A Comparative Approach to the Implementation of Drug Pedigree Discovery Systems

by

Indy Yu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

© Massachusetts Institute of Technology 2007. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
...007

Certified by.....
...
...ms
Ass... ...r of MIT Auto-ID Labs
Thesis Supervisor

Certified by.....
.....
Abel Sanchez
Research Sci... ...d Productivity
...esis Supervisor

Accepted by ...
.....
...rthur C. Smith
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 0 3 2007
LIBRARIES

BARKER

A Comparative Approach to the Implementation of Drug Pedigree Discovery Systems

by

Indy Yu

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As the use of RFID technology penetrates and reforms the supply-chain industry, standards are being produced at all levels of the RFID technology spectrum, ranging from hardware to software. The Electronic Product Code (EPC) standard uniquely identifies RFID-tagged products. An application that supports the usage of EPCs is an Electronic Drug Pedigree (E-Pedigree), which is a historical record that indicates the chain of custody of a particular drug product being passed from one supply-chain partner to another. In order to fully implement track-and-trace of pharmaceutical products, software systems need to be built so that pedigree documents can be effectively stored and searched. In this Thesis, two approaches that address the issue of pedigree document discovery are presented–one centralized, one decentralized. The centralized pedigree discovery service extracts metadata from pedigree documents submitted to a centralized server and uses them in a search engine, such as Google Base, to located desired documents that match client queries. The decentralized service allows pedigree documents to be stored locally by individual business owners. Each local server is attached to a Discovery Service Unit containing metadata of local pedigree documents, and these units communicate with each other to form a network. Both approaches are implemented as Web Services.

Thesis Supervisor: John R. Williams
Title: Associate Professor, Director of MIT Auto-ID Labs

Thesis Supervisor: Abel Sanchez
Title: Research Scientist, Laboratory of Manufacturing and Productivity

Acknowledgments

I would like to thank Dr. Abel Sanchez and Professor John R. Williams for their excellent guidance while I am working at MIT Auto-ID Labs for the past year. Their mastery in computer science and valuable insights have helped me to develop new perspectives in the field and strengthened by knowledge required for the research associated with this Thesis.

I would also like to thank my wonderful lab mates Fivos Constantinou, JinHock Ong, and Sergio Herrero for their help and support of my research, and for creating a lively and constructive working environment.

Finally, I would like to thank my parents for the support and guidance they have provided throughout my life.

Contents

1 Introduction **13**

1.1 RFID Technology 13

1.2 Electronic Product Code Applications 14

1.3 Motivations for Implementing E-Pedigree Discovery Service 15

1.4 Thesis Organization 16

2 Overview of Electronic Drug Pedigree **17**

2.1 Components and Usage 17

2.1.1 Document Components and Layout 18

2.1.2 Business Flow Scenarios 19

2.2 Document Authentication 22

2.2.1 Digital Signature Usage 22

2.2.2 X509 Certificate Usage 24

2.2.3 Common Registry 25

3 A Decentralized Approach to E-Pedigree Discovery Service **27**

3.1 Previous Work 27

3.1.1 Design Motivation 27

3.1.2 Existing Discovery Protocols 28

3.1.3 Overview of Salutation 28

3.2 Design Architecture 29

3.2.1 Design Assumptions 29

3.2.2 Architecture Overview 29

3.2.3 Network Communication 30
3.2.4 Design Components 32
3.3 Web Service Implementation 34
3.3.1 Implementation Overview 34
3.3.2 The LocalLookupService Object 37
3.3.3 The PartnerLookupService Object 38
3.3.4 The PartnerLookupService Client 40

4 A Centralized Approach to E-Pedigree Discovery Service **41**
4.1 Previous Work 41
4.1.1 Design Motivation 41
4.1.2 Overview of RSS 42
4.1.3 Overview of Google Base 43
4.2 Design Architecture 44
4.2.1 Design Assumptions 44
4.2.2 E-Pedigree Capture Process 45
4.2.3 E-Pedigree Query Process 46
4.3 Web Service Implementation 46
4.3.1 Implementation Overview 46
4.3.2 The PedigreeCapture Method 48
4.3.3 The PedigreeQuery Method 50
4.3.4 The Pedigree Server Client 52

5 Discussion and Future Work **55**
5.1 Analysis of the Two Design Approaches 55
5.1.1 Evaluation of the Decentralized Approach 55
5.1.2 Evaluation of the Centralized Approach 56
5.2 Future Work 57
5.2.1 Performance Analysis 57
5.2.2 Improvements 58
5.2.3 Extensibility to Other Applications 59

6 Conclusion 61

A Class Diagrams for the Decentralized Implementation 63

B Class Diagrams for the Centralized Implementation 67

List of Figures

1-1 EPC Architecture Framework 14

2-1 Pedigree Document Layers Generation 18
2-2 Example Repackaged Pedigree with Unsigned Received Pedigree Layer 21
2-3 Pedigree Layer Generation Flow Diagram 22
2-4 Pedigree Authentication Process 23
2-5 Example Certificate Data 23
2-6 Use of Public Key Infrastructure 24
2-7 Signature Element of a Pedigree Document 25

3-1 Architecture of Salutation 29
3-2 Pedigree Discovery Service Network 30
3-3 Communication among Discovery Service Units 31
3-4 Discovery Service Unit Architecture 32

4-1 Example RSS Feed 42
4-2 Example Google Base Entry 44
4-3 Pedigree Capture Process Flow Diagram 45
4-4 Pedigree Query Process Flow Diagram 46
4-5 Screen Capture of Web Service Interface 47

A-1 Class Diagram for the Local Lookup Service 64
A-2 Class Diagram of the Partner Lookup Service 65

B-1 Class Diagram of the Web Service 68

Chapter 1

Introduction

1.1 RFID Technology

Radio-frequency identification (RFID) is an automatic identification method that relies on remote storage and retrieval of data through devices called RFID tags or transponders. An RFID tag can be attached onto or incorporated into a product, an animal, or a person. It is used in a variety of applications such as e-passports systems, financial transaction systems, and automotive tracking systems. One of the most important uses of RFID technology is in the supply-chain industry, where products are tagged for the purpose of increasing business efficiency and the ability to track and trace items under transaction.

All RFID-tagged products can be uniquely identified through the use of Electronic Product Code (EPC) [19], embedded in the tags. The EPC Gen2 (in short for EPCglobal UHF Class 1 Generation 2) standard was proposed by EPCglobal, Inc., a subsidiary of GS1, who is the creator of the UPC barcode. The usage of EPCs in the supply-chain industry forms the backbone of RFID tag standards and is likely replace the barcode system in the near future. By exchanging EPC information, business partners–including manufactures, wholesalers, and retailers–can effectively react to business changes and expedite the process of formulating business transactions.

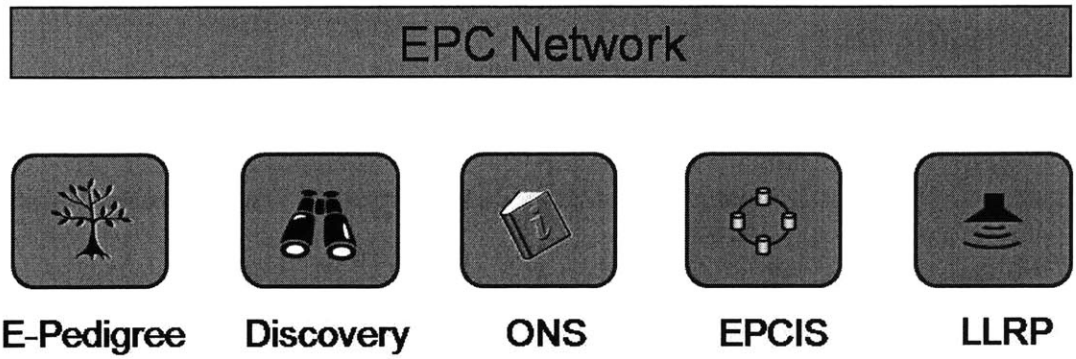

Figure 1-1: EPC Architecture Framework

1.2 Electronic Product Code Applications

In order to share EPC data among business partners, an agreement has to be reached as to how this information can be communicated. Consequently, a standard was proposed in 2005 by the EPC standard body EPCglobal to address this issue. They established an architecture framework to formalize and identify the hardware, software and data interface components needed to formulate the whole EPC Network [6]. Since then, many standards have been proposed for some of these components. A few of the major ones are illustrated in Figure 1-1.

The functions of each of the components in the EPC Network illustrated above are as follows:

- Electronic Drug Pedigree (E-Pedigree): A historical record that indicates the chain of custody of a particular drug product being passed from one pharmaceutical supply-chain partner to another.

- EPC Network Discovery: A system that specifies how EPC-related items can be visible and accessible within a large network of supply-chain partners.

- Object Naming Service (ONS): A name resolution system that specifies how a Domain Name System (DNS) is used to locate authoritative metadata and services associated with EPCs.

- EPC Information Service (EPCIS): A repository service that allows one to cap-

ture and query EPC-related events generated by readings from RFID Tags and Readers.

- Lower Level Reader Protocol: An interface protocol that bridges the communication between RFID Readers and software systems that controls these readers.

Currently standards have been proposed and specifications have been derived for all of the components enumerated above with the exception of the EPC Network Discovery. While no discovery standard exists, many designs in this area have been proposed and implemented by research institutions and business corporations. For instance, major players in the software industry such as Oracle and IBM have their own proprietary implementations of the EPC Discovery Service software [1]. Auto-ID Labs at Cambridge University, along with varies industry partners such as Sony and SAP, has been working on the BRIDGE (Building Radio Frequency Identification for the Global Environment) project, which attempts to research, develop and implement tools to enable the deployment of EPCglobal applications in Europe [14]. A large component of the BRIDGE project is the implementation of an EPC Discovery Service. Even within the standard body EPCglobal, detailed propositions on the implementation of discovery services for EPCIS and LLRP have are in the process of being addressed by industry-led work groups.

1.3 Motivations for Implementing E-Pedigree Discovery Service

As the implementations of other components in the network, such as EPCIS and E-Pedigree, become more complete, the Discovery component becomes increasingly indispensable. Currently it poses as the biggest hindrance to the advancement and completion of the EPC Network as a whole. The core component of my research focuses on working with the small building blocks of the discovery problem and experimenting with the various design approaches potentially suitable for the EPC Architecture Framework.

Of the different building blocks of the EPC Network, there are three components from which a discovery service can be build upon–EPCIS [7], LLRP [8] and E-Pedigree [9]. A discovery service for EPCIS can be defined as a service that returns event records associated with a particular EPC from all EPCIS Repositories in the network. A discovery service for LLRP can be defined as a service that returns RFID Readers to be controlled by Reader Access Controllers in the network. A discovery service for E-Pedigree can be defined as a service that returns drug pedigree documents associated with certain search criteria.

For this prototype discovery network, the Electronic Drug Pedigree is chosen for a few reasons. For one, there is an urgency in resolving the pharmaceutical product track-and-trace issue in the health care industry. Government mandates force pharmaceutical companies to generate pedigree documents to be view by the public. This effort in is led by the U.S. Food and Drug Administration (FDA) in attempt to reduce the amount of counterfeiting drugs in the country. Furthermore, from the implementation perspective, Electronic Drug Pedigrees are simple. They are XML documents and do not use special bindings, protocols, and security mechanisms like EPC Information Systems and LLRP Systems do.

1.4 Thesis Organization

Chapter 2 provides an overview for what an Electronic Drug Pedigree is and how it is used in the pharmaceutical supply-chain industry. Chapter 3 proposes a decentralized design for a pedigree discovery service and demonstrates an implementation of this system. Chapter 4 proposes a centralized approach for the pedigree discovery system and also illustrates a version of the implementation. Chapter 5 compares the two designs presented and discusses the benefits, potential problems, and future work needs to be done for each. Finally, Chapter 6 presents the conclusion of this thesis.

Chapter 2

Overview of Electronic Drug Pedigree

2.1 Components and Usage

An Electronic Drug Pedigree, or e-pedigree, is a custody record of a particular drug product that is exchanged within a pharmaceutical supply-chain. Although the content of an e-pedigree can vary by law, it is not difficult to derive a standard document format that includes a supper-set of data elements required by all of the laws. Florida is the first state to mandate e-pedigree documents for all drug products originated and handled within the state. Soon after, California joined the effort by making a strict set of pedigree requirements. These laws went into effect January 1st, 2007, and is currently considered the *unofficial* standard by the pharmaceutical pedigree community [2]. Around the same time in 2007, the Unified Drug Pedigree Coalition work group of EPCglobal ratified a pedigree document format standard [9]. The work group is composed of company representatives from pharmaceutical companies, their industry associations and various US state and federal regulatory agencies. The format was determined to be an XML document. A pedigree document schema that includes all of the necessary data elements were derived as part of the standardization process and the pedigree specification. (An XML schema is a description of an XML document, typically expressed in terms of constraints on the structure and content

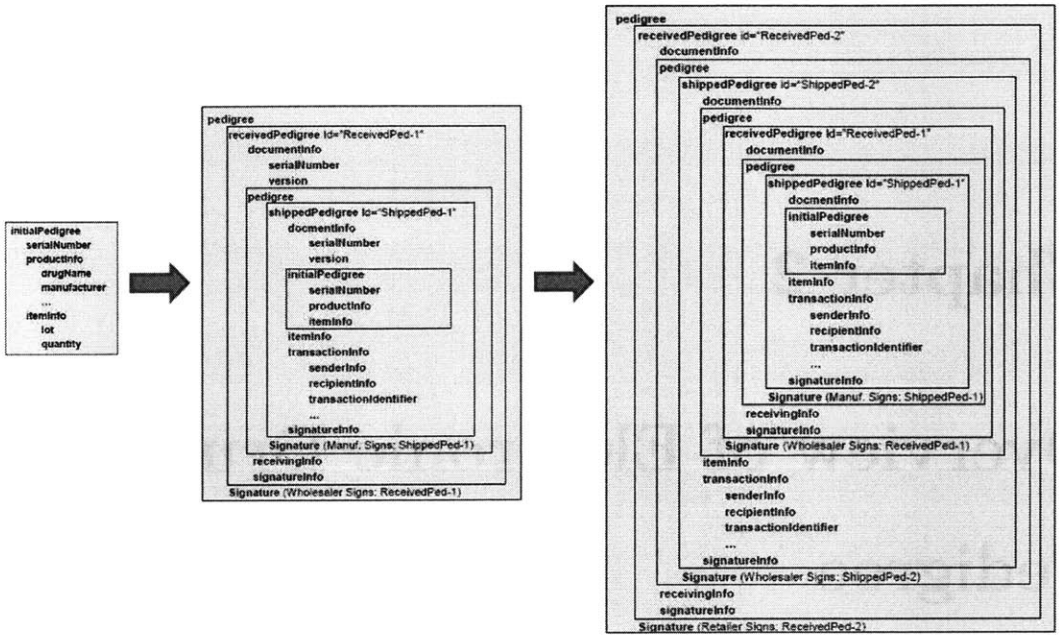

Figure 2-1: Pedigree Document Layers Generation

of the document, above and beyond the basic syntax constraints imposed by XML itself.)

2.1.1 Document Components and Layout

Initially, a pedigree contains drug product information only. As the corresponding drug product moves from one supply-chain partner to another, new layers of the pedigree are added each time to record the business transaction information. Each new layer of the pedigree wraps around an older layer. The more the number of business transactions carried out on a product, the more the number of layers of the associated pedigree document. Figure 2-1 is an example of the pedigree formation process.

When a new layer is added to a pedigree, it often requires the business partner creating the layer to apply a digital signature to authenticate all information contained, including previous layers created by other supply-chain partners. The details of the signing process is explained in Section 2.2. Of the five different types of pedigree layers can be used in a pedigree document, two requires signing. The definition of

the five layer types are shown below:

- Initial Pedigree: The innermost layer of the pedigree document that records initial product and packaging information of a particular drug product.

- Repackaged Pedigree: The innermost layer of the pedigree document that not only records product and packaging information of a particular drug product but also records the information of the previous products and pedigrees used in the new product.

- Shipped Pedigree: An outer layer of the pedigree document that records the business transaction information for a product that is ready to be shipped to the next supply-chain location. This layer can be used to wrap any of the other four pedigree layers and requires a digital signature of the shipping party.

- Received Pedigree: An outer layer of the pedigree document that records the business transaction information for a product that has been received from a previous supply-chain location. This layer can be used to wrap the Shipped Pedigree layer and requires a digital signature of the receiving party.

- Unsigned Received Pedigree: An intermediate layer of the pedigree document that records the business transaction information for a product that has been received from a previous supply-chain location. This layer can be used to wrap Shipped Pedigree layers and, as the name suggest, does not require a digital signature. (This layer must be generated concurrently with a wrapping Shipped Pedigree layer).

2.1.2 Business Flow Scenarios

There are twelve types of business flow scenarios in which a pedigree document can be used. For the purpose of this Thesis, I will not iterate each one but will simply show two examples to illustrate how pedigrees are used under different business contexts. The most common business flow scenario is the transaction from a manufacturer, to

a wholesaler, then to a retailer. A manufacture can initiate an Initial Pedigree layer and adds a Shipping Pedigree layer to the document when the product is ready to be transferred to the wholesaler, with signature applied. Once a wholesaler receives the pedigree and the associated drug products, the wholesaler would authenticate the information presented in the pedigree as well as the digital signature. When all information are authenticated, a Received Pedigree layer is added to the document and signed by the wholesaler. The process repeats as the pedigree moves from the wholesaler to the retailer. Figure 2-1 shown above is an example of this scenario.

Not all transactions has to originate from a manufacture. Sometimes a wholesaler can initiate a pedigree on behalf of the manufacture. The original transaction information from the manufacture to the wholesaler would simply be recorded inside an Initial Pedigree or a Repackaged Pedigree layer. For instance, a wholesaler can initiate a pedigree to indicate a return transaction to a kit manufacture. Initially the wholesaler creates a Repackaged Pedigree to indicate the previous product and previous pedigree information for each drug contained in the kit. Then it adds and signs a Shipped Pedigree layer for the return transaction. If the wholesaler decides to sale the same product to another retailer, it would add another receiving and shipping layer on to the current shipping layer. However, if both layers were to be added at the same time, the process can be simplified by replacing the Received Pedigree layer with the Unsigned Received Pedigree layer, so that only the outermost Shipped Pedigree layer has to be signed. Figure 2-2 shown below is an example of this scenario.

Regards of the business scenario, all pedigrees have to abide by the rules shown in Figure 2-3 when new layers are generated. Pedigree layers with *solid* arrows projecting out indicate that they cannot be stand-alone and must have additional layers wrapping around them. Layers with *dashed* arrows indicate that they can be stand-alone, and any additional layers are optional.

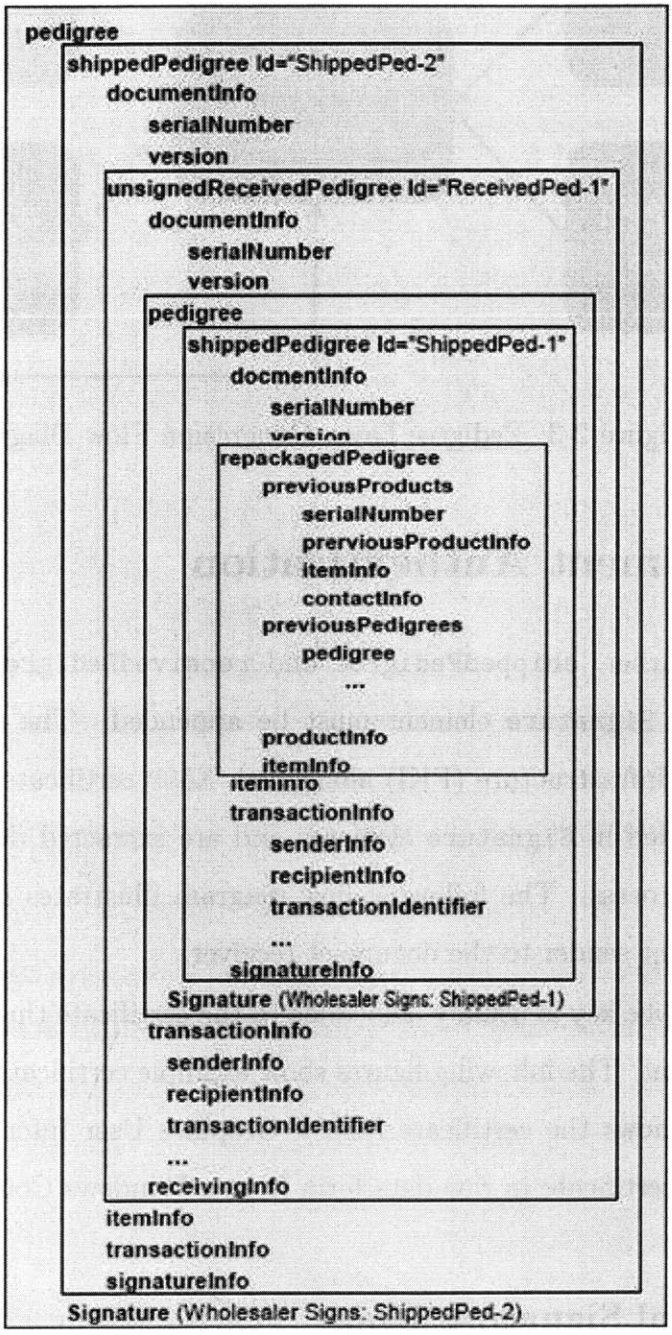

Figure 2-2: Example Repackaged Pedigree with Unsigned Received Pedigree Layer

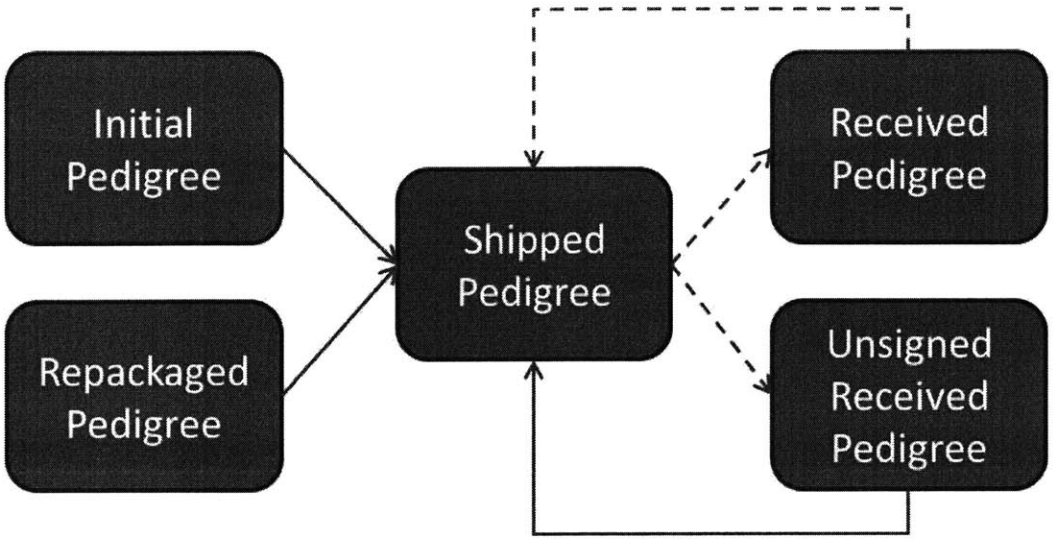

Figure 2-3: Pedigree Layer Generation Flow Diagram

2.2 Document Authentication

As mentioned earlier, `shippedPedigree` and `receivedPedigree` elements require signing; hence a `Signature` element must be appended. The signing mechanism uses Public Key Infrastructure (PKI) along with X509 certificates [17]. User certificates are embedded in `Signature` elements and are extracted during the signature authentication process. The following flow diagram illustrates this process from a pedigree document sender to the document receiver.

A signer's public key is usually embedded in the certificate that is attached to the pedigree document. The following figures show example certificate data (Figure 2-5). The left figure shows the certificate from a Graphics User Interface, and the right shows the same certificate in raw data form from a Windows Console.

2.2.1 Digital Signature Usage

The following figure (Figure 2-6) illustrates the signing and authentication process of PKI. During the signing process, a message digest over the message content being signed is first computed. Then the signer uses its Private Key to sign over the digest. During the authentication process, the authenticator would use the signer's Public

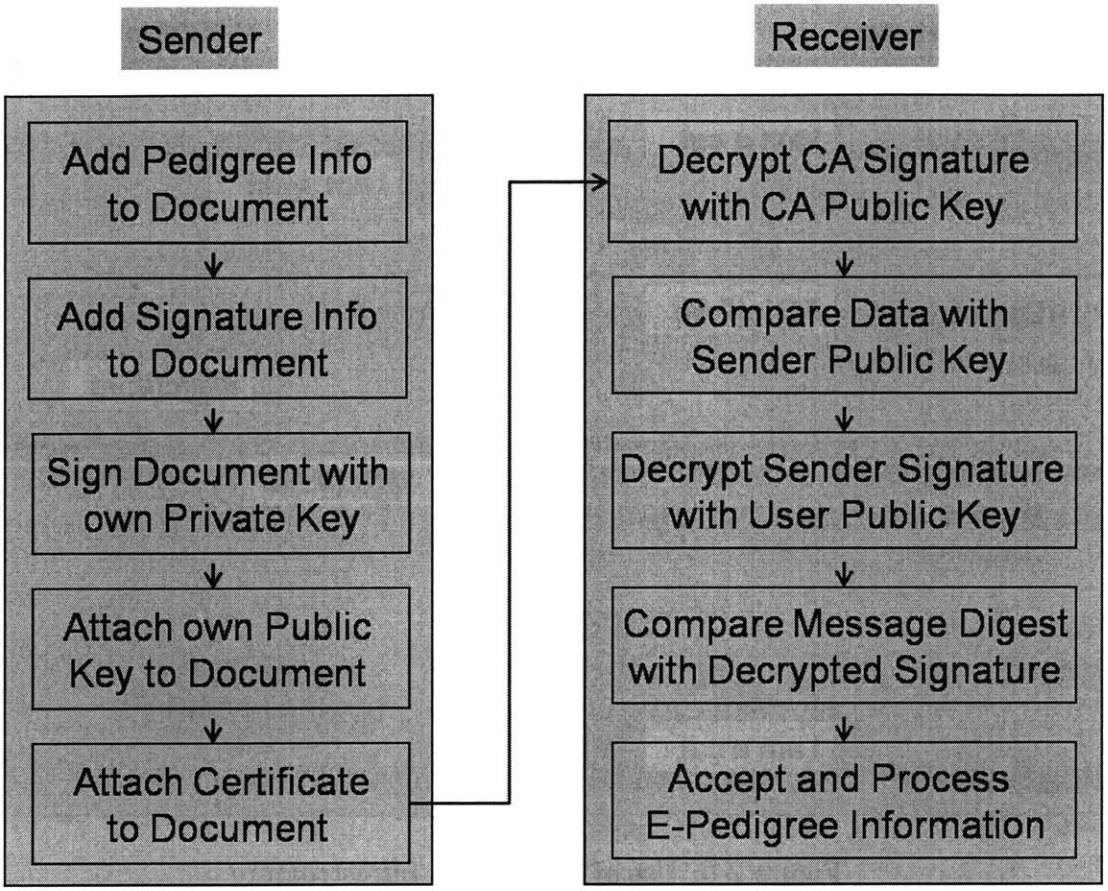

Figure 2-4: Pedigree Authentication Process

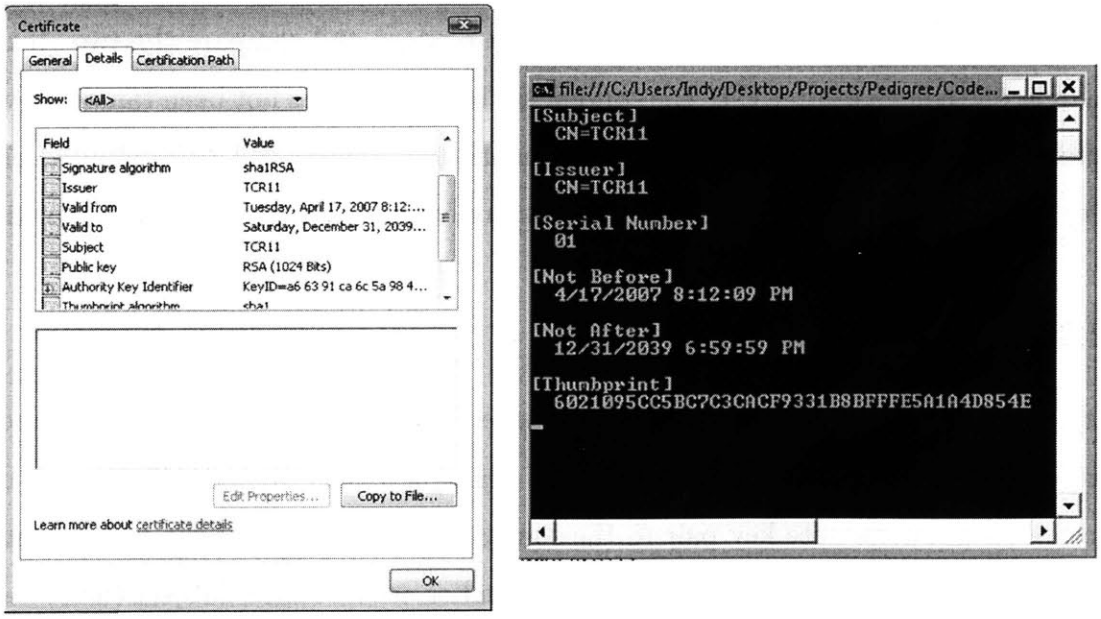

Figure 2-5: Example Certificate Data

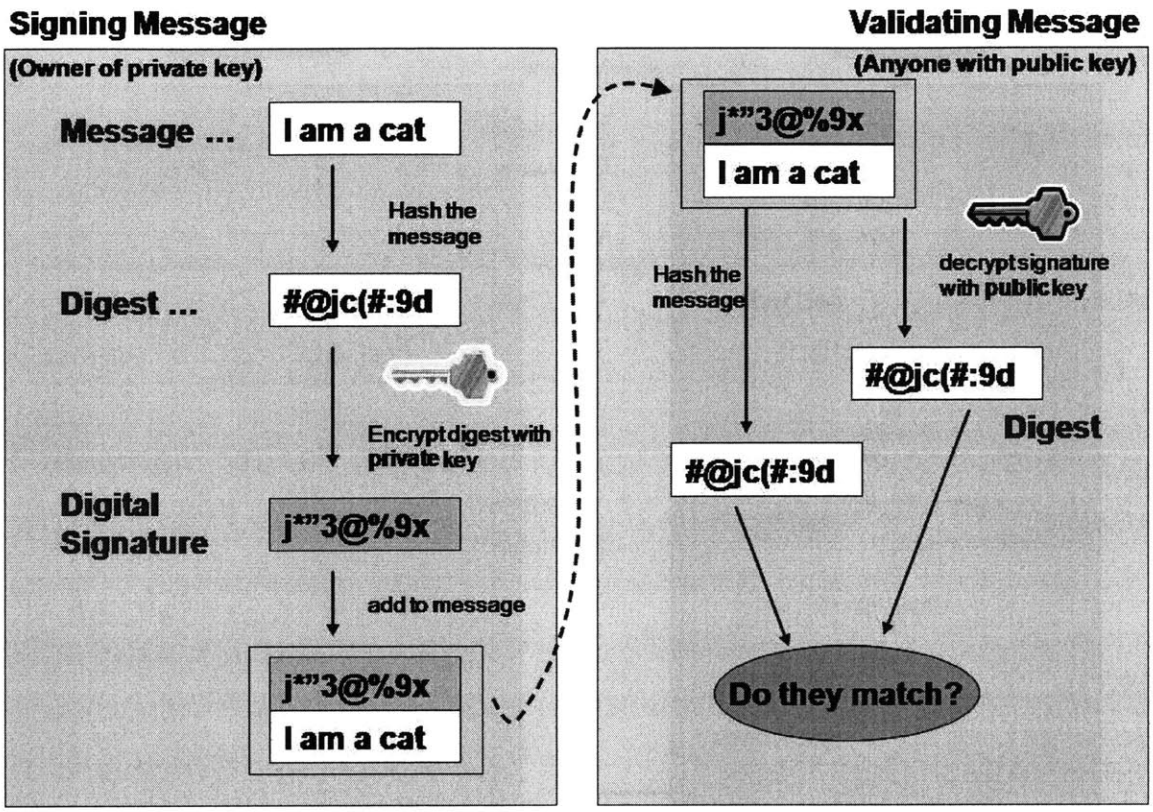

Figure 2-6: Use of Public Key Infrastructure

Key to decrypt the signature to obtain the original digest. Then the authenticator would compute another digest of the message content and compare the two. If they match, one can guarantee that the content of the message has not been tampered with after it is signed. For drug pedigree documents, message digests are computed using the SHA1 algorithm, and digital signatures are computed using the RSA algorithm.

2.2.2 X509 Certificate Usage

Digital signatures only guarantee that a signed pedigree has not been not tampered with but does not guarantee the identity of the signer; thus X509 certificates are used to bind a public-private key pair to the signer's identity. Some of the components of a certificates include validity period, serial number, issuer name, subject (user) name, and user public key [17]. To verify that the Public Key belongs to the signer, the authenticator must already have a trusting relationship with the Certificate Authority

Figure 2-7: Signature Element of a Pedigree Document

(CA) that issues the signer's certificate. In other words, he or she must have CA's Public Key. The certificate itself contains a digital signature, signed over the user Public Key using the CA's Private Key. The authenticator can use the CA's Public Key to verify the CA's signature over the user Public Key. Once it is clear that the user Public Key belongs to the signer, the authenticator can use the key to verify the pedigree content. An example `Signature` element of a pedigree document is shown below in 2-7. This XML block of document not only includes the certificate itself but also the transforms, canonicalization, and signature methods used to compute the digest and the digital signature value.

2.2.3 Common Registry

Although digital signatures and X509 certificates can be used to guarantee the authenticity of pedigree content and the identity of the signer, it does not prevent the signer to change the content of the pedigree and re-sign at a later time. Currently, all pedigree documents are maintained and stored locally by the individual businesses that have created these documents. Thus a prevention scheme may be necessary to prohibit companies from potentially manipulating their information. Many industry

participants have been actively promoting the use of a common registry. Currently the structure of the registry has not been finalized. It may be historical copies of all drug pedigree document exchanged within the entire supply-chain, or it may simply be a hash, or digest, of the documents. Many large companies, such as Verisign, have been competing for the position as the primary holder of the common registry. However, whether this registry is needed at all is still uncertain.

Chapter 3

A Decentralized Approach to E-Pedigree Discovery Service

3.1 Previous Work

3.1.1 Design Motivation

The purpose of a pedigree discovery service is to provide searching assistance for pedigree documents that reside on the EPC Network. A decentralized approach to discovery suggests that all pedigree documents associated with a particular business must be maintained in its local repository. This approach may be preferred from a business logistric's point of view. For one, pharmaceudical companies might be skeptical and reluctant to actively report their data into a centralized location for others to manage. In addition, if mistakes need to be revised, the revision process can be performed must faster if all data are maintained locally, thus reducing business costs. From the point of view of system design, a decentralized service reduces the possibility of potentially having a message loading bottleneck at the central storage server and having a single point of failure if the server breaks.

3.1.2 Existing Discovery Protocols

Many common discovery protocols were surveyed for the purpose of this design, including Jini [18], Salutation [4], UPnp [5], SLP [15], and UDDI [10]. However, all current protocols lack one important parallelism with what is needed for the Pedigree Discovery Service–All the discovery protocols enumerated above looks for any *one* service that would satisfy a client's needs, but a discovery service on the EPC network requires *all* services that satisfy the client's needs. Currently there is no available framework that fully implements the latter. The services that are close to satisfying this need are search engines like Google.

3.1.3 Overview of Salutation

Of all the discovery protocols mentioned above, the one that influenced the design presented in this Chapter the most is Salutation, developed by Salutation Consortium. It is also the only one that uses a decentralized network approach. The architecture of Salutation is composed of three fundamental components: Functional Units, Salutation Managers, and Transport Managers. Multiple clients and services can be attached to one Functional Unit, and Functional Units can communicate with each other about the services that they provide. A client only needs to connect to one unit to locate a desired service in the network.

Each Functional Unit contains a Salutation Manager and a Transport Manager. Salutation Managers serve as service brokers that 1) help clients to find desired services and 2) allow services to register their availability. They communicate with each other through remote procedure calls (RPC). Transport Managers isolate the detailed network specific protocol information from Salutation Managers, thereby giving Salutation network transport independence. Figure 3-1 is an overview of Salutation's architecture.

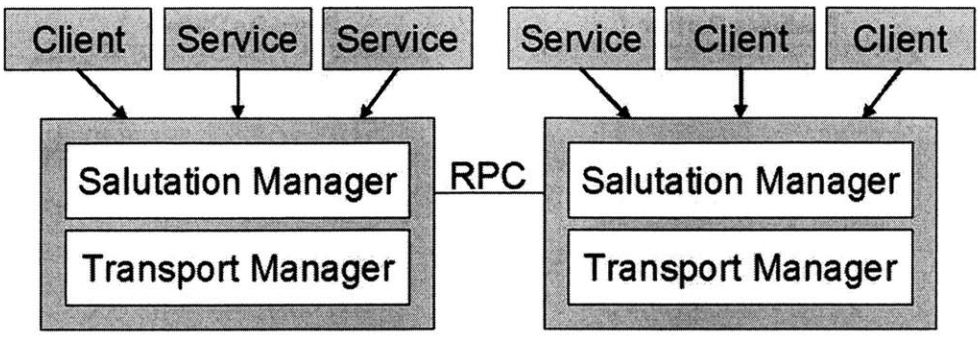

Figure 3-1: Architecture of Salutation

3.2 Design Architecture

3.2.1 Design Assumptions

In order to scale the prototype to contain only the minimum crucial portions of the design, many important assumptions were made. For one, since the design does not contain any security mechanism, it is assumed that all the service partners in the network may freely access each other's data. In addition, it is assumed that business relationships do not form and discontinue frequently. Thus, the overturn rate of businesses entering and leaving the network is low. For the initial implementation of this design, there is no mechanism that supports automatic registration of business partners. In addition, the total number of partners participating in the network must be scalable such that there are no significant delays caused by querying a large number of partners in the network. Finally, we assume that having stale data, caused by delays in content refreshing in local servers, is allowed.

3.2.2 Architecture Overview

The decentralized pedigree discovery service is defined as a network of services that returns a list of pedigree documents associated with a common attribute for which a client has used as searching criteria. A client can be a company, a consumer, or a government agency. This network is composed of identical service units, which serve a similar purpose as Functional Units in Salutation. These units, referred to as a

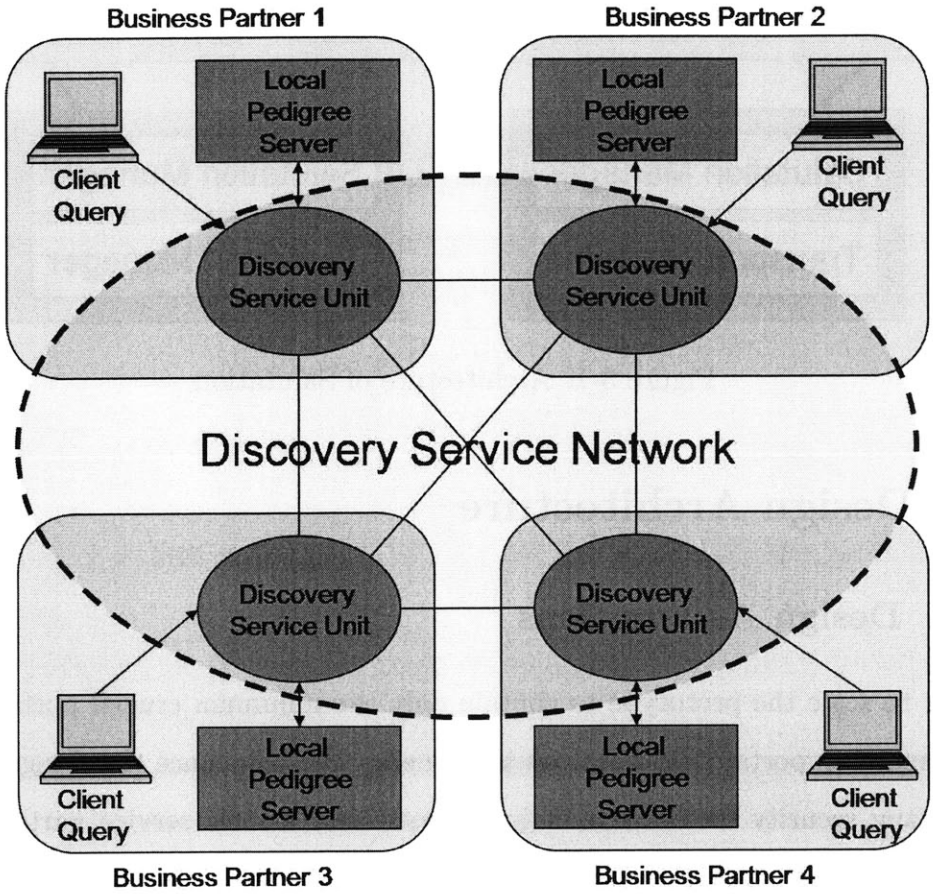

Figure 3-2: Pedigree Discovery Service Network

Discovery Service Units, reside locally with individual business partners. They are directly attached to local pedigree servers where pedigree documents are stored. A service unit has two purposes: 1) extract metadata from pedigree documents residing in local servers, and 2) answer client queries by communicating with a predefined list of service units to obtain pedigree documents. A service unit would always be connected to one or more other service units, and these interweaving connections form the Discovery Service Network. Figure 3-2 illustrates an overview of this network.

3.2.3 Network Communication

As mentioned above, a Discovery Service Unit has two functions, thus two design components. Just as there is a layer of separation between a Salutation Manager

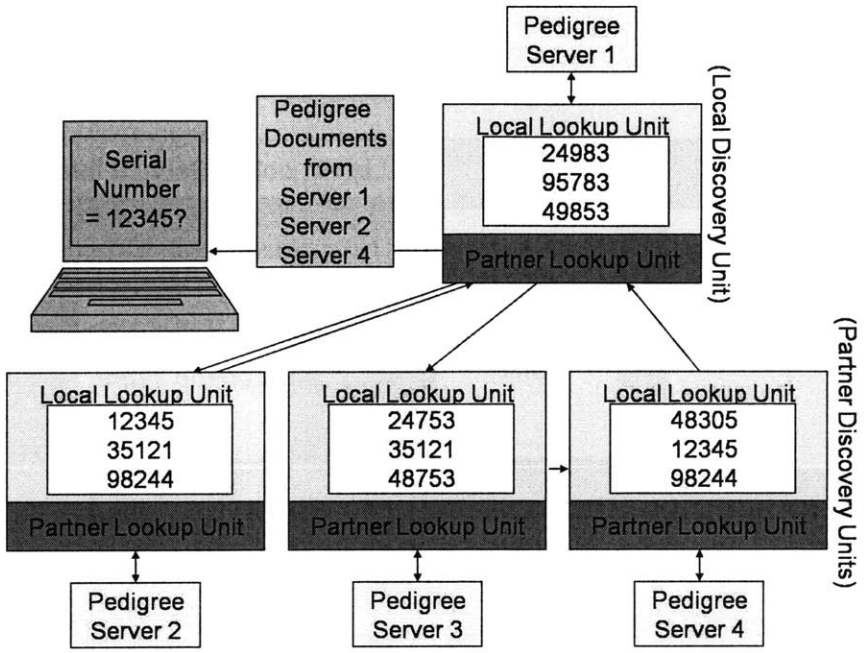

Figure 3-3: Communication among Discovery Service Units

and a Transport Manager in Salutation, there is complete isolation between the two major components that make up a Discovery Service Unit: the Partner Lookup Unit and the Local Lookup Unit. The Partner Lookup Unit accepts client queries and conducts searchs of pedigree documents that match client requests by communicating with other Discovery Service Units. The Local Lookup Unit accepts queries from Partner Lookup Units of other service units and locally searches its metadata in attempt to match client requests. Figure 3-3 is an example of how partner Discovery Service Units interact with each other. For explanation purposes, the metadata used in answering client queries are pedigree serial numbers. In reality, many different pedigree document attributes can be used as metadata beyond the simple serial number caching presented in the figure.

In a Discovery Service Unit, a Local Lookup Unit may contain a list of serial numbers referring to all pedigree documents residing in the local server. When a client queries for pedigree documents with specified serial numbers, the Partner Lookup Unit searches its local server as well as the Local Lookup Units of other Discovery Service Units. Once a client query reaches another service unit, it is forward onto other

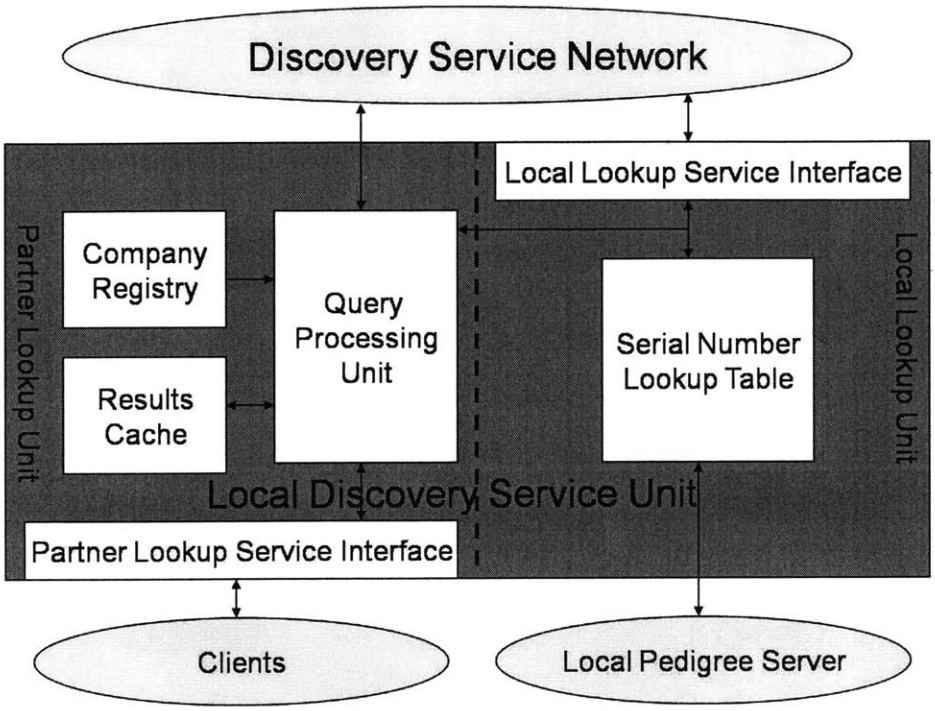

Figure 3-4: Discovery Service Unit Architecture

Discovery Service Units known by the current service, and the process continues. The above figure illustrates this principal. For instance, the client is connected to Unit 1, which is connected to Unit 2 and 3. But since Unit 3 is connected to Unit 4, the client query is forwarded onto Unit 4, which communicates with Unit 1 to answer the query. It is worthy to note that a service unit may refuse to forward a client query if the Time to Live (TTL) value of the query has been expired.

3.2.4 Design Components

This section describes in detail the design of the two components of the Discovery Service Unit. Figure 3-4 provides the layout of a typical Discovery Service Unit. Again, for explanation purposes, only pedigree serial numbers are mentioned instead of other pedigree attributes.

The Local Lookup Unit contains a Lookup Table composed of serial numbers and file location pointers. The table is updated as new pedigree documents are submitted to the local server. When a client request asking for documents with a particular

serial number is received, the Local Lookup Unit searches its Lookup Table, if the serial number exists, it retrieves the corresponding pedigree documents from its local server and forwards the query onto the Partner Lookup Unit. The Partner Lookup Unit would then forward the client query onto other Discovery Service Units, but only if the query's TTL has not yet been expired. The average pedigree document is usually associated with three businesses–such as a manufacture, a wholesaler, and a retailer; therefore a TTL value that permits the query to be sent to Discovery Service Units that are four degrees removed would be sufficient. Of course, this assumes that the client would have *some* justification for choosing the initial entry point where the query is made. However, this assumption may be dangerous. Choosing TTL values itself can be a challenging optimization problem, which I will discuss further in Section 5.2 of Chapter 5.

The Partner Lookup Unit is composed of a Partner Registry, a Results Cache and a Query Processing Unit. The Partner Registry contains a list of endpoints, or URLs, of each Discovery Unit known by the current unit. (In most situations, this list would contain Discovery Service Units that belong to frequent business partners.) The Results Cache contains query results that are recently returned by the network. When a client queries the Partner Lookup Unit, the service would first check with the Results Cache, and remote queries to partner Discovery Units are only made if the result is not already available in the cache. All cache results are kept for a predefined period of time, which may vary depending on network traffic. If the cache is full, it would function as a First In First Out (FIFO) queue. The Query Processing Unit handles the actual queries that flow through the network. When a query originated from a remote discovery service is being forwarded to another service, the Query Processing Unit sets its return address to that of the original service unit where the query is initiated, so results can be returned directly. As results come back from the network, (which do not have to be in the same order as it is sent out,) the Query Processing Unit packages them and returns them to the client.

3.3 Web Service Implementation

3.3.1 Implementation Overview

As mentioned earlier, pedigree metadata are extracted while files are being stored into local servers. These characteristic data may include serial number, drug name, business names, and other XML elements of a pedigree document. Clients may use any of these data fields as query parameters in their search. Again, for demonstration purposes, the implementation only considers pedigree serial number as a query parameter. Nonetheless, regardless of the query parameter used for searching, the process for how queries are performed remains the same.

The Discovery Service Unit is implemented using .NET technology as a set of two separated Web Services with different port locations. One implements the Partner Lookup Unit and the other the Local Lookup Unit. Each service contains only one method, which takes in a serial number as its input parameter and performed all the necessary processing for results to be returned. The service interface for each of the two services is shown below:

```
[ServiceContract]
public interface ILocalLookupService
{
[OperationContract]
    XmlDocument[] GetLocalData(Serial serial);
}
```

and

```
[ServiceContract]
public interface IPartnerLookupService
{
    [OperationContract]
    XmlDocument[] GetPartnerData(Serial serial);
}
```

The method `GetLocalData` takes in a serial number of user defined type `Serial` and searches its local Lookup Table, implemented as a `DataTable` object. If there is a match, it retrieves and returns pedigree documents in the format of `XmlDocument` arrays. Similarly, The method `GetPartnerData` also takes in a serial number of type `Serial`, sends a request to every Discovery Service Unit listed in the Partner Registry, and records the results in Results Cache before returning it to the client as `XmlDocument` arrays. The Partner Registry and Results Cache are also implemented as `DataTable` objects.

Windows Foundation Communication (WFC) is used, and both the `ILocalLookupService` and the `IPartnerLookupService` are self-hosted by performing the following:

```
ServiceHost host = new ServiceHost(typeof(LocalLookupService), baseURI);
host.Open();
```

and

```
ServiceHost host = new ServiceHost(typeof(PartnerLookupService), baseURI);
host.Open();
```

The `LocalLookupService` and `PartnerLookupService` inherit from the `ILocalLookupService` and `IPartnerLookupService` base class, respectively, and these services implement the logic behind the interfaces. The `baseURI` is a string that indicates the URL endpoint of the service.

When a Local Lookup service host opens, the Lookup Table is uploaded from local memory and updated as new pedigree documents are inserted or deleted from the local pedigree server. Uploading and deletion of data are performed atomically to prevent data inconsistency. When a Partner Lookup service host opens, its Partner Registry is loaded from memory and an empty Results Cache is created. The cache is gradually filled as client queries are processed. Again, this operation is performed atomically. Furthermore, since all results are stored in one instance of the cache, only

one instance of the service is created. All clients connect to the same instance. This is made possible by using WFC, which differs from a typical Web Service, where an instance is created for every client and discards after use. A WFC service remains alive as long as the host is open.

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
ConcurrencyMode = ConcurrencyMode.Multiple)]
```

In addition, to improve performance, requests sent to remote discovery units are made asynchronously, so as results come back (possibly in different order than it is being sent out), it can be assembled right away to be sent back to the client. A timer is set for each request so that the service does not wait forever for a response to comeback. Consequently, if certain remote services fails to respond, partial results can still be returned. The following code is taken from the Query Processing Unit, where queries are been sent out asynchronously to the Local Lookup Unit of another Discovery Service Unit:

```
ServiceEndpoint httpEndpoint = new ServiceEndpoint(
ContractDescription.GetContract(typeof(ILocalLookupService)),
    new WSHttpBinding(), new EndpointAddress(url));

ChannelFactory<ILocalLookupService> factory =
new ChannelFactory<ILocalLookupService>(httpEndpoint);
    ILocalLookupService svc = factory.CreateChannel();

IAsyncResult ar = svc.BeginGetLocalData(serial, null, null);
```

First, a `ServiceEndpoint` object is created to specify 1) the contract of the service `ILocalLookupService` and 2) the location of the service indicated by the `url` string. Then, the `ChannelFactory` creates an instance of `ILocalLookupService`, which calls `BeginGetLocalData` (instead of the normal synchronous method `GetLocalData`) to send a request to another Discovery Service Unit. Again, this method take in a

serial number of type `Serial`. As results come back, they are contained in the `IAsyncResult` object and can be retrieved by performing the following:

```
XmlDocument[] result = svc.EndGetLocalData(ar);
```

The `ILocalLookupService` calls `EndGetLocalData`, which takes in the `IAsyncResult` object and return the actual pedigree documents in `XmlDocument` array form.

3.3.2 The LocalLookupService Object

As mentioned earlier, the LocalLookupService takes in a serial number of type `Serial` and searches through its Lookup Table for a match. If a match exists, then it retrieves the corresponding pedigrees from the local server. This process is embedded in the method `GetLocalData` of class `LocalLookupService`. A section of this method is shown below:

```
DataTable table = LocalUnitHost.SerialLookUp.SerialTable;
DataRow row = table.Rows.Find(serial.Value);

XmlDocument[] result = null;
if (row != null) result = UtilitiesLocal.RetrivePedigrees(row);
```

The Lookup Table, called `SerialTable` is instantiated in the `LocalLookupService` host class `LocalUnitHost`. It contains two columns: 1) the serial number of type `Serial` and 2) the pointers to the corresponding pedigree file locations of type `string[]`. The `Value` property of the `Serial` object contains the actual serial number to be searched. If during the querying process a corresponding entry is found in the table, the helper method `RetrivePedigrees` would use the pointers to retrieve the pedigree documents from the server. In the current implementation, a separate program is in charge of filtering information from captured pedigrees and updating the Lookup Table. The current program only reloads the table from memory periodically, such as shown below:

```
if (TimeToReload())
{
    Thread thread = new Thread(new ThreadStart(UpdateTable));
    thread.IsBackground = true;
    thread.Start();
}
```

The `TimeToReload` method is set to *true* periodically, which spins out a (Thread) that updates the Lookup Table. The `UpdateTable` method simply loads the newly revised table from memory and merges the two tables.

```
DataTable newTable = UtilitiesLocal.LoadMyTable(FileLocation);
DataTable table = LocalUnitHost.SerialLookUp.SerialTable;
table.Merge(newTable, false);
```

The `FileLocation` parameter indicates the location of the Lookup Table in memory. Having the second parameter of the `Merge` method set to *false* guarantees that in case of conflict, incoming values can overwrite existing values in the table.

3.3.3 The PartnerLookupService Object

The PartnerLookupService answers client requests by forwarding the request onto Discovery Service Units listed in the Partner Registry. The query is sent out to the network through the Query Processing Unit if it is not found in Results Cache. This entire process is encapsulated in one method, called `GetPartnerData`, in the `PartnerLookupService` class. A section of this method is shown below:

```
DataTable rsltsTable = PartnerUnitHost.rsltsCache.ResultsTable;
DataRow row = rsltsTable.Rows.Find(serial);
if (row != null)
{
    return UtilitiesPartner.BuildResults(row);
}
```

```
else
{
    QueryProcessingUnit myQuery = new QueryProcessingUnit(serial);
    return myQuery.ExcuteQuery();
}
```

The method first checks the Results Cache, which is instantiated by the service host `PartnerUnitHost`. The cache `ResultsTable` is stored as a `DataTable` object. It has two columns: 1) The serial number of type `Serial`, 2) the previously returned pedigree results of type `XmlDocument[]`. If the query is found in the cache, the previously stored results would be returned by `BuildResults`; otherwise an instance of `QueryProcessingUnit` is created to send the search request out to the network. In order to send a query, the `ExcuteQuery` method must first obtain the URLs of its partner services from the Partner Registry. It then sends the request out asynchronously to Local Lookup Units of partner Discovery Service Unit, shown below:

```
DataTable registry = PartnerUnitHost.ptnerReg.RegistryTable;
ArrayList returnedRslts = new ArrayList();

foreach (DataRow row in registry.Rows)
{
XmlDocument[] result = UtilitiesPartner.SendRequest(row);
returnedRslts.AddRange(result);
}

XmlDocument[] results = (XmlDocument[]) returnedRslts.ToArray();
UtilitiesPartner.CacheInsert(serial, results);
return results;
```

Like the Results Ccache, the Partner Registry `RegistryTable` is also instantiated by the service host `PartnerUnitHost`. Client requests are sent out one-by-one asynchronously to partner service units through the method `SendRequest`. This asynchronous process is described in the above Section 3.3.1. As results come back from

the network, they are compiled into one arraylist named `returnedRslts`. Before results are returned, it is inserted into the cache by the `CacheInsert` method.

3.3.4 The PartnerLookupService Client

Anyone who wishes to utilize the discovery service can connect to the Partner Lookup Unit of a Discovery Service Unit. However, he or she would first need to build a service proxy for the PartnerLookupService and then connect to the proxy. For example, if a client wants to obtain all pedigree documents containing the serial number 12345, he or she would run the following program:

```
PartnerLookupServiceClient client = new PartnerLookupServiceClient();
DiscoveryServiceUnit.Serial serial = new DiscoveryServiceUnit.Serial();
serial.Value = "12345";

XmlDocuments[] resultPedigrees = client.GetPartnerData(serial);
```

The `PartnerLookupServiceClient` object, provided automatically by the proxy, allows the client to directly utilize the web service method `GetPartnerData`. The service endpoints are specified in a separate configuration file `app.config`. The following is a section of this file:

```
<client>
    <endpoint address="http://localhost:9000/PartnerLookupService"
        binding="wsHttpBinding"
        bindingConfiguration="WSHttpBinding_IPartnerLookupService"
        contract="IPartnerLookupService"
        name="WSHttpBinding_IPartnerLookupService">
        <identity>
            <userPrincipalName value="AUTO-A2F185FFE5\Indy" />
        </identity>
    </endpoint>
</client>
```

Chapter 4

A Centralized Approach to E-Pedigree Discovery Service

4.1 Previous Work

4.1.1 Design Motivation

A decentralized approach to electronic pedigree discovery service allows pedigree documents to be managed by individual supply-chain partners. However, this approach forces a client to wait for search results from multiply locations and is greatly dependent on network performance. A centralized approach, on the contrary, uses a much simpler search process. It simply requires a search to be conducted on the local file system. There would be no need to send search request out to the Internet, hence query performance can be greatly improved.

Furthermore, performing data search in local memory is a more manageable problem than architecting new network discovery protocols. Much research has been performed in this area. In this approach, the problem becomes 1) designing a search engine of high performance and 2) effectively managing large amounts of data. An example solution to the first problem is the Google File System [11]. An example solution to the second problem is the Google's BigTable, which implements a distributed storage system [3]. Another solution can be a distributed in-memory database pro-

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Liftoff News</title>
    <link>http://liftoff.msfc.nasa.gov/</link>
    <description>Liftoff to Space Exploration.</description>
    <language>en-us</language>
    <pubDate>Tue, 10 Jun 2003 04:00:00 GMT</pubDate>
    <lastBuildDate>Tue, 10 Jun 2003 09:41:01 GMT</lastBuildDate>
    <docs>http://blogs.law.harvard.edu/tech/rss</docs>
    <generator>Weblog Editor 2.0</generator>
    <managingEditor>editor@example.com</managingEditor>
    <webMaster>webmaster@example.com</webMaster>

    <item>
      <title>Star City</title>
      <link>http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp</link>
      <description>How do Americans get ready to work with Russians aboard the
        International Space Station? They take a crash course in culture, language
        and protocol at Russia's Star City.</description>
      <pubDate>Tue, 03 Jun 2003 09:39:21 GMT</pubDate>
      <guid>http://liftoff.msfc.nasa.gov/2003/06/03.html#item573</guid>
    </item>

    <item>
      <title>Astronauts' Dirty Laundry</title>
      <link>http://liftoff.msfc.nasa.gov/news/2003/news-laundry.asp</link>
      <description>Compared to earlier spacecraft, the International Space
        Station has many luxuries, but laundry facilities are not one of them.
        Instead, astronauts have other options.</description>
      <pubDate>Tue, 20 May 2003 08:56:02 GMT</pubDate>
      <guid>http://liftoff.msfc.nasa.gov/2003/05/20.html#item570</guid>
    </item>
  </channel>
</rss>
```

Figure 4-1: Example RSS Feed

posed by MIT Auto-ID Labs [16].

4.1.2 Overview of RSS

The centralized approach uses the Really Simple Synchronization (RSS) framework to capture and manage pedigree documents [20]. RSS forms a standard way to publish frequently updated digital content. Its most common application is in publishing of news blogs. A RSS Feed is analogous to a table of contents. Each entry in the Feed summarizes the content of some information located elsewhere and provides a link to the full content. RSS formats are specified in XML. Figure 4-1 illustrates a typical RSS feed.

Recently many extensions have been developed to support the use of RSS in different context. The extensions are predefined XML elements that fall under a

different namespace than that of the default namespace. The minimum requirement for an RSS entry, or an `<item>`, to be valid must include the following fields:

- Title
- Description
- Link

If extension elements are used, they are appended to the required elements and are child nodes of `<item>`. For instance, an extension to Yahoo! Weather feeds would include elements such as `<yweather:location>`, `<yweather:wind>`, and `<yweather:atmosphere>`, under predefined namespace `yweather`.

For the centralized e-pedigree discovery service, RSS is used along with the Google Base extension fields under the namespace g and `gbase`. This framework serves as the backbone to metadata storage of pedigree information, where characteristic attributes for each pedigree document is extracted and saved as an RSS entry in Google Base.

4.1.3 Overview of Google Base

Google Base allows Google account users to freely publish RSS feeds under their own accounts for the public to browse. The information posted by the public can be informational, such as recipes and personal profiles, or commercial, such as housing and products for sale. The following is an example RSS entry found on the website:

All items published are accessible through the use of the Google Query Language [12]. For instance, if one wants to search for a digital camera that is under $500, he or she perhaps would construct a query like the one shown below:

```
digital camera [price <= 500.0 USD]
```

A matching entry would contain the phrase `digital camera` in the `<title>` or `<content>` field, and it would also contains a field `<g:price>` whose value is less than 500 USD. The example entry shown above in Figure 4-2 satisfies these criteria.

```
<entry>
  <id>http://www.google.com/base/feeds/snippets/9026618904664888476</id>
  <published>2007-05-26T03:03:08.000Z</published>
  <updated>2007-05-26T04:47:14.000Z</updated>
  <category scheme='http://base.google.com/categories/itemtypes' term='Products'></category>
  <title type='text'>Canon Powershot SD550 Digital Camera Battery Door</title>
  <content type='html'>SquareTrade Â® AP6.0 Please Read Entire Description Before Bidding or Buying.Item Description: This is a
  <link rel='alternate' type='text/html' href='http://adfarm.mediaplex.com/ad/ck/711-5256-8196-2?loc=http%3A%2F%2Fcgi.ebay.com%2
  <link rel='self' type='application/atom+xml' href='http://www.google.com/base/feeds/snippets/9026618904664888476'></link>
  <author>
    <name>eBay</name>
    <email>rcross@ebay.com</email>
  </author>
  <g:brand type='text'>Digital Camera Battery</g:brand>
  <g:item_type type='text'>Products</g:item_type>
  <g:item_language type='text'>EN</g:item_language>
  <g:price type='floatUnit'>9.95 usd</g:price>
  <g:target_country type='text'>US</g:target_country>
  <g:image_link type='url'>http://thumbs.ebaystatic.com/pict/140122326763_1.jpg</g:image_link>
  <g:category type='text'>Cameras &amp;amp; Photo&amp;gt;Digital Cameras&amp;gt;Parts &amp;amp; Repair</g:category>
  <g:customer_id type='int'>11729</g:customer_id>
  <g:id type='text'>140122326763</g:id>
  <g:expiration_date type='dateTime'>2007-06-25T03:03:08.000Z</g:expiration_date>
</entry>
```

Figure 4-2: Example Google Base Entry

To further assist users, a programming API for Google Base was developed [13]. The GData API not only allows Google Base users to programmatically create, revise, delete their own items in Google Base but also provides a channel for users to search for publish items that can automatically be fed into user applications. Client libraries are available in C# and Java. For this implementation, the C# libraries were used along with ASP.NET to construct the Web Service framework described in Section 4.3.

4.2 Design Architecture

4.2.1 Design Assumptions

This design assumes that all pedigree documents submitted to the central pedigree server satisfies the standard document format stated in the *Pedigree Ratified Standard v1.0* [9], so that XML elements, or pedigree data fields, can be extracted to use as metadata. In addition, this design does not involve the use of any security mechanism. As stated earlier, pedigree documents by law must be open to the public. Furthermore, this design requires a constant connection to Google Base servers. Although in the future a separate search engine will be built, the current implementation relies heavily on Google Base to perform the search of pedigree documents. Finally, and most importantly, this design only attempts to solve the data discovery

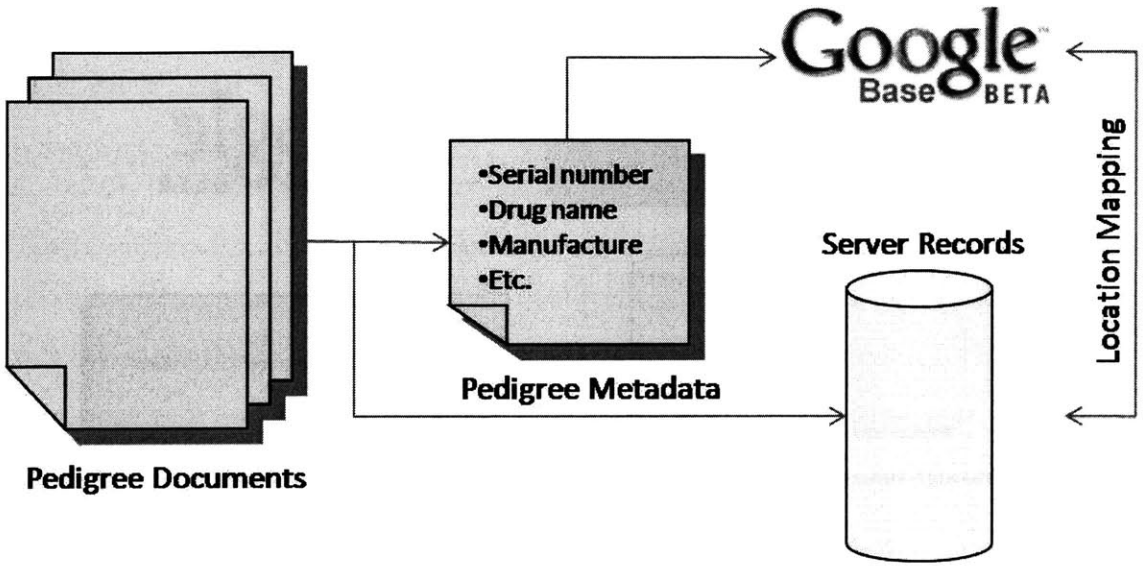

Figure 4-3: Pedigree Capture Process Flow Diagram

problem and does not propose a solution for the data storage problem. Hence, it is assumed that the central pedigree server has the memory capacity to store all the pedigree documents submitted to the server.

4.2.2 E-Pedigree Capture Process

The centralized pedigree discovery service can be broken down into two parts: Capture and Query. The pedigree capturing process is shown in the following figure.

As a pedigree document is submitted to the server, it is filtered to extract characteristic data. The data fields, or XML elements, extracted are the fields that are most representative of and whose values are the most unique to the pedigree documents that contains them. This metadata information is pushed onto Google Base as an `<item>` of a RSS feed, The full document is stored in the local file system. It is important to note that each entry fed to Google Base not only includes the metadata but also the pointer to the file location in the server, so one can effectively retrieve desired documents after matching metadata are found as a result of a client query.

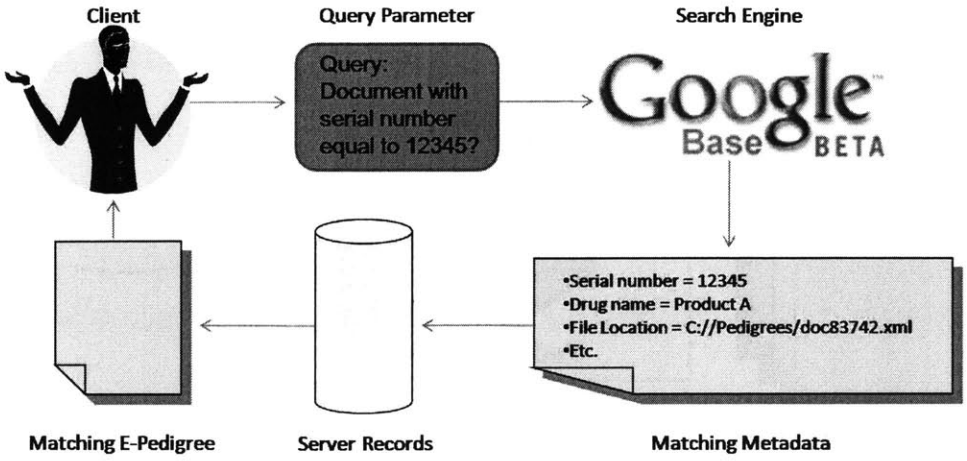

Figure 4-4: Pedigree Query Process Flow Diagram

4.2.3 E-Pedigree Query Process

An example pedigree querying process is shown below in Figure 4-4. For instance, if a client wants to obtain all pedigree documents containing the serial number 12345, the discovery service application would submit a query to Google Base, and Google Base would return all RSS entries with the `serialNumber=12345`. For each entry that is matched, the service application would use the corresponding file path, such as `fileLocation=C://Pedigrees/doc83742.xml`, to obtain the matched documents in the server. These documents are then returned to the client.

4.3 Web Service Implementation

4.3.1 Implementation Overview

Same as the decentralized discovery service implementation, the centralized version also uses .NET technology and the Web Services framework. The following figure is a screen capture of this service shown in a web browser.

As shown above, the web service only supports two web methods: `PedigreeCapture` and `PedigreeQuery`. The following code is a stripped-down version of this service.

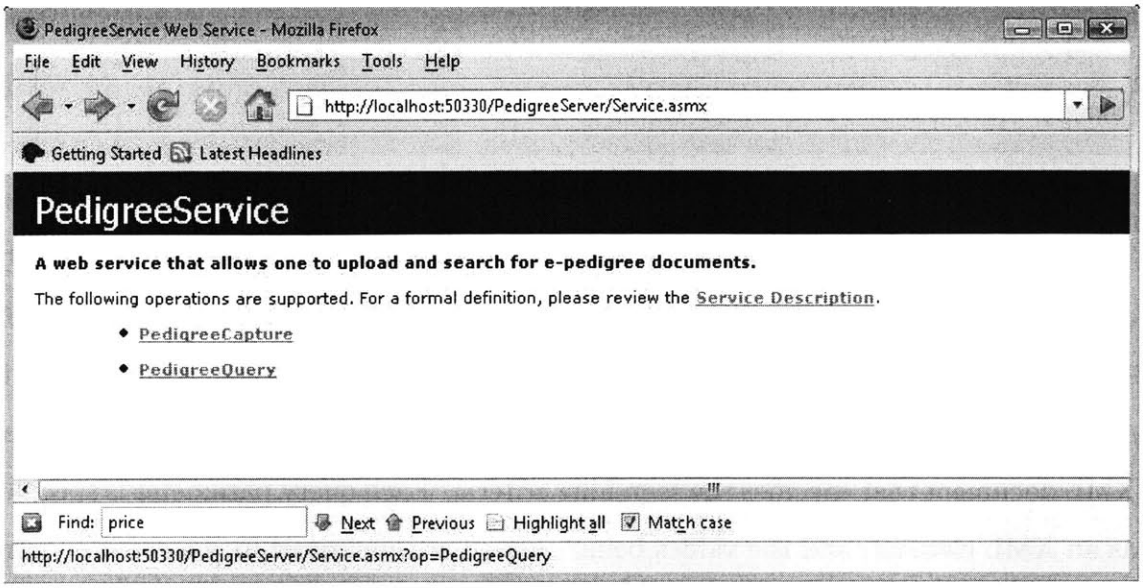

Figure 4-5: Screen Capture of Web Service Interface

```
[WebService(Namespace = "PedigreeServer",
    Description = "<b>A web service that allows one to upload
                  and search for e-pedigree documents.</b>")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class PedigreeService : System.Web.Services.WebService
{
[WebMethod]
public void PedigreeCapture(XmlDocument pedigree, string filename)
    {
    PedigreeCapture capture = new PedigreeCapture(pedigree, filename);
       capture.StorePedigree();
    }

    [WebMethod]
    public XmlDocument[] PedigreeQuery(XmlDocument queryDoc)
    {
       PedigreeQuery query = new PedigreeQuery(queryDoc);
        return query.Run();
```

```
    }
}
```

The name and the description of the service are specified as web service attributes. The pedigree document submitted through the `PedigreeCapture` method must be of type `XmlDocument`. The `StorePedigree` method stores the input pedigree in the server and pushes its corresponding metadata onto Google Base. If one wants to submit a query through the web method `PedigreeQuery`, he or she must submit an XML document that specifies the searching criteria. Each query parameter is specified as an XML element, and the values being queried are contained in the corresponding elements. The following example XML document queries for pedigrees with serial number equal to 12345 and drug name equal to Product A.

```
<serialNumber>12345<\serialNumbe>
<drugName>Product A<\drugName>
```

The `Run` method processes the query parameter document, searches though Google Base for matching pedigrees, and returns the pedigrees in `XmlDocument[]` format.

4.3.2 The PedigreeCapture Method

The `PedigreeCapture` captures pedigree document in three steps: 1) attribute extraction, 2) file storing, and 3) attribute publishing. The following code encapsulates what has just been described:

```
PedigreeAttributes pedAttr = ExtractAttributes();
GBaseService service = new GBaseService("Pedigree Discovery", developerKey);
service.setUserCredentials(username, password);
PublishEntry(pedAttr, service);
```

The `ExtractAttributes` method processes the XML document and extracts characteristic data as metadata. In the currently implementation, the following fields are being extracted:

- Serial Number
- Drug Name
- Transaction ID
- Sender Business Name
- Recipient Business Name
- Source Routing Code
- Destination Routing Code

One or more of these XML elements shown above may be empty in the pedigree document. If a search string uses a data field that is empty in the pedigree, the document will simply be skipped. The fields are stored as properties of the `PedigreeAttributes` object. After the desired elements are extracted, the pedigree is saved in the local file system and the path of the file location is returned. The file location is also saved as a property of `PedigreeAttributes`. The following code is a stripped-down version of `ExtractAttributes`:

```
PedigreeAttributes ped = new PedigreeAttributes();
XmlNodeList serials = GetNodes(pedigree.DocumentElement, "//serialNumber");
ped.SerialNumber = serials;
//Extract other properties
...
string fileLocation = GetStorageLocation();
ped.FileLocation = fileLocation;
return ped;
```

As shown above, XML elements are extracted one by one and assigned as pedigree attributes, including the file location. The `GetNodes` method extracts XML elements based on the XPath expression given. The `GetStorageLocation` method simply save the pedigree document to the local server and return a string that indicates its file path.

After attributes are extracted, a `GBaseService` is instantiated. The Google Base account where pedigree metadata is stored requires a username and password to prevent outsiders from tampering with this information. After security credentials are set, the `PublishEntry` method pushes the new pedigree information onto Google Base through the use of `GBaseService` methods. A stripped-down version of the `PublishEntry` method is shown below:

```
GBaseEntry entry = new GBaseEntry();
foreach (XmlNode serial in ped.SerialNumber)
{
    entry.GBaseAttributes.AddTextAttribute("serialNumber", serial.InnerText);
}
//Assign other attributes
...
GBaseEntry myEntry = service.Insert(GBaseUriFactory.Default.ItemsFeedUri,
    (GBaseEntry)entry);
```

An RSS entry of type `GBaseEntry` is instantiated. Metadata fields are assigned as `GBaseAttributes` objects. After all the attributes are assigned, the entry is inserted into Google Base through the `Insert` method of the `GBaseService` object.

4.3.3 The PedigreeQuery Method

The `PedigreeQuery` method first forms a query string based on the query parameters given in the client's input XML document, such as the one shown in Section 4.3.1. The method then submits the query to Google Base to retrieve an RSS feed containing the pedigree entries that have matched the request. This process is illustrated in the following code:

```
foreach (XmlNode query in queryParams.ChildNodes)
{
    if (query.LocalName.Equals(QueryType.SerialNumber))
    {
```

```
        GBaseQueryString = GBaseQueryString +
            " [SerialNumber: *" + query.InnerText + "*]";
    }
    else if (query.LocalName.Equals(QueryType.DrugName))
    {
        GBaseQueryString = GBaseQueryString +
            " [DrugName: *" + query.InnerText + "*]";
    }
    //Other query parameters are checked
    ...
}
GBaseService service = new GBaseService("Pedigree Discovery", developerKey);
service.setUserCredentials(username, password);
GBaseQuery GBquery = new GBaseQuery(GBaseUriFactory.Default.ItemsFeedUri);
GBquery.GoogleBaseQuery = GBaseQueryString;
GBaseFeed feed = service.Query(GBquery);
```

The `GBaseQueryString` is an concatenation of all the query parameter-value pairs expressed in the Google Query Language format mentioned in Section 4.1.3. After the query string is constructed, a new `GBaseService` is instantiated, and user credentials are set. Then, A `GBaseQuery` object is instantiated to encapsulate the query string. The `GBaseService` takes in this object and sends a request to Google Base using its `Query` method. The returned results are RSS entries encapsulated in the `GBaseFeed` object.

Once results are returned, each entry of the feed is extracted as an `GBaseEntry` object. Using the `GBaseAttribute` pointer location of each entry, the program obtains the corresponding XML documents from the local server. Finally, these XML documents are compiled and return as an `XmlDocument[]` object. The following code illustrates this process:

```
foreach (GBaseEntry entry in feed.Entries)
{
```

```
    foreach (GBaseAttribute attr in entry.GBaseAttributes)
    {
        if (attr.Name.Equals("serverlocation"))
        {
            XmlDocument doc = new XmlDocument();
            doc.Load(attr.Content);
            results.Add(doc);
        }
    }
}
```

4.3.4 The Pedigree Server Client

A client who wants to connect to this centralized pedigree discovery service intends to either submit a pedigree document or query the discovery network. Like all Web Services clients, anyone who wishes to utilize this service must first create a service proxy and add it as a web reference. Once the proxy is set up, web methods can be directly used by client applications. The following code instantiates a service proxy:

```
PedigreeServiceProxy.PedigreeService  service =
    new PedigreeServiceProxy.PedigreeService();
```

The endpoint URL of this service is specified in a separate configuration file. A section of this file is shown below:

```
<applicationSettings>
    <PedigreeServerClient.Properties.Settings>
        <setting name="PedigreeServerClient_PedigreeServiceProxy_PedigreeService"
            serializeAs="String">
            <value>http://localhost:50330/PedigreeServer/Service.asmx</value>
        </setting>
    </PedigreeServerClient.Properties.Settings>
</applicationSettings>
```

To submit a document into the server, one can call the `PedigreeCapture` method with the appropriate parameters, such as the example shown below:

```
service.PedigreeCapture(pedigree, "My Pedigree");
```

The first input parameter `pedigree` is an `XmlDocument` object containing the actual pedigree. The second input parameter is a name with which the client wants to associate his or her pedigree. This name will be used as the *Title* field to the associated metadata entry submitted Google Base.

Similarly, to query the network, the `PedigreeQuery` method should be called, such as the one shown below:

```
XmlDocument[] results = service.PedigreeQuery(queryParams);
```

The input parameter `queryParams` is an XML document similar to the one shown in Section 4.3.1. The results returned from the network are XML documents of type `XmlDocument[]`, which can potentially be further processed by clients applications.

Chapter 5

Discussion and Future Work

5.1 Analysis of the Two Design Approaches

Both the decentralized and the centralized approach presented in this Thesis have its benefits and weaknesses. The following sections analyze the design trade-offs of each approach.

5.1.1 Evaluation of the Decentralized Approach

The decentralized approach allows pedigree documents to be stored in business owner's local servers. First of all, this approach is perhaps preferred by most businesses, who are skeptical about a third-party intervening. Secondly, this approach eliminates the possibility of having a centralized point of failure. If one pedigree server fails, the performance of the network is completely unaffected, since a time-out is imposed on all client queries. If at a later time the broken server rejoins the network, again, no other servers would be affected by this change.

Although the decentralized approach is easier to manage and reduces the possibility of having a single point of failure, it has many weaknesses. For one, it is a *best effort* network and does not guarantee to return all the results that satisfy the searching criteria. The amount of results returned is greatly dependent on the entry point of the network, or the Discovery Service Unit to which the client has chosen to

connect. All service units have a predefined list of partner services that it is connected to. Consequently, there is a limitation to the querying range. To obtain quality results, clients must have some prior business context knowledge of the service unit to which he or she has connected. In the worst case, it is possible to have a client request expire without returning any desired results.

The biggest weakness of this design is the problem of scalability. As more Discovery Service Units join the network, the Partner Registry will grow with speed of $\Omega(n)$, where n is the number of discovery units in the network. Assuming the delay due to network traffic is insignificant and the service unit computation time is negligible, this value will be the amount of time it takes for a single client request to be processed. If it takes constant time, or $\Omega(1)$, to search through a finite list of pedigree documents records within one Discovery Service Unit, as the TTL value of a client request approaches ∞, the number of times the request get passed on from one unit to another is $\Omega(\lg(n))$. Consequently, in the worst case scenario, the time it takes for a client request to be answered is $\Omega(n) \times \Omega(\lg(n)) = \Omega(n \lg(n))$. This performance is inefficient. Hence, the decentralized design presented in this Thesis does not solve the problem of discovery and can only serve as a starting point.

5.1.2 Evaluation of the Centralized Approach

The search process for the centralized approach is much simpler than the decentralized approach. All pedigree documents are managed in one centralized pedigree server, so answering a client query does not require submitting requests to a network of remote servers. As a result, query processing speed can be greatly reduced. It takes constant time to extract metadata from a pedigree, to push the metadata onto Google Base, and to store the pedigree document into the server. Hence, the time it takes for a pedigree document to be submitted to a server is $\Omega(c) \approx \Omega(1)$, where c is a constant equal to the number of operations performed by the pedigree capturing service.

On the querying side, it takes constant time to form a query string and submit it to Google Base, approximately $\theta(m)$ to conduct a metadata search, where m is the number of RSS metadata entries stored, and constant time to retrieve the pedigree

document associate with the metadata. Altogether, a single client query takes $\Omega(m)$ to process. Although m >n, or the number of pedigree documents stored in the server is greater than the number of discovery units in the decentralized network, possibly even by orders of magnitude, it is still reasonable to believe that the centralized approach, by design, would outperform the decentralized approach, especially as the network grows and more pedigree documents are generated.

However, this centralized approach also has some weaknesses. For instance, having all pedigrees stored in one location introduces the possibility of having a single point of failure. If the web service endpoint breaks, clients can no longer submit pedigrees nor query them. If the hardware storing the documents becomes vastly damaged, pedigree data may be permanently lost. If too many clients are connected to the server, the endpoints can be overloaded and the network saturated. A solution to how client requests can be evenly distributed among the network entry points remains unsolved. As a result, if the centralized approach were to be implemented, many other associated issues like the ones mentioned above would need to be addressed concurrently.

5.2 Future Work

5.2.1 Performance Analysis

Although prototype implementations are available for both the decentralized and the centralized pedigree discovery services, their performance analysis is yet to be completed. For the decentralized design, a test is needed to indicate the scalability of the network by measuring the delay in client responses and the quality of results returned due to network traffic caused by sending large amounts of round-trip queries. This might involve exploring different TTL values for client requests. The goal it to find an optimal TTL value such that there is a balance between the time delay for a request and the number of desired pedigree documents returned. Furthermore, one may wants to test query load limitations to determine what threshold values would undesirably saturate the network. A another test may involve evaluating data

consistency and the robustness of the Results Cache, as multiple threads attempt to read and write the cache. Similarly, for the centralized design, it is important to measure the average amount of time it takes for a request to be returns 1) when many clients attempts to access the server concurrently and 2) when the number of pedigree document stored in the server becomes extremely large.

5.2.2 Improvements

Many improvements can be made and many extra features can be added to the decentralized pedigree discovery service. For instance, an automated registration system can be implemented so that as new discovery service units enter the network, it broadcasts itself to all other units in its Partner Registry so that other service units can have the option to add the new unit to their own registry. In addition, the processing of updating the pedigree Lookup Table can be automated. Instead of reloading the table periodically, a mechanism should be implemented such that whenever a pedigree is submitted to the server, its information is automatically added to the Lookup Table. A publication-subscription model may be ideal in this case. Furthermore, message forwarding needs to be added to this implementation, and a desired TTL value needs to be determined. Since the TTL value of client requests is crucial to network performance, it may be beneficial to keep TTL values as a dynamic variable linked to real-time network performance, and heuristics can be added to further optimize these values.

Similarly, further work needs to be performed on the centralized service. For instance, all pedigrees are currently stored in one server. As the number of pedigrees documents increases and more servers are added to provide extra storage space, a mechanism needs to resolve the issue of load balancing of documents among the servers. Furthermore, to present lost of information due to server failures, document replicas should be made. Hence, one needs to research and implement a mechanism to effectively create pedigree replicas and handle unexpected server failures. Finally and more importantly, since it would not be surprising if pharmaceutical companies are distrustful of using a third party company, such as Google, to manage their informa-

tion, a reliable search engine should be implemented to replace all the functionalities used in this application currently offered by Google Base.

5.2.3 Extensibility to Other Applications

The two approaches presented in this Thesis are for the RFID applications of Electronic Drug Pedigree. However, these approaches may also be applicable to other applications. They may serve as starting points for building other discovery service systems such as that for EPC Information Service. The EPCIS Repository contains information about business events related with particular EPCs. These data are managed locally by individual businesses with different EPCIS applications.

For the decentralized approach, a Discovery Service Unit may connect with a EPC Information Service (EPCIS). Its Lookup Table may store EPCs instead pedigree attributes. The information returned to the client may be EPC events instead of pedigree documents. For the centralized approach, the same logic applies–one may submit and query for EPCIS events instead of pedigree documents. The metadata extracted from EPC events used by the search engine can be EPCIS query parameters, such as EPC Values and Record Times, instead of Serial Numbers and Drug Names.

Chapter 6

Conclusion

In this Thesis we have presented two approaches for the design of the Electronic Drug Pedigree (e-pedigree) discovery system–one decentralize, one centralized. The decentralized service is desired when businesses in the health care industry–such as drug manufactures, wholesaler, and retailers–do not trust third parties to manage their company information and prefer to store and maintain their pedigree documents locally. A discovery service protocol is proposed for this approach, and the building blocks of the service are referred to as Discovery Service Units. Each Discovery Service Unit is attached to a local server belonging to a particular business. It has a Partner Registry of other service units in the network and answers client requests by communicating with its partners services. Client requests are forwarded from one service unit onto another.

For this approach, quality results are returned when a client has a good understanding of the service unit to which he or she has connected. An e-pedigree document is already a track-and-trace record and shows transactions between business partners. If a company's Discovery Service Unit includes all the businesses it works within the Partner Registry list, it would not be surprising if one or multiple of its business partners have information on a pedigree document that it also has. Hence, choosing the Discovery Service Unit with which to initiate a connection is critical in determining the time delay for a client to receive a response and the completeness of matching pedigree documents returned. For the same reason, choosing the Time-to-Live (TTL)

value of a client's request also plays an important role in determining the quality of results returned.

The centralized service does not require a new discovery protocol to be designed. It simply requires that e-pedigree documents be stored in one location and a search engine be implemented to effectively answer client queries. The search engine should use heuristic information of pedigrees, such as metadata of characteristic attributes, to query for desired results.

In the currently implementation, Google Base is used to fulfill the role of the search engine. When pedigree documents are being submitted, characteristic attributes are extracted and pushed onto Google Base along with the server file location. Metadata are being pushed as an RSS feed, and each pedigree is associated with one entry of the deed. When a client sends a request to the server, these attributes are can be used as query parameters. Therefore, we can see that the role of the search engine is the heart of this design. Furthermore, a good search engine cannot function without effective searching parameters. Determining what attribute data of a pedigree document from which the service should extract also becomes a significant factor that affects its performance.

In terms of pedigree capturing, both designs perform the same and can store document in constant time. In terms of pedigree querying, the performance of the two approaches vary significantly. For instance, the performance of the decentralize approach depends heavily on the size of the network, such as the number of service partners each discovery service unit have. The larger the network, the longer it would take a request to return matching results, the less complete the results would be.

On the contrary, the centralized approach is much more scalable. Its performance does not depend on the number of business participants, but simply the number of documents submitted to the server. If the server becomes overloaded, the documents can be redistributed among others servers. Replicas can be made to prevent server failures. Therefore, although further performance analysis needs to be conducted for both pedigree discovery service implementations, it seems that by design, the centralized approach may potentially outperform the decentralized approach.

Appendix A

Class Diagrams for the Decentralized Implementation

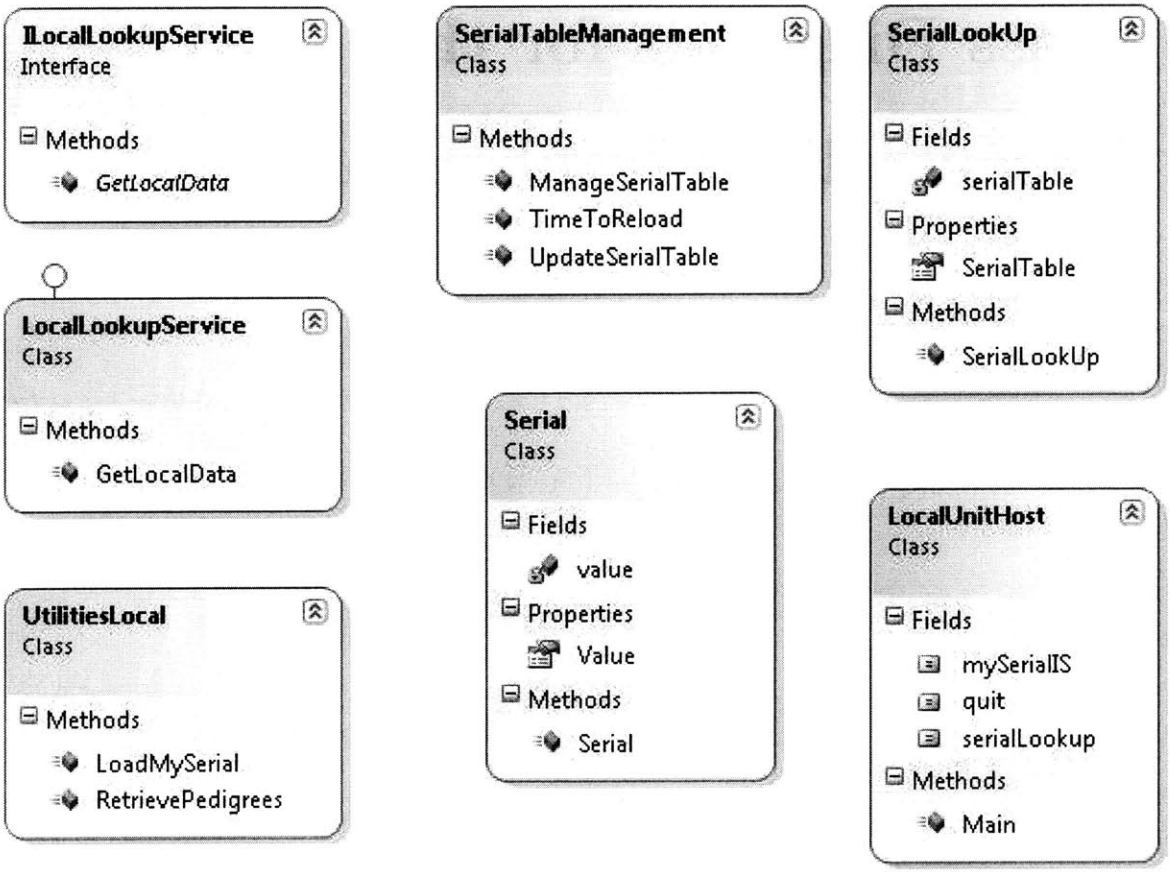

Figure A-1: Class Diagram for the Local Lookup Service

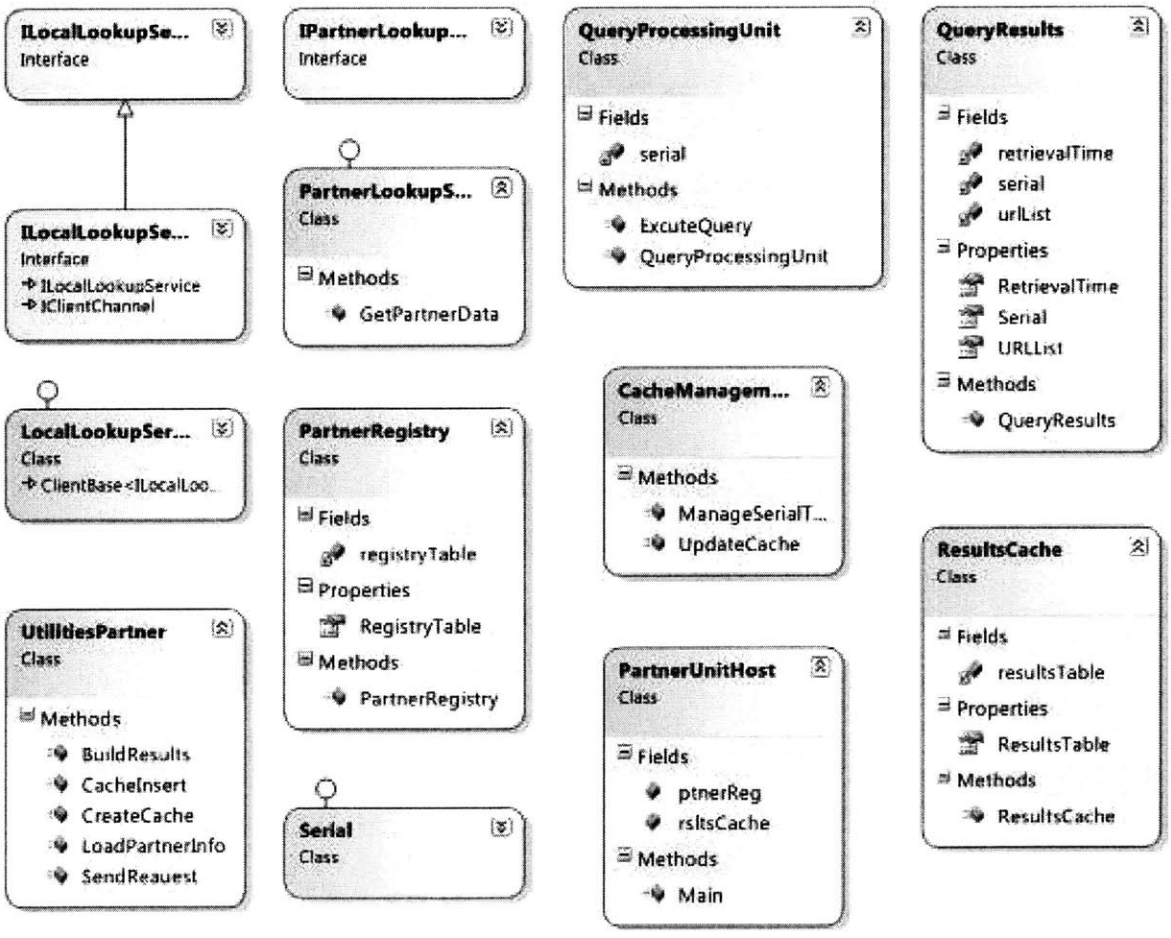

Figure A-2: Class Diagram of the Partner Lookup Service

Appendix B

Class Diagrams for the Centralized Implementation

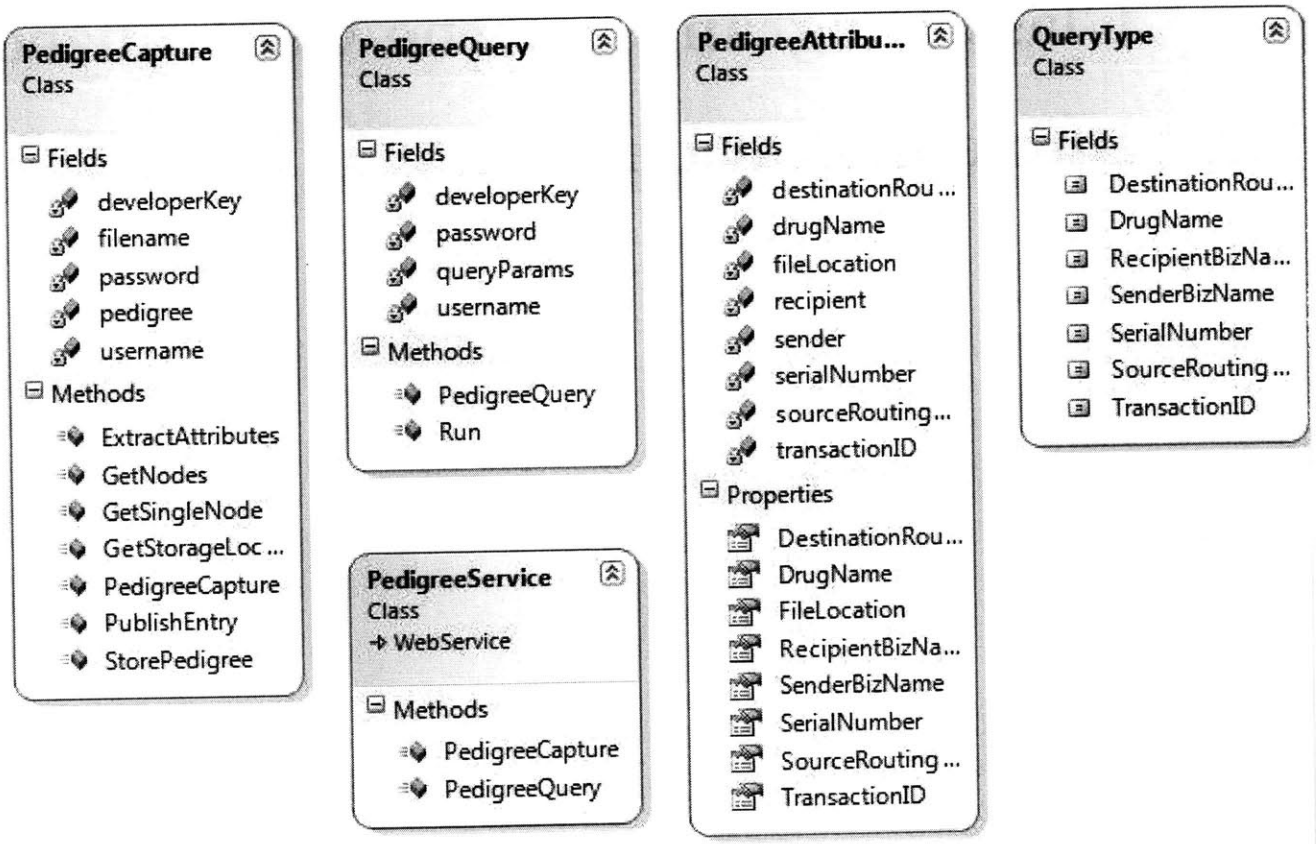

Figure B-1: Class Diagram of the Web Service

Bibliography

[1] Steve Beier, Tyrone Grandison, Karin Kailing, and Ralf Rantzau. Discovery services–enabling traceability in epcglobal networks. Delhi, India, December 2006. Indian Institute of Technology. International Conference on Management of Data.

[2] Mike Celentano, Daniel Engels, and Ted Ng. Staying ahead of pedigree laws, May 2007.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. Seattle, WA, USA, November 2006. USEMX. OSDI.

[4] Salutation Consortium. *Salutation Architecture Specification, Version 2.0c*, June 1999.

[5] Microsoft Corporation. *Universal Plug and Play Device Architecture, Version 1.0*, June 2000.

[6] EPCGlobal. *The EPCGlobal Architecture Framework.* EPCGlobal Standard Specification, July 2006.

[7] EPCGlobal. *EPC Information Services (EPCIS), Version 1.0.* EPCGlobal Standard Specification, April 2007.

[8] EPCGlobal. *Low Level Reader Protocol (LLRP), Version 1.0.* EPCGlobal Standard Specification, April 2007.

[9] EPCGlobal. *Ratified Pedigree Standard, Version 1.0.* EPCGlobal Standard Specification, January 2007.

[10] Alex Ferrara and Matthew MacDonald. *Programming .NET Web Services.* O'Reilly, September 2002.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. Bolton Landing, NY, USA, October 2003. ACM SIGOPS. SOSP.

[12] Google. Google query language specification, 2007. http://code.google.com/apis/base/query-lang-spec.html.

[13] Google. *Using the Google Base Data API C# Client Library,* 2007. http://code.google.com/apis/base/csharpdevguide.html.

[14] GS1. Building radio frequency identification solutions for the global environment, March 2005.

[15] E. Guttman, C. Perkins, J. Veizades, and M. Day. *Service Location Protocol, Version 2.* IETF Network Working Group, June 1999.

[16] Sergio Herrero, Abel Sanchez, and John Williams. Distributed in-memory database for epc information services. April 2007.

[17] R. Housley, W. Ford, W. Polk, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile.* IETF Network Working Group, January 1999.

[18] Sun Microsystems. *Jini Architecture Specification, Version 1.2,* December 2001.

[19] Sanjay Sarma, David Brock, , and Daniel Engels. Radio frequency identification and the electronic product code. *IEEE Micro,* 21(6):50–54, 2001.

[20] Dave Winer. *RSS Specification, Version 2.0.* Berkman Center for Internet and Society, July 2003.