interchange* specification [62].

The WSDL-S specification builds on top of WSDL 2.0, adding tags to pre-existing elements. Operations' semantics is specified by the means of adding *concepts* and constraints, which may be *preconditions* and *postconditions*, to the WSDL's element *operation*. A concept specified in an operation uses to specify semantic annotations explaining what does the operation do. Likewise OWL-S ones, pre- and post- conditions allow to specify constraints to operations. Concepts may also be specified in association with WSDL's elements *part*, thus specifying the meaning of input, output and fault through semantic annotations. Finally, the WSDL's *service* element is enhanced with the possibility of specifying a *domain*, which are taxonomy entries pointing out the service's category (i.e. elements of NAICS taxonomy), and *location*, which reports the geographic location of the service by the means of taxonomy entries (for instance, entries from the ISO 3166 geographic taxonomy).

Nowadays, WSDL-S is primarily supported by the METEOR-S, which is a framework aiming at enabling semantics in the complete lifecycle of Web processes. METEOR-S is presented in Section 2.4.3.

WSMO

The *Web Service Modeling Ontology* (WSMO) [48] provides both a conceptual framework and a formal language to semantically describe Web Services. WSMO is a meta-model based on ontologies, and it is derived from *Web Service Modeling Framework* (WSFM) [40]. The WSMO's meta-model is defined on the basis of the *Meta Object Facility* (MOF) [61]. MOF is a set of standard interfaces that can be used to define and manipulate a set of interoperable meta-models, such as the UML ones, and their corresponding models.

The WSMO's *top-level elements*, the main building blocks of a WSMO document, are:

- *Ontology*: provide the terminology used by other WSMO elements to describe the domains involved;
- *Web Service*: describe the computational entity providing access to services by the means of capabilities, interfaces and internal working; this is done by using the terminology defined by the ontologies;
- *Goal*: represent user aims, for fulfilling which which Web Services are invoked; goals may be described through Ontologies;
- *Mediator*: describe elements that assess interoperability between different WSMO elements that may arise, for instance, on the data, process and protocol level; mediators can solve mismatches between different used terminologies, communication means between Web services and combining Web Services and goals.

WSMO supports the reuse of external ontologies through importing. In case compatibility problem arise, they can be addressed through mediators. Web

service are described by the means of functional, non-functional and behavioral aspects.

The semantic descriptions developed according to WSMO meta model are expressed in the *Web Service Modeling Language* (WSML) [21]. The WSML and WSMO languages are deeply related: the WSMO provides the conceptual model for describing Web services, ontologies, goals and mediators, whereas WSFL provides a formal language for writing, storing and communicating such descriptions [72]. The reference implementation of WSMO is named WSMX, and it is presented in Section 2.4.3.

Comparing WSDL-S, OWL-S and WSMO

The three of WSDL-S, OWL-S and WSMO languages aim at semantically describing Web Services. Since they share a similar, goal, their peculiarities can be compared. Moreover, there have been efforts to adapt the languages so that they may interoperate, and enabling Semantic Web Services descriptions to be done taking the good from all the languages.

WSMO and OWL-S aims at describing Web Services by the means of ontologies. However, they achieve this goal in very different manners: OWL-S explicitly defines a set of ontologies that support reasoning about Web services, whereas WSMO defines a conceptual framework within which these ontologies will have to be created. Moreover, OWL-S and WSMO greatly differ in how they handle different types of Web Services. On the one hand, WSMO define the Mediators, that are programs providing convenient mappings between different Web services to have the latter interoperating. On the other hand, OWL-S simply does strive at solving directly this issue, but instead provides Web Services and their clients with the information needed to find existing mediators that can reconcile their mismatches, or creating such mediators through Web Service composition.

Since WSMO and WSDL-S are both about semantically describing Web Services, there have been efforts to align them. In [46], few additions to the WSDL-S specification are proposed so that it may cover the features of WSMO, and using the latter to be adopted as semantic annotation language. Another possible mean of inter-operation between the two languages is using WSDL-S as WSMO grounding, that is, explaining how Web Services described in WSMO can be actually contacted, how are structured the messages they exchange, and all the other Web Services' functional aspects that are proper of WSDL-S.

WSDL-S and OWL-S mainly differ in the design approach: OWL-S does not built on top of pre-existing WS-related specifications, whereas WSDL-S does. This is likely to facilitate the adoption of WSDL-S, which may take advantage of the consistent legacy of adopters of its ancestor, WSDL. By the way, there are also some relevant technical differences between the two languages: WSDL-S abstracts from the semantic representation language, which is not true for OWL-S. Another remarkable difference is how the Web Services' grounding are specified: with WSDL-S the correspondence must be given in the WSDL spec, whereas with OWL-S it is given in a separate OWL document [4].

2.4.2 Semantic Registries

Many Semantic Registries have been proposed, either built on top of the UDDI and ebXML infrastructures, or designed from scratch adopting different approaches.

Semantic Registries on top of Current Service Brokering Infrastructures

The UDDI architecture stands at the basis of several works aiming at producing Semantic Registries. One of the first attempts is presented in [22], in which the upper ontology of DAML-S is extended to relate Web Services with electronic catalogs, allow the description the complementary services and Web Services retrieval on the basis of complementary functionality (e.g., a Web Service that provides "Sell" with one that offers "Delivery of goods") the properties of products or services. The DAML-S documents are classified on the UDDI Registry similarly to WSDL documents, and stored as *OverView Documents* on the *tModels* that are assigned to both the generic services as well as its implementations.

An approach similar to the one described above is presented in [69], where the requirements the implementation and the integration of a DAML-S aware matchmaking algorithm with UDDI Registries are discussed. When DAML-S evolved into OWL-S, the idea of using semantic markups to describe Web Services in the UDDI Registries kept being explored. In [80] is presented a OWL-S/UDDI matchmaker that matches OWL-S *Profile Descriptions* documents stored within a UDDI Registry on the basis of the mappings with the UDDI core data exposed in [64]. The adopted UDDI Registry is Apache jUDDI[5]. Specialized *tModels* are used to contain OWL-S Profile's *Input*, *Output*, *Service Parameter* and similar, likewise the WSDL-to-UDDI mapping proposed in [18] by the OASIS Committee. The matching algorithm supports *degrees* of match: *exact*, where the service description fits perfectly the user's request, *plug in*, where the service capabilities is very likely to satisfy the request because it provides more general functionalities (subsumption), *subsume*, where the service may not be able to satisfy all the user's needs, because it provides less general capabilities than the ones requests, and *fail*, where there is no match between the service capabilities and the one needed.

Similarly to the UDDI Architecture, the ebXML one has been adopted as foundation for a Semantic Registry. Storing and managing in ebXML Web Services' descriptions enhanced with OWL-S is presented in [23, 24]. Differently from UDDI Registries, ebXML provides facilities (the *ClassificationScheme* described in the ebXML Registry Information Model specification [32]) enabling metadata to be stored in the registry. The semantic of individual service instances are to be stored externally to the registry (as "ExtrinsicObject" in ebXML terminology). Since the content of external objects can not be queried through ebXML queries, it is proposed a workaround based on concatenations of queries based on fixed templates.

[5]Apache jUDDI Home Page: `http://ws.apache.org/juddi/`

Distributed Semantic Service Discovery Architectures

Distributed Semantic Service Discovery had a lot of contributions in the past few years. HyperCup [75] is a peer-to-peer network in which peers share and consume Web Services. Peers are organized in a graph structure based on hypercubes. The topology of the network takes into account ontologies, too. Semantic Web Services can be described by using various ontologies in parallel, augmenting a service ontology by domain ones. On the basis of the ontologies that describe their services and their interests, the peers are organized into *concept clusters*. This reduces the number of query routing needed to retrieve matching services. The query propagation mechanism exploits logic minimization to identify queries' logical minterms, that denote a group of concept clusters. Broadcasts of queries are made only within the concept clusters that represent the concepts the queries comprise. The performances in query routing and the less number of messages exchanged to route the queries are balanced by the need of creating and maintaining the topology of the network when peers join or leave it, or when new services are available.

In [65], peer-to-peer networks adopting the Gnutella protocol are augmented with the capability of finding Web Services on a semantic manner, sharing and discovering service descriptions encoded in the DAML-S language. Every node on the Gnutella-based peer-to-peer network contains DAML-S description of its capabilities, the associated engines for parsing ontologies, and a peer-to-peer discovery module. The information contained within the DAML-S *Process Model* and *Grounding* are used to carry out interaction with other Web services, whereas the data of *Profile* and *Matchmaking* come in hand to manage the discovery and location of providers.

Chord underlies the framework for Web Service discovery with ranking support introduced in [36]. The system proposed is a structured peer-to-peer system and considering the functionality, behavior and the reputation of the Web services during discovery, this enabling the ranking of the search results according to the trust and service quality ratings of the matching Web Services. Service Descriptions are subdivided in *implementation*, *service* and *requests*. The implementation can be considered as the BPEL/DAML-S description of the Web Service and it is represented by a finite automaton. Likewise the implementation, also the service is represented as finite automaton, which describes the set of interactions observed by the users of the Web Service. The requests is made of the set of finite automata corresponding to other Web Services on which the current Web Service depends.

2.4.3 Semantic Web Services Frameworks

Along with the enhancement of existing Service Registries through the support for semantic descriptions of services, entire frameworks have been proposed, providing functionalities that can lead and support the whole lifecycle of Semantic Web Services. In this Section we overview the most prominent frameworks: WSMX, IRS and METEOR-S.

WSMX

The *Web Service Modelling eXecution Environment* (WSMX) [90, 17, 74] is the reference implementation of WSMO (see Section 2.4.1). WSMX supports providers and requesters providing them with features enabling discovery, selection, mediation, invocation and interoperation of Web Services based on semantic descriptions.

WSMX is a distributed framework. Its nodes can be arranged as peers in a peer-to-peer network; peers receive goal definitions and use the discovery components they retain to search for matching services. The main components of the WSMX architecture are the following:

- *Adapter*: are components allowing external software modules to integrate with WSMX, for instance converting, on the basis of transformation rules preserving the semantic, the received information in the WSML language that WSMX uses, or carrying out the delivery of messages to the internal components of the WSMX architecture;

- *Compiler*: checks the syntactic validity of WSML documents and stores parsed information persistently;

- *Matchmaker*: finds the appropriate Web Services to achieve goals. In order to retrieve Web Services to match, it may rely on either an internal repository of known services, or on an external UDDI Registry;

- *Data Mediator*: fulfills the needs of mediating between two different data formats by loading the stored mappings, creating the mapping rules, and applying them to the input data (it corresponds to a subset of the functionalities provided by the *ooMediator* described in WSMO);

- *Choreography Engine*: provides the means for facing issues related with the heterogeneity in communication patterns that may arise between service provide and requester; this is done through the runtime analysis of two given choreography instances and to use the mediators to compensate the possible mismatches that may appear, for instance by generating dummy acknowledgement messages, grouping several messages in a single one, changing messages' order or even dropping some of the messages;

- *Composition*: executes complex compositions of services, by using either hard-coded business rules or adopting an external process language, to achieve a certain goal;

- *Communication Manager*: handles the various invocations that may come from requesters and feeds the latter with the results outcoming the invocation of Web Services.

WSMX is an Open Source Project, which can be downloaded at the URL `http://www.wsmx.org/downloads.html`.

IRS

The *Internet Reasoning Service* (IRS) is a framework and implemented infrastructure, whose main goal is to support the publication, location, composition and execution of heterogeneous web services, augmented with semantic descriptions of their functionalities. IRS provides its users with flexible mappings between services and problem specifications and dynamic, knowledge-based service selection. Moreover, since it explicitly separates task specifications, that is the problems to be solved, from the method specifications, problems' solutions, from the domain models, it enables *capability-driven* service invocation: given a problem, IRS finds a service providing the needed solution. During the development of IRS, three different mainstream versions of the system have been published: IRS, first release of the system released in 2002 [19], IRS-II [55], which follows the *Unified problem-solving modeling language* (UPML) framework [38, 39], and IRS-III [14, 26, 25], which builds upon the previous implementation and supports WSMO-based Semantic Web Services.

The IRS-II's architecture is made of three main components: the *IRS Server*, the *IRS Publisher*, and the *IRS Client*. The IRS server keeps the descriptions of semantic web services as UPML knowledge level description of tasks, problem solving methods and domain models. The information about Web Services providing the problem solving methods are stored in an internal registry. It is possible to have multiple services described by the same semantic specifications, as well as multiple semantic specifications of the same service. The IRS Publisher links Web Services to their semantic descriptions within the IRS server, and automatically generates a set of wrappers which turn standalone code (written in either Java or LISP) into a web service described by a problem solving method. Finally, the IRS Client provides its users with the aforementioned capability driven Web Services invocation functionalities, by providing an interface and a set of APIs which are both task centric.

IRS-III extends the IRS-II architecture by supporting UPML based types of knowledge models (added goal models, mediator models and web service models), implementing of the the WSMO ontology, which is also supported by a WSMO specific Java API, and the customization of the IRS reflecting the WSMO adoption.

The IRS-II and IRS-III clients are available for download at the address `http://kmi.open.ac.uk/projects/irs/#download`.

METEOR-S

The *Managing End-To-End OpeRations: for web Services* (METEOR-S) [89, 63, 1, 78] discovery engine is a framework that makes use of WSDL-S (see Section 2.4.1) and OWL ontologies to perform logical reasoning inherent in ontological representations. METEOR-S can be subdivided into two parts: the *front-end*, providing facilities for developing, annotating and discovering Web Services, and the *back-end*, which allows service providers to bind services based on the given abstract process, requirements and the process constraints.

The front-end is made of the following elements:

- the *Semantic Web Service Designer*: a graphical user interface to design and develop Semantic Web Services;
- the *Semantic Description Generator*: a tool that produces semantic service descriptions in either annotated WSDL or WSDL-S;
- the *Publishing Interface*: it is a client which uses the functionalities offered by a back-end component, *Discovery Engine*.

The back-end is structured around the *Enhanced-UDDI* component, which builds on top of UDDI v2, using the core data structures as place holders of semantic information. The *Discovery Engine* is based primarily on the semantic descriptions and constraints advertised by the service provider, retrieving the matching data from the Enhanced-UDDI by employing heuristics based on subsumption-relations, data-type matching between requester specified constraints and provider-advertised concepts, and others. The *Abstract Process Designer* creates representations of Web processes on the basis of BPEL4WS, representing the requirements of the services in the process by specifying service templates, and specifying process constraints for optimization purposes. Service templates allow process designer to either bind them to known Web Services, or specify semantic descriptions of the needed ones. The *Constraint Analyzer* dynamically selects services from the candidate ones find out by the Discovery Engine. Finally, the *Binder* binds the abstract process created by the Abstract Process Designer to the optimal set of services found to generate an executable process.

METEOR-S has been released as an Open Source project in August 2004, and it is available for download at:
`http://lsdis.cs.uga.edu/projects/meteor-s/downloads/`.

Chapter 3

The Federation System

In a sense, the state of the art in Web Services brokering (Chapter 2) spans on two different axis: the architecture of the system, which can be centralized, hybrid or fully decentralized, and the adoption of semantic service description languages as building blocks of the system. Examples of service brokers that build on top of semantic service description languages are WSMX (Section 2.4.3) and METEOR-S (Section 2.4.3). However, service brokers that do not rely on semantic-enabled languages do not necessarily prohibit to use the latter for services' descriptions. For instance, OWL-S (Section 2.4.1) documents may be stored in a UDDI Registry as tModels, even if there are no facilities directly relying on the information these documents contain. According to this categorization, the system we propose, named *Federation System* (FS), is a hybrid/semantic-less service broker. We believe that structuring a service brokering architecture in a hybrid centralized/decentralized fashion can grant the desired characteristics of scalability and reliability, without compromising the feasibility of the functionalities. From the point of view of semantics, we adopt an approach that is agnostic with respect to the languages adopted to describe the published services. That is, the Federation System allows its users to specify service descriptions in terms of semantics, but it never uses explicitly this information to provide additional functionalities, such as automatical service composition or similar.

One of the key concepts of the Federation System is the notion of *publication*. Publications are XML documents describing the Web Services published by the Peers comprised within the Federations. Each published Web Service corresponds to exactly one publication; the latter defines the former in terms of service's technical descriptions and service-descriptive data. A suitable example of technical description contained within a publication is a WSDL document consisting of the interface's description of the Web Service to which the publication corresponds, whereas an example of service-descriptive data may be a set of taxonomic entries defining the Web Service's business target activity. For instance according to the Universal Standard Products and Services Classification (UNSPSC) or the North American Industry Classification System (NAICS) taxonomies. Thus, Web Services' discovery within a Federation becomes an issue of matching the requirements defined by the Web Services'

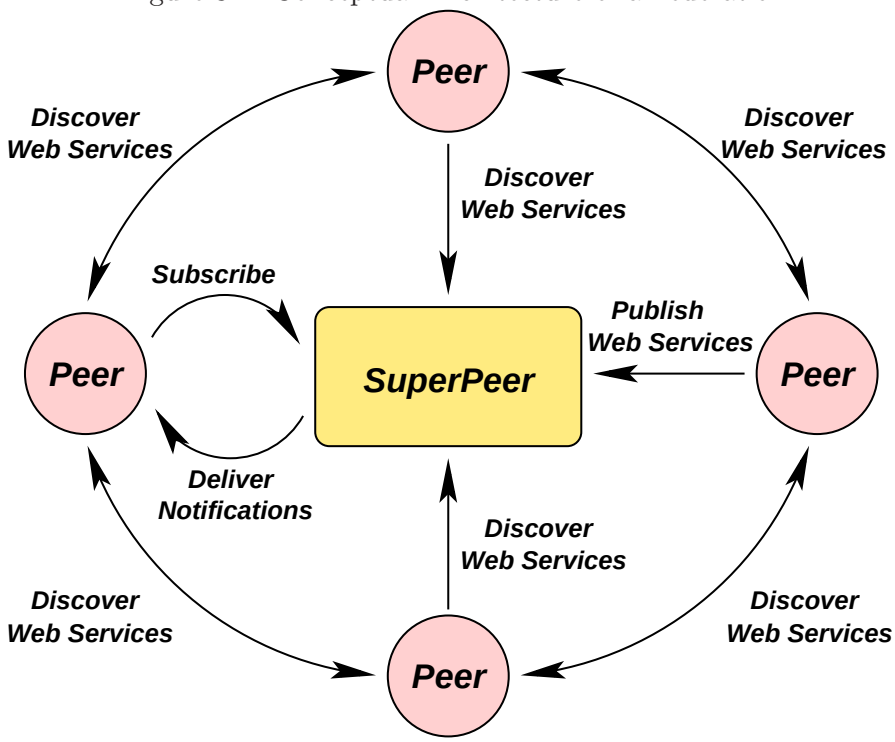
requesters against the currently available publications.

The Federations are formed for specific, specialized areas of interest such as e-travel, finances, marketing, automotive industry and similar. The Federations promote on-demand services by offering sets of related services through the Canonical Service Declarations (CSDs), that is, a set of Web Service prototypes. Each Federation declares its CSDs, which are likely to be more than one per each Federation, according to the kind of the promoted services. Every publication has to *inherit* from a CSD. A publication is said to inherit from a given CSD if every information contained within the CSD applies to the publication as well. Indeed, a CSD is a set of service-descriptive information fitting all the services described by the inheriting publications. In a publication, a subset (or even the whole set) of the data regarding the described service are acquired from the CSD the publication inherits from. In a sense, the relation between Web Services, publications and CSDs is similar to the one associating instances, classes and abstract classes: the CSDs are abstract classes defining a minimal set of attributes and methods that the inheriting classes/publications have to expose, that are then realized by the actual instances of the publications, which are the published Web Services. However, there is a difference to point out when comparing classes to publications and instances to Web Services: on the one hand, more than one instance is allowed for a given class, on the other hand, each publication corresponds to one and only one Web Service.

When a Peer wants to publish a service within a publication, it has to pick a CSDs properly describing the service. If multiple fitting CSDs are available, the publisher Peer still has to select one of them. This design choice is based on the same reasons motivating the single inheritance mechanism which belongs to many object oriented languages. Having publications inheriting from only one CSD at a time avoids the need of resolving potentially conflicting specifications. The goal is to classify published services into homogeneous groups with respect to the available information. In particular, this is effective in the case of a CSD containing a description of a service interface. Since all the services described by publications inheriting from the same CSDs would have to expose the same interface, service consumers would be freed from the charge of integrating into their applications different Web Service clients to interact with services offering similar capabilities but having different interfaces.

To exemplify these concepts let us consider the Federation CarRental4All focused on Car Rental activities. The Federation declares two CSDs, one, named GeneralCarRental, prototyping a general car rental service and the other, called AirportCarRental, focused on services provided by car rental agencies located within airports. The GeneralCarRental CSD contains the NAICS entry for car rental services and a WSDL document describing the Web Service's interface exposing two operations: car booking and a car booking withdrawn. The car booking operation requires as input the model of the car to be rent, the time the car has to be available and the length in time of the car's rent. The AirportCarRental CSD is still made of the NAICS entry for car rental services, but the service's interface it contains is richer, comprising all the operations exposed by the general car rental services' interface and adding another capability: booking a car by specifying the car's model, the air flight with which

Figure 3.1: Conceptual Architecture of a Federation.



the renter is expected to arrive at the airport (data used by the system will to automatically determine when the rent is going to start) and the length of the rent. In a sense, the requirements expressed by the AirportCarRental CSD are a superset of the ones mandated by the GeneralCarRental CSD, thus every Web Service that comply with the former, also comply with the latter. If the car rental agency RentALot located within the Malpensa airport (Milan, Italy) had a Web Service exposing all the operations prescribed by both the CSDs, and it would like to publish its Web Service within the CarRental4All Federation, it would have to choose which CSD to refer to. Let us suppose that the selected CSD is the AirportCarRental one. Then the publication describing the RentALot Web Service would inherit from the AirportCarRental CSD. Moreover, the publication may contain additional information, such as the entry representing the Malpensa airport in some global airport listing, or the physical location of the agency according to some geographical taxonomy.

A conceptual architecture of a Federation is presented in Figure 3.1. A Federation is made of a *SuperPeer*, providing the Peers with publishing, centralized service discovering and subscription functionalities. The Peers join and leave the Federation, publish publications, sign subscriptions, and retrieve information about publication by accessing the SuperPeer's functionalities on a request-response basis. Peers can interact with their equivalents to retrieve publications without relying on the SuperPeer. The notifications are delivered by the SuperPeer to the Peers in response to their subscriptions. The notification delivery, as well as the interactions among the Peers themselves, are carried

out through an event-notification mechanism based on the messagging facilities provided by the Simple Object Access Protocol (SOAP) [13, 43] protocol.

Each Federation is focused on certain types of services represented by the published CSDs. Peers are expected to select the Federation to join according to the CSDs the latter provides: then, by the time they join, newcoming Peers already know the available CSDs provided by the Federation. When a Peer publishes or removes a service, or updates the service-descriptive data, it invokes a dedicated operation on the SuperPeer. Every change performed on the data regarding a published Web Service directly affects the corresponding publication. The SuperPeer matches the involved publication against the signed subscriptions, and then it spreads the notifications to the Peers to which the matched subscriptions belong. Subscriptions are strongly related with the CSDs mandated within the environmental Federation. When a Peer signs a subscription, it submits to the SuperPeer the desired requirements, which are a target CSD and eventually additional technical and service-descriptive requirements. Upon the submission of the subscription to the SuperPeer, the Peer is provided with all the currently available matching publications. There is not any upper bound to the number of subscription a Peer can sign, and subscriptions can be retired at any moment. In order to join a Federation, Peers are not forced to provide publications, nor to sign subscriptions: they may only inquire the SuperPeer for publications in the same way UDDI clients interact with the UDDI registries in order to retrieve data about published Web Services.

When a Peer joins a Federation, it is aware only of the SuperPeer. The Peers know each others upon gaining knowledge of other's publications, by means of the subscription or the inquire mechanism. When a Peer gains knowledge of the existence of an another Peer, the latter is said to become an *acquainted Peer* of the former. The set containing all the acquainted Peers of a given Peer is known as *Peer Acquaintance Group* (PAG). It is likely to be that different Peers do not mutually belong to the other's acquaintance group. For instance, if a Peer is only into providing publications, then it would never sign subscriptions nor it would inquire for other Peers' publications and its PAG would be empty; on the other hand, since the Peer retains publications, it is probably comprised into some other Peers' PAGs. There is no upper bound to the number of PAGs in which a Peer is comprised, nor it is limited the number of acquainted Peers a Peer has.

The acquaintance groups are the basis on which the distributed service discovering facility is structured. The Peer rising a query is said to be the *originating Peer*. The originating Peer spreads the query to all its acquainted Peers, and listens for incoming results. When a Peer receives a query, it matches the latter against its own publications, and then it spreads the query unchanged to all its acquainted Peers. Whenever a Peer detects a matching between a query and one or more of its publications, it delivers a result to the originating Peer. Given a query, each Peer has to process it at most once. In fact, if a Peer was comprised within more than one PAG, it may be provided with the same query twice or more[1]. In order to avoid the repetition of the matching

[1]Actually, if a Peer belongs to n PAGs, it may receive the same query up to n times.

process, the delivery of the results to the originating Peer and also the further spreading of the query, each Peer blacklists the queries it has already processed, thus avoiding wasting of resources.

When describing the Federation System, we often refer to the UDDI standard and its data structures. This is due to the fact that, even through its different architecture and approach to service brokering, the Federation System plays the same role the UDDI implementations do. Since having similar roles usually means sharing many notions and concepts. The UDDI standard is a good comparison for it is widely known and adopted in the Web Service community, we often juxtapose it with the Federation System when presenting the latter. For the same reason, since the Federation System can use third parties languages to describe services and their publishers, examples involving a subset of the UDDI data structures will be vised.

3.1 Architectural overview

Scalability, modularity, extensibility and interoperability with third parties provided service/business descriptive languages are requirements of the Federation System. Moveover, the system must be able to operate on top of middleware layers which rely on different integration-oriented platforms, such as the *Web Services Stack* (WSS) [20, 47].

Conforming with the extended Service Oriented Architecture (SOA) approach [68], the Federation System high-level functionalities are composed by underpinning interactions among the services provided by the SuperPeer and the ones exposed by the various Peers. The SuperPeer and the Peers are applications operating a set of *Application Program Interfaces* (APIs). They communicate with one another by invoking APIs' operations on the corresponding services. The invocation of API's operations is based on exchanging of documents encoded in a XML vocabulary, called *Federation Description Language* (FDL), which is introduced in Section 3.5.

The publishing and subscription functionalities are provided by the *SuperPeer*, which exposes two Web Services called *Inquire Service* and *Publication Service*. The former provides the centralized service retrieving capabilities, whereas the latter sums up all the registering facilities and service publication/subscription management features. In order to access the Federation System's functionalities, the participants have to operate a client software module, named *Peer*, which exposes a pair of Web Services: the *Public Service*, carrying out the direct interactions with the other Peers, and the *Notification Service*, which receives the event-notifications the SuperPeer dispatches.

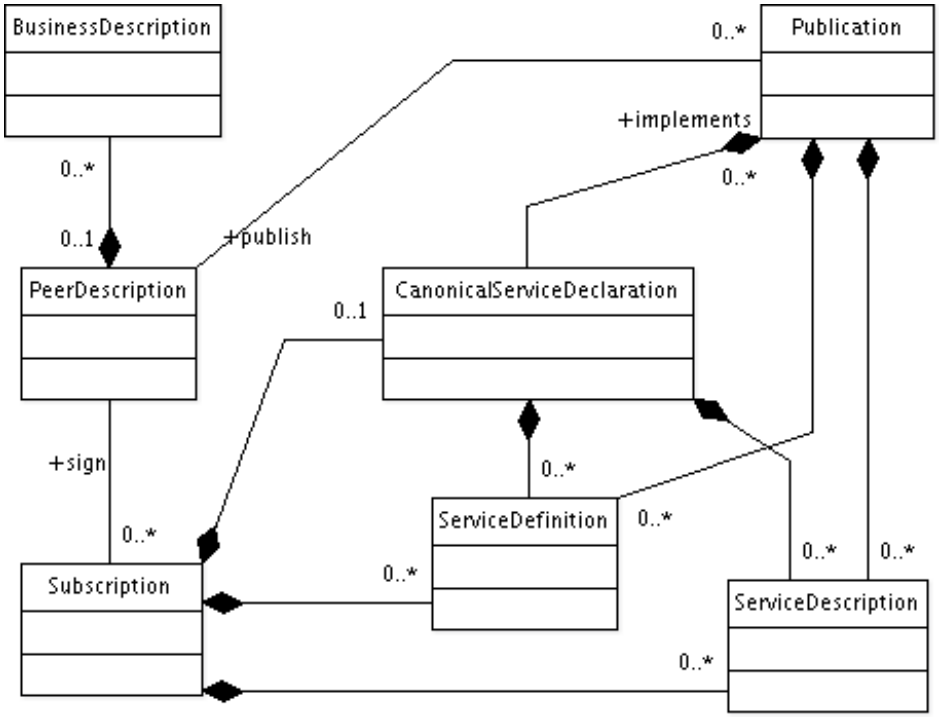
Since the SuperPeer application dispatches event notifications by invoking methods on to the Notification Services belonging to the target Peers, the former needs to know the *endpoints* of that services. An endpoint is the network address of the corresponding Web Service expressed as an *Universal Resource Locator* (URL). The SuperPeer gains information about the Peers upon their registrations: the data submitted during the request of registration comprise the Notification Service's and Publish Service's endpoints, the name the registering

user would like to adopt, contact and business information and so on.

The main definitions tied to the elements of the Federation System, which hereby will be called *concepts*, are the following:

- *csd*: represents a Canonical Service Declaration;
- *peerDescription*: describes a Peer;
- *publication*: represents a Web Service published within the Federation;
- *subscription*: vies a Peer's Subscription;
- *businessDescription*: contains descriptive information about a Peer, such as descriptions in multiple languages, contact and classification information;
- *serviceDefinition*: consists of the technical definition of a Service;
- *serviceDescription*: represents a logical service classification, made in terms of an industrial categorization or a peculiar family of technical services, based on standard taxonomies.

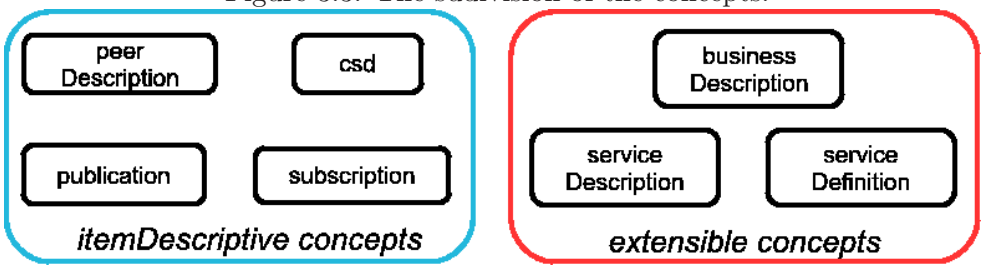
Figure 3.2: The concept's hierarchy.



Certain concepts include others. In this sense, there is an hierarchy among the concepts, presented in Figure 3.2 in the form of a Class Diagram. The arrows with the blank head represent the reference relation correlating two different item-descriptive concepts, whereas the arrows with the black head show the inclusion relation associating an item-descriptive concept and an extensible

one. In order to express the multiplicity of the relations, we adopt a notation similar to the UML's one. Next to the arrows' heads and tails are reported the multiplicities characterizing the relations: for instance, each csd may be referred by any number of publications, whereas each subscription refers to exactly one peerDescription.

Figure 3.3: The sudivision of the concepts.



The concepts listed so far can be parted into two, as shown in Figure 3.3: the first one is made of the csd, peerDescription, publication and subscription concepts, which we call *item-descriptive*, whereas the second contains the *extensible* concepts businessDescription, serviceDefinition and serviceDescription. The item-descriptive elements describe the basic entities populating the Federation System from an abstract point of view: the Peers, the Canonical Service Declarations, the published Web Services and the signed Subscriptions. The extensible elements allow expressing descriptive information about csds, peerDescriptions, publications and subscriptions according to third parties languages, such as a subset of the UDDI Datastructure Schema[60]. This realizes the requirement of having the Federation System using third parties languages to describe services, by the means of serviceDefinitions and serviceDescriptions, and their publishers, through businessDescriptions.

The *csd* concept represents the concept of Canonical Service Declaration, the latter of which is a sort of prototype of the Web Services suitable for publication within the Federation. The csd concept is made of the following items:

- *csdIdentifier*: uniquely identifies the csd;
- *serviceDefinition*: describes technical requirements of the Canonical Service Declaration. Each csd can have any number of serviceDefinitions, which may describe different types of requirements (service's interface, Quality of Service, and so on), or maybe the same requirements in equivalent languages;
- *serviceDescription*: contain service descriptive information about the Canonical Service Declaration, such as logical service classification, made in terms of an industrial categorization or a peculiar family of technical services, based on standard taxonomies. Canonical Service Declarations might contain none or several serviceDescriptions.

Each Federation can bear any number of Canonical Service Declaration which fits the needings.

A *peerDescription* describes the respective Peer by the means of the following items:

- *peerIdentifier*: it identifies univocally the respective Peer among all the others and it is assigned by the SuperPeer upon the Peer's registration;
- *notificationServiceURL*: it is the endpoint of the Peer's Web Service providing the Notification API functionalities;
- *publicServiceURL*: it is the endpoint of the Peer's Web Service implementing the Public API functionalities;
- *businessDescription*: it is a set of business-descriptive data describing the Peer, such as contact information belonging to the business actor operating the Peer. Any number of businessDescriptions can be specified.

A peerDescription does not contain any information about neither the publications its respective Peer owns, nor the subscriptions it has signed. On the other hand, since the peerDescriptions contain the endpoints of the Peers' Public and Notification Web Services, they retain all the information required to interact with the respective Peers. The businessDescriptions allow Peers to express their business-descriptive data according to various different formats, or even in natural language, as well as specifying different documents containing different information.

The Federation System represents offered Web Services through the publications. The publications are published by the Peers aiming at providing Web Services. Each publication has to *inherit* from one of the csds mandated within the Federation, which means that all information contained within the csd has to apply to the inheriting publications as well. Publishers are in charge of making sure that the Web Services represented by publications suite the requirements specified in the inherited csds. The *publication* concept is built upon the following items:

- publicationIdentifier: an identifier which is distinctive of the publication among all the others
- csdIdentifier: the csdIdentifier of the inherited csd
- peerIdentifier: the publisher's peerIdentifier
- serviceDefinition: additional technical-related data
- serviceDescription: additional service-descriptive data

There are no constraints on how many serviceDefinitions and serviceDescriptions publications can contain. *Subscriptions* allow Peers to be notified of the modifications involving publications which match predefined requisites. Since subscriptions are meant to keep track of the publications within the Federation, they are defined similarly to the way publications are specified. When

signing a subscription, Peers can specify a csd (through its csdIdentifier) as well as additional serviceDefinitions and serviceDescriptions. In order to match a given subscription, a publication has to comply to all the requirements the former specifies. Signing a subscription referring to a given csd and containing additional serviceDefinitions and serviceDescriptions comes in hand whenever the subscriber desires to keep track of a subset of the publications inheriting from that csd. For instance, let us consider a csd prototyping hotel booking services without defining a geographical area in which the hotels are placed. However a subscriber might be interested only in hotel booking services within a given geographical region, thus it might sign a subscription referring to the hotel booking csd and specify an additional serviceDescription containing taxonomic entries which identify the region of interest. The subscription concept is made of the following items:

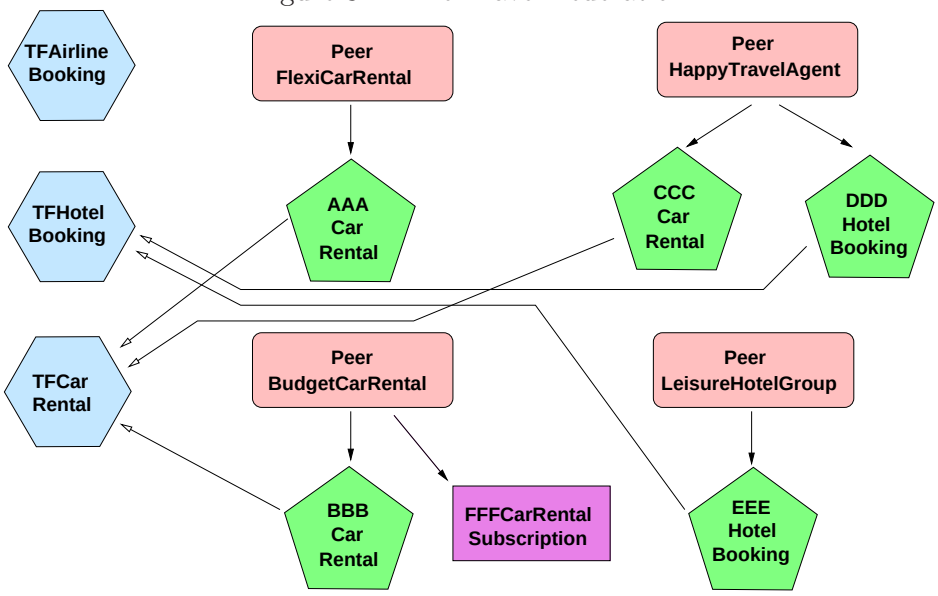
- subscriptionIdentifier: a unique identifier which is distinctive of each subscription
- peerIdentifier: the peerIdentifier of the Peer which has signed the subscription
- csdIdentifier: the csdIdentifier of the csd matching publications have to inherit from
- serviceDefinition: additional technical-related data
- serviceDescription: additional service-descriptive data

Often through paper, instances of the same concept will be matched. The comparison between instances of the same item-descriptive concept is rather intuitive: in fact, for instance, two different peerDescription are said to be equal if all their fields (peerIdentifier, notificationServiceURL and publicServiceURL) bear the same value, and the nested concepts' instances are respectively equivalent. Differently, the extensible concepts are not given of a fixed syntax, as they contain documents encoded into third parties languages that are not know a priori. Moreover, different instances of the same extensible concept may contain very different kind of information, and thus being not directly comparable: for instance, consider two instances of serviceDefinition, one consisting of a WSDL document describing a service interface, and another one made of a OWL-S [50, 51] document describing the semantics of the service's operations. Nonetheless, during an overview of the implementation of the FS, presented in Section 4, we are presenting a solution addressing the comparison of extensible concepts' instances.

3.2 The Travel Federation

The item-descriptive and extensible concepts are the building blocks upon which the API the SuperPeer and Peer modules provide are built. In order to exemplify the functionalities provided by the APIs, we introduce a Federation scenario focused on an imaginary federation named Travel Federation. The

Figure 3.4: The Travel Federation.



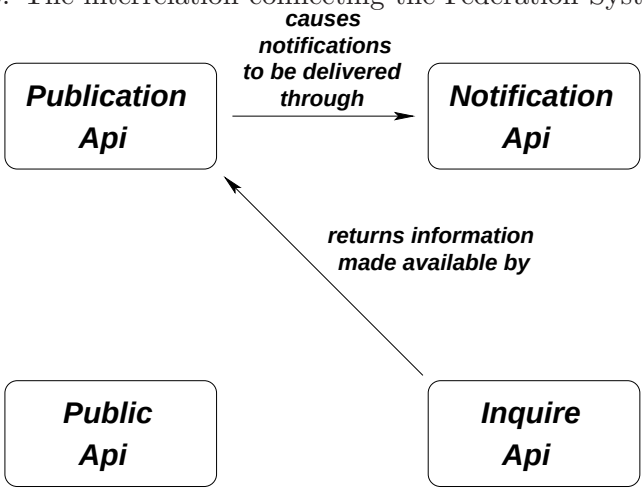
state of the Travel Federation is shown in Figure 3.4. The Travel Federation is populated by the Peers BudgetCarRental, FlexiCarRental, HappyTravelAgent and LeisureHotelGroup, and it provides support for Web Services searching for availability and booking reservation in the airline, hotel and car rental industry. For each one of those activities, the Travel Federation provides a Canonical Service Description, respectively named TFAirlineBooking, TFHotelBooking and TFCarRental. Each Travel Federation's CSD contains a WSDL document describing the prototypical interface the inheriting publications have to expose, as well as a set of industrial taxonomy's entries providing service categorization. In Figure 3.4, the CSDs are presented as hexagons (i.e., TFCarRental), the subscriptions as rectangles (i.e., FFFCarRentalSubscription), the publications are reported as pentagons (i.e., AAACarRental) and the Peers are shaped as rectangles with rounded corners. Some example of the relations among Peers, CSDs, publications and subscriptions are the following: the FlexiCarRental Peer owns the publication named AAACarRental which inherits from the TFCarRental CSD whereas the Peer HappyTravelAgent has both the publication CCCCarRental inheriting from the same CSD, and the publication DDDHotelBooking inheriting from the TFHotelBooking CSD. The subscription FFFCarRental is owned by the Peer BudgetCarRental. The TFAirlineBooking CSD is not inherited by any publication.

3.3 The Federation System's APIs

The Federation System's APIs are realized by the *Inquire*, *Publication Notification* and *Public* services. Their interrelations are shown in Figure 3.5. In particular, invocations of the operations comprised in the Publication API may cause delivery of notifications through the Notification API. Moreover, the In-

quire API operates the retrieval of the results working on the information made available through the Publication API. The Publication, Inquire and Notification API are correlated as they cooperate in realizing the centralized part of the system, which is orthogonal with respect to the distributed part of the system relying on the Public API; this actually explains why the Public API is not connected with relations to the other APIs.

Figure 3.5: The interrelation connecting the Federation System's APIs.



Hereby we introduce the four different APIs with an insight view on the operations they are made of.

3.3.1 The Publication API

The Publication API provides the Peers with the functionalities of joining and leaving a Federation, as well as publishing, updating and removing the shared Web Services and allows the Peers to sign and dismiss subscriptions to the SuperPeer.

The Publication's operations, which are listed in Table 3.1, are designed according to the request-response approach: when a Peer invokes a publication operation on the SuperPeer, a report stating the resulting success or the failure is returned. The Publication API's operations are divided into three subsets according to the type of data the operations handle. The first one, comprising the PeerRegister, PeerUpdate and PeerResign operations, allow Peers to join a Federation by submitting its peer-descriptive data (that is the endpoints of its Public and Notification services, and eventually business-descriptive information), modifying their data and leaving their Federation. The second subset handles changes to the Federation's data concerning published Web Services: it is made the PublicationSubmission, PublicationUpdate and PublicationRemoval, which respectively address the Peers' publication, updating and removal of information about Web Services. The third group is made of the SubscriptionSubmission and SubscriptionRemoval operations: they make invoking Peers signing and withdrawing subscription on the SuperPeer.

Table 3.1: The Publication API.

Operation Name	**Achievement**
PeerRegister	The invoker Peer joins the Federation by submitting its peerDescription
PeerUpdate	Updates the invoker's peerDescription
PeerResign	The invoking Peer resigns from the Federation, and its peerDescription and its publications are removed
PublicationSubmission	Makes available a publication
PublicationUpdate	Allows publishing Peers to update publications' service-describing data
PublicationRemoval	Removes a publication
SubscriptionSubmission	The invoker Peer signs a subscription
SubscriptionRemoval	Allows Peers to withdraw a previously signed subscriptions

The three different operations' subsets require the invoking Peer to submit to the SuperPeer different types of data. The operations involved in managing the Peer's registration to the Federation require the invoker to submit a peerDescription. For instance, if another Peer, operated by an airline travel agency, wants to join the Travel Federation. It prepares a peerDescription containing its proposal for peerIdentifier (AirlineTravel), the URLs of its Public and Notification Web Services, and two documents, one encoded according to the UDDI datastructure and the other one rely one the ebXML specifications, containing a set of contacts and characterizes the agency as an airline services supplier according to the NAICS taxonomy. The AirlineTravel Peer invokes the PeerRegister operation on the SuperPeer and submits its peerDescription. The SuperPeer firstly checks if the peerIdentifier has not already been assigned to another Peer in the Federation, and thereafter checks if the URLs of the Web Services are present and well formed. If the proposed peerIdentifier was not suitable, the SuperPeer would have provided one. If the notificationServiceURL or the publicServiceURL were not present or well formed, the request of registration would have been rejected and the invocation of the PeerRegister would have resulted in an error. At this point the registration is accepted. The SuperPeer stores the businessDescription into its Registry, and then it generates the Peer's *credentials*, which is the equivalent of a password for accounts. Then the authenticationDataSet is returned to the AirlineTravel Peer, completing the registration.

The pair made of peerIdentifier and credentials of a Peer is called *authenticationDataSet*. The Peer is expected to submit its authenticationDataSet to the SuperPeer every time it invokes Publication API's operation but the Peer-

Register one. This provides a trivial security mechanism, which is meant to be improved in the further specifications of the Federation System.

The PeerUpdating operation allows a Peer to change some of the data contained within its peerDescription. The peerIdentifier is fixed upon the Peer registration, thus the data that can be modified are the endpoints of the Public and Notification Web Services and the businessDescriptions. The PeerResign operation allows Peers to leave the Federation. When a Peer leaves the Federation, its peerDescription and its publications are removed, too. Peers that have signed a subscription matching a publication removed this way are notified of the occurrence.

Peers can publish, update and remove their publications through the operations PublicationSubmission, PublicationUpdate and PublicationRemoval allow the . Let us consider again the Travel Federation (Figure 3.4). Suppose the Peer LeisureHotelGroup wants to keep track of the evolution of the publications about car rental services in order to compose them with its hotel-booking service and offer to its customers a car in use for the time they lodge. It may use the Inquire API of the SuperPeer in order to retrieve at a given time all the publications inheriting from the TFCarRental CSD. However, it would only have a snapshot of the publications offered by the time it inquires the SuperPeer. If cheaper or better services were provided after the inquiry, then the Peer LeisureHotelGroup would not be aware of that. Provided this, in order to adapt its business proposals to the car rental market's changes, the Peer LeisureHotelGroup signs a subscription containing as requirements the TFCarRental csd and a serviceDescription defining the geographical location of the car rental agencies . Let us suppose that the publications AAACarRental and CCCCarRental, belonging respectively to the Peers FlexiCarRental and HappyTravelAgent, have a serviceDescription claiming they are located in Rome, whereas the publication BBBCarRental, which is located in Milan, does not. Upon the invocation of the operation SubscriptionSubmission, passing its subscription as argument, the Peer LeisureHotelGroup is returned with the subscriptionIdentifier, might it be LeisureSubscription, which uniquely identifier its signed subscription. Moreover, through the Notification API, presented in Section 3.3.3, the Peer is notified of the publications AAACarRental and CCCCarRental, which fit all the subscription's requirements. Suppose now that the Peer BudgetCarRental, the publication BBBCarRental of which did not previously fit the LeisureSubscription subscription due to the unsatisfactory geographical location, moves its car rental agency from Milan to Rome. In order to reflect the changes in its service, the Peer BudgetCarRental updates its BBBCarRental publication by modifying the serviceDeclaration stating the geographical location. Since after the updating the BBBCarRental publication actually fits the requirements of the LeisureSubscription subscription, the Peer LeisureHotelGroup is notified of the existence of the newly matching publication.

Table 3.2: The Inquire API.

Operation Name	**Achievement**
FindCSDs	Allows Peers inquiry the SuperPeer for the Canonical Service Declarations published within the Federation
FindPublications	Retrieve publications matching the specified criteria
FindPeerDescriptions	Retrieve information about the Peers that match the criteria

3.3.2 The Inquire API

The Inquire API, summarized in Table 3.2, realizes the Federation System's centralized service discovery feature. Peers inquiry the SuperPeer in order to retrieve csds, publications or peerDescriptions matching the specified criteria, and they are returned with the matching outcome in a request-response manner. According to the different kind of needed information, the Peers are expected to submit queries shaped in different ways. The queries are matched by the SuperPeer against its database, which has a complete knowledge of the whole Federation, the published publications and the participating Peers.

Peers submit documents of the same kind of those they are looking for, filled with the fields that have to match the ones contained within the results. For instance, if the Peer HappyTravelAgent wants to retrieve the peerDescriptions containing a two given businessDescriptions, it invokes the FindPeerDescriptions by passing a document containing a peerDescription comprising that businessDescriptions. The Peer would receive all the peerDescriptions containing at least those businessDescriptions; naturally, peerDescriptions containing those businessDescriptions together with others would be returned as well. In fact, the Inquire API's matching mechanism is based on the concept of minimal set of data: the inquirer specifies its requirements as minimal bunch of information that have to be present in the results, and it is returned with all the documents containing at least them. A particular case is the one where the document submitted to the SuperPeer does not contain any information at all: let us suppose that a Peer wants to find every single csd available in the current Federation, then it has to inquire the SuperPeer with an empty csd, without any information in it. Since every document matches an empty one, then all the csds would be suitable and returned as results.

3.3.3 The Notification API

The Notification API allows the SuperPeer to deliver the notification regarding changes in the publications to the Peers which have signed matching subscriptions. The Notification API is implemented by the Notification Web Services exposed by the Peers. Upon any modification made through the Publication API and involving a publication, the SuperPeer matches the latter against the

Table 3.3: The Notification API.

Operation Name	**Achievement**
PublicationSubmissionNotification	Notifies target Peers the availability of a newly published publication
PublicationUpdateNotification	Notifies target Peers the updating of an already existing publication
PublicationRemovalNotification	Notifies target Peers the removal of a publication

Table 3.4: The Public API.

Operation Name	**Achievement**
DistributedFindPublicationsQuery	Submit a PublicationQuery to the acquainted Peer
DistributedFindPublicationsResult	Submit a PublicationResult to the originating Peer

subscription by the Peers. Whenever a matching is detected, the SuperPeer delivers to the Peer which have signed the subscription a notification containing the publication and the publisher's peerDescription. The three types of modifications that occur to a publication are its newly availability, an updating of the data it comprises, and its removal. The respective notification's types are ServicePublicationNotification, ServiceUpdateNotification and ServiceRemovalNotification.

When delivering notifications through the Notification API, the SuperPeer is not returned with documents by the notified Peers. Unlikely the Inquire and Publication API, that are designed in a request/response manner, the Notification API is designed to be based on messagging facilities, enabling distributed communication that is loosely coupled.

3.3.4 The Public API

Differently from the centralized discovery mechanism, which allows Peers to find Canonical Service Declarations, publications and other Peers, the distributed one is focused only on retrieving publications. The documents containing the criteria that matching publications have to satisfy are called *publication queries*. The intuition beneath the whole mechanism is that, to retrieve the matching publications, the Peer which is looking for publications, named *originating Peer*, spreads publication queries among its equivalents, named *processing Peers*, instead of inquiring the centralized SuperPeer. Processing Peers return matching publications to the originating one through the delivery of *publication results*. The distributed service discovery mechanism relies on the operations provided by the Public API. Similarly to the operations belonging to the Notification

API, the Public API's operations DistributedFindPublicationsQuery and DistributedFindPublicationsResult are message-oriented.

From a point of view of their internal structure, publication queries are similar to the subscriptions. Publication queries are made of one of the Federation's CSD (by the means of the relative csdIdentifier) and, eventually, additional serviceDefinitions and serviceDescriptions. In order to match a publication query, a publication has both to inherit from the specified CSD and also to comply to the additional serviceDefinitions and serviceDescriptions, if these are present. In the same way subscriptions have their distinctive subscriptionIdentifiers, publication queries are univocally identified by their publicationQueryIdentifiers. A structural difference between subscriptions and publication queries is that the latter contain in addition the originating Peer's peerDescription, which contains the endpoint of the Public service to which publication results are dispatched. This is due to the fact that every Peer processing a query must be able to deliver the found matching publications to the originating Peer; this means that the processing Peer needs to know the endpoint of the originating one's Public service. A publication query is routed from a Peer to another through and invocation of the DistributedFindPublicationsQuery operation performed by the former on the Public service exposed by the latter.

A publication result is a document containing the processing Peer's peerDescription together with the matching publications the latter owns. To return the matching publications found, the processing Peers forge publication results and then deliver the latter to the originating Peer through an invocation of the DistributedFindPublicationsResult operation on the Public service exposed by the latter.

The distributed discovery mechanism is based upon the concept of *Peer Acquaintance Group* (PAG). A Peer's PAG is the set of all the Peers the PAG's owner is aware of. Peers comprised within a PAG are said to be *acquainted peers* of the PAG's owner, whereas the relationship associating a Peer and its acquainted Peers is called *acquaintance relation*. The acquaintance relation is:

- irreflexive: a Peer does not include itself in its PAG;
- not symmetric: it is not granted that two Peers are always mutually acquainted to each other;
- not transitive: let A, B and C, be three Peers, if B is an acquainted Peer of A, and C is an acquainted Peer of B, it is not necessarilt the case that C is an acquainted Peer of A.

Each Peer in a Federation is likely to know only the other Peers matching its own subscription needs; however, by inquiring the SuperPeer for publications through the Inquiry API, as well as being given publication queries to process, Peers may know any other Peer registered to the same Federation. That is, each Peer contains the descriptions of its published service together with data about a subset of all the Peers participating in its Federation.

Once a processing Peer is given a publication query, it performs the following four steps:

Table 3.5: An example of Peers and PAGs.

Peer	**PAG**
FlexiCarRental	HappyTravelAgent, BudgetCarRental
HappyTravelAgent	BudgetCarRental, LeisureHotelGroup
BudgetCarRental	FlexiCarRental, HappyTravelAgent
LeisureHotelGroup	BudgetCarRental
AirlineTravel	HotelBookingOnline, Rent-O-Nline
HotelBookingOnline	AirlineTravel
Rent-O-Nline	HotelBookingOnline, YourCarRental
YourCarRental	

1. *publication query booking*: checks the publicationQueryIdentifier distinctive of the publication query, if it has already been processed a query with the same publicationQueryIdentifier, then the publication query is discarded and the process ends; otherwise the publicationQueryIdentifier is stored and the algorithm proceeds;

2. *publication matching*: matches the publication query it against its publications;

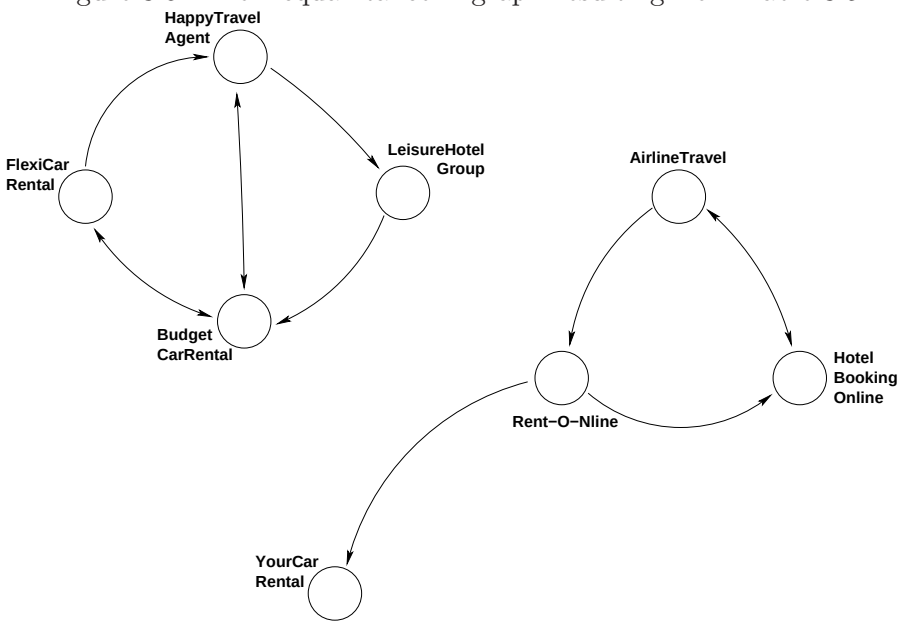
3. *publication results delivery*: forges the publication result and delivers it to the originating Peer;

4. *publication query routing*: routes the query unchanged towards all its acquainted Peers.

Through subsequent routings, an originating Peer spreads its publication query over all the other consecutive processing Peers, while two nodes of a graph are said to be consecutive if there exists an edge linking the first with the second. We assume that during the retrieving of the results for a the query, neither new matching publications are published, nor the Peers' acquaintance groups are modified. Let us consider a digraph, the nodes of which are the registered Peers: a digraph is a set of vertices V and set of ordered pairs (a,b) (where a, b are in V) called edges. The vertex a is the initial vertex of the edge and b the terminal vertex. Let the acquaintance relationship be represented on the acquaintance digraph as directed edges linking the corresponding nodes: the owner of the PAG is the initial vertex of the edges, whereas its acquainted Peers comprised within the former are the terminal vertexes. We name such a graph *acquaintance digraph*. Since two Peers may be mutually acquainted with each others, the acquaintance digraph admits bidirected edges. Thus, the publication query generated by a given node is spread over the acquaintance digraph according to the paths which link the generating node with the other ones it is connected with.

Let us consider an example based upon the Travel Federation scenario enriched with the additional Peers AirlineTravel, HotelBookingOnline, Rent-O-Nline and YourCarRental. The Peers and their respective PAGs are reported in Table 3.5, whereas the acquaintance digraph is presented in Figure 3.6. The

Figure 3.6: The Acquaintance Digraph resulting from Table 3.5.



digraph's edges are directed, and an edge starting from a node and ending in a second means that the second node is comprised in the first one's PAG. In the above scenario, we highlight three different sets of connected nodes:

- FlexiCarRental, HappyTravelAgent, BudgetCarRental and LeisureHotel-Group
- AirlineTravel, HotelBookingOnline, Rent-O-Nline
- YourCarRental

The node YourCarRental is in a set of its own, because it has the incoming edge starting from node Rent-O-Nline, but no outgoing edge or path going to.

The query booking step of the publication query processing algorithm presented above avoids infinite loops in publication query routing. The simplest scenario in which we find such an issue is the case in which two Peers are mutually acquainted, that means they are comprised within a connected component. If the processing of a publication query is allowed more than once, the Peers would keep on routing the publication query to each other infinitely often. In fact, the query booking step guarantees that each Peer in a Federation processes a given publication query at most once. Let us consider the acquaintance digraph presented in Figure 3.6, and let us focus on the connected component made of the nodes AirlineTravel, HotelBookingOnline, Rent-O-Nline. If a publication query originated in the node AirlineTravel, it would be initially routed to both the nodes HotelBookingOnline and Rent-O-Nline. At the second step, node Rent-O-Nline would route the publication query to the node HotelBookingOnline, which at the same time would forward the query to the AirlineTravel node. It can be easily seen that this would form an infinite loop,

in which the same query would be processed infinite times by any of the nodes comprised within the connected component.

3.4 The Underpinning Mathematical Framework

The Federation system relies on a number of fundamental operations for matching, managing publications and providing notification. Next, we provide a formal treatment of these concepts based on the following points:

1. mathematical representation of the Federation System's concepts
2. definition of the *projection* operators
3. definition of the *matching* operators
4. definition of the *federation-level* operators

In Figure 3.7 we summarize the basic notation we use for describing concepts. In particular we report the domain of each concept, where a domain is a set containing all the possible different instances of a certain concept. For instance, all the possible instances of the concept *csd* are comprised within the domain $\mathbb{CSDS}$.

Figure 3.7: Concepts' Metavariables and Domains.

CONCEPT	METAVAR.	DOMAIN	INTUITIVE MEANING
csd	csd	$\mathbb{CSDS}$	prototype of publications
peerDescription	pdesc	$\mathbb{PDESCS}$	representation of a Peer
publication	pub	$\mathbb{PUBS}$	representaton of a published service
subscription	sub	$\mathbb{SUBS}$	representation of a subscription for services
businessDescription	bdesc	$\mathbb{BDESCS}$	container for white-pages information
serviceDefinition	sdef	$\mathbb{SDEFS}$	container for yellow-pages information
serviceDescription	sdesc	$\mathbb{SDESCS}$	container for green-pages information

The item-descriptive concepts (Section 3.1) contain identifiers univocally identifying their instances. Given a Federation, at any time all instances of the same concept have different identifiers. This constraint does not force instances

of different concept to have different identifiers: it may happen that, for instance, a peerDescription and a publication share the same identifier. However, even though the context always clarifies the kind of concept the identified instance belongs to, from a mathematical point of view it is preferable to impose disjointed sets of identifiers for the different concept's domains. In Figure 3.8 the sets of identifiers are listed, the adopted metavariables and the domains in which they are used, which we call *associated domains*: for instance, the set of identifiers $ID_{\mathbb{PUBS}}$ identify instances of the concept publications (in fact the associated domain is $\mathbb{PUBS}$), and they will usually be represented with the metavariable id_{pub}.

Figure 3.8: Identifiers for the Concepts' Instances.

IDENTIFIERS' SETS	**METAVAR.**	**ASSOCIATED DOMAIN**
$\mathbb{ID}_{\mathbb{CSDS}}$	id_{csd}	$\mathbb{CSDS}$
$\mathbb{ID}_{\mathbb{PDESCS}}$	id_{pdesc}	$\mathbb{PDESCS}$
$\mathbb{ID}_{\mathbb{PUBS}}$	id_{pub}	$\mathbb{PUBS}$
$\mathbb{ID}_{\mathbb{SUBS}}$	id_{subs}	$\mathbb{SUBS}$

The item-descriptive concepts are formalized as tuples. Each tuple is made of an identifier and other inner elements. The formalization is presented in Figure 3.9. For instance, let us consider the formalization of the csd concept:

$$\mathrm{csd}' := (\mathrm{id}_{\mathrm{csd}} \in \mathbb{ID}_{\mathbb{CSDS}}, \mathbb{SDESCS}_{\mathrm{csd}} \subseteq \mathbb{SDESCS}, \mathbb{SDEFS}_{\mathrm{csd}} \subseteq \mathbb{SDEFS})$$

The csd csd' is a tuple made of the identifier $\mathrm{id}_{\mathrm{csd}}$ belonging to the $\mathbb{ID}_{\mathbb{CSDS}}$, the subset $\mathbb{SDESCS}_{\mathrm{csd}}$ of $\mathbb{SDESCS}$ and the subset $\mathbb{SDEFS}_{\mathrm{csd}}$ of $\mathbb{SDEFS}$. Equivalently, the domain $\mathbb{CSDS}$ is defined in the following way:

$$\mathbb{CSDS} := \mathbb{ID}_{\mathbb{CSDS}} \times 2^{\mathbb{SDESCS}} \times 2^{\mathbb{SDEFS}}$$

With respect to the first structure of the item-descriptive concepts, the concepts csd, publication and subscription are the straightforward set-based mathematical formalization. However, the same does not hold in the case of the peerDescription concept. The peerDescription concept has been previously structured in terms of a peerIdentifier, a notificationServiceURL, a publicServiceURL and a set of businessDescriptions. The mathematical formalization we have hereby provided drops the notificationServiceURL and the publicServiceURL elements. The notificationServiceURL and the publicServiceURL are not relevant for the

Figure 3.9: Concept Definitions.

CONCEPT	FORMALIZATION
peerDescription	$\text{pdesc} := (\text{id}_{\text{pdesc}} \in \mathbb{ID}_{\mathbb{PDESCS}}, \mathbb{BDESCS}_{\text{pdesc}} \subseteq \mathbb{BDESCS}, \text{PAG}_{\text{pdesc}} \subseteq (\mathbb{ID}_{\mathbb{PDESCS}} \backslash \{\text{id}_{\text{pdesc}}\}))$
csd	$\text{csd} := (\text{id}_{\text{csd}} \in \mathbb{ID}_{\mathbb{CSDS}}, \mathbb{SDESCS}_{\text{csd}} \subseteq \mathbb{SDESCS}, \mathbb{SDEFS}_{\text{csd}} \subseteq \mathbb{SDEFS})$
publication	$\text{pub} := (\text{id}_{\text{pub}} \in \mathbb{ID}_{\mathbb{PUBS}}, \text{id}_{\text{pdesc}} \in \mathbb{ID}_{\mathbb{PDESCS}}, \text{id}_{\text{csd}} \in \mathbb{ID}_{\mathbb{CSDS}}, \mathbb{SDESCS}_{\text{csd}} \subseteq \mathbb{SDESCS}, \mathbb{SDEFS}_{\text{csd}} \subseteq \mathbb{SDEFS})$
subscription	$\text{sub} := (\text{id}_{\text{sub}} \in \mathbb{ID}_{\mathbb{SUBS}}, \text{id}_{\text{pdesc}} \in \mathbb{ID}_{\mathbb{PDESCS}}, \text{id}_{\text{csd}} \in \mathbb{ID}_{\mathbb{CSDS}}, \mathbb{SDESCS}_{\text{csd}} \subseteq \mathbb{SDESCS}, \mathbb{SDEFS}_{\text{csd}} \subseteq \mathbb{SDEFS})$

mathematical formalization. Moreover, the formalization of the peerDescription contains the representation $\text{PAG}_{\text{pdesc}}$ of the PAG. The Peer Acquaintance Group is of interest only to the peer represented by a the peerDescription, however including it in the formal representation greatlys help when formalizing the distributed service discovery mechanism.

The extensible elements businessDescription, serviceDescription and serviceDefinition have been taken into consideration when defining the concept's domains; although a formal definition of the instances of such concepts has not been provided. This is due to the fact that the Federation System does not mandate any syntax to encode such data, whereas it aims at relying on languages provided by third parties. This means that in this mathematical formalization we are not going to make any assumption on the actual structure of the extensible elements but we rather assume they can be data-wise compared one with each others. The way such comparison is carried out will be specified more in detail when the matching operators are introduced.

From an abstract point of view, we can describe a Federation (that is, a running instance of the Federation System) by the information and data it holds. That is, we have to specify its registered peers, the available publications, the published canonical service declarations and the signed subscriptions. According to this principle, the formalization of the Federations is the one shown in Figure 3.10: the Federation fed is a tuple made of the subset $\mathbb{CSDS}_{\text{fed}}$ of $\mathbb{CSDS}$, the subset $\mathbb{PDESCS}_{\text{fed}}$ of $\mathbb{PDESCS}$, the subset $\mathbb{PUBS}_{\text{fed}}$ of $\mathbb{PUBS}$ and the subset $\mathbb{PUBS}_{\text{fed}}$ of $\mathbb{PUBS}$. The set of all the data known by the Federation fed, namely $\mathbb{CSDS}_{\text{fed}} \cup \mathbb{PDESCS}_{\text{fed}} \cup \mathbb{PUBS}_{\text{fed}} \cup \mathbb{SUBS}_{\text{fed}}$, is called federation's *knowledge set*. When modelling a real instance of the Federation System, the federation's knowledge set evolves together with the operations carried out within the Federation, and in particular with the operations belonging to the Publication API. For instance, when a peer publishes a new publication pub, the set $\mathbb{PUBS}_{\text{fed}}$ is

Figure 3.10: The Federation's Knowledge Set.

fed	:=	$(\mathbb{CSDS}_{\text{fed}}, \mathbb{PDESCS}_{\text{fed}}, \mathbb{PUBS}_{\text{fed}}, \mathbb{SUBS}_{\text{fed}})$
$\mathbb{CSDS}_{\text{fed}}$	$\subseteq$	$\mathbb{CSDS}$
$\mathbb{PDESCS}_{\text{fed}}$	$\subseteq$	$\mathbb{PDESCS}$
$\mathbb{PUBS}_{\text{fed}}$	$\subseteq$	$\mathbb{PUBS}$
$\mathbb{SUBS}_{\text{fed}}$	$\subseteq$	$\mathbb{SUBS}$

accordingly updated. From a formal point of view, the federation's knowledge evolves through federation-level operators, which model the semantics of the operation comprised in the APIs provided by the Federation System.

The *projection operators* π are defined as shown in Figure 3.11. They provide access to the inner elements of concepts. The different projection operators are described according to the different domains. The $\pi^{comp}_{domain}(instance)$ notation is equivalent to $\pi_{domain}(comp, instance)$. Please note that some projection operator might return a single entity or a set according to the projected component: for instance, given the publication pub, π^{pdesc}_{pub}(pub) returns an identifier, whereas π^{sdef}_{pub}(pub) returns a set of serviceDefinitions. Though not formally correct, it is a shortcut to avoid the definition of a lot of different projection operators with the same domain, actually one for each inner element in the given entity type.

We introduce the following notation: be A a set, and be a an element such that $a \in A$, be b an element such that $b \notin A$, if we want substitute the element a with b in the set A, we write:

$$A\,[a \backslash b] := (A \backslash \{a\}) \cup \{b\}$$

One of the main concerns of the Federation System is matching the available publications with either the signed subscriptions or the queries that are submitted through both the distributed and centralized service discovery mechanisms. The building blocks of the formalization of such procedures are the *matching operators*, which compair in Figure 3.12. The matching operators are a family of functions aiming at comparing pairs of concepts' instances. The matching operators are divided into two groups: the ones comparing item-descriptive elements with a set of requirements, which may be expressed as other item-descriptive concept's instances or extensible ones, and the other comparing extensible elements among themselves. The matching operators comparing item-descriptive elements are the following:

- $\sqsubseteq_{\text{pub}}$: for comparing a publication with a set of requirements involving which csd it inherits from and, optionally, a set of serviceDescriptions and serviceDefinitions;

Figure 3.11: Projection Operators.

$$comp \in \{id, csd, pdesc, pag, pub, subs, bsd, sdef, sdesc\}$$

$$\pi_{csd}(comp, \text{csd}) = \pi^{comp}_{csd}(\text{csd}) := \begin{cases} \text{id}_{\text{csd}} & \text{if } comp = id \\ \mathbb{SDESCS}_{\text{csd}} & \text{if } comp = sdef \\ \mathbb{SDEFS}_{\text{csd}} & \text{if } comp = sdesc \end{cases}$$

$$\pi_{pdesc}(comp, \text{pdesc}) = \pi^{comp}_{pdesc}(\text{pdesc}) := \begin{cases} \text{id}_{\text{pdesc}} & \text{if } comp = id \\ \mathbb{BDESCS}_{\text{pdesc}} & \text{if } comp = bsd \\ \text{PAG} & \text{if } comp = pag \end{cases}$$

$$\pi_{pub}(comp, \text{pub}) = \pi^{comp}_{pdesc}(\text{pub}) := \begin{cases} \text{id}_{\text{pub}} & \text{if } comp = id \\ \text{id}_{\text{pdesc}} & \text{if } comp = pdesc \\ \text{id}_{\text{csd}} & \text{if } comp = csd \\ \mathbb{SDESCS}_{\text{csd}} & \text{if } comp = sdesc \\ \mathbb{SDEFS}_{\text{csd}} & \text{if } comp = sdef \end{cases}$$

$$\pi_{subs}(comp, \text{sub}) = \pi^{comp}_{subs}(\text{sub}) := \begin{cases} \text{id}_{\text{sub}} & \text{if } comp = id \\ \text{id}_{\text{pdesc}} & \text{if } comp = pdesc \\ \text{id}_{\text{csd}} & \text{if } comp = csd \\ \mathbb{SDESCS}_{\text{csd}} & \text{if } comp = sdesc \\ \mathbb{SDEFS}_{\text{csd}} & \text{if } comp = sdef \end{cases}$$

- $\sqsubseteq_{\text{pdesc}}$: for checking whether the businessDescriptions contained within the peerDescription matches with the specified businessDescriptions;
- $\sqsubseteq_{\text{csd}}$: returns true if both the serviceDefinitions and the serviceDescriptions contained within the csd match the serviceDefinitions and the serviceDescriptions specified;
- $\sqsubseteq_{\text{subs}}$: subscriptions are sets of requirements that identify publications to be notified to subscribing peers; this operator is a shorthand matching publications against subscriptions.

The matching operators comparing item-descriptive concepts' instances are built on top of the matching operators comparing extensible concepts' instances. The extensible matching operators are:

- $\sqsubseteq_{\text{bsd}}$: matches two businessDescriptions;
- $\sqsubseteq_{\text{sdesc}}$: matches two serviceDescriptions;
- $\sqsubseteq_{\text{sdef}}$: matches two serviceDefinitions.

They way the comparison is carried out depends on what kind of data the extensible concept are meant to contain. An intuitive meaning based on sets. Consider two instances of the same extensible element, such as the businessDescription one. Let e and e' be two instances of the businessDescription concept. Both e and e' can be abstracted as sets of information. Let S_e and $S_{e'}$ be the

sets of data abstracted respectively from e and e'. Then $\sqsubseteq_{\text{bsd}}(e, e') = \top$, and we say that e matches e' if and only if

$$\sqsubseteq_{\text{bsd}}(e, e') = \top \iff (S_e \Rightarrow S_{e'}) \iff (S_e \subseteq S_{e'})$$

Roughly speaking, one extensible instances matches one another if and only if the former contains at least the same set of data than the latter, that is, if the former is a super set of information of the latter. Let us consider an example: two businessDescriptions bdesc and bdesc$'$ have to be compared, and they contain two UDDI's businessElement. For the sake of simplicity we consider only a subset of the information usually comprised within a businessElement: the categoryBag element and the description attribute. The description entry is an attribute, and therefore it is matched for equality, that is, two attributes matches if they are they have the same value. The categoryBag can be seen as a set of taxonomical entries for service categorization, and the matching is based on set's inclusion (meaning that the bdesc's categoryBag matches bdesc$'$'s one if and only if it the former contains at least the same taxonomical entries than the latter). Then bdesc matches bdesc$'$ if and only if both the description and the categoryBag contained within bdesc match with the respective entry in bdesc$'$.

The matching operators are the foundation atop of which the *federation-level* ones are built. The federation-level operators model how a Federation evolves upon the execution of an operation listed in one of the Publication, Notification and Inquire APIs by providing a formal semantics for the APIs of the System. The federation-level operators can be subdivided according to the API they refer to:

- *notification operators* modelling the Notification API
- *publication operators* referring to the Publication API
- *inquire operators* formalizing the results retrieved through the Inquire API

The *notification operator*, shown in Figure 3.13, models the delivery of notifications to the peers with respect to the signed subscriptions. They do not affect the federation's knowledge, they just express the need to notify a peer of something. From a formal point of view, they evualiate to true or to false accordingly to the success of the notification's delivery.

- PublicationSubmissionNotification(*peerDescription* pdesc, *publication* pub): notifies the peer pdesc the availability of the newly published publication pub;
- PublicationUpdateNotification(*peerDescription* pdesc, *publication* pub): notifies the peer pdesc the availability of update of the publication pub;
- PublicationRemovalNotification(*peerDescription* pdesc, *publication* pub): Notifies the peer pdesc the removal of the publication pub.

Figure 3.12: Matching Operators.

$$
\begin{array}{lcll}
\sqsubseteq_{\text{pub}} & := & \mathbb{PUBS} \times \mathbb{CSDS} \times 2^{\mathbb{SDESCS}} \times 2^{\mathbb{SDEFS}} & \rightarrow \quad \{\top, \bot\} \\
 & & (\text{pub}, \text{id}_{\text{csd}}, \mathbb{SDESCS}', \mathbb{SDEFS}') & \mapsto \quad \begin{cases} \top & \text{if } \pi^{csd}_{pub}(\text{pub}) \\ & = \pi^{id}_{csd}(\text{csd}) \ \wedge \ \forall \ \text{sdesc} \ \in \ \mathbb{SDESCS}' \\ & \exists \ \text{sdesc}' \ \in \ \pi^{sdesc}_{pub}(\text{pub}) \\ & \sqsubseteq_{\text{sdesc}} (\text{sdesc}, \text{sdesc}') \ \wedge \ \forall \ \text{sdef} \ \in \\ & \in \ \mathbb{SDEFS}' \ \exists \ \text{sdef}' \ \in \\ & \in \pi^{sdef}_{pub}(\text{pub}) \ \sqsubseteq_{\text{sdef}} (\text{sdef}, \text{sdef}') \\ \bot & \text{otherwise} \end{cases} \\
\sqsubseteq_{\text{pdesc}} & := & \mathbb{PDESCS} \times 2^{\mathbb{BDESCS}} & \rightarrow \quad \{\top, \bot\} \\
 & & (\text{pdesc}, \mathbb{BDESCS}') & \mapsto \quad \begin{cases} \top & \text{if} \ \forall \ \text{bdesc} \ \in \ \mathbb{BDESCS}' \\ & \exists \ \text{bdesc}' \ \in \\ & \in \ \pi^{bsd}_{pdesc}(\text{pdesc}) \ \sqsubseteq_{\text{bsd}} (\text{sdesc}, \text{sdesc}') \\ \bot & \text{otherwise} \end{cases} \\
\sqsubseteq_{\text{csd}} & := & \mathbb{CSDS} \times 2^{\mathbb{SDESCS}} \times 2^{\mathbb{SDEFS}} & \rightarrow \quad \{\top, \bot\} \\
 & & (\text{csd}, \mathbb{SDESCS}', \mathbb{SDEFS}') & \mapsto \quad \begin{cases} \top & \text{if} \ \forall \ \text{sdesc} \ \in \ \mathbb{SDESCS}' \\ & \exists \ \text{sdesc}' \ \in \\ & \in \ \pi^{sdesc}_{csd}(\text{csd}) \ \sqsubseteq_{\text{sdesc}} (\text{sdesc}, \text{sdesc}') \wedge \\ & \wedge \forall \ \text{sdef} \ \in \ \mathbb{SDEFS}' \exists \ \text{sdef}' \ \in \\ & \in \ \pi^{sdef}_{csd}(\text{csd}) \ \sqsubseteq_{\text{sdef}} (\text{sdef}, \text{sdef}') \\ \bot & \text{otherwise} \end{cases} \\
\sqsubseteq_{\text{subs}} & := & \mathbb{PUBS} \times \mathbb{SUBS} & \rightarrow \quad \{\top, \bot\} \\
 & & (\text{pub}, \text{sub}) & \mapsto \quad \sqsubseteq_{\text{pub}} (\text{pub}, \pi^{csd}_{subs}(\text{sub}), \pi^{sdesc}_{subs}(\text{sub}), \pi^{sdef}_{subs}(\text{sub})) \\
\sqsubseteq_{\text{bsd}} & := & \mathbb{BDESCS} \times \mathbb{BDESCS} & \rightarrow \quad \{\top, \bot\} \\
\sqsubseteq_{\text{sdesc}} & := & \mathbb{SDESCS} \times \mathbb{SDESCS} & \rightarrow \quad \{\top, \bot\} \\
\sqsubseteq_{\text{sdef}} & := & \mathbb{SDEFS} \times \mathbb{SDEFS} & \rightarrow \quad \{\top, \bot\}
\end{array}
$$

Figure 3.13: Notification Operators.

PublicationSubmissionNotification	:=	$\mathbb{PDESCS} \times \mathbb{PUBS}$	$\rightarrow$	$\{\top, \bot\}$
		$(\text{pdesc}, \text{pub})$	$\mapsto$	$\begin{cases} \top & \text{if the delivery is successful} \\ \bot & \text{otherwise} \end{cases}$
PublicationUpdateNotification	:=	$\mathbb{PDESCS} \times \mathbb{PUBS}$	$\rightarrow$	$\{\top, \bot\}$
		$(\text{pdesc}, \text{pub})$	$\mapsto$	$\begin{cases} \top & \text{if the delivery is successful} \\ \bot & \text{otherwise} \end{cases}$
PublicationRemovalNotification	:=	$\mathbb{PDESCS} \times \mathbb{PUBS}$	$\rightarrow$	$\{\top, \bot\}$
		$(\text{pdesc}, \text{pub})$	$\mapsto$	$\begin{cases} \top & \text{if the delivery is successful} \\ \bot & \text{otherwise} \end{cases}$

The *publication operators* formalize how the Publication API's operations affect a Federation. The arguments of a publication operator may have to fulfil certain requirements, expressed in the form of logical constraints.

- PeerRegister(*peerDescription* pdesc): the invoking Peer joins the Federation by submitting its peerDescription pdesc

  $$\mathbb{PDESCS}_{\text{fed}} := \mathbb{PDESCS}_{\text{fed}} \cup \{\text{pdesc}\}$$

  Before the update to $\mathbb{PDESCS}_{\text{fed}}$ is applied, the uniqueness of the identifier has to be checked. Formally, pdesc has to satisfy the following constraint:

  $$\nexists\ \text{pdesc}' \in \mathbb{PDESCS}_{\text{fed}} \ \mid\ \pi^{id}_{pdesc}(\text{pdesc}) = \pi^{id}_{pdesc}(\text{pdesc}')$$

- PeerUpdate(*peerDescription* pdesc): updates the invoker's peerDescription. Be pdesc' the previously submitted peerDescription of the updating Peer, then the set of Federation's peerDescriptions $\mathbb{PDESCS}_{\text{fed}}$ is affected as follows:

  $$\mathbb{PDESCS}_{\text{fed}} := \mathbb{PDESCS}_{\text{fed}} \left[\text{pdesc} \backslash \text{pdesc}'\right]$$

  The only inner component of the peerDescription which has to be left unchanged is the identifier. Formally, the following constraint has to be fulfilled:

  $$\pi^{id}_{pdesc}(\text{pdesc}) = \pi^{id}_{pdesc}(\text{pdesc}')$$

- PeerResign(*peerDescription* pdesc): the invoking Peer resigns from the Federation, and its peerDescription and its publications are removed. The following updates are applied to the Federation:

  $$\begin{aligned} &\mathbb{PUBS}_{\text{fed}} \quad := \\ &\quad \mathbb{PUBS}_{\text{fed}} \backslash \{\text{pub} \in \mathbb{PUBS}_{\text{fed}} \mid \pi^{id}_{pdesc}(\text{pdesc}) = \pi^{pdesc}_{pub}(\text{pub})\} \\ &\mathbb{SUBS}_{\text{fed}} \quad := \\ &\quad \mathbb{SUBS}_{\text{fed}} \backslash \{\text{sub} \in \mathbb{SUBS}_{\text{fed}} \mid \pi^{id}_{pdesc}(\text{pdesc}) = \pi^{pdesc}_{subs}(\text{sub})\} \\ &\mathbb{PDESCS}_{\text{fed}} \quad := \\ &\quad \mathbb{PDESCS}_{\text{fed}} \backslash \{\text{pdesc}\} \end{aligned}$$

Upon the removal of the publications, involved subscribing Peers are notified:

$$\forall \text{pub} \in \{\text{pub} \in \mathbb{PUBS}_{\text{fed}} \mid \pi^{id}_{pdesc}(\text{pdesc}) = \pi^{pdesc}_{pub}(\text{pub})\} \forall \text{pdesc} \in$$
$$\in \{\text{pdesc} \in \mathbb{PDESCS}_{\text{fed}} \mid \exists \text{sub} \in \mathbb{SUBS}_{\text{fed}} \,.\, \pi^{pdesc}_{subs}(\text{sub}) = \pi^{pdesc}_{pub}(\text{pub}) \wedge$$
$$\wedge \sqsubseteq_{\text{subs}} (\text{pub}, \text{sub})\} \; \mathsf{PublicationRemovalNotification}(\text{pdesc}, \text{pub})$$

- PublicationSubmission(*peerDescription* pdesc, *publication* pub): makes available the new publication pub published by the Peer pdesc. In order not to be rejected, the publication is required to inherit from an existing csd, and to have the peerIdentifier component set accordingly to the identifier of the publishing Peer. That is, the following condition have to be verified:

$$\pi^{id}_{pdesc}(\text{pdesc}) = \pi^{pdesc}_{pub}(\text{pub}) \wedge \exists \text{csd} \in \mathbb{CSDS}_{\text{fed}} \,.\, \pi^{id}_{csd}(\text{csd}) = \pi^{csd}_{pub}(\text{pub})$$

Upon the publication of pub, the Federation's knowledge is updated as follows:

$$\mathbb{PUBS}_{\text{fed}} \;:=\; \mathbb{PUBS}_{\text{fed}} \cup \{\text{pdesc}\}$$

Every Peer which have signed a subscription matching the newly available publication is notified. The set of Peers to be notified is the following:

$$\forall \text{pdesc} \in \{\text{pdesc} \in \mathbb{PDESCS} \mid \exists \text{sub} \in \mathbb{SUBS} \,.\, \pi^{pdesc}_{subs}(\text{sub}) =$$
$$\pi^{pdesc}_{pub}(\text{pub}) \wedge \sqsubseteq_{\text{subs}} (\text{pub}, \text{sub})\}$$
$$\mathsf{PublicationSubmissionNotification}(\text{pdesc}, \text{pub})$$

- PublicationUpdate(*peerDescription* pdesc, *publication* pub): allows publishing Peers to update previously published publications. Let pub$'$ be the previous version of the currently updating peerDescription, pub is requested to satisfy all the conditions for the PublicationSubmission operation; moreover, pub is supposed to inherit from the same csd of pub$'$, and of course belong to the same Peer. Formally, in order to update pub$'$ with pub, the following condition has to hold:

$$\pi^{id}_{pdesc}(\text{pdesc}) = \pi^{pdesc}_{pub}(\text{pub}) \wedge \exists \text{csd} \in \mathbb{CSDS}_{\text{fed}} \,.\, \pi^{id}_{csd}(\text{csd}) =$$
$$\pi^{csd}_{pub}(\text{pub}) \wedge \pi^{id}_{pub}(\text{pub}) = \pi^{id}_{pub}(\text{pub}') \wedge$$
$$\wedge \pi^{pdesc}_{pub}(\text{pub}) = \pi^{pdesc}_{pub}(\text{pub}') \wedge$$
$$\wedge \pi^{csd}_{pub}(\text{pub}) = \pi^{csd}_{pub}(\text{pub}')$$

If the constraints are verified, the publication pub$'$ is substituted in the set $\mathbb{PUBS}_{\text{fed}}$ with pub

$$\mathbb{PUBS}_{\text{fed}} := \mathbb{PUBS}_{\text{fed}} \left[\text{pub} \backslash \text{pub}'\right]$$

Since it is not granted that the new version of the publication matches exactly the same subscriptions the previous one did, we notify every Peer

having a subscription matching either the previous version of the publication or the updated one. The notification process is carried out as follows:

$$\forall \text{pdesc} \in \{\text{pdesc} \in \mathbb{PDESCS} \mid \exists \text{sub} \in \mathbb{SUBS} \,.\, \pi^{pdesc}_{subs}(\text{sub}) = \pi^{pdesc}_{pub}(\text{pub}) \wedge (\sqsubseteq_{\text{subs}} (\text{pub}, \text{sub}) \vee \sqsubseteq_{\text{subs}} (\text{pub}', \text{sub}))\}$$
$$\mathsf{PublicationSubmissionNotification}(\text{pdesc}, \text{pub})$$

Notice that the design choice of notifying Peers of the updating of a publication, leaves to the Peers the responsibility of determining the usefulness of the new version of the publication. Another approach which would address this issue would stand of introducing a dedicated type of notification, for instance a PublicationNoLongerMatchingNotification. We have preferred to adopt the previously explained solution to keep small the number of types of notifications; nevertheless, ongoing improvements of the Federation System would certainly involve an increase in the complexity of the FDL, which would result in an even more complex underlying mathematical framework.

- PublicationRemoval(*peerDescription* pdesc, *publication* pub): removes a publication. In order to be removed, the publication pub has to exist in the knowledge of the Federation, and it has to belong to the same Peer requesting the removal:

$$\text{pub} \in \mathbb{PUBS}_{\text{fed}} \wedge \pi^{pdesc}_{pub}(\text{pub}) = \pi^{id}_{pdesc}(\text{pdesc})$$

If the constraints are satisfied, the publication pub is removed from the set $\mathbb{PUBS}_{\text{fed}}$:

$$\mathbb{PUBS}_{\text{fed}} := \mathbb{PUBS}_{\text{fed}} \backslash \{\text{pub}\}$$

The process of notifying Peers holding matching subscription is almost identical to the PublicationSubmission process, except for the kind of notification delivered:

$$\forall \text{pdesc} \in \{\text{pdesc} \in \mathbb{PDESCS} \mid \exists \text{sub} \in \mathbb{SUBS} \,.\, \pi^{pdesc}_{subs}(\text{sub}) = \pi^{pdesc}_{pub}(\text{pub}) \wedge \sqsubseteq_{\text{subs}} (\text{pub}, \text{sub})\}$$
$$\mathsf{PublicationRemovalNotification}(\text{pdesc}, \text{pub})$$

- SubscriptionSubmission(*peerDescription* pdesc, *subscription* sub): the invoker Peer represented by pdesc signs the subscription sub. In order to have the subscription accepted, Peers have to submit subscriptions satisfying conditions similar to the ones for the publications: the subscription has to refer to an existing csd, and it has to belong to the invoking Peer. Formally:

$$\pi^{id}_{pdesc}(\text{pdesc}) = \pi^{pdesc}_{subs}(\text{sub}) \wedge \exists \text{csd} \in \mathbb{CSDS}_{\text{fed}} \,.\, \pi^{id}_{csd}(\text{csd}) = \pi^{csd}_{subs}(\text{sub})$$

The acceptance of a submitted subscription involves the update of the Federation's knowledge as follows:

$$\mathbb{SUBS}_{\text{fed}} := \mathbb{SUBS}_{\text{fed}} \cup \{\text{sub}\}$$

Moreover, the subscribing Peer is fed with all the currently available publications matching the newly signed subscription; that is, the Peer pdesc is returned with the following set of publications:

$$\{\text{pub} \in \mathbb{PUBS}_{\text{fed}} \mid \sqsubseteq_{\text{subs}} (\text{pub}, \text{sub})\}$$

- SubscriptionRemoval(*peerDescription* pdesc, *subscription* sub): allows Peers to withdraw a previously signed subscriptions. Similarly to the PublicationRemoval operation, the following condition has to hold:

$$\text{sub} \in \mathbb{SUBS}_{\text{fed}} \wedge \pi^{pdesc}_{subs}(\text{sub}) = \pi^{id}_{pdesc}(\text{pdesc})$$

The only operation to be carried out is to update the Federation's knowledge:

$$\mathbb{SUBS}_{\text{fed}} := \mathbb{SUBS}_{\text{fed}} \backslash \{\text{sub}\}$$

The *inquire operators* model how the Peers retrieve publications, peerDescriptions and csds by inquiring the SuperPeer through the Inquire API or other Peers through the Public API. The *centralized inquire operators* formalize the results retrieved through the Inquire API, whereas the *distributed inquire operators* refer to the distributed service discovery mechanism provided by the Public API.

The centralized inquire operator express the results that the SuperPeer, which holds the complete knowledge of the Federation, feeds to the Peers invoking the Inquire API. Since the SuperPeer has the complete knowledge of the Federation, the results retrieved are *complete*. The Inquire API provide Peers with the functionalities of retrieving csds, publications and peerDescriptions; these functionalities are modeled by the following operators:

- FindCSDs(*serviceDeclarations* $\mathbb{SDESCS}'$, *serviceDefinitions* $\mathbb{SDEFS}'$): allows Peers to inquire the SuperPeer for the Canonical Service Declarations published within the Federation matching the criteria specified by the set of serviceDeclaration $\mathbb{SDESCS}'$ and the set of serviceDefinitions $\mathbb{SDEFS}'$. The inquiring Peer is returned with the following set of Canonical Service Declarations:

$$\{\text{csd} \in \mathbb{CSDS}_{\text{fed}} \mid \sqsubseteq_{\text{csd}} (\text{csd}, \mathbb{SDESCS}', \mathbb{SDEFS}')\}$$

- FindPublications(*csd* csd,
serviceDeclarations $\mathbb{SDESCS}'$, *serviceDefinitions* $\mathbb{SDEFS}'$): retrieve publications matching the specified criteria. The set of publications which is fed to the inquiring Peer is the following:

$$\{\text{pub} \in \mathbb{PUBS}_{\text{fed}} \mid \sqsubseteq_{\text{pub}} (\text{pub}, \text{csd}, \mathbb{SDESCS}', \mathbb{SDEFS}')\}$$

- FindPeerDescriptions(*businessDescriptions* $\mathbb{BDESCS}'$): retrieve information about the Peers that match the criteria specified as a set $\mathbb{BDESCS}'$ made of businessDescriptions. The peerDescriptions to be returned are obtained as follows:

$$\{\text{pdesc} \in \mathbb{PDESCS}_{\text{fed}} \mid \sqsubseteq_{\text{pdesc}} (\text{pdesc}, \mathbb{BDESCS}')\}$$

The distributed inquire operator, named DistributedFindPublications, models how Peers retrieve publications by inquiring their equivalents. Each Peer knows a subset of the whole set $\mathbb{PDESCS}_{\text{fed}}$. This knowledge is represented by the $\text{PAG}_{\text{pdesc}}$, which is obtained straightforwardly by the set of the known Peers. Suppose that $\mathbb{PDESCS}'$ contains all the Peers known by the Peer pdesc. $\text{PAG}_{\text{pdesc}}$ is a subset of $(\mathbb{PDESCS}_{\text{fed}} \backslash \{\text{pdesc}\})$: since the only function of the PAG is to keep track of the Peers to which queries are spread to, a Peer has no interest in including itself in its own PAG. Given $\mathbb{PDESCS}'$ the set of Peers known by the Peer pdesc, then $\text{PAG}_{\text{pdesc}}$ is the following set:

$$\text{PAG}_{\text{pdesc}} := \bigcup_{\text{pdesc}' \in \mathbb{PDESCS}'} \{\pi^{id}_{pdesc}(\text{pdesc}')\}$$

The *acquaintance digraph* G_{fed} of the Federation fed is shaped as a digraph with the identifiers of the Peers as nodes and the edges represent the *acquaintance relation*. Each node represents the Peer to which the identifier belongs to. A node representing a Peer pdesc has outgoing edges connecting it with all the nodes representing the Peers comprised within the PAG of pdesc. Formally, G_{fed} is defined as follows:

$$\begin{aligned} G_{\text{fed}} &:= (V, E) \\ V &= \textstyle\bigcup_{\text{pdesc} \in \mathbb{PDESCS}_{\text{fed}}} \{\pi^{id}_{pdesc}(\text{pdesc})\} \\ E &= \{(\text{id}_{\text{pdesc}}, \text{id}_{\text{pdesc}'}) \mid \{\text{id}_{\text{pdesc}}, \text{id}_{\text{pdesc}'}\} \subseteq V \wedge \text{id}_{\text{pdesc}'} \in \pi^{pag}_{pdesc}(\text{pdesc})\} \end{aligned}$$

Assume that the execution of the operations of the Public API are atomic: that is, the environment in which the operation is run does not change during the execution. According to the Federation System's structure, this means that there are no new publications, nor the PAGs of the Peers change during the run of the distributed service discovery mechanism. Under this assumption, we can model the results retrieved through the DistributedFindPublications operator according to the structure of the Federation's *acquaintance digraph*. We say that a Peer pdesc$'$ is *acquaintance-reachable* by the Peer pdesc in the Federation fed, if in the acquaintance digraph G_{fed} there exists a path connecting the nodes id_{pdesc} and $\text{id}_{\text{pdesc}'}$. In order to be consistent with the formal structure we have given to the acquaintance digraph, in the Federation fed the acquaintance-reachable relation $\prec$ is specified according to the Peers' identifiers: when a Peer pdesc$'$ is acquaintance-reachable by the Peer pdesc, we write $\text{id}_{\text{pdesc}} \prec_{\text{fed}} \text{id}_{\text{pdesc}'}$, where $\text{id}_{\text{pdesc}} = \pi^{id}_{pdesc}(\text{pdesc})$ and $\text{id}_{\text{pdesc}'} = \pi^{id}_{pdesc}(\text{pdesc}')$. The operator DistributedFindPublications is defined as follows:

$$\begin{aligned} \text{DistributedFindPublications} &:= \mathbb{PDESCS} \times \mathbb{CSDS} \times 2^{\mathbb{SDESCS}} \times 2^{\mathbb{SDEFS}} \rightarrow 2^{\mathbb{PDESCS}} \\ &\quad (\text{pdesc}, \text{csd}, \mathbb{SDESCS}', \mathbb{SDEFS}') \mapsto RESULTS \end{aligned}$$

$$\begin{aligned} RESULTS = \{&\text{pub} \in \mathbb{PUBS}_{\text{fed}} \\ &| \sqsubseteq_{\text{pub}} (\text{pub}, \text{csd}, \mathbb{SDESCS}', \mathbb{SDEFS}') \\ &\wedge \pi^{id}_{pdesc}(\text{pdesc}) \prec \\ &\pi^{pdesc}_{pub}(\text{pub})\} \end{aligned}$$

Intuitively, the returned publications are the ones matching the criteria specified by the csd, the serviceDefinitions and the serviceDescriptions, and that belong

to Peers that are acquaintance-reachable by the Peer which has made the search. Notice that, as it has been pointed out when introducing the Public API and the distributed service discovery mechanism, the DistributedFindPublications is not complete: there may exist publications matching the specified criteria which are not returned by to the inquiring Peer pdesc because they belong to Peers not acquaintance-reachable from pdesc.

3.5 The FDL language

The interactions among the SuperPeer and the Peers, and among the Peers themselves, are realized by exchanging documents written in the Federation Description Language (FDL). The FDL is a XML vocabulary encoding instances of the Federation System's item-descriptive and extensible concepts. Its structure is specified in the form of a set of XML Schemas (XSDs) [37]. The item-descriptive and extensible concepts are respectively mapped to the *item-descriptive* and *extensible* FDL types. In addition to the item-descriptive and extensible types, the FDL contains the *wrapped* types, allowing the adoption of the *Document/Literal* style in the interactions among the Web Services provided by the SuperPeer and the Peers.

The item-descriptive types are declared in the *fdl_base* schema, identified by the namespace `urn:federation-org:fdl:base`. The item-descriptive types are the straightforward transposition of the concepts described in Section 3.1 according to XSDs. Each item-descriptive type has the specification of an identifier: for instance, the *PeerDescription* type has an attribute of type *PeerDescriptionIdentifier* named *peerIdentifier*. All the item-descriptive types' identifiers, which in Section 3.4 we have assumed to be natural numbers only for the sake of simplicity, are represented as *qualified names*, and therefore mapped to the XSD type QName.

Likely the item-descriptive types, the extensible types are declared in the *fdl_base* schema and they belong to the `urn:federation-org:fdl:base` namespace. The extensible types are the following:

- *BusinessDescription*;
- *ServiceDescription*;
- *ServiceDefinition*.

They represent the omonimous concepts presented in Section 3.1. Although in the mathematical formalization we have not made any assumption on the structure of the extensible elements, the extensible types contain the specification of attributes acting as identifiers, respectively of types *BusinessDescriptionIdentifier*, *ServiceDescriptionIdentifier* and *ServiceDefinitionIdentifier*. Likely the item-descriptive types' identifiers, also the identifiers of the extensible types are specified to rely on the XML Schema's type QName.

The extensible types are abstract: their instances can not appear in well-formed FDL documents. In fact, they are meant to be extended in other XSD schemas in order to integrate the FDL Language with third party descriptive

languages. An example of such a schema containing extensions of the extensible types in order to make use of datastructures belonging to some other language is the *fdl_uddi* schema. The *fdl_uddi* schema is presented in Appendix A.3; it declares the types *UDDIBusinessDescription*, *UDDIServiceDescription* and *UDDIServiceDefinition*, respectively extending the FDL extensible types *BusinessDescription*, *ServiceDescription* and *ServiceDefinition*. The types *UDDIBusinessDescription*, *UDDIServiceDescription* and *UDDIServiceDefinition* respectively contain instances of the UDDI Datastructure's types *businessEntity*, *businessService* and *tModel*, thus enwrapping UDDI datastructure in FDL types.

When designing Web Services, one of the choices to take is the type of messages' encoding style to be adopted. The WSDL 1.1 Specifications list three different types of interaction styles:

- RPC/Encoded Style
- RPC/Literal Style
- Document/Literal Style

The interaction styles are the combination of two different aspects of Web Service design approaches: how the SOAP messages are formatted, that is the choice between the RPC or the Document Styles, and the encoding, which may be either Encoded or Literal, concerning how types are represented in XML.

The RPC styles are essentially two formats following the classical "remote procedure call" style in which a client sends a synchronous request to a server to execute an operation. The difference between the RPC/Encoded and the RPC/Literal styles is that, unlikely the former, the latter does not provide data type definition by the referenced XML Schema. The adoption of an RPC style tends to make the Web Service's WSDL description intuitive as it follows the well-known remote-procedure-call approach. Moreover, the RPC/Encoded style allows the adoption of data graphs or polymorphism in the Web Service's interface definition, whereas the RPC/Literal Web Services tend to have higher throughput performances as the type encoding is eliminated from the message. However, both the RPC styles suffer of the following drawbacks:

- The contract between the service and the clients (that is, the Web Service's interface) are tightly coupled: any changes to the interface would break the contract between the service and the client, thus requiring modifications in the implementations of both the Web Service and the Web Service Client software modules
- It is usually hard (or even impossible) to validate the data transferred by the SOAP message
- The RPC styles are not yet supported by the WSI conformance standard

The major difference between the Document style and the RPC-oriented styles *RPC/Literal* and *RPC/encoded* is that the service's consumer send the

invocation parameters in a normal XML document to the Web Service, instead of a discrete set of parameter values of methods. This makes the Document/Literal style more loosely coupled style of interaction than the RPC format, due to the fact that the SOAP Message does not contain any type encoding information, enhancements and changes can be made to the underpinning datastructure's XML schema without breaking the Web Service's interface and that the Document style services can maintain application state if multiple procedures must be called in a particular sequence. However, it may happen that removing the operation name from the SOAP message makes difficult or impossible the dispatching of the request and the appliance of routing and loading balance policies. This drawback made the Web Service's community adopt a "new" Document/Literal style, named Document/Literal wrapped, which is a sort of extension of the canonical Document/Literal one. The Document/Literal wrapped style aims at putting the operation name back into the SOAP message and still preserving the desirable characteristics of the Document/Literal style. This goal is addressed by developing the WSDL of the Web Service according to the following principles:

- For each operation, the input message has a single part
- The part is an element
- The element has the same name as the operation.

This style has still some drawback: it does not allow overloaded operations (in some sense the firms of the operations are made only by the operations' names) and the resulting WSDL is way more complicated than the one which would be obtained by applying one of the RPC styles. Although there is no specification in WSDL 1.1 standard for the Document/Literal wrapped style, many current SOAP toolkits supports it, and it is compliant with the WS-I Basic Profile [6].

The FDL *wrapped* types allow the use of the Document/Literal wrapped style in the request/response interactions among the Federation System's software modules, as well as the structure for the messages delivered through the message-oriented APIs, that is the notifications sent through the Notification API and the messages sent by the Public API's operations. The wrapped types are specified in the *fdl_api* schema, identified by the namespace `urn:federation-org:fdl_api`. The wrapped types are organized in a type hierarchy according to the APIs they refer to: the *FDLWrappingElement* is the abstract base of all the wrapped types. The abstract types *FDLPublicationOperation*, *FDLInquireOperation*, *FDLPublicOperation* and *FDLNotification* are respectively the base for the wrapped types involved in the Publication, Inquire, Public and Notification APIs. The structure of the wrapped types reflect the operations they have to represent. For instance, let us consider the FindPublications operation belonging to the Inquire API as it is presented in SubSection 3.2: it requires in input the csd the matching publications have to implement, and additional serviceDefinitions and serviceDeclarations maybe specified. The result of the operation is a collection containing all the publications that match all the specified criteria and the peerDescription representing

the Peers that provide the matching publications. The wrapped type that represent the request towards the Inquire service is the FindPublicationsQuery. The XML schema specification of the wrapped type FindPublicationsQuery specifies that it contains a CanonicalServiceDeclaration (the FDL representation of the csd concept), and any number of ServiceDefinition and ServiceDescription elements, representing the additional serviceDefinitions and serviceDeclarations. The FindPublicationsResult type represents the response message that is returned by the Inquire service to the invoking Peer: it is made of a sequence of Publication and PeerDescription elements, representing respectively the matching publications and the peers providers.

Chapter 4

An Implementation of the Federation System

We implemented the Federation System relying on the family of J2EE Technologies and Agent-oriented development. The information system is composed of the following elements:

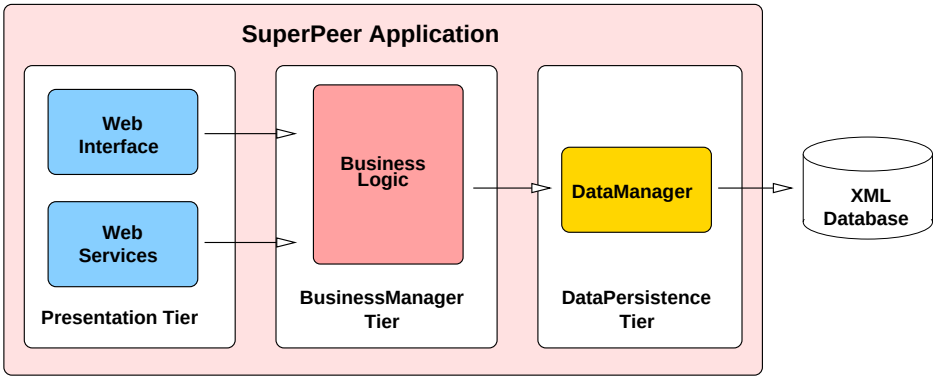
- the SuperPeer is implemented as a EAR J2EE 1.4 compliant application
- the Peers are implemented as agents in an Agent-Platform, named Federation Simulator

4.1 The SuperPeer Application

The SuperPeer application is designed as an Enterprise Application to run on top of an application server. The SuperPeer application's architecture is shown in Figure 4.1. It is made of the following modules:

- *WebInterface*: a WAR Module providing the SuperPeer administration facilities
- *WebServices*: a WAR Module exposing the Publication and Inquire Web Services

Figure 4.1: The architecture of the SuperPeer application.



- *BusinessLogic*: an EJB Module implementing the business logic of the SuperPeer Application
- *DataPersistence*: an EJB Module providing the data persistence facilities and management of the database backend

Both the WAR Module access the data rely on functionalities provided by the BusinessLogic EJB Module. The BusinessLogic Module implements the business logic of the application, manages the delivery of the notifications towards the Peers and makes use of the data persistence capabilities provided by the DataPersistence Module. Hereby we discuss the implementation details of the WebServices and DataPersistence, that are, in our opinion, the most interesting from the point of view of both adopted technologies and development approaches.

4.1.1 WebServices Module

The Web Services the SuperPeer exposes are realized on top of Apache Axis2[1] and comprised within the WebServices module. They do not rely on the Axis2's marshalling/unmarshalling capabilities provided by the *Axis Data Binding* (ADB) framework, but they instead handle the incoming and outcoming messages as raw XML trees through the *AXis Object Model* (AXIOM)[2] APIs. Since AXIOM is based on the novel XML's handling *pull parsing*[3] approach, it provides higher performances easing the amount of resources required by the Web Services support. In fact, the pull parsing approach provides a rich API *Document Object Label* (DOM) like, together with on-demand building of the object tree, delegating the parsing of the XML documents to the underlying *Streaming API for XML* (StAX) framework. Moreover, since all the data exchanged among the participants of the Federation System are encoded as FDL documents, and that we have already provided a library for parsing FDL documents into corresponding Java data structures, using a Data Binding framework would have been redundant.

4.1.2 DataPersistence Module

The SuperPeer application that keeps track of the current state of the Federation, that is the CanonicalServiceDeclarations available to the enrolled Peers, the PeerDescriptions and credentials of latter, as well as the Publications and Subscriptions the Peers manage. All the data the SuperPeer has to keep track of, can be formatted and expressed according to the FDL, which has been introduced in Section 3.5. Since the FDL is an XML Vocabulary, the documents compliant to it are themselves XML documents. Then, it comes naturally realizing the SuperPeer's data storage, retrieval and querying capabilities on top of an XML Database.

[1] Apache Axis2 Home Page: `http://ws.apache.org/axis2/`
[2] AXIOM's Project Home Page: `http://ws.apache.org/commons/axiom/`
[3] Sometimes this is also called "pull-through"

Indeed, the present Section presents the process and the outcomes of developing the data-managements features of the Federation's SuperPeer on top of the eXist[4] XML Database exploiting several world-wide adopted technologies and standards, such as XPath [11], XQuery [12], and XUpdate [49]. The Section is structured as follows: the requirements that the resulting system has to fulfill and the functionalities it has to provide are enlisted in Section 4.1.2. The XSD Schema according to which the data stored in the Database is presented in Section 4.1.2. Due to the size of the queries that have been realized to retrieve and modify the data, they are presented in Appendix B.2, together with an insight view of the technologies used in realizing the SuperPeer's data-managements capabilities presented in Appendix B.1.

Database Requirements

The database serving the SuperPeer application has to store the following datastructures borrowed from the FDL:

- the CanonicalServiceDeclarations available within the Federation
- the PeerDescriptions describing the Peers that have joined the Federation
- the Peers's credentials, namely their authentication information
- the Publications made available by the Peers
- the Subscriptions signed by the Peers

The FDL data structures are interconnected by several relations, which have been previously presented in Figure 3.2 in the shape of a UML Class Diagram. The relation occurring among the FDL data structures are the following:

- each Publication refers to exactly one CanonicalServiceDeclaration (N-1 relationship)
- each Publication refers to exactly one PeerDescription (N-1 relationship)
- each Subscription refers to exactly one CanonicalServiceDeclaration (N-1 relationship)
- each Subscription refers to exactly one PeerDescription (N-1 relationship)
- each PeerDescription may comprise any number of nested BusinessDescriptions (1-N relationship)
- each CanonicalServiceDeclaration may comprise any number of nested ServiceDescriptions (1-N relationship)
- each CanonicalServiceDeclaration may comprise any number of nested ServiceDefinitions (1-N relationship)

[4] eXist's Project Home Page: `http://exist.sourceforge.net`

- each Publication may comprise any number of nested ServiceDescriptions (1-N relationship)
- each Publication may comprise any number of nested ServiceDefinitions (1-N relationship)
- each Subscription may comprise any number of nested ServiceDescriptions (1-N relationship)
- each Subscription may comprise any number of nested ServiceDefinitions (1-N relationship)

On the one hand, the relation connecting the extensible elements, that is the BusinessDescriptions, ServiceDescriptions and ServiceDefinitions, with PeerDescriptions, Publications, Subscriptions and CanonicalServiceDeclarations, are implicit in the structure of the FDL data. In fact, according to the FDL Schema, presented in Appendix A.1, the extensible elements are contained within the item-descriptive ones. On the other hand, the relations interconnecting the item-descriptive elements are realized with an approach that resembles the adoption of primary and foreign keys in the relational databases.

The non-functional requirements for the SuperPeer's XML Database backend concern the use of APIs that may abstract the particular database adopted, and the scalability and performances that the resulting system has to achieve. In particular, the non-functional requirements are:

- The programmatical access to the database must rely on the XML:DB APIs (XUpdate, XPath 2.0 and XQuery 1.0).
- The database must scale enough to provide convenient support to Federations comprising up to one hundred Peers simulated via the Federation Simulator (see Section 4.2), assuming that each of the Peers owns ten Publications and ten Subscriptions on average at a time.
- The database application must be deployable in a J2EE container, or a Servlet Container compliant with the Servlet 2.4 specifications.

The functional requirements of the XML Database backend consist of a some required feature, such as authentication support, and a list of queries through which data can be retrieved and modified:

- The access to the database must be secured through the authentication mechanisms supported by the XML:DB APIs.
- Queries are to be decoupled from the code of the application, so that they can be modified in a convenient manner. For example, they may be stored in text files, and loaded and cached on request by a mechanism similar to the I18N (internationalization) supports.

The information stored within the database is retrieved through the following queries:

- *Authenticate Peer*: given a PeerIdentifier and a Credentials, it is returned the PeerDescription identified by that PeerIdentifier if the associated Credentials match, otherwise the query fails.

- *Authenticate Publication Owner*: given a PeerIdentifier and a PublicationIdentifier, the query succeeds only if exists a Publication with the specified PublicationIdentifier, which belongs to the Peer identified by the specified PeerIdentifier; otherwise the query fails. Upon success, the PeerDescription corresponding to the PeerIdentifier is returned.

- *Authenticate Subscription Owner*: given a SubscriptionIdentifier and a PeerIdentifier, the query succeeds only if exists a Subscription with the specified SubscriptionIdentifier, which belongs to the Peer identified by the specified PeerIdentifier; otherwise the query fails. Upon success, the PeerDescription corresponding to the PeerIdentifier is returned.

- *Get CanonicalServiceDeclaration*: taking in input a CanonicalServiceDeclarationIdentifier, it is returned the CanonicalServiceDeclaration identified by the former. The query fails if no CanonicalServiceDeclaration is found with the given CanonicalServiceDeclarationIdentifier.

- *Get PeerDescription*: given a PeerIdentifier, the PeerDescription identified by the former is returned . The query fails if no PeerDescription is found with the given PeerIdentifier.

- *Get Publication*: given a PublicationIdentifier, it is returned the Publication identified by the former. The query fails if no Publication is found with the given PublicationIdentifier.

- *Get Publications by Peer*: given a PeerIdentifier, all the Publications belonging to the Peer identified by the specified PeerIdentifier are returned. If no Peer with the given PeerIdentifier is found, then the query fails.

- *Get Publishers by Publications*: given set of PublicationIdentifiers, are returned all the PeerDescriptions describing the Peers owning one or more of the Publications identified by the given PublicationIdentifiers. The query fails if any of the PublicationIdentifiers specified does not correspond to an existing Publication.

- *Get Subscription*: given a SubscriptionIdentifier, it is returned the Subscription identified by the former. The query fails if no Subscription is found with the given SubscriptionIdentifier.

Moreover, a set of queries implementing the *matching operators* specified in Section 3.4 are required, in particular, the following query operations:

- *Get Matching CanonicalServiceDeclarations*: given a set of ServiceDescriptions and ServiceDefinitions, it returns all the matching CanonicalServiceDeclarations.

- *Get Matching PeerDescriptions*: given a BusinessDescriptions, it returns all the matching PeerDescriptions.
- *Get Matching Publications*: given a CanonicalServiceDeclarationIdentifier, and optionally a set of ServiceDescriptions and ServiceDefinitions, it returns all the matching Publications. The query fails if it does not exist a CanonicalServiceDeclaration identified by the specified CanonicalServiceDeclarationIdentifier
- *Get Matching Publications by Subscription*: given a SubscriptionIdentifier, it returns all the Publications matching the Subscription's CanonicalServiceDeclarationIdentifier, ServiceDescriptions and ServiceDefinitions. The query fails if it does not exist a Subscription identified by the specified SubscriptionIdentifier.
- *Get Matching Subscribers*: given a PublicationIdentifier, it returns the PeerDescriptions of all the Peers that own any of the Subscriptions matching the Publication identified by the specified PublicationIdentifier. The query fails if it does not exist a Publication identified by the specified PublicationIdentifier.
- *Get Matching Subscriptions*: given a PublicationIdentifier, are returned all the Subscriptions matching the CanonicalServiceDeclarationIdentifier, ServiceDescriptions and ServiceDefinitions of the Publication identified by the specified PublicationIdentifier. The query fails if it does not exist a Publication identified by the specified PublicationIdentifier.

The content of the database can be modified through the following queries:

- *Add CanonicalServiceDeclaration*: adds the CanonicalServiceDeclaration specified to the database. The CanonicalServiceDeclarationIdentifier that identifies the CanonicalServiceDeclaration must exist and not be already present within the database, otherwise the query fails.
- *Add PeerDescription*: adds the specified PeerDescription to the database. The PeerIdentifier that identifies the PeerDescription must exist and not be already present within the database, otherwise the query fails.
- *Add Publication*: adds the specified Publication into the database. The PublicationIdentifier that identifies the Publication must exist and not be already present within the database, otherwise the query fails. Moreover, the Publication's CanonicalServiceDeclarationIdentifier must exist and correspond to an existing CanonicalServiceDeclaration. Similarly the PublisherPeer must exist and correspond to a PeerIdentifier identifying a Peer in the database.
- *Add Subscription*: adds the specified Subscription into the database. The SubscriptionIdentifier that identifies the Subscription must exist and not be already present within the database, otherwise the query fails. Moreover, the Subscription's CanonicalServiceDeclarationIdentifier must exist

and correspond to an existing CanonicalServiceDeclaration. Similarly the SubscriberPeer must exist and correspond to a PeerIdentifier identifying a Peer in the database.

- *Modify CanonicalServiceDeclaration*: substitutes the ServiceDeclarations and ServiceDefinitions of the specified CanonicalServiceDeclaration. If it does not exist a CanonicalServiceDeclaration identified by the CanonicalServiceDeclarationIdentifier comprised within the specified the CanonicalServiceDeclaration the query fails.

- *Modify PeerDescription*: substitutes the BusinessDescription of the specified PeerDescription. If it does not exist a PeerDescription identified by the PeerIdentifier comprised within the specified the PeerDescription the query fails.

- *Modify Publication*: substitutes the ServiceDeclarations and ServiceDefinitions of the specified Publication. If it does not exist a Publication identified by the PublicationIdentifier comprised within the specified the Publication the query fails. Neither the CanonicalServiceDeclarationIdentifier nor the PublisherPeer of the involved Publication are affected by the modifications.

- *Modify Subscription*: substitutes the ServiceDeclarations and ServiceDefinitions of the specified Subscription. If it does not exist a Subscription identified by the SubscriptionIdentifier comprised within the specified the Subscription the query fails. Neither the CanonicalServiceDeclarationIdentifier nor the SubscriberPeer of the involved Publication are affected by the modifications.

- *Remove CanonicalServiceDeclaration*: given a CanonicalServiceDeclarationIdentifier, removes the CanonicalServiceDeclaration identified by the former.

- *Remove PeerDescription*: given a PeerIdentifier, removes the PeerDescription identified by the former.

- *Remove Publication*: given a PublicationIdentifier, removes the Publication identified by the former.

- *Remove All Publications by Peer*: given a PeerIdentifier, removes all the Publications owned by the Peer identified by the specified PeerIdentifier. The query fails if it does not exist a Peer identified by the specified PeerIdentifier.

- *Remove Subscription*: given a SubscriptionIdentifier, removes the Subscription identified by the former.

- *Remove All Subscription by Peer*: given a PeerIdentifier, removes all the Subscriptions owned by the Peer identified by the specified PeerIdentifier. The query fails if it does not exist a Peer identified by the specified PeerIdentifier.

During all the lifecycle of the database, there are constraints related to the FDL data structures that have to be enforced. These constraints, hereby listed, are directly derived from the relations interconnecting the FDL data structures:

- each Publication refers to one CanonicalServiceDeclaration through its field CanonicalServiceDeclarationIdentifier
- each Publication refers to one PeerDescription through its field PublisherPeer
- each Subscription refers to one CanonicalServiceDeclaration through its field CanonicalServiceDeclarationIdentifier
- each Subscription refers to one PeerDescription through its field SubscriberPeer

The Database's XML Schema

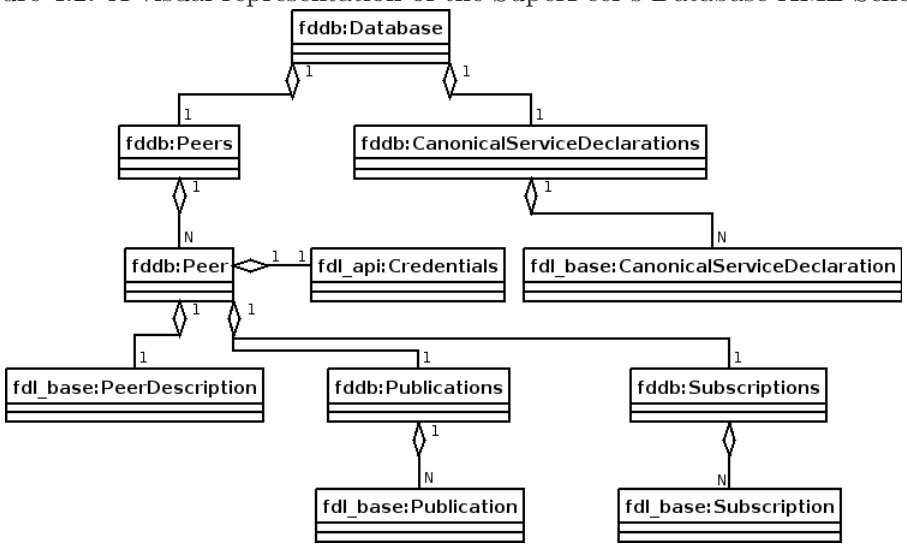
The requirements expressed in Section 4.1.2 are met through the adoption of XML-enabled data retrieval technologies, such as XPath B.1.1, XUpdate B.1.2 and XQuery B.1.3, applied in conjunction with an XML Database, namely eXist B.1.4.

The information retained by the SuperPeer are stored in a XML Database application, that organizes them accordingly to a user-defined XML Schema. In fact, an XML Database can be seen as (possibly large) XML document which, likely all the other XML documents, comply to a pre-defined structure, expressed by the means of an *XML Schema* (XSD) [37]. The schema according to which the information hold by the SuperPeer are organized is the following:

```
<xsd:schema
 targetNamespace="urn:federation-org:SuperPeer:database:xmldb"
 xmlns:xmldb="urn:federation-org:SuperPeer:database:xmldb"
 xmlns:fdl_base="urn:federation-org:fdl:base"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:complexType name="Database">
  <xsd:sequence>
   <xsd:element name="CanonicalServiceDeclarations"
    type="xmldb:CanonicalServiceDeclarations"/>
   <xsd:element name="Peers" type="xmldb:Peers"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="CanonicalServiceDeclarations">
  <xsd:sequence>
   <xsd:element ref="fdl_base:CanonicalServiceDeclaration"
    minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="Peers">
```

```
  <xsd:sequence>
   <xsd:element name="Peer" type="xmldb:Peer" minOccurs="0"
    maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="Peer">
  <xsd:sequence>
   <xsd:element ref="fdl_base:PeerDescription"/>
   <xsd:element ref="fdl_api:Credentials"/>
   <xsd:element name="Subscriptions"
    type="xmldb:Subscriptions"/>
   <xsd:element name="Publications" type="xmldb:Publications"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="Subscriptions">
  <xsd:sequence>
   <xsd:element ref="fdl_base:Subscription"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="Publications">
  <xsd:sequence>
   <xsd:element ref="fdl_base:Publication"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:schema>
```

Figure 4.2: A visual representation of the SuperPeer's Database XML Schema.



A visual representation of the structure of the above XML Schema, with emphasis on the cardinality of the relations between nesting data structures, is presented in Figure 4.2. The SuperPeer's Database XML Schema has the

namespace `urn:federation-org:SuperPeer:database:xmldb`, and its default prefix is *fddb*. It borrows elements from both the FDL_BASE schema (identified by the namespace `urn:federation-org:fdl:base`, prefix *fdl_base*, the whole schema is reported in Appendix A.1) and the FDL_API one (namespace `urn:federation-org:fdl:api`, prefix *fdl_api*, the whole schema is reported in Appendix A.2). The root element is named **Database**, and it contains an instance each of the types **CanonicalServiceDeclarations** and **Peers**. The **CanonicalServiceDeclarations** element plays the role of a container for all the FDL elements **CanonicalServiceDeclaration** that represent the CanonicalServiceDeclaration available within the Federation. Similarly, the element **Peers** comprises the elements **Peer**, each one describing a Peer in the Federation by the means of its PeerDescription, its credentials, and its publications and subscriptions that are respectively comprised within the **Publications** and **Subscriptions** elements. Thou this organization of the FDL documents would not be strictly necessary, due to the versatility of the XQuery language and the use of unique identifiers in all the item-descriptive and extensible FDL elements (see Section 3.5), it has proven several times to be useful during both the development of the queries and the debugging of the latter. Moreover, it is very likely to allow higher performances of the database: likewise the SQL, in which holds the "golden rule" *always push a projection inside as possible*, also XML Databases are affected by the dimensions of the slices of data that are examined (which are sequences in this case). Intuitively, it costs lesser to navigate through the tree-structure of the XML document organized in layers per contained data structure instead of searching for the matching elements in a sort of "flat domain" with all the FDL documents children of the root element **Database**.

Similarly, having the FDL Documents representing the Publications and Subscriptions belonging to a Peer nested within the **Peer** element representing the latter, underline the fact that those elements are extremely often accessed in conjunction with information regarding their Peer: for instance, Publications and Subscriptions can be retrieved by the identifier of their Peer owner.

An insight view of the standards adopted and the third parties technologies exploited to implement the Database is reported in Section B.1, while the queries through which the underlying DBMS is inquired are listed in Section B.2.

4.2 The Federation Simulator

The Federation's Peers are implemented as agents in an Agent Platform, named *Federation Simulator*, based on JADE 3.4[5], a framework that simplifies the implementation of multi-agent systems through a middle-ware that complies with *The Foundation of Intelligent Physical Agents* (FIPA) specifications[6]. In the Federation Simulator, each Peer corresponds to a *PeerAgent*. The PeerAgents interoperate with the Enterprise Application-based implementation of the Su-

[5]JADE Home Page: `http://jade.tilab.com`

[6]The Foundation of Intelligent Physical Agents (FIPA): `http://www.fipa.org/`

perPeer invoking the Web Services exposed by the latter, receive the FDLNotification through their Notification API, and they interact with each others exchanging messages corresponding to the ones described by the Public API.

We had the Federation Simulator running smoothly with two hundreds of PeerAgents acting randomly. Each PeerAgent had intervals of ten seconds (on average, it ranged from 5 to 15 seconds) between two subsequent actions. That is, the average Federation Simulator's workload was 20 actions/second. We are using the Federation Simulator to study how the Federations evolve. In particular, we are interested in studying how evolve the Peers' PAGs, and the acquaintance digraphs they determine.

4.2.1 Simulator's Requirements

The requirements that the Federation Simulator has to fulfill are the following:

- Implements the Peers as agents realizing both the Notification and Public APIs. The Peer agents interact to the Inquire and Publication APIs exposed as Web Services by the SuperPeer application.
- The Simulator can support up to one hundred of PeerAgents at the same time.
- The Peer agents can act in two different manners:
  - Execute operations that are them requested from outside the Agent Platform
  - Act randomly

  The operations that the Peers can carry out are:
  - Invoke Publication API's operations on the SuperPeer through a WebService's client
  - Invoke Inquire API's operations on the SuperPeer through a WebService's client
  - Invoke Public API's operations exposed by other Peer agents
  - Modify on request the composition of their PAGs by
    - adding acquainted Peers to the PAG by specifying the former's PeerDescriptions
    - removing acquainted Peers from the PAG by specifying the former's PeerDescriptions
  - Shutdown
- The Peer agents generate events that are delivered to the listeners registered to the Simulator under the following circumstances:
  - registration to the Federation
  - updating of the Peer agent's PeerDescription to the Federation
  - resign from the Federation

  - submission of a new Publication
  - update of an already-existing Publication
  - removal of a Publication
  - submission of a new Subscription
  - removal of a Subscription
  - reception of a DistributedFindPublications document via Public API
  - reception of a DistributedFindPublicationsResponse via Public API
  - modifications of the Peer Acquaintance Group
  - agent's shutdown

- Peer agents can be inquired from outside the Simulator to retrieve their status, which is made of:
  - whether the Peer is registered to a Federation or not
  - if the inquired Peer is registered to a Federation, the following data are also returned:
    * its PeerDescription
    * the Publications it owns
    * the Subscriptions it owns
    * the Peers listed in its PAG

- The Simulator proxies the notifications outcoming the SuperPeer and to be delivered to the Peers, so that multiple notification-delivery mechanisms can be implemented without modifying the Peer agents

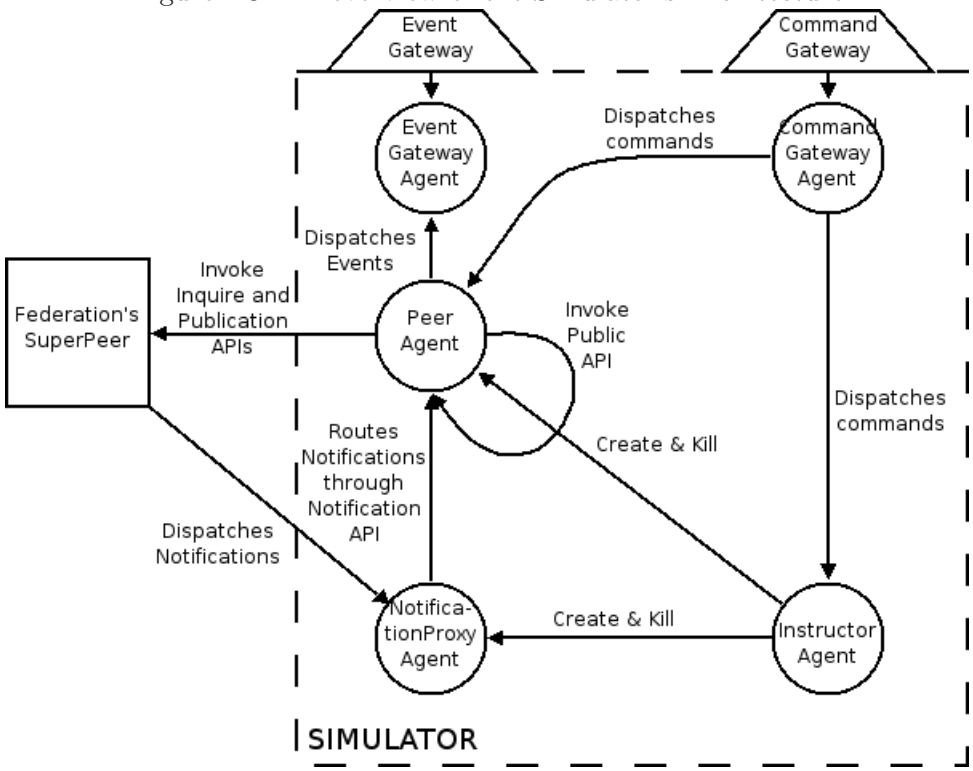
4.2.2 Architectural Overview

The Simulator provides its functionalities relying on the underlying interactions among instances of several types of agents:

- InstructorAgent
- PeerAgent
- NotificationProxyAgent
- CommandGatewayAgent
- EventGatewayAgent

A schema of the interactions occurring among the agents, as well as occurring with the rest of the Federation, is presented in Figure 4.3. The InstructorAgent has the role of managing the Simulator, by starting and shutting down the latter, adding and removing the NotificationProxyAgent and the PeerAgents and, more generally, executing the commands that are dispatched by the CommandGateway via the CommandGatewayAgent. The EventGateway relies on the EventGatewayAgent in order to collect the events that are generated by the

Figure 4.3: An overview of the Simulator's Architecture.



PeerAgents. The PeerAgents interact with each others invoking the operations of the Public API on their peers, and also interact with the SuperPeer invoking the latter's implementations of the Inquire and Publication APIs. Moreover, the PeerAgents execute the commands that are them delivered by the CommandGateway via the CommandGatewayAgent. The Notifications generated by the SuperPeer according to the Peers' Subscriptions are delivered to the NotificationProxyAgent, which then routes them to the respective PeerAgents.

The implementation of the above introduced architecture fulfills all the requirements listed in Section 4.2.1. The Federation Simulator has been implemented on top of JADE 3.4.

4.2.3 Simulator Agents

As mentioned in Section 4.2.2, the Simulator realizes the provided functionalities through interactions occurring among different types of agents, all extending the **SimulatorAgent** class, providing basilar facilities, such as logging and registration on startup to the DirectoryFacilitator and deregistration from the latter on take down. Therefore, all the agents running in the Simulator are called *SimulatorAgents*. The different SimulatorAgents are therefore examined more in detail.

InstructorAgent

The InstructorAgent is the first agent that is created within the simulator, and it is the last one to shutdown. Indeed, the InstructorAgent has sets up and takes down all the other agents in the Simulator upon the latter's startup and shutdown. When the Simulator is started, an InstructorAgent is created, and starts waiting for commands to execute. The InstructorAgent can perform the following commands:

- Start the Simulator
- Shutdown the Simulator
- Create a PeerAgent
- Remove a PeerAgent
- List PeerAgents

The InstructorAgent starts and shuts down the Simulator with the steps listed in Section 4.2.4 and 4.2.4 respectively. When the InstructorAgent receives the request to create a new Peer agent, it firstly instantiates the new PeerAgent and then starts the latter. The InstructorAgent retains the **AgentController** instances of all the created PeerAgents in order to be able to list them when receiving the List PeerAgents command. The removal of a PeerAgent is carried out invoking the **kill** method on the respective **AgentController**.

The commands to be executed are dispatched to the InstructorAgent by the CommandGatewayAgent encoded according to the **InstructorAgentCommandsOntology** presented in Section 4.2.5.

PeerAgent

The PeerAgent plays the role of a Peer in the Federation. Each PeerAgent represents a Peer. PeerAgents may join and leave the Federation, as well as invoke all the operations comprised within the Inquire and Publication APIs exposed by the Federation's SuperPeer, as well as the ones exposed by the Public APIs provided by the other PeerAgents available. The PeerAgents keep track of the Publications and Subscriptions they own, and of the Peers comprised within their Peer Acquaintance Groups.

The Public and Notification APIs are implemented as cyclic behaviors awaiting for messages matching proper **MessageTemplates**. The Notification API is provided by the **NotificationServiceListenerBehaviour**. The Public API is made of the **PublicServiceRequestsListenerBehaviour**, listening for incoming DistributedFindPublications documents, and the **PublicServiceResponsesListenerBehaviour** processes the incoming DistributedFindPublicationsResponse documents. These behaviors then typically add to the agents they belong to other behaviors that carry out the processing of the incoming messages and the composition and delivery of responses.

The PeerAgents are required to be able to behave in two different manners: acting randomly and awaiting for commands. Each way of operating is

reflected in a behavior which is set upon the PeerAgent creation according to the arguments that are passed. The "act randomly" manner is carried out by the **RandomExecutorBehaviour**, which is a **FSMBehaviour** that, in order to pick the next action to be taken by the Peer, takes into consideration both the state of Peer's registration to the Federation, as well as the state of its Publications and Subscription (that is, for example, a PeerAgent acting randomly will never try to submit a Publication if it was not registered to the Federation). Otherwise, the **BOTExecutorBehaviour** simply awaits for commands dispatcher by the CommandGatewayAgent, that are straightforwardly executed.

The PeerAgents make use of several ontologies:

- the **PeerAgentCommandsOntology**, which encodes the commands dispatched by the CommandPeerAgent to the target Peers
- the **FDLPublicOntology**, according to which are encoded the messages exchanged via the Public API
- the **FDLNotificationOntology**, encoding the notifications that are delivered to the Peers by the SuperPeer via the NotificationProxyAgent

The Peer agents generate events that are delivered to the listeners registered to the Simulator under the following circumstances:

- registration to the Federation
- updating of the PeerAgent's PeerDescription to the Federation
- resign from the Federation
- submission of a new Publication
- update of an already-existing Publication
- removal of a Publication
- submission of a new Subscription
- removal of a Subscription
- reception of a DistributedFindPublications document via Public API
- reception of a DistributedFindPublicationsResponse via Public API
- modifications of the Peer Acquaintance Group
- PeerAgent's shutdown

The events are dispatched to the EventGatewayAgent, which delivers them outside the Simulator to the listening applications.

NotificationProxyAgent

The NotificationProxyAgent works as a bridge connecting the PeerAgents with all the different implementations of the Notification delivery mechanism. Each running Simulator has exactly one NotificationProxyAgent, which is created and started by the InstructorAgent upon the Simulator Startup, and killed by the InstructorAgent when shutting down the Simulator.

The agent runs the **NotificationsListenerBehaviour**, which receives the FDLNotification documents that have to be routed to the Peers. Upon the reception of a FDLNotification document, the one-shot behavior **DeliverNotificationBehaviour** carries out the routing to the target PeerAgent. The messages that the NotificationProxyAgent receives and dispatches are encoded according to the **FDLNotificationOntology**.

CommandGatewayAgent

The CommandGatewayAgent is a refinement of the JadeGateway mechanism provided by the JADE Platform 3.4, in order to overcome a serious limitations: since the JadeGateway facility is implemented according to the pattern Singleton, it is not possible to have more than one agent executing command dispatched from outside the JADE Platform per Java Virtual Machine. The refinement was needed because both the CommandGatewayAgent and the EventGatewayAgent needed to rely on JadeGateway-like facilities, and thus the standard JadeGateway implementation shipped with the platform did not fit.

The CommandGatewayAgent awaits for commands from outside the Simulator. Any number of CommandGatewayAgent instances can run at the same time, according to how many external software modules aim at invoking commands inside the Simulator. The commands are represented by instances of classes extending the **SimulatorCommand** one. The commands are intended to be executed by either the InstructorAgent or one of the PeerAgents. The commands meant to be run by the InstructorAgent extend the class **InstructorAgentCommand**, whereas the ones to be run by a PeerAgent inherit from the **PeerCommand** one. According to the type of command received, the CommandGatewayAgent forges a message according to either the **InstructorAgentCommandsOntology** or the **PeerAgentCommandsOntology**, and sends it to the target agent blocking itself until a response message is returned; afterwhat the command is released and the result is returned to the invoking application outside the Simulator.

EventGatewayAgent

Likewise the CommandGatewayAgent, the EventGatewayAgent aims at allowing exchange of data among software modules interoperating with the Simulator and the latter's agents. The EventGatewayAgent enqueues and buffers the events generated by the PeerAgents, and dispatches them when polled from outside the Simulator. There is an instance of EventGatewayAgent per EventListener registered to the Simulator. If events are generated and there are no EventListeners registered, the event is discarded.

4.2.4 Simulator Workflows

In this Section, the schematic representations of the Simulator Initialization and Shutdown Workflows are presented.

Simulator Initialization Workflow

The Simulator is initialized when a SimulatorCommandGateway processes a **StartSimulatorCommand**. The steps are the following:

1. an InstructorAgent is created
2. the newly created InstructorAgent runs a **StartSimulatorBehaviour**
3. the **StartSimulatorBehaviour** creates and starts the NotificationProxyAgent
4. if configurations have been retrieved by the InstructorAgent, a set of PeerAgents are accordingly created and started immediately
5. the InstructorAgent starts running the cyclic **CommandsListenerBehaviour**, awaiting for commands to be dispatched by a SimulatorCommandGateway

Simulator Shutdown Workflow

The Simulator is taken down upon the dispatching of a **ShutDownSimulatorCommand** by a SimulatorCommandGateway. The following steps are taken:

1. the InstructorAgent starts running a **ShutDownSimulatorBehaviour**
2. the **ShutDownSimulatorBehaviour** deletes the InstructorAgent
3. during the InstructorAgent's **takeDown** method's execution, the NotifixactionProxyAgent and all the PeerAgents are therefore deleted

That is, the Simulator stands inactive until it is initialized again.

4.2.5 Ontologies

The Simulator relies on several Ontologies prescribing how the messages exchanged among the SimulatorAgents are encoded. All the Ontologies are compliant with the *Vocabulary* pattern [10]. The Simulator's Ontologies are the following:

- **FDLBaseOntology**: describes how to encode FDL item-descriptive and extensible elements in an ACL-compliant manner
- **FDLApiOntology**: it is based on the **FDLBaseOntology**, describes the elements that are in common among the **FDLInquireOntology**, **FDLPublicationOntology** and the **FDLPublicOntology**

- **FDLInquireOntology**: encodes the FDL documents involved into invocations of the Inquire API
- **FDLNotificationOntology**: encodes the FDL documents involved into invocations of the Notification API
- **FDLPublicationOntology**: encodes the FDL documents involved into invocations of the Publication API
- **FDLPublicOntology**: encodes the FDL documents involved into invocations of the Public API
- **FDLFullOntology**: sums up all the Ontologies based on the **FDLBaseOntology** one, and thus it provides a way of encoding all the FDL elements
- **BaseCommandsOntology**: provides the basis for the encoding of the commands
- **InstructorAgentCommandsOntology**: encodes the commands run by the InstructorAgent relying on the facilities offered by the **BaseCommandsOntology**
- **PeerAgentCommandsOntology**: based on the **BaseCommandsOntology**, encodes the commands run by the PeerAgent

All the Ontologies involve the encoding of FDL documents. Thus, they are organized incrementally according to the structure of the FDL language, enhancing modularity and reusability.

Chapter 5

Conclusions

The provisioning of service providers is a fundamental part for the success of Service Oriented Architectures. Service consumers need to consider all possible available services efficiently to choose and bind dynamically to the appropriate ones for their needs. Service brokers fill the role of providing information to the consumers and collect providers' publications, thus realizing the ultimate vision of dynamic open service discovery.

There are several approaches to the service brokering which range from the completely centralized architecture of UDDI v2 to the entirely peer-to-peer proposals. If, on the one hand, the centralized approach guarantees completeness when searching for services, on the other hand, it is effected by the usual drawbacks as single point of failure, and bottlenecks. Moreover, due to the delegation from the service providers to the service brokers there could be out of date information kept in the centralized brokers. In the peer-to-peer approach, conversely, there is an overlap between the role of service providers and service brokers which guarantees that the information published is mostly up to date. The price to pay for the higher information quality is the slower query processing time and the possible unreachability of published information.

To take the best of both centralized and peer-to-peer worlds, we proposed the Federation System, a hybrid architecture in which publication and subscription are managed centrally, while discovery is carried out in either centralized or in a peer-to-peer way depending on the consumers' wishes. The Federation System is a service broker relying on the idea of federations of providers with shared business goals. Furthermore, the Federation System abstracts from the specificity of third party service descriptive languages as different federations may use different standards or even proprietary formats. The only requirement is to extend the proposed FDL to wrap the underpinning descriptive languages. We provided a formal specification underlying the proposed language and architecture and an overview of a related implementation of the SuperPeer and the Federation Simulator, and Agent Platform providing support for simulating the Peers as agents.

The Federation System is widely applicable. Acting as a Web Service integration platform aimed at serving similar or strictly interrelated business operated by communities of industrial consortia, such as the vertical e-marketplaces,

as well as commercial syndicates. Furthermore, its relying on peer-to-peer makes it scalable and extensible, thus being also suitable for open environments which may even go beyond e-marketplaces.

The introduction of the peer-to-peer architecture in the brokering, gives advantages in terms of scalability and of ownership of published information. Though, it introduces the potential problem of reachability of published information. When introducing peer acquaintance groups, we have remarked how some peer may be not reachable from others. Isolated Peer can not make use of the distributed service discovery mechanism because they have not acquainted Peers to spread the publication queries to. To overcome this problem, one can devise different management of the peers, e.g., decoupling the PAG mechanisms from the peer-to-peer connectivity by resorting to a JXTA[1] like framework.

Further extensions of the architecture presented may regard improvements in the structure of the SuperPeer, which may be implemented as cloud-like architecture in the manner of UDDI Registries compliant to the UDDI specifications 3.0. Moreover, the Federation System may be extended to become even more "federated", by providing a mechanism for distributed subscription, thus adding a subscription at the peer level, and having different federations interacting with each other. Finally, future may provide further insight on policies according to which the Peers form their Peer Acquaintance Groups in order to maximize the mutual reachability among the Peers and minimize the size of the PAGs, still achieving a desireable redundancy of interconnections easing possible networking-related problems.

[1] JXTA Project's Home Page: `http://www.jxta.org/`

Appendix A

Federation Description Language's XML Schema

A.1 FDL-Base Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
 targetNamespace="urn:federation-org:fdl:base"
 xmlns:fdl_base="urn:federation-org:fdl:base"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:complexType name="FDLElement" abstract="true"/>
 <xsd:complexType name="FDLItemDescriptiveElement"
  abstract="true">
  <xsd:complexContent>
   <xsd:extension base="fdl_base:FDLElement"/>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:complexType name="FDLExtensibleElement"
  abstract="true">
  <xsd:complexContent>
   <xsd:extension base="fdl_base:FDLElement"/>
  </xsd:complexContent>
 </xsd:complexType>
 <!-- Item Descriptive DataStructures -->
 <xsd:element name="PeerDescription"
  type="fdl_base:PeerDescription"/>
 <xsd:complexType name="PeerDescription">
  <xsd:complexContent>
   <xsd:extension base="fdl_base:FDLItemDescriptiveElement">
    <xsd:sequence>
     <xsd:element name="PeerIdentifier"
      type="fdl_base:PeerIdentifier"/>
     <xsd:element name="NotificationServiceLocation"
      type="fdl_base:ServiceLocation"/>
```

```
    <xsd:element name="PublicServiceLocation"
     type="fdl_base:ServiceLocation"/>
    <xsd:element ref="fdl_base:BusinessDescription"
     maxOccurs="unbounded" minOccurs="0"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="CanonicalServiceDeclaration"
 type="fdl_base:CanonicalServiceDeclaration"/>
<xsd:complexType name="CanonicalServiceDeclaration">
 <xsd:complexContent>
  <xsd:extension base="fdl_base:FDLItemDescriptiveElement">
   <xsd:sequence>
    <xsd:element name="CanonicalServiceDeclarationIdentifier"
     type="fdl_base:CanonicalServiceDeclarationIdentifier"/>
    <xsd:element ref="fdl_base:ServiceDefinition"
     maxOccurs="unbounded" minOccurs="0"/>
    <xsd:element ref="fdl_base:ServiceDescription"
     maxOccurs="unbounded" minOccurs="0"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Publication"
 type="fdl_base:Publication"/>
<xsd:complexType name="Publication">
 <xsd:complexContent>
  <xsd:extension
   base="fdl_base:FDLItemDescriptiveElement">
   <xsd:sequence>
    <xsd:element name="PublicationIdentifier"
     type="fdl_base:PublicationIdentifier"/>
    <xsd:element
     name="CanonicalServiceDeclarationIdentifier"
     type="fdl_base:CanonicalServiceDeclarationIdentifier"/>
    <xsd:element name="PublisherPeer"
     type="fdl_base:PeerIdentifier"/>
    <xsd:element ref="fdl_base:ServiceDefinition"
     minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="fdl_base:ServiceDescription"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Subscription"
```

```
 type="fdl_base:Subscription"/>
<xsd:complexType name="Subscription">
 <xsd:complexContent>
  <xsd:extension
   base="fdl_base:FDLItemDescriptiveElement">
   <xsd:sequence>
    <xsd:element name="SubscriptionIdentifier"
     type="fdl_base:SubscriptionIdentifier"/>
    <xsd:element
     name="CanonicalServiceDeclarationIdentifier"
     type="fdl_base:CanonicalServiceDeclarationIdentifier"/>
    <xsd:element name="SubscriberPeer"
     type="fdl_base:PeerIdentifier"/>
    <xsd:element ref="fdl_base:ServiceDefinition"
     minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="fdl_base:ServiceDescription"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<!-- Extensible Structures -->
<xsd:element name="BusinessDescription"
 type="fdl_base:BusinessDescription"/>
<xsd:complexType name="BusinessDescription"
 abstract="true">
 <xsd:complexContent>
  <xsd:extension
   base="fdl_base:FDLExtensibleElement">
   <xsd:sequence>
    <xsd:element
     ref="fdl_base:BusinessDescriptionIdentifier"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ServiceDescription"
 type="fdl_base:ServiceDescription"/>
<xsd:complexType name="ServiceDescription"
 abstract="true">
 <xsd:complexContent>
  <xsd:extension base="fdl_base:FDLExtensibleElement">
   <xsd:sequence>
    <xsd:element
     ref="fdl_base:ServiceDescriptionIdentifier"/>
   </xsd:sequence>
  </xsd:extension>
```

```
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ServiceDefinition"
 type="fdl_base:ServiceDefinition"/>
<xsd:complexType name="ServiceDefinition"
 abstract="true">
 <xsd:complexContent>
  <xsd:extension base="fdl_base:FDLExtensibleElement">
   <xsd:sequence>
    <xsd:element
     ref="fdl_base:ServiceDefinitionIdentifier"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="ServiceLocation">
 <xsd:restriction base="xsd:anyURI">
  <xsd:maxLength value="255"/>
 </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="PeerIdentifier">
 <xsd:restriction base="xsd:QName">
  <xsd:maxLength value="255"/>
 </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="SubscriptionIdentifier">
 <xsd:restriction base="xsd:QName">
  <xsd:maxLength value="255"/>
 </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType
 name="CanonicalServiceDeclarationIdentifier">
 <xsd:restriction base="xsd:QName">
  <xsd:maxLength value="255"/>
 </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="PublicationIdentifier">
 <xsd:restriction base="xsd:QName">
  <xsd:maxLength value="255"/>
 </xsd:restriction>
</xsd:simpleType>
<xsd:element name="BusinessDescriptionIdentifier"
 type="fdl_base:BusinessDescriptionIdentifier"/>
<xsd:simpleType name="BusinessDescriptionIdentifier">
 <xsd:restriction base="xsd:QName">
  <xsd:maxLength value="255"/>
 </xsd:restriction>
```

```
 </xsd:simpleType>
 <xsd:element name="ServiceDefinitionIdentifier"
  type="fdl_base:ServiceDefinitionIdentifier"/>
 <xsd:simpleType name="ServiceDefinitionIdentifier">
  <xsd:restriction base="xsd:QName">
   <xsd:maxLength value="255"/>
  </xsd:restriction>
 </xsd:simpleType>
 <xsd:element name="ServiceDescriptionIdentifier"
  type="fdl_base:ServiceDescriptionIdentifier"/>
 <xsd:simpleType name="ServiceDescriptionIdentifier">
  <xsd:restriction base="xsd:QName">
   <xsd:maxLength value="255"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:schema>
```

A.2 FDL-Api Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
 targetNamespace="urn:federation-org:fdl:api"
 xmlns:fdl_api="urn:federation-org:fdl:api"
 xmlns:fdl_base="urn:federation-org:fdl:base"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:import namespace="urn:federation-org:fdl:base"
  schemaLocation="fdl_base.xsd"/>
 <xsd:complexType name="FDLWrappingElement"
  abstract="true">
  <xsd:complexContent>
   <xsd:extension base="fdl_base:FDLElement"/>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:complexType name="FDLOperation"
  abstract="true">
  <xsd:complexContent>
   <xsd:extension base="fdl_api:FDLWrappingElement"/>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:complexType name="FDLAuthenticatedOperation"
  abstract="true">
  <xsd:complexContent>
   <xsd:extension base="fdl_api:FDLOperation">
    <xsd:sequence>
     <xsd:element ref="fdl_api:AuthenticationDataSet"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:complexType name="FDLInquireOperation"
  abstract="true">
  <xsd:complexContent>
   <xsd:extension base="fdl_api:FDLOperation"/>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:complexType name="FDLPublicOperation"
  abstract="true">
  <xsd:complexContent>
   <xsd:extension base="fdl_api:FDLOperation">
    <xsd:sequence>
     <xsd:element ref="fdl_api:OperationIdentifier"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
```

```
</xsd:complexType>
<xsd:complexType name="FDLPublicationOperation"
 abstract="true">
 <xsd:complexContent>
  <xsd:extension
   base="fdl_api:FDLAuthenticatedOperation"/>
 </xsd:complexContent>
</xsd:complexType>
<!-- Authentication -->
<xsd:element name="AuthenticationDataSet"
 type="fdl_api:AuthenticationDataSet"/>
<xsd:complexType name="AuthenticationDataSet">
 <xsd:sequence>
  <xsd:element name="PeerIdentifier"
   type="fdl_base:PeerIdentifier"/>
  <xsd:element ref="fdl_api:Credentials"/>
 </xsd:sequence>
</xsd:complexType>
<xsd:element name="OperationIdentifier"
 type="fdl_api:OperationIdentifier"/>
<xsd:simpleType name="OperationIdentifier">
 <xsd:restriction base="xsd:QName">
  <xsd:maxLength value="255"/>
 </xsd:restriction>
</xsd:simpleType>
<xsd:element name="Credentials"
 type="fdl_api:Credentials"/>
<xsd:simpleType name="Credentials">
 <xsd:restriction base="xsd:string">
  <xsd:maxLength value="255"/>
 </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="FDLResponse" abstract="true">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLWrappingElement"/>
 </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="FDLPublicationResponse"
 abstract="true">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLResponse"/>
 </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="FDLInquireResponse"
 abstract="true">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLResponse"/>
```

```
 </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="FDLPublicResponse"
 abstract="true">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLResponse">
   <xsd:sequence>
    <xsd:element ref="fdl_api:OperationIdentifier"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="FDLFault" abstract="true">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLWrappingElement">
   <xsd:sequence>
    <xsd:element name="FaultCode"
     type="fdl_api:FaultCode"/>
    <xsd:element name="FaultMessage"
     type="fdl_api:FaultMessage"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PublicationFault"
 type="fdl_api:PublicationFault"/>
<xsd:complexType name="PublicationFault">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLFault"/>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="InquireFault"
 type="fdl_api:InquireFault"/>
<xsd:complexType name="InquireFault">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLFault"/>
 </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="FDLNotification"
 abstract="true">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLWrappingElement"/>
 </xsd:complexContent>
</xsd:complexType>
<!-- Simple Types -->
<xsd:simpleType name="FaultCode">
 <xsd:restriction base="xsd:string"/>
```

```
</xsd:simpleType>
<xsd:simpleType name="FaultMessage">
 <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
<!-- Wrapping DataStructures -->
<!-- Publication -->
<xsd:element name="PeerRegister"
 type="fdl_api:PeerRegister"/>
<xsd:complexType name="PeerRegister">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationOperation">
   <xsd:sequence>
    <xsd:element ref="fdl_base:PeerDescription"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PeerRegisterResponse"
 type="fdl_api:PeerRegisterResponse"/>
<xsd:complexType name="PeerRegisterResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationResponse">
   <xsd:sequence>
    <xsd:element ref="fdl_api:AuthenticationDataSet"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PeerUpdate" type="fdl_api:PeerUpdate"/>
<xsd:complexType name="PeerUpdate">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationOperation">
   <xsd:sequence>
    <xsd:element ref="fdl_base:PeerDescription"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PeerUpdateResponse"
 type="fdl_api:PeerUpdateResponse"/>
<xsd:complexType name="PeerUpdateResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationResponse"/>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PeerResign" type="fdl_api:PeerResign"/>
<xsd:complexType name="PeerResign">
```

```
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationOperation"/>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PeerResignResponse"
 type="fdl_api:PeerResignResponse"/>
<xsd:complexType name="PeerResignResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationResponse"/>
 </xsd:complexContent>
</xsd:complexType>
<!-- Service Publication, Update and Removal -->
<xsd:element name="PublicationSubmission"
 type="fdl_api:PublicationSubmission"/>
<xsd:complexType name="PublicationSubmission">
 <xsd:complexContent>
  <xsd:extension
   base="fdl_api:FDLPublicationOperation">
   <xsd:sequence>
    <xsd:element ref="fdl_base:Publication"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PublicationSubmissionResponse"
 type="fdl_api:PublicationSubmissionResponse"/>
<xsd:complexType name="PublicationSubmissionResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationResponse">
   <xsd:sequence>
    <xsd:element ref="fdl_base:Publication"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PublicationUpdate"
 type="fdl_api:PublicationUpdate"/>
<xsd:complexType name="PublicationUpdate">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationOperation">
   <xsd:sequence>
    <xsd:element ref="fdl_base:Publication"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PublicationUpdateResponse"
```

```
 type="fdl_api:PublicationUpdateResponse"/>
<xsd:complexType name="PublicationUpdateResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationResponse"/>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PublicationRemoval"
 type="fdl_api:PublicationRemoval"/>
<xsd:complexType name="PublicationRemoval">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationOperation">
   <xsd:sequence>
    <xsd:element ref="fdl_base:Publication"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PublicationRemovalResponse"
 type="fdl_api:PublicationRemovalResponse"/>
<xsd:complexType name="PublicationRemovalResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationResponse"/>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SubscriptionSubmission"
 type="fdl_api:SubscriptionSubmission"/>
<xsd:complexType name="SubscriptionSubmission">
 <xsd:complexContent>
  <xsd:extension
   base="fdl_api:FDLPublicationOperation">
   <xsd:sequence>
    <xsd:element ref="fdl_base:Subscription"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SubscriptionSubmissionResponse"
 type="fdl_api:SubscriptionSubmissionResponse"/>
<xsd:complexType name="SubscriptionSubmissionResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationResponse">
   <xsd:sequence>
    <xsd:element name="SubscriptionIdentifier"
     type="fdl_base:SubscriptionIdentifier"/>
    <xsd:element ref="fdl_base:PeerDescription"
     minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="fdl_base:Publication"
```

```
      minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SubscriptionRemoval"
 type="fdl_api:SubscriptionRemoval"/>
<xsd:complexType name="SubscriptionRemoval">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLPublicationOperation">
   <xsd:sequence>
    <xsd:element ref="fdl_base:Subscription"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SubscriptionRemovalResponse"
 type="fdl_api:SubscriptionRemovalResponse"/>
<xsd:complexType name="SubscriptionRemovalResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLResponse"/>
 </xsd:complexContent>
</xsd:complexType>
<!-- Subscription Notifications -->
<xsd:complexType name="PublicationNotification"
 abstract="true">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLNotification">
   <xsd:sequence>
    <xsd:element name="SubscriptionIdentifier"
     type="fdl_base:SubscriptionIdentifier"/>
    <xsd:element ref="fdl_base:PeerDescription"
     minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="fdl_base:Publication"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element
 name="PublicationSubmissionNotification"
 type="fdl_api:PublicationSubmissionNotification"/>
<xsd:complexType
 name="PublicationSubmissionNotification">
 <xsd:complexContent>
  <xsd:extension
   base="fdl_api:PublicationNotification"/>
```

```
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PublicationUpdateNotification"
 type="fdl_api:PublicationUpdateNotification"/>
<xsd:complexType name="PublicationUpdateNotification">
 <xsd:complexContent>
  <xsd:extension
   base="fdl_api:PublicationNotification"/>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="PublicationRemovalNotification"
 type="fdl_api:PublicationRemovalNotification"/>
<xsd:complexType name="PublicationRemovalNotification">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:PublicationNotification"/>
 </xsd:complexContent>
</xsd:complexType>
<!-- Inquire -->
<xsd:element name="FindCanonicalServiceDeclarations"
 type="fdl_api:FindCanonicalServiceDeclarations"/>
<xsd:complexType name="FindCanonicalServiceDeclarations">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLInquireOperation">
   <xsd:sequence>
    <xsd:element ref="fdl_base:ServiceDefinition"
     minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="fdl_base:ServiceDescription"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="FindCSDsResponse"
 type="fdl_api:FindCSDsResponse"/>
<xsd:complexType name="FindCSDsResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLInquireResponse">
   <xsd:sequence>
    <xsd:element
     ref="fdl_base:CanonicalServiceDeclaration"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="FindPublications"
 type="fdl_api:FindPublications"/>
```

```
<xsd:complexType name="FindPublications">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLInquireOperation">
   <xsd:sequence>
    <xsd:element
     ref="fdl_base:CanonicalServiceDeclarationIdentifier"
     minOccurs="0" maxOccurs="1"/>
    <xsd:element ref="fdl_base:ServiceDefinition"
     minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="fdl_base:ServiceDescription"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="FindPublicationsResponse"
 type="fdl_api:FindPublicationsResponse"/>
<xsd:complexType name="FindPublicationsResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLInquireResponse">
   <xsd:sequence>
    <xsd:element
     ref="fdl_base:PeerDescription"
     minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="fdl_base:Publication"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="FindPeerDescriptions"
 type="fdl_api:FindPeerDescriptions"/>
<xsd:complexType name="FindPeerDescriptions">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLInquireOperation">
   <xsd:sequence>
    <xsd:element ref="fdl_base:BusinessDescription"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="FindPeerDescriptionsResponse"
 type="fdl_api:FindPeerDescriptionsResponse"/>
<xsd:complexType name="FindPeerDescriptionsResponse">
 <xsd:complexContent>
  <xsd:extension base="fdl_api:FDLInquireResponse">
```

```
    <xsd:sequence>
     <xsd:element ref="fdl_base:PeerDescription"
      minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
 <!-- Public -->
 <xsd:element name="DistributedFindPublications"
  type="fdl_api:DistributedFindPublications"/>
 <xsd:complexType name="DistributedFindPublications">
  <xsd:complexContent>
   <xsd:extension base="fdl_api:FDLPublicOperation">
    <xsd:sequence>
     <xsd:element name="OriginatingPeer"
      type="fdl_base:PeerDescription"/>
     <xsd:element
      ref="fdl_base:CanonicalServiceDeclarationIdentifier"
      minOccurs="0" maxOccurs="1"/>
     <xsd:element ref="fdl_base:ServiceDefinition"
      minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="fdl_base:ServiceDescription"
      minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:element name="DistributedFindPublicationsResponse"
  type="fdl_api:DistributedFindPublicationsResponse"/>
 <xsd:complexType name="DistributedFindPublicationsResponse">
  <xsd:complexContent>
   <xsd:extension base="fdl_api:FDLPublicResponse">
    <xsd:sequence>
     <xsd:element ref="fdl_base:PeerDescription"
      minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element ref="fdl_base:Publication"
      minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
</xsd:schema>
```

A.3 FDL-Uddi Schema

```
<!--
  Federation Description Language - UDDI Extension
-->
<xsd:schema
 targetNamespace="urn:federation-org:fdl:ext:uddi"
 xmlns:fdl_base="urn:federation-org:fdl:base"
 xmlns:fdl_uddi="urn:federation-org:fdl:ext:uddi"
 xmlns:uddi="urn:uddi-org:api_v3"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:import namespace="urn:federation-org:fdl:base"
  schemaLocation="fdl_base.xsd"/>
 <xsd:import namespace="urn:uddi-org:api_v3"
  schemaLocation="http://uddi.org/wsdl/uddi_v3.xsd"/>
 <xsd:element name="UDDIBusinessDescription"
  type="fdl_uddi:UDDIBusinessDescription"/>
 <xsd:complexType name="UDDIBusinessDescription">
  <xsd:complexContent>
   <xsd:extension base="fdl_base:BusinessDescription">
    <xsd:sequence>
     <xsd:element ref="uddi:businessEntity"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:element name="UDDIServiceDescription"
  type="fdl_uddi:UDDIServiceDescription"/>
 <xsd:complexType name="UDDIServiceDescription">
  <xsd:complexContent>
   <xsd:extension base="fdl_base:ServiceDescription">
    <xsd:sequence>
     <xsd:element ref="uddi:businessService"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>
 <xsd:element name="UDDIServiceDefinition"
  type="fdl_uddi:UDDIServiceDefinition"/>
 <xsd:complexType name="UDDIServiceDefinition">
  <xsd:complexContent>
   <xsd:extension base="fdl_base:ServiceDefinition">
    <xsd:sequence>
     <xsd:element ref="uddi:tModel"/>
    </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
```

```
 </xsd:complexType>
</xsd:schema>
```

Appendix B

SuperPeer Data Persistence Implementation's Details

B.1 Technologies

To realize the implementation of the XML-based backend for the SuperPeer application we took advantage of the XPath, XUpdate, XQuery languages, and of the XML native database eXist. Next, we provide an overview of these and how they have been used in the development of the Federation System.

B.1.1 XPath

The *XML Path Language* (XPath) is a non-XML syntax for addressing portions of an XML document. To put is simple, an XML document is a tree made up of nodes. Some nodes have children, and one root node ultimately is ancestor of all the other nodes. XPath allows the compositions of queries that point to particular nodes in the XML document's tree, indicates nodes by position, relative position, type, content, and several other criteria. XPath expressions can also represent numbers, strings, or Booleans, providing support for simple arithmetic and string manipulation.

The most recent version of XPath is the 2.0 one [11]. The version 2.0 of the standard is not a mere improvement of the previous releases, but it is the outcome of the W3C XSL and XML Query Working Groups working closely together at sharing as much as possible between XSLT 2.0 and XQuery 1.0. The resulting set of specifications has name "XPath 2.0". Technically speaking, XPath 2.0 is a strict syntactic subset of XQuery 1.0.

XPath 2.0 is ultimately a language for processing sequences of data, may they be nodes, strings, boolean values, or similar. Indeed, all the possible expressions in XPath 2.0 return sequences, which may contain duplicates. However, sequence of sequences are not allowed. Iteration over sequences are provided by the **for** operator, which returns a new value for each member in the argument sequence. Moreover, XPath 2.0 adds the support to conditional expressions, as well as intersections, differences, unions of sequences. Finally, comments are supported, though with the rather unusual delimiters *(:* and *:)*.

B.1.2 XUpdate

XUpdate [49] is a product of the XML:DB[1] group. It is an XML-based host language that makes extensive use of XPath expressions, and therefore is somewhat similar to XSTL, tailored for update tasks (which may go beyond XML Databases).

The features provided by XUpdate comprise, among the others, the following use cases:

- inserting elements, attributes, text, processing instructions and comments into existing nodes
- appending nodes into the hierarchy of the document
- updating the content of existing nodes
- removal of nodes
- renaming nodes

Despite the lack of features such as the possibility of substituting entire nodes (the update operation allows the modification of the content of a node, not the node itself), XUpdate has over a dozen implementations, several of which can be used independently from XML Databases. Among the various implementations, the followings are notably the most relevant for the Federation System Project:

- XML:DB XUpdate[2]: reference XUpdate implementation in Java
- eXist: XML DBMS in Java, examined more in detail in Section B.1.4

B.1.3 XQuery

XQuery is a language designed for processing XML data of several types, such as files in XML format, but also other data including databases whose structure is similar to XML. The basic building block of XQuery is the expression, which is a sequence of Unicode characters compliant to the XQuery BNF syntax, presented in [12]. More technically, XQuery 1.0 is an extension of XPath Version 2.0 (B.1.1). As in XPath 2.0, every result of an XQuery query is a sequence. XQuery is a strongly typed, side-effect free, functional programming language supporting, among the many features, iterations over sequences, declarations of functions and modules (that are aggregates of functions, and therefore almost equivalent to libraries for common programming languages), sorting of sequences, declarations of contexts and type specification.

One of the most relevant programming constructs of the XQuery language is the *FOR, LET, WHERE, ORDER BY, RETURN* (FLWOR), that integrates the XPath expression syntax addressing specific parts of an XML document,

[1] `http://xmldb-org.sourceforge.net/`
[2] `http://sourceforge.net/projects/xmldb-org/`

with an SQL-like functionality for performing joins. Syntax allowing construction of new XML documents is also available. An XML-like syntax can be used in the case the names of elements and attributes to be constructed are known in advance. Otherwise, built-in expressions, called dynamic node constructors, are available. All these constructs are defined as expressions within the language, and can be arbitrarily nested. XQuery 1.0 does not address features for updating XML documents or databases, as is done by XUpdate B.1.2, even though many implementations, such as the eXist XML Database's one B.1.4, fill this lack by extending the core language with ad-hoc functions.

Nowadays, the support of XQuery is a feature shared by almost all the XML Databases available. Moreover, several vendors, such as IBM, Oracle and Microsoft, are providing support for it on top of their relational databases, thanks to the increasing native support of XML data structures in the most widely adopted DBMSs.

B.1.4 eXist

The eXist is an Open Source native XML Database written in Java and released under the terms of the GNU Lesser General Public License (LGPL). It is available as both a standalone server (integrating the light-weight Jetty servlet container), or as a Web Application. The first versions of eXist used a relational database back end, but the current version is a native XML database optimized to handle XML. Queries are accelerated with the help of indexes that eXist invisibly creates for its important internal components. To avoid memory-intensive document tree traversals during query processing, eXist uses an efficient index structure which is based on a numerical indexing scheme for identifying XML nodes in the index. The indexes are based on B+trees for elements and words, collections (for mapping collection paths to collection objects) and DOM objects (for rapid location of a document's nodes).

Though programming access is available through the XML:DB Java API, and both the XPath and XUpdate standard services are supported, XQuery plays a central role in eXist. In fact, both the XPath and XUpdate queries submitted to the eXist database are actually mapped on XQuery ones. This is possible due to the extensions that eXist realizes on top of XQuery, enhancing the latter with the functionalities provided by XUpdate, that is the capability of modifying, removing, inserting and appending the nodes of the documents contained in the collections.

B.2 Queries

Next we present the queries that realize the requirements specified in Section 4.1.2. The queries that do not require modification of the database are implemented as monolithic XQuery expressions. Otherwise, in order to avoid subtle bugs affecting the eXist's update-enabling extensions to the XQuery, many of the queries that involve modifications of the database are split into two: a XQuery expression verifying the input values and the constraints on the database, and a XUpdate query actually carrying out the modifications. Such

a couple of XQuery - XUpdate queries are executed subsequently by the XML Database, as if they were a single XQuery expression via a synchronization mechanism in the code of the SuperPeer application. In fact, eXist unfortunately lacks of the support to the transactions and locking mechanisms that are foreseen in the XML:DB API. Thus the need of preventing concurrent and inconsistent modifications in the client application, instead that having them handled by the XML Database.

The names of the input variables in the XQuery expressions are systematized by the following convention: the input parameters passed to the query are represented as the variables *$v[n]*, where *[n]* is their index number, starting from 0. That is, the first parameter passed to the query is represented with the variable *$v0*, the second as the variable *$v1*, and so on. In the XQueries, the first parameter *$v0* is always the name of the XML Database's collection that is inquired.

In case that a query is split into a XQuery expression and a XUpdate query, both are listed.

Authenticate Peer

Description: given a PeerIdentifier and a Credentials, it is returned the PeerDescription identified by that PeerIdentifier if the associated Credentials match, otherwise the query fails. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";
declare namespace fdl_api =
  "urn:federation-org:fdl:api";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := document($v0)/fddb:Database
let $peerIdentifierData := $v1
let $credentialsData := $v2
let $peer :=
  $database/fddb:Peers/fddb:Peer
  [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier) =
  $peerIdentifierData]
let $peerCredentialsData := data($peer/fdl_api:Credentials)

return
if ( empty($peerIdentifierData) )
  then fn:error( fn:QName($errorNamespace,
    "NoPeerIdentifierSet") )
  else

if ( empty($credentialsData) )
  then fn:error( fn:QName($errorNamespace,
    "NoCredentialsSet") )
  else

if ( empty($peer) )
  then fn:error( fn:QName($errorNamespace, "NoPeerFound") )
  else

if ( empty($peerCredentialsData) )
  then fn:error( fn:QName($errorNamespace,
    "NoCredentialsOnPeer") )
  else

if ( not ($credentialsData = $peerCredentialsData) )
  then fn:error( fn:QName($errorNamespace,
    "WrongCredentials") )
```

```
  else $peer/fdl_base:PeerDescription
```

Authenticate Publication Owner

Description: given a PeerIdentifier and a PublicationIdentifier, the query succeeds only if exists a Publication with the specified PublicationIdentifier, which belongs to the Peer identified by the specified PeerIdentifier; otherwise the query fails. Upon success, the PeerDescription corresponding to the PeerIdentifier is returned. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := document($v0)/fddb:Database
let $peerIdentifierData := $v1
let $publicationId := $v2
let $peer := $database/fddb:Peers/fddb:Peer
  [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier) =
  $peerIdentifierData]

return
if ( empty($peerIdentifierData) )
  then fn:error( fn:QName($errorNamespace,
    "NoPeerIdentifierSet") )
  else

if ( empty($publicationId) )
  then fn:error( fn:QName($errorNamespace,
    "NoPublicationIdentifierSet") )
  else

if ( empty($peer) )
  then fn:error( fn:QName($errorNamespace,
    "NoPeerFound") )
  else

if ( not ($peer/fddb:Publications/fdl_base:Publication
  [data(fdl_base:PublicationIdentifier) =
  $publicationId]) )
  then fn:error( fn:QName($errorNamespace,
    "PeerNotOwner") )
  else $peer/fdl_base:PeerDescription
```

Authenticate Subscription Owner

Description: given a SubscriptionIdentifier and a PeerIdentifier, the query succeeds only if exists a Subscription with the specified SubscriptionIdentifier, which belongs to the Peer identified by the specified PeerIdentifier; otherwise the query fails. Upon success, the PeerDescription corresponding to the PeerIdentifier is returned. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := document($v0)/fddb:Database
let $peerIdentifierData := $v1
let $subscriptionId := $v2
let $peer := $database/fddb:Peers/fddb:Peer
  [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier) =
  $peerIdentifierData]

return
if ( empty($peerIdentifierData) )
  then fn:error( fn:QName($errorNamespace,
    "NoPeerIdentifierSet") )
  else

if ( empty($subscriptionId) )
  then fn:error( fn:QName($errorNamespace,
    "NoSubscriptionIdentifierSet") )
  else

if ( empty($peer) )
  then fn:error( fn:QName($errorNamespace,
    "NoPeerFound") )
  else

if ( not ($peer/fddb:Subscriptions/fdl_base:Subscription
  [data(fdl_base:SubscriptionIdentifier) =
  $subscriptionId]) )
  then fn:error( fn:QName($errorNamespace,
    "PeerNotOwner") )
  else $peer/fdl_base:PeerDescription
```

Get CanonicalServiceDeclaration

Description: taking in input a CanonicalServiceDeclarationIdentifier, it is returned the CanonicalServiceDeclaration identified by the former. The query fails if no CanonicalServiceDeclaration is found with the given CanonicalServiceDeclarationIdentifier. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $database := document($v0)/fddb:Database
let $csds :=
  $database/fddb:CanonicalServiceDeclarations

return $csds/fdl_base:CanonicalServiceDeclaration
  [data(fdl_base:CanonicalServiceDeclarationIdentifier)
  = $v1]
```

Get PeerDescription

Description: given a PeerIdentifier, the PeerDescription identified by the former is returned . The query fails if no PeerDescription is found with the given PeerIdentifier. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $database := document($v0)/fddb:Database
let $peers := $database/fddb:Peers/fddb:Peer

for $x in $peers/fdl_base:PeerDescription
where $x[data(fdl_base:PeerIdentifier) = $v1]
return $x
```

Get Publication

Description: given a PublicationIdentifier, it is returned the Publication identified by the former. The query fails if no Publication is found with the given PublicationIdentifier. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $database := document($v0)/fddb:Database
let $peers := $database/fddb:Peers/fddb:Peer

for $x in $peers/fddb:Publications/fdl_base:Publication
where $x[data(fdl_base:PublicationIdentifier) = $v1]
return $x
```

Get Publications by Peer

Description: given a PeerIdentifier, all the Publications belonging to the Peer identified by the specified PeerIdentifier are returned. If no Peer with the given PeerIdentifier is found, then the query fails. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $database := document($v0)/fddb:Database
let $peers := $database/fddb:Peers/fddb:Peer

for $x in $peers
where $x
  [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier)
  = $v1]
return $x/fddb:Publications/fdl_base:Publication
```

Get Publishers by Publications

Description: given set of PublicationIdentifiers, are returned all the PeerDescriptions describing the Peers owning one or more of the Publications identified by the given PublicationIdentifiers. The query fails if any of the PublicationIdentifiers specified does not correspond to an existing Publication. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";
declare namespace fdl_uddi =
  "urn:federation-org:fdl:ext:uddi";
declare namespace fn =
  "http://www.w3.org/2003/05/xpath-functions";
declare namespace util =
  "http://exist-db.org/xquery/util";

declare function
  local:unmarshallElements($marshalledNodes as xs:string*)
  as element()*
{
let $res := ( for $string in $marshalledNodes
  return util:eval( $string ) )
return ( $res )
};

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := document($v0)/fddb:Database
let $pDescs :=
  $database/fddb:Peers/fddb:Peer/fdl_base:PeerDescription
let $marshalledPublications := $v1
let $publications := local:unmarshallElements($v1)
let $peerDescriptions :=
  $database/fddb:Peers/fddb:Peer/fdl_base:PeerDescription
let $peerPublisherIdentifiersData :=
  data($publications/fdl_base:PublisherPeer)

let $publishers := ( for $peerPublisherIdData
  in $peerPublisherIdentifiersData
return $peerDescriptions[data(fdl_base:PeerIdentifier) =
  $peerPublisherIdData] )

return ( $publishers )
```

Get Subscription

Description: given a SubscriptionIdentifier, it is returned the Subscription identified by the former. The query fails if no Subscription is found with the given SubscriptionIdentifier. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $database := document($v0)/fddb:Database
let $peers := $database/fddb:Peers/fddb:Peer

for $x in $peers/fddb:Subscriptions/fdl_base:Subscription
where $x[data(fdl_base:SubscriptionIdentifier) = $v1]
return $x
```

Get Matching CanonicalServiceDeclarations

Description: given a set of ServiceDescriptions and ServiceDefinitions, it returns all the matching CanonicalServiceDeclarations. *XQuery*:

```
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";
declare namespace
  fdl_uddi = "urn:federation-org:fdl:ext:uddi";
declare namespace
  fn = "http://www.w3.org/2003/05/xpath-functions";
declare namespace
  util = "http://exist-db.org/xquery/util";

declare function
  local:getMatchingCanonicalServiceDeclarations
  ($csds as element(fdl_base:CanonicalServiceDeclaration)*,
  $serviceDescriptions as element()*,
  $serviceDefinitions as element()*) as element()*
{
let $matchingCsds := (
  for $csd in $csds
  where local:matchCanonicalServiceDeclaration
    ($csd, $serviceDescriptions, $serviceDefinitions)
  return ( $csd ) )
return ( $matchingCsds )
};

declare function local:matchCanonicalServiceDeclaration
  ($csd as element(fdl_base:CanonicalServiceDeclaration),
  $serviceDescriptions as element()*,
  $serviceDefinitions as element()*) as xs:boolean
{
let $csdServiceDescriptionIDsData :=
  data( $csd//fdl_base:ServiceDescriptionIdentifier )
let $csdServiceDefinitionIDsData :=
  data( $csd//fdl_base:ServiceDefinitionIdentifier )

let $serviceDescriptionIDsData :=
  data($serviceDescriptions//fdl_base:ServiceDescriptionIdentifier)
let $serviceDefinitionIDsData :=
  data($serviceDefinitions//fdl_base:ServiceDefinitionIdentifier)

let $matchesSDescs := if ( empty( $serviceDescriptionIDsData ) )
  then ( true() )
  else ( fn:contains( $csdServiceDescriptionIDsData,
    $serviceDescriptionIDsData ) )
let $matchesSDefs :=  if ( empty( $serviceDefinitionIDsData ) )
  then ( true() )
  else ( fn:contains( $csdServiceDefinitionIDsData,
    $serviceDefinitionIDsData ) )
```

```
let $res := $matchesSDescs and $matchesSDefs

return ( $res )
};

declare function local:unmarshallElements
  ($marshalledNodes as xs:string*) as element()*
{
let $res := (
  for $string in $marshalledNodes
  return util:eval( $string ) )
return ( $res )
};

(:
let $v0 := "/federation/database"
let $v1 := ''
let $v2 := ''
:)

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := document($v0)/fddb:Database
let $csds :=
  $database/fddb:CanonicalServiceDeclarations
  /fdl_base:CanonicalServiceDeclaration
let $marshalledServiceDescriptions := $v1
let $marshalledServiceDefinitions := $v2

let $matchingCsds :=
  local:getMatchingCanonicalServiceDeclarations( $csds,
  local:unmarshallElements( $marshalledServiceDescriptions ),
  local:unmarshallElements( $marshalledServiceDefinitions ) )

return ( $matchingCsds )
```

Get Matching PeerDescriptions

Description: given a BusinessDescriptions, it returns all the matching PeerDescriptions. *XQuery*:

```
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";
declare namespace
  fdl_uddi = "urn:federation-org:fdl:ext:uddi";
declare namespace
  fn = "http://www.w3.org/2003/05/xpath-functions";
declare namespace
  util = "http://exist-db.org/xquery/util";

declare function local:getMatchingPeerDescriptions
  ($pDescs as element(fdl_base:PeerDescription)*,
  $businessDescriptions as element()*) as element()*
{
let $matchingPDescs := (
  for $pDesc in $pDescs
  where local:matchPeerDescription($pDesc, $businessDescriptions)
  return ( $pDesc ) )
return ( $matchingPDescs )
};

declare function local:matchPeerDescription
  ($pDesc as element(fdl_base:PeerDescription),
  $businessDescriptions as element()*) as xs:boolean
{
let $pDescBusinessDescriptionIDsData :=
  data( $pDesc//fdl_base:BusinessDescriptionIdentifier )

let $businessDescriptionIDsData :=
  data( $businessDescriptions//fdl_base:BusinessDescriptionIdentifier )

let $res := ( fn:contains( $pDescBusinessDescriptionIDsData,
  $businessDescriptionIDsData ) )

return ( $res )
};

declare function local:unmarshallElements
  ($marshalledNodes as xs:string*) as element()*
{
let $res := (
  for $string in $marshalledNodes
  return util:eval( $string ) )
return ( $res )
};

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"
```

```
let $database := document($v0)/fddb:Database
let $pDescs :=
  $database/fddb:Peers/fddb:Peer/fdl_base:PeerDescription
let $marshalledBusinessDescriptions := $v1

let $matchingPDescs := local:getMatchingPeerDescriptions
  ( $pDescs,
  local:unmarshallElements( $marshalledBusinessDescriptions ) )

return ( $matchingPDescs )
```

Get Matching Publications

Description: given a CanonicalServiceDeclarationIdentifier, and optionally a set of ServiceDescriptions and ServiceDefinitions, it returns all the matching Publications. The query fails if it does not exist a CanonicalServiceDeclaration identified by the specified CanonicalServiceDeclarationIdentifier *XQuery*:

```
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";
declare namespace
  fdl_uddi = "urn:federation-org:fdl:ext:uddi";
declare namespace
  fn = "http://www.w3.org/2003/05/xpath-functions";
declare namespace
  util = "http://exist-db.org/xquery/util";

declare function local:getMatchingPublications
  ($pubs as element(fdl_base:Publication)*,
  $csdId as element(fdl_base:CanonicalServiceDeclarationIdentifier),
  $serviceDescriptions as element()*,
  $serviceDefinitions as element()*) as element()*
{
let $matchingCsds := (
  for $pub in $pubs
  where local:matchPublication($pub, $csdId, $serviceDescriptions,
    $serviceDefinitions)
  return ( $pub ) )
return ( $matchingCsds )
};

declare function local:matchPublication
  ($pub as element(fdl_base:Publication),
  $csdId as element(fdl_base:CanonicalServiceDeclarationIdentifier),
  $serviceDescriptions as element()*,
  $serviceDefinitions as element()*) as xs:boolean
{
let $pubCanonicalServiceDeclarationIdData :=
  data( $pub/fdl_base:CanonicalServiceDeclarationIdentifier )
let $pubServiceDescriptionIDsData :=
  data( $pub//fdl_base:ServiceDescriptionIdentifier )
let $pubServiceDefinitionIDsData :=
  data( $pub//fdl_base:ServiceDefinitionIdentifier )

let $csdIdData := data( $csdId )
let $serviceDescriptionIDsData :=
  data( $serviceDescriptions//fdl_base:ServiceDescriptionIdentifier )
let $serviceDefinitionIDsData :=
  data( $serviceDefinitions//fdl_base:ServiceDefinitionIdentifier )

let $matchesCsdId :=
  if ( empty( $csdIdData ) )
    then ( true() )
```

```
    else ( $pubCanonicalServiceDeclarationIdData = $csdIdData )
let $matchesSDescs :=
  if ( empty( $serviceDescriptionIDsData ) )
    then ( true() )
    else ( fn:contains( $pubServiceDescriptionIDsData,
      $serviceDescriptionIDsData ) )
let $matchesSDefs :=
  if ( empty( $serviceDefinitionIDsData ) )
    then ( true() )
    else ( fn:contains( $pubServiceDefinitionIDsData,
      $serviceDefinitionIDsData ) )

let $res :=
  ( $matchesCsdId ) and ( $matchesSDescs ) and ( $matchesSDefs )

return ( $res )
};

declare function local:unmarshallElements
  ($marshalledNodes as xs:string*) as element()*
{
let $res := (
  for $string in $marshalledNodes
  return util:eval( $string ) )
return ( $res )
};

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := document($v0)/fddb:Database
let $pubs :=
  $database/fddb:Peers/fddb:Peer/fddb:Publications/fdl_base:Publication
let $marshalledCSDeclarationIdentifier := $v1
let $marshalledServiceDescriptions := $v2
let $marshalledServiceDefinitions := $v3

let $matchingPubs := local:getMatchingPublications
  ( $pubs,
  local:unmarshallElements( $marshalledCSDeclarationIdentifier ),
  local:unmarshallElements( $marshalledServiceDescriptions ),
  local:unmarshallElements( $marshalledServiceDefinitions ) )

return ( $matchingPubs )
```

Get Matching Subscriptions

Description: given a PublicationIdentifier, are returned all the Subscriptions matching the CanonicalServiceDeclarationIdentifier, ServiceDescriptions and ServiceDefinitions of the Publication identified by the specified PublicationIdentifier. The query fails if it does not exist a Publication identified by the specified PublicationIdentifier. *XQuery*:

```
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";
declare namespace
  fdl_uddi = "urn:federation-org:fdl:ext:uddi";
declare namespace
  fn = "http://www.w3.org/2003/05/xpath-functions";
declare namespace
  util = "http://exist-db.org/xquery/util";

declare function local:getMatchingSubscriptions
  ($subs as element(fdl_base:Subscription)*,
  $csdId as element(fdl_base:CanonicalServiceDeclarationIdentifier),
  $serviceDescriptions as element()*,
  $serviceDefinitions as element()*) as element()*
{
let $matchingCsds := (
  for $sub in $subs
  where local:matchSubscription($sub, $csdId, $serviceDescriptions,
    $serviceDefinitions)
  return ( $sub ) )
return ( $matchingCsds )
};

declare function local:matchSubscription
  ($sub as element(fdl_base:Subscription),
  $csdId as element(fdl_base:CanonicalServiceDeclarationIdentifier),
  $serviceDescriptions as element()*,
  $serviceDefinitions as element()*) as xs:boolean
{
let $subCanonicalServiceDeclarationIdData :=
  data( $sub/fdl_base:CanonicalServiceDeclarationIdentifier )
let $subServiceDescriptionIDsData :=
  data( $sub//fdl_base:ServiceDescriptionIdentifier )
let $subServiceDefinitionIDsData :=
  data( $sub//fdl_base:ServiceDefinitionIdentifier )

let $csdIdData := data( $csdId )
let $serviceDescriptionIDsData :=
  data( $serviceDescriptions//fdl_base:ServiceDescriptionIdentifier )
let $serviceDefinitionIDsData :=
  data( $serviceDefinitions//fdl_base:ServiceDefinitionIdentifier )

let $matchesCsdId :=
  if ( empty( $csdIdData ) )
```

```
    then ( true() )
    else ( $subCanonicalServiceDeclarationIdData = $csdIdData )
let $matchesSDescs :=
  if ( empty( $serviceDescriptionIDsData ) )
    then ( true() )
    else ( fn:contains(
      $subServiceDescriptionIDsData, $serviceDescriptionIDsData ) )
let $matchesSDefs :=
  if ( empty( $serviceDefinitionIDsData ) )
    then ( true() )
    else ( fn:contains( $subServiceDefinitionIDsData,
      $serviceDefinitionIDsData ) )

let $res :=
  ( $matchesCsdId ) and ( $matchesSDescs ) and ( $matchesSDefs )

return ( $res )
};

declare function local:unmarshallElement
  ($marshalledNodes as xs:string) as element()
{
let $res := (
  for $string in $marshalledNodes
  return util:eval( $string ) )
return ( $res )
};

declare function local:getPublicationByPublicationIdentifier
  ( $pubs as element(fdl_base:Publication)*,
  $pubId as element(fdl_base:PublicationIdentifier) )
  as element(fdl_base:Publication)
{
let $matchingPubs :=
  $pubs[data(fdl_base:PublicationIdentifier) = data( $pubId )]
return ( $matchingPubs )
};

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := document($v0)/fddb:Database
let $marshalledPublicationId := $v1
let $pubId := local:unmarshallElement( $marshalledPublicationId )
let $pubs :=
  $database/fddb:Peers/fddb:Peer/fddb:Publications/fdl_base:Publication
let $pub :=
  local:getPublicationByPublicationIdentifier( $pubs, $pubId )

let $canonicalServiceDeclarationIdentifier :=
  $pub/fdl_base:CanonicalServiceDeclarationIdentifier
let $serviceDescriptions :=
  $pub/child::*[fdl_base:ServiceDescriptionIdentifier]
```

```
let $serviceDefinitions :=
  $pub/child::*[fdl_base:ServiceDefinitionIdentifier]

let $peers := $database/fddb:Peers/fddb:Peer
let $subs := $peers/fddb:Subscriptions/fdl_base:Subscription
let $matchingPubs :=
  local:getMatchingSubscriptions ( $subs,
    $canonicalServiceDeclarationIdentifier,
    $serviceDescriptions, $serviceDefinitions )

return ( $matchingPubs )
```

Add CanonicalServiceDeclaration

Description: adds to the database the CanonicalServiceDeclaration specified. The CanonicalServiceDeclarationIdentifier by which the CanonicalServiceDeclaration is identified must exist and not be already present within the database, otherwise the query fails. *XQuery*:

```
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";
declare namespace
  fn = "http://www.w3.org/2003/05/xpath-functions";
declare namespace
  util = "http://exist-db.org/xquery/util";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := document($v0)/fddb:Database
let $csds := $database/fddb:CanonicalServiceDeclarations
let $csdIdData := $v1

return
(: Check $csdIdData exists :)
if ( empty($csdIdData) )
  then fn:error(fn:QName($errorNamespace,
    "NoCanonicalServiceDeclarationIdentifierSet"))
  else

(: Check no CanonicalServiceDeclarationIdentifier duplication :)
if ( count($csds/fdl_base:CanonicalServiceDeclaration
  [fdl_base:CanonicalServiceDeclarationIdentifier = $csdIdData]) > 0 )
  then ( fn:error(fn:QName($errorNamespace,
    "CanonicalServiceDeclarationIdentifierDuplicated")) )
  else 0
```

XUpdate:

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate"
  xmlns:fddb="urn:federation-org:SuperPeer:database:xmldb"
  xmlns:fdl_base="urn:federation-org:fdl:base">
  <xupdate:append
    select="/fddb:Database/fddb:CanonicalServiceDeclarations" >
    { fdl_base:CanonicalServiceDeclaration }
  </xupdate:append>
</xupdate:modifications>
```

Add PeerDescription

Description: adds the specified PeerDescription to the database. The PeerIdentifier that identifies the PeerDescription must exist and not be already present within the database, otherwise the query fails. *XQuery*:

```
declare namespace
  fn = "http://www.w3.org/2003/05/xpath-functions";
declare namespace
  util = "http://exist-db.org/xquery/util";
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $peers :=
  document($v0)/fddb:Database/fddb:Peers
let $peerIdData := $v1

return
(: Check $peerIdData is not the empty sequence :)
if ( fn:empty($peerIdData) )
  then fn:error( fn:QName($errorNamespace,
    "NoPeerIdentifierSet") )
  else
if ( count($peers/fddb:Peer/fdl_base:PeerDescription
    [data(fdl_base:PeerIdentifier) = $peerIdData]) > 0 )
  then ( fn:error(fn:QName($errorNamespace,
    "PeerIdentifierDuplicated")) )
  else 0
```

XUpdate:

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate"
  xmlns:fddb="urn:federation-org:SuperPeer:database:xmldb"
  xmlns:fdl_base="urn:federation-org:fdl:base">
  <xupdate:append
    select="/fddb:Database/fddb:Peers">
    <fddb:Peer>
      { fdl:PeerDescription }
      <fddb:Publications/>
      <fddb:Subscriptions/>
    </fddb:Peer>
  </xupdate:append>
</xupdate:modifications>
```

Add Publication

Description: adds the specified Publication into the database. The PublicationIdentifier that identifies the Publication must exist and not be already present within the database, otherwise the query fails. Moreover, the Publication's CanonicalServiceDeclarationIdentifier must exist and correspond to an existing CanonicalServiceDeclaration. Similarly the PublisherPeer must exist and correspond to a PeerIdentifier identifying a Peer in the database. *XQuery*:

```
declare namespace
```

```
  fn = "http://www.w3.org/2003/05/xpath-functions";
declare namespace
  util = "http://exist-db.org/xquery/util";
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := doc($v0)/fddb:Database
let $peers := $database/fddb:Peers
let $publicationIdData := $v1
let $publisherPeerData := $v2
let $csdIdData := $v3

return
(: Check $publicationIdData exists as a variable :)
if ( empty($publicationIdData) )
  then fn:error( fn:QName($errorNamespace,
    "NoPublicationIdentifierSet") )
  else

(: Check $csdIdData exists as a variable :)
if ( empty($csdIdData) )
  then fn:error( fn:QName($errorNamespace,
    "NoCanonicalServiceDeclarationIdentifierSet") )
  else

(: Check $publisherPeerData exists as a variable :)
if ( empty($publisherPeerData) )
  then fn:error( fn:QName($errorNamespace,
    "NoPublisherPeerSet") )
  else

(: Check does not exist a Publication with
  PublicationIdentifier = $publicationIdData :)
if (
  data($peers/fddb:Publications/fdl_base:Publication\\
    /fdl_base:PublicationIdentifier) = $publicationIdData )
  then fn:error( fn:QName($errorNamespace,
    "PublicationIdentifierDuplicated") )
  else

(: Check exists a Peer with PeerIdentifier =
  $publisherPeerData :)
if ( not ( data($peers/fddb:Peer/fdl_base:PeerDescription\\
  /fdl_base:PeerIdentifier) = $publisherPeerData ) )
  then fn:error( fn:QName($errorNamespace,
    "NoPublisherPeerFound") )
  else
```

```
(: Check exists a Csd with
  CanonicalServiceDeclarationIdentifier = $csdIdData :)
if ( not ( data($database/fddb:CanonicalServiceDeclarations/\\
  fdl_base:CanonicalServiceDeclaration/\\
  fdl_base:CanonicalServiceDeclarationIdentifier) = $csdIdData ) )
  then fn:error( fn:QName($errorNamespace,
    "NoCanonicalServiceDeclarationFound") )
  else 0
```

XUpdate:

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate"
  xmlns:fddb="urn:federation-org:SuperPeer:database:xmldb"
  xmlns:fdl_base="urn:federation-org:fdl:base">
  <xupdate:append
    select="/fddb:Database/fddb:Peers/fddb:Peer
    [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier)
    = { fdl_base:PeerIdentifier }]/fddb:Publications">
    { fdl_base:Publication }
  </xupdate:append>
</xupdate:modifications>
```

Add Subscription

Description: adds the specified Subscription into the database. The SubscriptionIdentifier that identifies the Subscription must exist and not be already present within the database, otherwise the query fails. Moreover, the Subscription's CanonicalServiceDeclarationIdentifier must exist and correspond to an existing CanonicalServiceDeclaration. Similarly the SubscriberPeer must exist and correspond to a PeerIdentifier identifying a Peer in the database. *XQuery*:

```
declare namespace
  fn = "http://www.w3.org/2003/05/xpath-functions";
declare namespace
  util = "http://exist-db.org/xquery/util";
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := doc($v0)/fddb:Database
let $peers := $database/fddb:Peers
let $subscriptionIdData := $v1
let $subscriberPeerData := $v2
let $csdIdData := $v3

return
(: Check $subscriptionIdData exists as a variable :)
if ( empty($subscriptionIdData) )
  then fn:error( fn:QName($errorNamespace,
```

```
    "NoSubscriptionIdentifierSet") )
  else

(: Check $csdIdData exists as a variable :)
if ( empty($csdIdData) )
  then fn:error( fn:QName($errorNamespace,
    "NoCanonicalServiceDeclarationIdentifierSet") )
  else

(: Check $subscriberPeerData exists as a variable :)
if ( empty($subscriberPeerData) )
  then fn:error( fn:QName($errorNamespace,
    "NoSubscriberPeerSet") )
  else

(: Check does not exist a Subscription with
  SubscriptionIdentifier = $subscriptionIdData :)
if ( data($peers/fddb:Subscriptions/fdl_base:Subscription\\
  /fdl_base:SubscriptionIdentifier) = $subscriptionIdData )
  then fn:error( fn:QName($errorNamespace,
    "SubscriptionIdentifierDuplicated") )
  else

(: Check exists a Peer with PeerIdentifier
  = $subscriberPeerData :)
if ( not ( data($peers/fddb:Peer/fdl_base:PeerDescription\\
  /fdl_base:PeerIdentifier) = $subscriberPeerData ) )
  then fn:error( fn:QName($errorNamespace,
    "NoSubscriberPeerFound") )
  else

(: Check exists a Csd with
  CanonicalServiceDeclarationIdentifier = $csdIdData :)
if ( not ( data($database/fddb:CanonicalServiceDeclarations/\\
  fdl_base:CanonicalServiceDeclaration/\\
  fdl_base:CanonicalServiceDeclarationIdentifier) = $csdIdData ) )
  then fn:error( fn:QName($errorNamespace,
    "NoCanonicalServiceDeclarationFound") )
  else 0
```

XUpdate:

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate"
  xmlns:fddb="urn:federation-org:SuperPeer:database:xmldb"
  xmlns:fdl_base="urn:federation-org:fdl:base">
  <xupdate:append
    select="/fddb:Database/fddb:Peers/fddb:Peer
    [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier)
    = { fdl_base:PeerIdentifier }]/fddb:Subscriptions">
    { fdl_base:Subscription }
  </xupdate:append>
</xupdate:modifications>
```

Modify CanonicalServiceDeclaration

Description: substitutes the ServiceDeclarations and ServiceDefinitions of the specified CanonicalServiceDeclaration. If it does not exist a CanonicalServiceDeclaration identified by the CanonicalServiceDeclarationIdentifier comprised within the specified the CanonicalServiceDeclaration the query fails. *XUpdate*:

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate"
  xmlns:fddb="urn:federation-org:SuperPeer:database:xmldb"
  xmlns:fdl_base="urn:federation-org:fdl:base">
  <xupdate:update
    select="/fddb:Database/fddb:CanonicalServiceDeclarations\\
    /fdl_base:CanonicalServiceDeclaration\\
    [data(fdl_base:CanonicalServiceDeclarationIdentifier)
    = { fdl_base:CanonicalServiceDeclarationIdentifier }]">
    { fdl_base:CanonicalServiceDeclarationIdentifier }
    { fdl_base:ServiceDescriptions }
    { fdl_base:ServiceDefinitions }
  </xupdate:update>
</xupdate:modifications>
```

Modify PeerDescription

Description: substitutes the BusinessDescription of the specified PeerDescription. If it does not exist a PeerDescription identified by the PeerIdentifier comprised within the specified the PeerDescription the query fails. *XUpdate*:

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate"
  xmlns:fddb="urn:federation-org:SuperPeer:database:xmldb"
  xmlns:fdl_base="urn:federation-org:fdl:base">
  <xupdate:update
    select="/fddb:Database/fddb:Peers/fddb:Peer\\
    /fdl_base:PeerDescription\\
    [data(fdl_base:PeerIdentifier)
    = { fdl_base:PeerIdentifier }]">
    { fdl_base:PeerIdentifier }
    { fdl_base:NotificationServiceLocation }
    { fdl_base:PublicServiceLocation }
    { fdl_base:BusinessDescriptions }
  </xupdate:update>
</xupdate:modifications>
```

Modify Publication

Description: substitutes the ServiceDeclarations and ServiceDefinitions of the specified Publication. If it does not exist a Publication identified by the PublicationIdentifier comprised within the specified the Publication the query fails. Neither the CanonicalServiceDeclarationIdentifier nor the PublisherPeer of the involved Publication are affected by the modifications. *XQuery*:

```
declare namespace
  fn = "http://www.w3.org/2003/05/xpath-functions";
```

```
declare namespace
  util = "http://exist-db.org/xquery/util";
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := doc($v0)/fddb:Database
let $peers := $database/fddb:Peers
let $publicationIdData := $v1
let $publisherPeerData := $v2
let $csdIdData := $v3
let $publicationOld :=
  $peers/fddb:Peer
    [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier) =
    $publisherPeerData]/fddb:Publications/fdl_base:Publication
    [data(fdl_base:PublicationIdentifier) =  $publicationIdData]

return
(: Check $publicationOld exists as a variable :)
if ( empty($publicationOld) )
  then fn:error( fn:QName($errorNamespace,
    "NoMatchingPreviousPublication") )
  else

(: Check if it does not change the
  CanonicalServiceDeclarationIdentifier between the two versions :)
if ( not (
  data($publicationOld/\\
    fdl_base:CanonicalServiceDeclarationIdentifier) = $csdIdData ) )
  then fn:error( fn:QName($errorNamespace,
    "MismatchingCanonicalServiceDeclarationIdentifier") )
  else 0
```

XUpdate:

```
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate"
  xmlns:fddb="urn:federation-org:SuperPeer:database:xmldb"
  xmlns:fdl_base="urn:federation-org:fdl:base">
  <xupdate:update
    select="/fddb:Database/fddb:Peers/fddb:Peer\\
    /fdl_base:PeerDescription\\
    [data(fdl_base:PeerIdentifier)
    = { fdl_base:PeerIdentifier }]/fddb:Publications\\
    /fdl_base:Publication[data(fdl_base:PublicationIdentifier)
    = { fdl_base:PublicationIdentifier } ]">
    { fdl_base:PublicationIdentifier }
    { fdl_base:PublisherPeer }
    { fdl_base:CanonicalServiceDeclarationIdentifier }
    { fdl_base:ServiceDescriptions }
```

```
    { fdl_base:ServiceDefinitions }
  </xupdate:update>
</xupdate:modifications>
```

Modify Subscription

Description: substitutes the ServiceDeclarations and ServiceDefinitions of the specified Subscription. If it does not exist a Subscription identified by the SubscriptionIdentifier comprised within the specified the Subscription the query fails. Neither the CanonicalServiceDeclarationIdentifier nor the SubscriberPeer of the involved Publication are affected by the modifications. *XQuery*:

```
declare namespace
  fn = "http://www.w3.org/2003/05/xpath-functions";
declare namespace
  util = "http://exist-db.org/xquery/util";
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := doc($v0)/fddb:Database
let $peers := $database/fddb:Peers
let $subscriptionIdData := $v1
let $subscriberPeerData := $v2
let $csdIdData := $v3
let $subscriptionOld :=
  $peers/fddb:Peer
  [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier) =
  $subscriberPeerData]/fddb:Subscriptions/fdl_base:Subscription
  [data(fdl_base:SubscriptionIdentifier) =  $subscriptionIdData]

return
(: Check $subscriptionIdData exists as a variable :)
if ( empty($subscriptionOld) )
  then fn:error( fn:QName($errorNamespace,
    "NoMatchingPreviousSubscription") )
  else

(: Check if it does not change the
  CanonicalServiceDeclarationIdentifier between the two versions :)
if ( not (
  data($subscriptionOld/fdl_base:CanonicalServiceDeclarationIdentifier)
    = $csdIdData ) )
  then fn:error( fn:QName($errorNamespace,
    "MismatchingCanonicalServiceDeclarationIdentifier") )
  else 0
```

XUpdate:

```
<xupdate:modifications version="1.0"
```

```
  xmlns:xupdate="http://www.xmldb.org/xupdate"
  xmlns:fddb="urn:federation-org:SuperPeer:database:xmldb"
  xmlns:fdl_base="urn:federation-org:fdl:base">
  <xupdate:update
    select="/fddb:Database/fddb:Peers/fddb:Peer\\
    /fdl_base:PeerDescription\\
    [data(fdl_base:PeerIdentifier)
    = { fdl_base:PeerIdentifier }]/fddb:Subscriptions\\
    /fdl_base:Subscription[data(fdl_base:SubscriptionIdentifier)
    = { fdl_base:SubscriptionIdentifier } ]">
    { fdl_base:SubscriptionIdentifier }
    { fdl_base:SubscriberPeer }
    { fdl_base:CanonicalServiceDeclarationIdentifier }
    { fdl_base:ServiceDescriptions }
    { fdl_base:ServiceDefinitions }
  </xupdate:update>
</xupdate:modifications>
```

Remove CanonicalServiceDeclaration

Description: given a CanonicalServiceDeclarationIdentifier, removes the CanonicalServiceDeclaration identified by the former. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $database := document($v0)/fddb:Database

for $x in $database/fddb:CanonicalServiceDeclarations\\
  /fdl_base:CanonicalServiceDeclaration
where $x
  [data(fdl_base:CanonicalServiceDeclarationIdentifier) = $v1]
return update delete $x
```

Remove PeerDescription

Description: given a PeerIdentifier, removes the PeerDescription identified by the former. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =\\
  "urn:federation-org:fdl:base";

let $database := document($v0)/fddb:Database

for $x in $database/fddb:Peers/fddb:Peer
where $x
  [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier)
  = $v1]
return update delete $x
```

Remove Publication

Description: given a PublicationIdentifier, removes the Publication identified by the former. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $database := document($v0)/fddb:Database
let $peers := $database/fddb:Peers/fddb:Peer
let $publications :=
  $peers/fddb:Publications/fdl_base:Publication

for $x in $publications
where $x
  [data(fdl_base:PublicationIdentifier) = $v1]
return update delete $x
```

Remove Subscription

Description: given a SubscriptionIdentifier, removes the Subscription identified by the former. *XQuery*:

```
declare namespace fddb =
  "urn:federation-org:SuperPeer:database:xmldb";
declare namespace fdl_base =
  "urn:federation-org:fdl:base";

let $database := document($v0)/fddb:Database

for $x in $database/fddb:Peers/fddb:Peer/\\
  fddb:Subscriptions/fdl_base:Subscription
where $x
  [data(fdl_base:SubscriptionIdentifier) = $v1]
return update delete $x
```

Remove All Subscription by Peer

Description: given a PeerIdentifier, removes all the Subscriptions owned by the Peer identified by the specified PeerIdentifier. The query fails if it does not exist a Peer identified by the specified PeerIdentifier. *XQuery*:

```
declare namespace
  fddb = "urn:federation-org:SuperPeer:database:xmldb";
declare namespace
  fdl_base = "urn:federation-org:fdl:base";

let $errorNamespace :=
  "urn:federation-org:SuperPeer:database:xmldb:errors"

let $database := document($v0)/fddb:Database
let $peerIdentifierData := $v1
```

```
let $peer := $database/fddb:Peers/fddb:Peer
  [data(fdl_base:PeerDescription/fdl_base:PeerIdentifier) = $v1]
let $subscriptions :=
  $peer/fddb:Subscriptions/fdl_base:Subscription
let $subscriptionIdentifiers :=
  $subscriptions/fdl_base:SubscriptionIdentifier

update remove $subscriptions

return $subscriptionIdentifiers
```

Ringraziamenti

In virtú dell'indispensabile apporto (e supporto) fornito all'autore durante la stesura della ivi presente Tesi Specialistica, é doveroso ringraziare:

- Il Ghetto: per aver evitato che loschi figuri che parlano solo per anagrammi gettassero nel panico gli studenti delle facoltá normali.
- Il segnale "Pericolo: Materiale Radioattivo" all'Interno 3: durante quest'anno ha evitato la morte di innumerevoli incauti curiosi, nonché svariati periodi di quarantena al Taramelli causa fuga materiale tossico.
- Le ruote lisce della Micra: per non averci mai fatto uscire di strada, nonostante gli accorati e profondamente sentiti tentativi del suo guidatore.
- L'orticello: per aver tenuto viva la speranza di Mamma Marisa di aver i meritatissimi nipotini in questa vita.
- Il risotto col TastaSal e i "Quattro Salti in Padella: Burro al Burro con Burro aggiunto": per aver sostentato e mantenuto in salute gli Ale, ed evitare che cibi troppo sani nuocessero al delicato metabolismo dell'Esemplare Maschio.
- I mortai con alzo 90 °: superfluo specificare le motivazioni.
- Il televisore dell'Interno 3: il solo televisore al mondo che, causa morte prematura del colore verde nel tubo catodico, dá la sensazione che tutte le partite di calcio siano giocate su campi di terra battuta.
- L'alpino Taramello Taramelli: le sue perle di saggezza mi accompagneranno per tutta la vita, indicandomi la via nei momenti bui.
- La Bettola: la peperonata il 15 Agosto dá la carica giusta per affrontare una dura e spossante giornata di lavoro!

Questi sopra riportati sono i ringraziamenti "per scherzare", componente di rito di ogni Tesi che si rispetti. Ma, in via del tutto eccezionale, questa Tesi riporterá dei veri ringraziamenti, rivolti a persone che veramente hanno contribuito a rendere possibile il completamento della mia Laurea Specialistica:

- Il mio Relatore, Dott. Aiello Marco: si é sempre rivelata una persona su cui fare cieco affidamento, sempre presente quando necessario, e sempre in

grado di dare il consiglio giusto al momento giusto. Per di piú, é riuscito a sopportarmi per quasi 4 anni, ed ha letto e corretto e riletto ancora questa noia mortale di Tesi ... Io dico, hanno canonizzato gente per meno! Santo subito.

- La mia famiglia: non mi hanno mai fatto mancare l'appoggio e il supporto (finanziaro e non), necessario a fare quello che ho fatto. certo, ci mettono del loro a rendermi la vita complicata, ma tirando le somme sono in netto credito.
- I miei amici piú cari: pochi, selezionati, soprattutto in virtú della loro infinita pazienza. Perché sono ben conscio che quando mi ci metto (e anche quando non mi ci metto ...), ci vuole davvero impegno per non prendermi a badilate sul coppino.

A tutti, un sentito grazie

Bibliography

[1] Rohit Aggarwal, Kunal Verma, John A. Miller, and William Milnor. Constraint Driven Web Service Composition in METEOR-S. In *IEEE Service Computing Conference (SCC)*, pages 23–30, 2004.

[2] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth, and Kunal Verma. *Web Service Semantics (WSDL-S) 1.0*, November 2005. http://www.w3.org/Submission/WSDL-S/.

[3] Rashid J. Al-Ali, Ali ShaikhAli, Omer F. Rana, and David W. Walker. Supporting QoS-Based Discovery in Service-Oriented Grids. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, page 101.2, 2003.

[4] Anupriya Ankolekar, David Martin, Deborah McGuinness, Sheila A. McIlraith, Massimo Paolucci, and Bijan Parsia. *OWL-S' Relationship to Selected Other Technologies*. World Wide Web Consortium, November 2004. http://www.w3.org/Submission/OWL-S-related.

[5] Anupriya Ankolenkar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne, and Katia P. Sycara. DAML-S: Web Service Description for the Semantic Web. In *Proceedings of The First International Semantic Web Conference (ISWC)*, pages 348–363, 2002.

[6] Keith Ballinger, David Ehnebuske, Christopher Ferris, Martin Gudgin, Canyang K. Liu, Mark Nottingham, and Prasad Yendluri. *Basic Profile Version 1.1*. The Web Services-Interoperability Organization, August 2004. http://www.ws-i.org/Profiles/BasicProfile-1.1.html.

[7] Sujata Banerjee, Sujoy Basu, Shishir Garg, Sukesh Garg, Sung-Ju Lee, Pramila Mullan, and Puneet Sharma. Scalable Grid Service Discovery based on UDDI. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[8] Rebhi S. Baraka. UDDI vs. ebXML, 2003. http://www.risc.uni-linz.ac.at/people/rbaraka/Registry.html.

[9] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. *OWL Web Ontology Language Reference*. World Wide Web Consortium, February 2004. http://www.w3.org/TR/owl-ref/.

[10] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco, and Giovanni Rimassa. Jade Programmer's Guide, November 2005.

[11] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. *XML Path Language (XPath) 2.0*, June 2006. http://www.w3.org/TR/xpath20/.

[12] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. *XQuery 1.0: An XML Query Language*, June 2006. http://www.w3.org/TR/xquery/.

[13] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, and Dave Winer. *SOAP Version 1.1*, May 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[14] Liliana Cabral and John Domingue. Mediation of Semantic Web Services in IRS-III. In *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005)*, volume 168 of *CEUR Workshop Proceedings*, pages 1–16. CEUR-WS.org, 2005.

[15] Liliana Cabral, John Domingue, Enrico Motta, Terry Payne, and Farshad Hakimpour. Approaches to Semantic Web Services: an Overview and Comparisons. In *The Semantic Web: Research and Applications: First European Semantic Web Symposium (ESWS 2004)*, pages 225–239, 2004.

[16] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 1.2*, July 2002.

[17] Emilia Cimpian and Adrian Mocan. WSMX Process Mediation Based on Choreographies. In *Proceedings of the Business Process Management Workshops (BPM)*, pages 130–143, 2005.

[18] John Colgrave and Karsten Januszewski. *Using WSDL in a UDDI Registry, Version 2.0.2*. OASIS UDDI Specifications TC, June 2004. http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v2.htm.

[19] Monica Crubézy, Mark A. Musen, Enrico Motta, and Wenjin Lu. Configuring Online Problem-Solving Resources with the Internet Reasoning Service. *IEEE Intelligent Systems*, 18(2):34–42, 2003.

[20] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next step in Web services. *Communication of the ACM*, 46(10):29–34, 2003.

[21] Jos de Bruijn, Holger Lausen, Reto Krummenacher, Axel Polleres, Livia Predoiu, Michael Kifer, and Dieter Fensel. *The Web Service Modeling Language (WSML) version 0.2*. DERI, March 2005. http://www.wsmo.org/TR/d16/d16.1/v0.2/.

[22] Asuman Dogac, Ibrahim Cingil, Gokce B. Laleci, and Yildiray Kabak. Improving the Functionality of UDDI Registries through Web Service Semantics. In *Proceedings of the Third international Workshop on Technologies For E-Services (TES)*, pages 9–18, 2002.

[23] Asuman Dogac, Yildiray Kabak, and Gokce B. Laleci. Enriching ebXML Registries with OWL Ontologies for Efficient Service Discovery. In *Proceedings of the 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04)*, pages 69–76, 2004.

[24] Asuman Dogac, Yildiray Kabak, Gokce B. Laleci, Carl Mattocks, Farrukh Najmi, and Jeff Pollock. Enhancing ebXML Registries to Make them OWL Aware. *Distributed and Parallel Databases*, 18(1):9–36, 2005.

[25] John Domingue, Liliana Cabral, Farshad Hakimpour, Denilson Sell, and Enrico Motta. IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services. In *Proceedings of the Workshop on WSMO Implementations (WIW 2004) (WIW 2004)*, 2004.

[26] John Domingue, Stefania Galizia, and Liliana Cabral. The Choreography Model for IRS-III. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, volume 3, page 62c, 2006.

[27] ebXML.org. *ebXML Business Process Specification Schema*, May 2001. www.ebxml.org/specs/ebBPSS.pdf.

[28] ebXML.org. *ebXML Collaboration-Protocol Profile and Agreement Specification*, May 2001. www.ebxml.org/specs/ccOVER.pdf.

[29] ebXML.org. *ebXML Core Component Dictionary*, May 2001. http://www.ebxml.org/specs/ccDICT.pdf.

[30] ebXML.org. *ebXML Core Component Overview*, May 2001. www.ebxml.org/specs/ccOVER.pdf.

[31] ebXML.org. *Message Service Specification v 1.0*, May 2001. http://www.ebxml.org/specs/ebMS.pdf.

[32] ebXML.org. *Registry Information Model v 2.0*, December 2001. http://www.ebxml.org/specs/ebrim2.pdf.

[33] ebXML.org. *Registry Services Specification v 1.0*, May 2001. http://www.ebxml.org/specs/ebRS.pdf.

[34] ebXML.org. *Registry Services Specification v 1.0*, May 2001. http://www.ebxml.org/specs/ebRIM.pdf.

[35] ebXML.org. *Using UDDI to find ebXML Reg/Reps*, May 2001. http://www.ebxml.org/specs/rrUDDI.doc.

[36] Fatih Emekçi, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. A Peer-to-Peer Framework for Web Service Discovery with Ranking. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 192–199, 2004.

[37] David C. Fallside and Priscilla Walmsley. *XML Schema*, October 2004. http://www.w3.org/XML/Schema.

[38] Dieter Fensel, Richard R. Benjamins, Enrico Motta, and Bob J. Wielinga. UPML: A Framework for Knowledge System Reuse. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI '99)*, pages 16–23, 1999.

[39] Dieter Fensel, Richard V. Benjamins, Stefan Decker, Mauro Gaspari, Rix Groenboom, William Grosso, Mark Musen, Enrico Motta, Enric Plaza, Guus Schreiber, Rudi Studer, and Bob Wielinga. The component model of UPML in a nutshell. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, 1999.

[40] Dieter Fensel and Christoph Bussler. Web Service Modeling Framework (WSMF). *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[41] Dieter Fensel, Christoph Bussler, and Alexander Maedche. Semantic Web Enabled Web Services. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 1–2, 2002.

[42] Florian Forster and Hermann de Meer. Discovery of Web Services with a P2P Network. In *International Conference on Computational Science (ICCS 2004)*, pages 90–97, June 2004.

[43] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik F. Nielsen. *SOAP Version 1.2*, June 2003. http://www.w3.org/TR/soap12-part1/.

[44] Wolfgang Hoschek. A Unified Peer-to-Peer Database Protocol. Technical Report DataGrid-02-TED-0407, DataGrid, April 2002.

[45] Wolfgang Hoschek. Peer-to-Peer Grid Databases for Web Service Discovery. *Concurrency: Practice and Experience*, 00:1–7, 2002.

[46] Jacek Kopecky. *Aligning WSMO and WSDL-S*. DERI, August 2005. http://www.wsmo.org/TR/d30/v0.1/.

[47] Heather Kreger. Fulfilling the Web services promise. *Communication of the ACM*, 46(6):29–34, 2003.

[48] Holger Lausen, Axel Polleres, and Dumitru Roman. *Web Service Modeling Ontology (WSMO)*. World Wide Web Consortium, June 2005. http://www.w3.org/Submission/WSMO/.

[49] Andreas Laux and Lars Martin. *XUpdate*, September 2000. http://xmldb-org.sourceforge.net/xupdate/.

[50] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia P. Sycara. *OWL-S: Semantic Markup for Web Services*. DAML-S Coalition, 2004. http://www.daml.org/services/owl-s/1.1/overview/.

[51] David Martin, Mark Burstein, Ora Lassila, Massimo Paolucci, Terry Payne, and Sheila A. McIlraith. *Describing Web Services using OWL-S and WSDL*. DAML-S Coalition, October 2003. http://www.daml.org/services/owl-s/1.0/owl-s-wsdl.html.

[52] Deborah L. McGuinness and Frank van Harmelen. *OWL Web Ontology Language Overview*. World Wide Web Consortium, February 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[53] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic Web services. *IEEE Intelligent Systems*, 16(2):46–53, March 2001.

[54] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of Hilbert space-filling curve. Technical Report UMIACS-TR-96-20, University of Maryland Institute for Advanced Computer Studies, 1996.

[55] Enrico Motta, John Domingue, Liliana Cabral, and Mauro Gaspari. IRS-II: A Framework and Infrastructure for Semantic Web Services. In *Second International Semantic Web Conference (ISWC 2003)*, pages 306–318, 2003.

[56] Farrukh Najmi. Web Content Management Using the OASIS ebXML Registry Standard, April 2004. http://www.idealliance.org/papers/dx_xmle04/papers/04-02-02/04-02-02.pdf.

[57] OASIS UDDI Specifications TC. *UDDI Version 2.0*, June 2001. http://www.oasis-open.org/specs/index.php#uddiv2.

[58] OASIS UDDI Specifications TC. *UDDI Version 2.0 Replication Specification*, July 2002. http://uddi.org/pubs/Replication_v2.htm.

[59] OASIS UDDI Specifications TC. *UDDI Version 3.0*, July 2002. http://uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[60] OASIS UDDI Specifications TC. *UDDI Version 3.0.2*, October 2004. http://uddi.org/pubs/uddi_v3.htm.

[61] Object Management Group. *Meta Object Facility (MOF) Specification version 1.4*, April 2002. http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf.

[62] Object Management Group. *XML Metadata Interchange Specification 2.0*, September 2005. http://www.omg.org/cgi-bin/doc?formal/2005-09-01.

[63] Nicole Oldham, Christopher Thomas, Amit P. Sheth, and Kunal Verma. METEOR-S Web Service Annotation Framework with Machine Learning Classification. In *Proceedings of the Semantic Web Services and Web Process Composition, First International Workshop (SWSWPC)*, pages 137–146, 2004.

[64] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Importing the Semantic Web in UDDI. In *Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web (CAiSE/WES)*, pages 225–236, 2002.

[65] Massimo Paolucci, Katia P. Sycara, Takuya Nishimura, and Naveen Srinivasan. Using DAML-S for P2P Discovery. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 203–207, 2003.

[66] Mike P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE)*, pages 3–12, 2003.

[67] Mike P. Papazoglou and Dimitrios Georgakopoulos. Service-Oriented Computing: Introduction. *Communications of the ACM*, 46(10):24–28, 2003.

[68] Mike P. Papazoglou, Bernd J. Krämer, and Jian Yang. Leveraging Web-Services and Peer-to-Peer Networks. In *Advanced Information Systems Engineering, 15th International Conference (CAiSE)*, pages 485–501, 2003.

[69] Thomi Pilioura, Aphrodite Tsalgatidou, and Alexandros Batsakis. Using WSDL/UDDI and DAML-S in Web Service Discovery. In *Proceedings of WWW 2003 Workshop on E-Services and the Semantic Web (ESSW' 03)*, 2003.

[70] Axel Polleres, Ruben Lara, and Dumitru Roman. *Formal Comparison WSMO/OWL-S*. DERI, March 2004. http://www.wsmo.org/2004/d4/d4.2/v01/.

[71] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. *IEEE Internet Computing Journal*, 6(1):50–57, January/February 2002.

[72] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Ruben Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[73] Pornpong Rompothong and Twittie Senivongse. A Query Federation of UDDI registries. In *Proceedings of the 1st international symposium on Information and communication technologies (ISICT '03)*, pages 561–566, 2003.

[74] Brahmananda Sapkota, Laurentiu Vasiliu, Ioan Toma, Dumitru Roman, and Christoph Bussler. Peer-to-Peer Technology Usage in Web Service Discovery and Matchmaking. In *Proceedings of the Web Information Systems Engineering (WISE)*, pages 418–425, 2005.

[75] Mario T. Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. HyperCuP - Hypercubes, Ontologies, and Efficient Search on Peer-to-Peer Networks. In *Agents and Peer-to-Peer Computing, First International Workshop (AP2PC 2002)*, pages 112–124, 2002.

[76] Cristina Schmidt and Manish Parashar. A Peer-to-Peer Approach to Web Service Discovery. *World Wide Web*, 7(2):211–229, 2004.

[77] Ali ShaikhAli, Omer F. Rana, Rashid Al-Ali, and David W. Walker. UDDIe: An Extended Registry for Web Services. In *Proceedings of Workshop on Service Oriented Computing (SOC)*, pages 85–90, 2003.

[78] Kaarthik Sivashanmugam, Kunal Verma, Amit P. Sheth, and John A. Miller. Adding semantics to web services standards. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 395–401, 2003.

[79] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. *OWL Web Ontology Language Guide*. World Wide Web Consortium, February 2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[80] Naveen Srinivasan, Massimo Paolucci, and Katia P. Sycara. Adding OWL-S to UDDI, implementation and throughput. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, pages 6–9, 2004.

[81] Stencil Group. *The Evolution of UDDI*, June 2002. http://www.uddi.org/pubs/the_evolution_of_uddi_20020719.pdf.

[82] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, pages 149–160, 2001.

[83] Michael Stollberg, Uwe Keller, and Dieter Fensel. Partner and Service Discovery for Collaboration Establishment with Semantic Web Services. *Proceedings of the International Conference on Web Services*, pages 473–480, 2005.

[84] Chenliang Sun, Yi Lin, and Bettina Kemme. Comparison of UDDI Registry Replication Strategies. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 218–225, 2004.

[85] Mahantesh Surgihalli and K Vidyasankar. A Lazy Replication Scheme for Loosely Synchronized UDDI Registries. In *Proceeding of 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, 2005.

[86] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the third international conference on on Parallel and distributed information systems (PDIS '94)*, pages 140–150, 1994.

[87] UDDI Spec Technical Committee Draft. Tc. *UDDI Version 1.0*, September 2000. http://www.oasis-open.org/committees/uddi-spec/doc/contribs.htm#uddiv1.

[88] UDDI.org Consortium. *UDDI as the registry for ebXML Components*, February 2004. http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-uddi-ebxml.htm.

[89] Kunal Verma, Kaarthik Sivashanmugam, Amit Sheth, Abhijit Patil, Swapna Oundhakar, and John Miller. METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Information Technology and Management*, 6(1):17–39, January 2005.

[90] Michal Zaremba, Matthew Moran, and Thomas Haselwanter. WSMX Architecture, June 2005. http://www.wsmo.org/TR/d13/d13.4/.

[91] Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee. QoS-Aware and Federated Enhancement for UDDI. *International Journal of Web Services Research*, 1(2):58–85, 2004.

[92] Chen Zhou, Liang-Tien Chia, Bilhanan Silverajan, and Bu-Sung Lee. UX - An Architecture Providing QoS-Aware and Federated Support for UDDI. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 171–176, 2003.