INSTITUTO TECNOLOGICO Y DE ESTUDIOS
SUPERIORES DE MONTERREY
CAMPUS MONTERREY
GRADUATE PROGRAM IN MECHATRONICS AND
INFORMATION TECHNOLOGIES

TECNOLÓGICO
DE MONTERREY®

ACCESS AND BRANCH MANAGEMENT ON DISTRIBUTED PERSONAL DIGITAL LIBRARIES

THESIS

PRESENTED AS A PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE WITH MAJOR IN
INFORMATION TECHNOLOGY

BY

MARTIN ALEXANDER CASTELLANOS ALVAREZ

MONTERREY, N. L. DECEMBER, 2011

INSTITUTO TECNOLOGICO Y DE ESTUDIOS
SUPERIORES DE MONTERREY
CAMPUS MONTERREY
GRADUATE PROGRAM IN MECHATRONICS AND
INFORMATION TECHNOLOGIES

TECNOLÓGICO
DE MONTERREY®

ACCESS AND BRANCH MANAGEMENT ON DISTRIBUTED PERSONAL DIGITAL LIBRARIES

THESIS

PRESENTED AS A PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE WITH MAJOR IN
INFORMATION TECHNOLOGY

BY

MARTIN ALEXANDER CASTELLANOS ALVAREZ

MONTERREY, N. L. DECEMBER, 2011

ACCESS AND BRANCH MANAGEMENT ON DISTRIBUTED PERSONAL DIGITAL LIBRARIES

BY

MARTIN ALEXANDER CASTELLANOS ALVAREZ

THESIS

GRADUATE PROGRAM IN MECHATRONICS AND INFORMATION TECHNOLOGIES

THIS THESIS IS A PARTIAL REQUIREMENT FOR THE DEGREE OF MASTER OF SCIENCE WITH MAJOR IN INFORMATION TECHNOLOGY

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

DECEMBER 2011

Dedication

This thesis is dedicated to my mother, who has always supported me and has been there when I needed the most, she is my role model and I thank her for teaching me to work hard for the things I want in life.

Acknowledgments

I would like to offer my sincerest gratitude to my advisor, Dr. Juan Lavariega, who has very patiently guided and supported me through the entire thesis process.

I thank the open source community, specially the people responsible for the development of CouchDB and its supporting tools for their excellent work and the effort they do supporting the community on their free time.

Abstract

Due to the great volume of digital documents that currently exist, several technologies have appeared to organize, maintain and help users process such amount of information, among them are the personal digital libraries, they allow the creation and administration of collections of digital documents. There are plenty of personal digital library systems available, from general purpose to very specialized versions focusing on a single topic of the domain area, some of them are very popular and have been deployed on several organizations, but there are still areas of opportunity that have not been covered and one that's fundamental for team work and that most of them lack, is a distributed collaboration model without the intervention of a central coordinator, the creation of such model this would give users the necessary tools to implement distributed library cooperation, discover, consume and contribute to public material available in other libraries, and achieve true portability and ubiquitous access even without an Internet connection. The present work proposes a collaboration model that allows users to discover and branch collections built by other users of personal digital libraries while giving users the control and management tools to handle changes and conflicts between collections.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays everyone of us have a great amount of documents and digital files that play a very important role in the daily activities, therefore it is important to be able to access this information in an ubiquitous way, this is possible now because of the Internet and technologies such as digital libraries.

Personal digital libraries systems have been created in order to fulfill the need of maintaining and organizing documents into collections defined by the users according to their needs. Personal digital libraries give the users the capability to manage and access their information in an efficient and convenient way.

Personal digital libraries can achieve another important function, which is provide users with the means to collaborate in order to create and manage better collections of documents, users in different locations can create, manage and share their collections or contribute into other users or libraries collections.

As part of this work we present a system for basic digital library services such as collection creation and management, metadata organization, local and distributed search, and user administration. This system will be the foundation of the project on which we will proposes the collaboration model that allows users to discover and branch collections built by other users, this model improves the reliability of the system by distributing the data, enhances the mobility and performance aspects by giving users the possibility of working with local data and giving them the tools to manage conflicts and merge their work with other servers once they are done with their changes, all of this available through a RESTful interface that gives users the possibility of extending the system.

The importance of this research comes from the need to have better tools to collaborate with others that are working on similar topics, current personal digital libraries mechanism are very limited, only providing users with simple replication of external sources but they don't provide services to turn that collaboration into an ongoing process, teams need to be able to work with each other asynchronously, independently of geographic location or limited connectivity. The proposed system gives users and administrators to massively

distribute the application, this could be a valuable tool for collaboration environments like universities and colleges, where vast amounts of documents are produced everyday, the implementation of this system on such environment would generate a very rich knowledge repository within a short time. Another important aspect is that new research could take advantage of the implemented platform that has been created during this research.

1.1 Problem Definition

There's not a comprehensive collaboration mechanism for the development of collections in personal digital library systems, this problem prevents users of the library from discovering, referencing and contributing to the large amount of high quality documents that are generated on the library, limits the creation of important connections with colleagues working on the same subjects, forces users to rely on external tools to handle the collaboration and merging aspects of the work, makes users depend on a unreliable centralized system to hold their information, creates a dependency to the availability of a network connection which might not be available at all times and when available it is the reason of delays which could be avoided by taking advantage of local data.

For all of these problems we propose a solution based on a RESTful peer to peer model of collaboration and interoperability that allows users to search, branch and contribute to public collections on different library servers, giving them the freedom to work and access these collections from their local environment, therefore improving performance and removing the network connection dependency.

1.2 General Objective

- Define a distributed collaboration model for collections in personal digital libraries.

1.3 Specific Objectives

- Allow users to search across public libraries in distributed servers.
- Allow multiple levels of branching of a collection.
- Provide users with the necessary tools for merging their work with other collaborators.
- Provide access to the collections in the system during down time or with limited connectivity.

1.4 Hypothesis

It's possible to create a multi-user collaboration model for the development and maintenance of distributed personal digital library collections using a RESTful architecture and replication services based on a branch-pull-merge mechanism.

With the purpose of showcasing the advantages that this model would bring to the users of digital libraries, here are some scenarios where this model brings get advantages to the users' library:

- A student is looking for a research topic for his thesis. He has interest in digital libraries so he goes to his personal digital library and searches about this topic, a global search is issued and the results are presented to him. Upon inspecting the results, he notices that the library of the department of wireless and mobility department of ITESM has published several papers and thesis projects on the subject, now he can browse inspect those documents and find out about available areas of research and future work on the department's ongoing research. After contacting the director, they agree on a topic for the research and the student creates a branch of the digital libraries collection from the department's server and starts working on his project. When his research ends, he will be able to promote his work and contribute to the collection of the department. In this case the student was able to found a research topic, found a great amount of high quality organized information about the subject and the department collection gained a new researcher and more valuable assets for their digital libraries collection.

- A researcher is working on a project and learns that a colleague is an expert on a related topic he would like to include in his work, the researcher can request his colleague to branch his collection a give him an opinion about the project, add comments or write some technical details. They can both work at the same time and merge efforts as they progress. At the same time the tutor of the researcher could also branch the project and receive updates and give feed back to the user. This scenario shows a collection that is constantly receiving input from different sources and merging them into one joined effort.

There are several technological challenges involved with the development of this project, among the most relevant we can find:

- Replication system for branching collections and documents over the network and maintaining the hierarchy of the tree and branch revisions.
- Discovery and maintenance of a distributed servers directory.
- Performing distributed search queries efficiently.
- Resolution of conflicts when merging from different branches.

- The ability to manage and maintain different versions of the collection documents in order to provide users with a solid recovery method in case problems arise.

1.5 Thesis organization

This chapter introduces the problem, the importance and benefits from its solution, the general and specific objectives of the research project and the hypothesis on which the following work is based. The remaining of this work is organized as follows.

Chapter 2 describes and explains the concepts and technological foundation on which this research is based. We begin with the digital libraries and the related works in that area, then we discuss storage technologies, focusing on database management systems and their different types, application architecture styles and patterns and finally some concepts from version control systems that seemed relevant to the project.

On chapter 3 we elaborate on the proposed system, it's architecture, deployment components, the library services, interoperability layer and collaboration mechanism. Chapter 4 showcases a prototype of the system that implements the model defined on chapter 3, and finally on chapter 5 we give the conclusions and future work.

Chapter 2

Background

2.1 Introduction

The following chapter will introduce the concepts and technologies involved with the creation of a personal digital library system that proposes a new collaboration and versioning model for collection administration and development.

2.2 Digital Libraries

While physical libraries have existed in communities since ancient times, digital libraries emerged just about 15 years ago, but during this time, some of them have become very important and influential institutions of this new age. The information revolution no only provides the machinery to support digital libraries but it also satisfies the increasing demand for storing, organizing and accessing information.

In order to further explain the relevance of digital libraries, we need to define a digital library [3] as a managed collection of information with associated services, where the information is stored in digital formats and is accessible over a network. The most important part of that definition is the "managed" word, digital libraries differentiate themselves from the Internet or document repositories in the fact that the information is organized systematically into collections of information, there's an important process of selection, categorization and aggregation of data to the materials contained in it, which in turn gives us a sort of guarantee about the quality of the information that can be found on digital libraries.

One important aspect of digital libraries are its users, these can be divided in two groups, users that search and browse through the content of the collections, and librarians that manage and maintain the collections. The job of the librarians is to improve the content of the collections in order for users to take advantage of the services the library provides.

The services the library provides are one of the distinct characteristics each library systems has, but in general the library must help its users in the task of creating and managing the collections of the library, some of the basic services that every digital library system should provide are the following:

- Collection and metadata management: Users should have a mechanism to organize their documents into catalogs and define their own sets of metadata to describe the items in the library.
- Access management and security: Digital libraries should provide the means to manage and define constraints in order to protect the digital library information from unauthorized access.
- User interface: Since the most important thing about a digital library is the content, the library must provide a way for users to browse, access and view the material contained in the library.
- Search and Retrieval: Digital libraries have to take advantage of the access to the information text and metadata, therefore it must provide indexing and searching services over its documents and collections.
- Sharing and Collaboration: Libraries and archives contain much information that is unique, placing digital information on a network makes it available to everybody giving an enormous value to the ability to share and collaborate on the creation and administration of documents and collections.
- Universal access: Users should be able to access their information from any device and location.
- Interoperability: It refers to the ability of a library to interact with external services using established protocols.

2.2.1 Metadata

One of the key features of digital libraries is metadata, in the core of digital library services lives the metadata, it's the foundation to organize and categorize everything, it allows better searching, gives information about quality and allows users to define or use standards in order to describe information. According to [16], metadata can be defined as structured descriptors of information resources, designed to promote information retrieval. Metadata can be grouped into specific frameworks called schemes, a scheme provides a formal structure designed to identify the knowledge structure of a given discipline and to link that structure to the information of the discipline through the creation of an information system that will assist the identification, discovery, and use of information within that discipline.

Some of the most recognized metadata schemes or sets are [8]:

- The Dublin Core metadata element set (Dublin Core), is a simple set of fifteen descriptive data elements intended to be generally applicable to all types of resources.
- The Visual Resources Association Core Categories (VRA Core), was developed primarily to describe items held in visual resources collections, which typically hold surrogates of original works of art and architecture.

- The Encoded Archival Description (EAD), was developed as a way of representing archival finding aids in electronic form. Finding aids are a form of archival description that generally begin with narrative information about the collection as a whole and provide progressively more detailed descriptions of the components of the collection.

- AACR2/MARC cataloging isn't exactly a metadata scheme in the sense that the preceding metadata element sets are schemes. However, together, the suite of rule sets and format specifications used in traditional library cataloging do functionally constitute a metadata scheme, These include the International Standard Bibliographic Description (ISBD), the American Cataloging Rules, the MARC21 specifications, and a number of related documents.

2.2.2 Personal Digital Libraries

A personal digital library [2] adapts the traditional digital library services to an individual user, providing a personalized experience based on a user-defined schema for creation and administration of their own libraries, sharing of the collections with other users and allowing universal access to his data. Because personal libraries are about the individuals, the content of the library tends to go with the user interests and his daily activities.

A very important aspect of a personal digital library is the customization, this type of libraries offer personalized versions of regular digital library services, usually through the creation of a user profile and tracking regular user operations and topics of interest in order take advantage of the knowledge about the user.

2.2.3 Copyright

Digital libraries [12] can easily be made far more accessible than physical ones, which causes a couple of problems: access to the information in digital libraries is generally less controlled than it is in physical collections. Putting information into a digital library has the potential to make it immediately available to a virtually unlimited audience.

Possessing a copy of a document certainly does not constitute ownership in terms of copyright law. Though there may be many copies, each document has only one copyright owner. This applies not just to physical copies of books, but to computer files too, whether they have been digitized from a physical work or created electronically in the first place—"born digital." When you buy a copy of a document, you can resell it, but you certainly do not buy the right to redistribute it. That right rests with the copyright owner.

For this reason is very important to consider this point when designing a digital library, specially one focused in collaboration, even if the system is not responsible for the use the users give it, it's not ethical nor legal to distribute copyrighted material.

2.2.4 Related work

PDLib PDLib [2] proposes a personal digital library system, universally available. It's personal because each user is provided with a general purpose document repository and it's universally accessible because each user can access its library virtually from any device with a network connection. PDLib has an interoperability mechanism based on P2P that allows searching and copying collections between collections but it's bound by a couple of important restrictions, in the case of the search, the address of the target server must be provided by the user, and for the copying of collections, they can only be made for different devices of the same user, which means that users can't copy libraries from other users, also if the documents already exist in the collection they will duplicated with each copy. PDLib can also connect with other libraries through OAI and it offers an API using XML-RPC that allows the creation clients.

The proposed work is similar to PDLib in purpose and architectural style in the sense that they use both, a client server style for user interactions and a peer to peer style for interoperability between servers, but there's a difference between the underlying technologies on these projects, the present work relies on Restful Web services for both ends because it's a more accessible technology and currently has become the mainstream for communication between distributed clients because it provides full language and capability independence.

UpLib UpLib [14] is a personal digital library system that consists of a full-text indexed repository accessed through an active agent via a Web interface. It is suitable for personal collections comprising tens of thousands of documents and provides for ease of document entry and access as well as high levels of security and privacy. UpLib emphasizes image-domain representation of documents and different representations of the information, even though it doesn't offer any interoperability options it has some services available through HTTP for the development of external clients.

Phronesis Phronesis [1] is an open source software intended for the creation of general purpose digital libraries over the Internet. It allows the creation, administration and maintenance of distributed digital libraries. Phronesis supports interoperability through the OAI and Z39.50 protocols.

Greenstone Greenstone is a suite of software for building and distributing digital library collections. It provides a new way of organizing information and publishing it on the Internet or on CD-ROM. The aim of the software is to empower users, particularly in universities, libraries and other public service institutions, to build their own digital libraries.

In terms of purpose, the main characteristic that differentiates this project from the others is that it focus in the collaboration and work flow management problems between different users working on an asynchronous and distributed fashion. PDLib covers less ground in this area but it has excellent universal access support,

UpLib is mostly concerned with the representation of data and Phronesis and Greenstone are more oriented to the publishing and maintenance aspect of the digital libraries.

2.3 RESTful HTTP API

An application programming interface (API) is the basic interface offered to developers so they can build applications that interact with a library or server. Representational State Transfer (REST) refers to the fact that resource access is available via a series of simple web services that are implemented in Hypertext Transfer Protocol (HTTP) and adhere to the principles of REST.

The following concepts are fundamental to explain the meaning of REST.

- Resource: A resource is anything that's important enough to be referenced as a thing in itself, usually is something that can be stored on a computer and represented as a stream of bits: a document, a row in a database, or the result of running an algorithm.
- URIs (Resource names): The URI is the name and address of a resource. URIs should have a structure, they should vary in predictable ways.
- Representation: A representation is just some data about the current state of a resource. Most resources are themselves items of data (like a list of products), so an obvious representation of a resource is the data itself, but in order to be considered a representation it has to have an specific file format and language, for example the list of products con be represented as an XML document, a Web page, or as a comma-separated text.
- Links give you access to other resources from within a resource.

In order for an API to be considered RESTful it needs to conform to four simple properties.

1. Addressability, an application is addressable if it exposes the interesting aspects of its data set as resources. Since resources are exposed through URIs, an addressable application exposes a URI for every piece of information it might conceivably server.
2. Statelessness means that every HTTP request happens in complete isolation. When the client makes an HTTP request, in includes all information necessary for the server to fulfill that request. The server never relies on information from previous requests.
3. Connectedness is the quality of having links, a Web service is connected to the extent that you can put the service in different status just by following links and filling out forms.
4. A uniform interface, it's important that every service uses the HTTP's interface the same way. This interface is formed by several verbs, among the most important are: GET to read, PUT to add, DELETE to remove and POST to change.

2.4 Database Management Systems

Every digital library has to store a big amount of files and information about documents, most of them make use of a database management systems for this purpose, the characteristics and theoretical foundation about this topic are discussed next.

A database [9] is a collection of related data that is logically coherent with some inherent meaning, is designed, built, and populated with data for an specific purpose and it represents some aspect of the real world. A database can be of any size and of varying complexity.

A database management system (DBMS) is a collection of programs that enables users to create and maintain a database. The DBMS is hence a general-purpose software system that facilitates the process of defining, constructing, and manipulating databases for various applications.

2.4.1 Classification of Database Management Systems

DBMSs can be classified by multiple characteristics, one of the most important is the data model [9][17][15]:

- Relational: Represents a database as a collection of tables, where each table can be stored as a separate file. Most relational databases use the high-level query language called SQL and support a limited form of user views.
- Object based: Defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a class, and classes are organized into hierarchies.
- Sorted ordered column-oriented: Data is stored in a column-oriented way, where each unit of data can be thought of as a set of key/value pairs, where the unit itself is identified with the help of a primary identifier, this identifier is used to sort and order data in the database.
- Key/value model: An associative array is the simplest data structure that can hold a set of key/value pairs. Such data structures provide a very efficient running time for accessing data.
- Document oriented: The data stored in the database comprises a series of documents, each of which can contain a series of fields and values. Each document is independent of one another, and there is no strict schema that they must adhere to.

The next criteria that is relevant for this research is the number of sites over which the database is distributed:

- Centralized: The data is stored at a single computer site, a centralized DBMS can support multiple users, but the DBMS and the database themselves resides totally at a single computer site.

- Distributed: Can have the actual database and DBMS software distributed over many sites, connected by a computer network. Homogeneous DDBMSs use the same DBMS software at multiple sites, while heterogeneous don't.

2.4.2 Document Oriented Database

The data stored in the database comprises a series of documents, each of which can contain a series of fields and values. Each document is independent of one another, and there is no strict schema that they must adhere to. Traditional databases that adhered to the relational model stored data in a series of tables; they were made up of rows and columns of data. In a relational database, you must predefine the schema that all data in each table will adhere to, and all the data contained in the table must strictly conform to that schema [15].

Data in a Document Oriented Database database is stored in a series of uniquely named documents, objects made up of various named fields. The values stored in the document can be strings, numbers, dates, booleans, lists, maps, or other data types. Each document in the database has a unique ID, and the documents are stored in the database on a flat address space. There is no limit to the number of fields a document may have or on the size of values stored in the document. In addition to data fields, each document includes metadata that is maintained by the server itself, such as a revision number and more.

2.4.3 CouchDB

CouchDB [15] is a document-oriented database management system, released under the open source Apache License. In contrast to most database systems, it stores data in a schema-free manner. This means that, unlike traditional SQL-based databases, there are no tables and columns, primary and foreign keys, joins, and relationships. Instead, CouchDB stores data in a series of documents and offers a JavaScript-based view model for aggregating and reporting on the data.

According to the CouchDB wiki, Couch stands for "Cluster Of Unreliable Commodity Hardware," indicating that CouchDB is intended to run distributed across a cluster of cheap servers. Anyone who has dealt with replication in databases before will know that it is rarely a simple task, but the exact opposite applies when it comes to CouchDB. Add to this the fact that CouchDB is developed in Erlang OTP, a fault-tolerant programming language that offers excellent concurrency features, and you know that your CouchDB database will scale well without a loss of reliability and availability.

CouchDB's design [13] borrows heavily from Web architecture and the concepts of resources, methods, and representations. It augments this with powerful ways to query, map, combine, and filter your data. Add fault tolerance, extreme scalability, and incremental replication, and CouchDB defines a sweet spot for document databases.

2.4.4 Replication

Replication [9] is useful in improving the availability of data. It can improve the availability remarkable because the system can continue to operate as long as at least one site is up. It also improves performance of retrieval for global queries, because the result of such a query can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module.

Distributed database systems replication types:

- Full replication: Is the most extreme case is replication of the whole database at every site in the distributed system.
- No replication: Each fragment is stored at exactly one site.
- Partial replication: Some fragments of the database may be replicated whereas others may not.

2.5 Application Architecture

Software architecture [10] is about the design of a system and the impact it has on the system's qualities, qualities like performance, security, and modifiability, more specifically, software architecture is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. We could categorize these structures as architectural styles and patterns.

2.5.1 Application Architecture Styles

An architectural style is a kind of pattern that occurs at an architectural level and applies to architectural elements like components and modules. The most relevant styles for this research are:

Client-server style & N-tier In the client-server architectural style, clients request services from servers. The request is usually synchronous and across a request-reply connector, but can vary. There is an asymmetry between client and server in that the client can request that the server do work, but not the reverse.

Peer-to-peer style In the peer-to-peer architectural style, nodes communicate with each other as peers and hierarchical relationships are prohibited. Each node has the ability, but not the obligation, to act both as a client and as a server. The result is a network of nodes operating as peers, where any node can request or provide services to any other node.

JGroups In terms of peer to peer systems, there's a need for a supporting communication layer, this need can be fulfilled by JGroups. JGroups [4] is a toolkit for reliable group communication. Processes can join a group, send messages to all members or single members and receive messages from members in the group.

The system keeps track of the members in every group, and notifies group members when a new member joins, or an existing member leaves or crashes. A group is identified by its name. Groups do not have to be created explicitly; when a process joins a non-existing group, that group will be created automatically. Member processes of a group can be located on the same host, within the same LAN, or across a WAN. A member can be a part of multiple groups.

2.5.2 Application Architecture Patterns

Layering [11] is one of the most common techniques that software engineers use to break apart a complicated software system. When thinking of a system in terms of layers, you imagine the principal subsystems in the software arranged in some form of layer cake, where each layer rests upon a lower layer. In this scheme the higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer. Furthermore each layer usually hides its lower layers from the layers above.

Breaking down a system into layers has a number of important benefits:

- You can understand a single layer as a coherent whole without knowing much about the other layers.
- You can substitute layers with alternative implementations of the same basic services.
- You minimize dependencies between the layers.
- Layers make good places for standardization.
- Once you have a layer built you can use it for many higher level services.

There are three primary layers that can be divided into: Presentation, domain, and data source.

- Presentation logic is about how to handle the interaction between the user and the software. The primary responsibilities of the presentation is to display information to the user and to interpret commands from the user into actions upon the domain and data source.
- Data source logic is about communicating with other systems that carry out tasks on behalf of the application. For most systems the biggest piece of data source logic is a database which is primarily responsible for storing persistent data.
- Domain logic, also referred to a business logic, is the work that this application needs to do for the domain you're working with. It involves calculations based on inputs and stored data, validation of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch depending on commands received from the presentation.

2.6 Version Control

The core mission of a version control system [6] is to enable collaborative editing and sharing of data while preventing the users from stepping on each other's feet. The following are the different strategies used to achieve this:

- The Lock-Modify-Unlock solution: In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks. A user must lock a file before he can begin making changes to it, once the user has locked a file, no one else can also lock it, and therefore cannot make any changes to that file. The problem with the lock-modify-unlock model is that it's a bit restrictive and often becomes a roadblock for users.
- The Copy-Modify-Merge solution: Each user's client contacts the project repository and creates a personal working copy, a local reflection of the repository's files and directories. Users then work simultaneously and independently, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately, a human being is responsible for making it happen correctly.

2.6.1 google-diff-match-patch

Merging changes is a very hard and important part of the Copy-Modify-Merge solution, few libraries are available for this task, one of the most prominent is the Diff Match and Patch libraries, which offer robust algorithms to perform the operations required for synchronizing plain text:

- Diff: Compare two blocks of plain text and efficiently return a list of differences.
- Match: Given a search string, find its best fuzzy match in a block of plain text. Weighted both accuracy and location.
- Patch: Apply a list of patches onto plain text. Uses best-effort to apply patch even when the underlying text doesn't match.

2.7 Real Life Applications of Document and Metadata Collaboration

2.7.1 OLCL WorldCat database

The OLCL WorldCat database [5] is the largest bibliographic database in the world. In 2009, OCLC eliminated many of the restrictions it had on editing the bibliographic records in its database, greatly expanding the number of institutions authorized to edit records.

The changed was initially called the Expert Community Experiment. When the experiment was judged successful, the name was changed to Expert Community, and the program acquired a permanent status and the current trend is that catalogers are adopting a wikilike process for editing records, which is cumulatively making small changes to records that add up to fulllevel records over time.

2.7.2 Ohio Memory Online Scrapbook Project

One approach taken by the Ohio Memory Online Scrapbook project [7], was to invite the state's institutions to contribute images and associated catalog records to a central point for review and input. Although it was said that "more than 320 cultural heritage repositories in Ohio contributed records to a common electronic catalog without even knowing it," that is not strictly true as those contributing information realized that they were participating in a common catalog, even if they would not create the final record.

In this case the central authority was based at the Ohio Historical Society which was responsible for the consistency of description, particularly in the choice of subject terms applied to individual objects. The added value of this centralized leadership was that it addressed imbalances of experience, equipment, and the ability to maintain the new resource of electronic collection information. The goal was access to the great range of visual material and the delivery of a rich imagebased result set in response to a search. For this reason, all objects were illustrated and records for handwritten documents included a page image as well as transcriptions.

2.7.3 King County Snapshots Project

The goal of this project [7] was direct access to images and it was achieved using CONTENT and a collaborative approach to metadata development, headed by specialists at the University of Washington Libraries and the Museum of History and Industry (MOHAI), Seattle. An article on this project discusses the difficulties of achieving and maintaining metadata standards, given that several of the twelve project participants already had well established thesauri for their specific subjectbased collections. Cataloging standards for the project were developed by the project manager and the two metadata specialists who traveled to each of the participating organizations following cataloging training workshops designed as part of the project.

Existing and desired access points were discussed between the participants and the metdata specialists and mapped to Dublic Core elements.

2.7.4 The Experience Music Project(EMP)

The EMP's mission [7] was to explore creativity as expressed in American popular music. Public availability of detailed information on the museum's collections, including images and audio and video clips, was

a stated intent of the project from the beginning. This online equivalent of an exhibition catalog was also to include as much of a virtual experience of the objects and performances as the technology could provide.

The resultant product combined elements from more than one standard developed by libraries, archives, and other cultural heritage repositories, all incorporated into the EMP cataloging manual.

2.8 Conclusion

The theoretical background and technologies involved with the development of this project were introduced in this chapter, with an special focus on digital libraries, the services it provides and the organization and management of collections. The other topics discussed were database technologies and some important concepts like replication and distribution of data and enterprise application architectures and technologies.

CouchDB has been selected as the persistence mechanism for the development of this project because of the many characteristics that make it a very good match for a personal digital library. Being a document oriented database allows great flexibility in the information that's stored in the database, this kind of flexibility is ideal for metadata handling because fields don't have to be defined in advance and the space required to store each document is optimum. Another important feature is the ability to keep versions of the documents stored in the DB which plays a key role in the proposed conflict resolution services. CouchDB makes perfect sense for a digital library because it can store actual files as binary attachments for documents meaning there's no need for an external file system solution. In terms of reliability and scalability CouchDB can run as cluster of connected servers or as standalone distributed servers, it can even run on modern mobile devices, this plays a very important role in the massive distribution and portability of the application, on top of these distributed CouchDB also provides replication services that are of great importance for the project as they will be the foundation for the branching and merging mechanisms.

In terms of the architecture of the server, the style is a mix of client-server and peer to peer, and most important pattern in which the application will be built is the MVC layering pattern has been chosen for the internals of the server using the Spring MVC framework because this model keeps a clean separation of concerns and helps with the code reutilization required to provide the different services of the server that will be divided in the form of a Web client and a RESTful API for interoperability that can also be used by anyone to create thin clients for the application that could be running on a mobile phone or any other device with access to a network connection.

JGroups has been chosen in order to create and maintain a peer to peer network because of the simplicity of its concept, it's flexibility in terms of underlying protocol support and the building blocks to interconnect the different servers in the network.

For the collaboration mechanism some concepts will be borrowed from the Copy-Modify-Merge solution existing in some version control systems. The main library used for the merging of the documents will be google-diff-match-patch which were selected because of its clear API, availability on different programming languages and robust algorithms.

Chapter 3

Access and Branch Management on Distributed Personal Digital Libraries

3.1 Introduction

We have already explained the problem and the theoretical foundation and technological stack that could be involved in the development and implementation of the proposed collaboration model. In the following chapter we are going to take a look at the model from an implementation perspective using a prototype for this purpose.

We are going to start from a global perspective and then dive deeper at each important piece of the platform, starting with the system architecture and design of the application, followed by the REST layer approach. Then the data source implementation which is a very key aspect of the prototype, the search infrastructure which is the first step towards getting users to work together, and finally explaining the implementation of the core operations in the interoperability and collaboration model.

3.2 Global Components Architecture

The current work proposes a client-server model for the interaction with users, and a peer to peer model for interoperability between the servers. The communication between all the components of the application goes through HTTP which means every component can be replaced easily and allows the implementation of technology agnostic clients that can run virtually anywhere just by using the RESTful services available. Figure 3.1 shows the proposed deployment diagram which shows an instance of the Library Application Server that interacts with a user through the Web application, mobile and desktop clients, each client could host an instance of the database for faster access to the information and portability. The server also communicates with other servers and the data source using HTTP RESTful services.

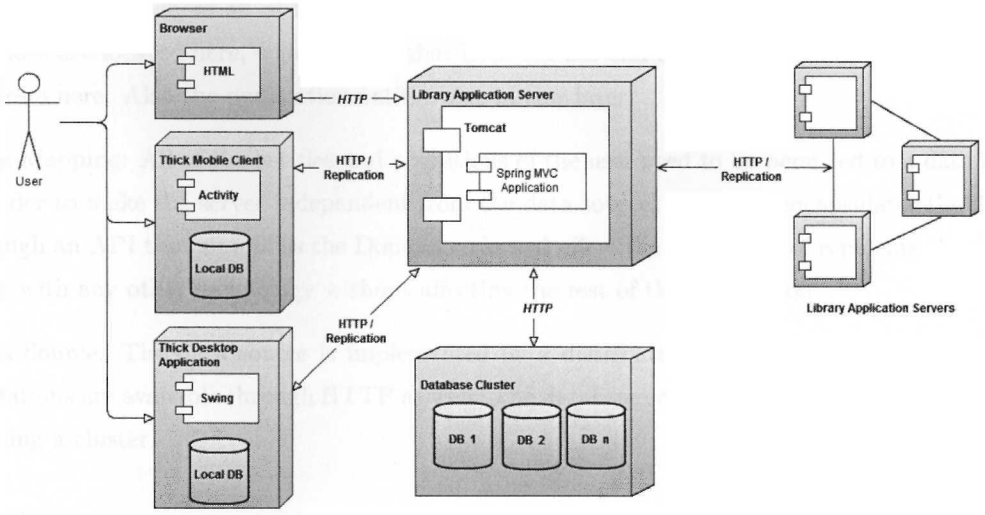

Figure 3.1: Global deployment diagram.

3.3 Server Architecture

This section is going to focus on the client-server part of the application, leaving the details of interoperability for a later section. The server application uses a layering scheme based on the foundation described in chapter one with the difference that a couple of mediating layers have been added to reduce the complexity and responsibilities of each layer, figure 3.2 represents the layers graphically. The following are the application layers and the purpose of each of them:

- Presentation: Contains the screens and views in which the users can have access to the library services. The views are basically Web pages, by embracing the Web as the default platform for the application the application is available to a great amount of users from many different devices without relying on a specific operating system or technology. This layer runs in the client, which means that the application is ready to be extended by anyone with any presentation technology like for example a native desktop application or a mobile application.

- Controller: It's in charge of communicating the presentation and domain layers, it's main functions are receiving the requests from the clients, routing them to the domain layer components and sending the response back to the clients. Within this layer we can find two types of controllers, regular controllers which serve the Web application and reply with the HTML pages users can see in the application and RESTful controllers that can respond with JSON or XML, these controllers are the ones that make possible the creation of new types of clients.

- Domain: Here's where all of the server logic is implemented, all of the services that the application provides are located here, which means that both regular and RESTful controllers end up routing the requests here. Also the application entities live in this layer.
- Data-Mapping: All of the entities and operations of the user need to be persisted to a data repository, in order to make the server independent from the data source, this layer encapsulates the data source through an API that simplifies the Domain code and offers the possibility of replacing the data source layer with any other technology without affecting the rest of the application.
- Data Source: The data source is implemented by a distributed, document oriented database whose operations are available through HTTP as well. The database can scale by adding more server instances forming a cluster.

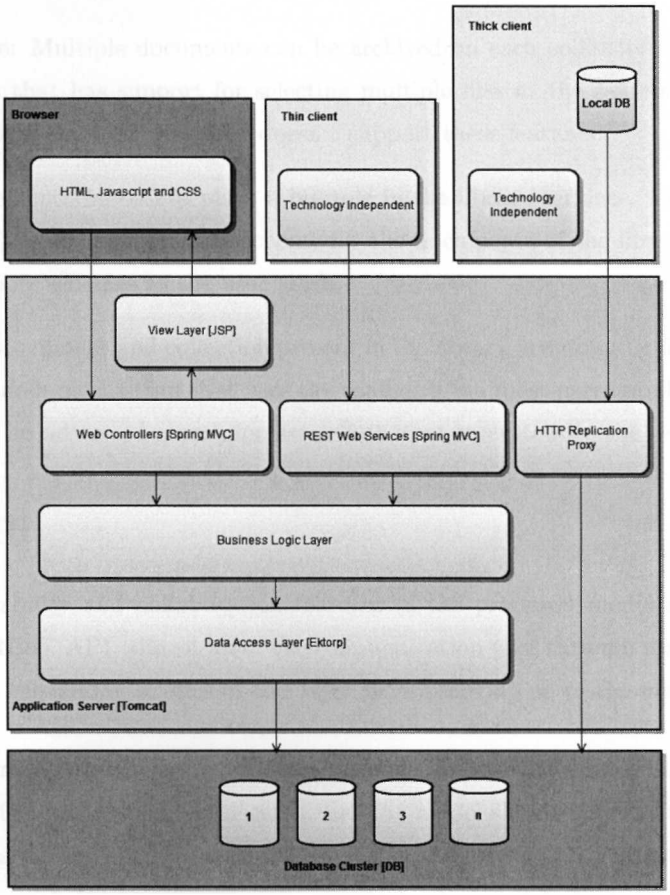

Figure 3.2: Application server layer architecture.

3.4 Aplication Services

In order to meet with the personal digital library criteria of the project, several services are proposed in the current work:

- User Administration: Users are managed using a role model, every user can have many roles. There are two types of roles available: The administrator role, allows users to manage the server and interoperability settings, allowing them to manually add new servers that are on remote locations. The user role, gives users access to the application services.
- Collection Management: User are able to create a manage multiple collections as part of its library. Each collection can have multiple children collections and documents conforming a tree like structure. Also each collection can be shared by setting it's public value on, this gives access to other users of the library to the public library.
- Document Storage: Multiple documents can be archived on each collection, there's a convenient uploader component that has support for selecting multiple files at the same time, drag and drop and gracefully degrades if the user's browser doesn't support these features.
- Metadata Management: Metadata plays a big role in the library services. By default the library has adopted Dublin Core as the metadata scheme for the documents of the library but it has support to add and use as many schemes as the user needs.
- Searching: Every document and collection present in the library is automatically indexed by the server. There's a quick search mechanism that uses the configuration most users would want to use but at the same time there's an advanced search for users that want to customize the search parameters.

3.5 REST API

The server interoperability and collaboration features of the proposed model are supported in a very important way by the REST API, almost all of the communication goes through using this layer, that's why we are going to explain the characteristics of this layer before moving on to the main model operations.

In order to have a greater flexibility in terms of interoperability with future clients, which in turn will allow the expansion of the application, we decided that the model should support at least JSON and XML as the transport objects for the operations of the model, in the future, other representations can be added as needed and as long as it helps to connect with more technologies and clients, table 3.1 presents a partial list of the operations available in the REST API.

Method	URL	Description
GET	/api/collections/{collectionId}	Get Collection: Returns a collection by its ID.
POST	/api/collections/{collectionId}	Save Collection: Saves or updates the collection received in the body.
DELETE	/api/collections/{collectionId}	Delete Collection: Deletes a collection by its ID.
GET	/api/collections/children/{collectionId}	Get Collection Children: Returns the children collections and documents of the specified collection.
GET	/api/collections/tree/{collectionId}	Get Collection Tree: Returns the full tree of the specified collection.
GET	/api/collections/branch/{collectionId}/to/{serverId}	Branch Collection: Copies the full collection contents to the specified server ID.
POST	/api/collections/pullrequest/{collectionId}	Save Pull Requests: Saves a new pull request from one of the client servers.
GET	/api/collections/commit/{collectionId}	Commit Collection: Commits pending changes to the source server of this collection.
GET	/api/collections/changes/{collectionId}/since/{sequenceId}	Get Changes: Returns a list of changes since the specified server sequence ID.
GET	/api/documents/{documentId}	Get Document: Returns a document by its ID.
POST	/api/documents/{documentId}	Save Document: Saves or updates the document received in the body.
DELETE	/api/documents/{documentId}	Delete Document: Deletes a document by its ID.
GET	/api/documents/download/{documentId}	Download Document: Returns a document attachment by its ID.
GET	/api/server/lastsequenceid	Get Last Sequence ID: Returns the last sequence ID of the server.
GET	/api/search/criteria	Search: Finds any public collections or documents that match the given criteria.

Table 3.1: List of available RESTful services.

3.6 Server Interoperability

The main focus of this project is to enable collaboration between users on different servers, but in order to accomplish that, there needs to be an underlying communication mechanism between servers, in this section we explain the technological details of the interoperability model.

The model consists on a peer to peer network that operates on the HTTP protocol with some help from the other protocols for auto discovery and multicasting. The process can be described with the following operations:

Auto discovery One of the goals of the system was to create instances that could auto discover and organize by themselves, this allows the communication among servers without users or administrators depending on the knowledge of the underlying network details. In order to implement this feature every server is subscribed to a communication channel that's configured on every instance. Servers on the channel can send messages using building blocks, the server that issues the request then receives and groups the result from each of the servers connected to the channel.

Upon server startup, one of the initialization tasks is to notify all the servers that a new instance has started, sending them the network information required to communicate and synchronize the servers directory. Each subscribed server checks whether they already have the information from that server and if it needs to be updated, in which case, there will be a synchronization for users and server directory information.

There might be some scenarios where it's not possible to auto discover a server, on such cases there's functionality to manually add a new server from the server administration page, which automatically activates the interoperability layer for that server.

Search One of the biggest reasons behind interoperability on digital libraries is to make available more content to users by aggregating results from different sources, which is where the search operation plays a very big role. Search is implemented as a building block on top of the communication channel, this allows efficient multicasting to all of the servers in the network, each server runs a local search and sends the results back to the requester which is in charge of merging the results and showing them to the user, a sequence diagram of this process can be seen in figure 3.4.

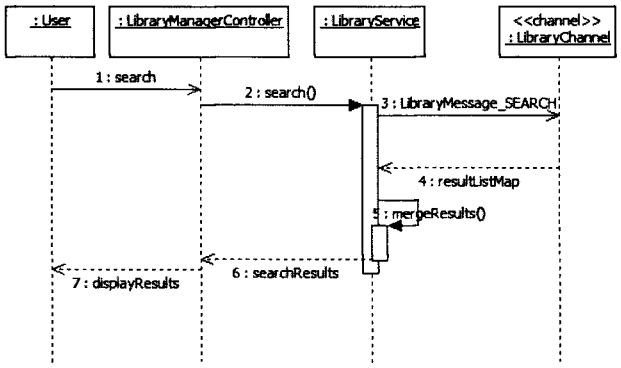

Figure 3.3: Sequence diagram of client server sending a search message to the channel and merging the results back to the client.

Every search result from another server is an opportunity for the user to discover more information about his topic of interest. Because documents are organized in collections, it could be interesting for a user to see the collection of documents paired with one of the search results.

Browse In order to do so, servers can also fetch and give users the ability to browse the source collection of a remote document. In order to do this, the server uses the RESTful API that's available for anyone to extend the application. Servers communicate using JSON, which is very lightweight and flexible.

3.7 Server Collaboration

The feature that really differentiates this project from the rest is the ability to collaborate and work on the same set of documents in different servers asynchronously and be able to integrate that work later, this is supported by the same RESTful API that allows users to create clients and extend the application.

Keeping track of changes Because of the nature of the model is important to know about the state of each server in the network, in order to tackle this problem, we propose keeping track of changes in the repository through the use of a sequence number that gets updated with each operation that's made in the server, using this sequence number we can establish the state of a copy when it was branched, and wether or not is necessary to update or send pull requests to other servers.

Branching Once a user has found a collection of his interest, he can make a copy on his local server for future reference or to collaborate with the collection, this is called branching. By branching a collection,

users get the ability to make changes to the collection on his local environment without disrupting the owner of the collection.

The branching process is highly related to the distributed search, it begins when a user issues a global search to the network and receives results from different servers. At this point, the user is able to move through the collection tree of any of the search results. In order to prevent having orphan collections, collections can only be branched from the root, so once the user makes its way to the root of the desired collection, he can issue a Branch command. In order to explain the process we will refer to the server that wants to branch the collection as the client server and the server that host the collection as the source server. The client server issues a branch command to the source server's API asking for a replication of the desired collection, the source server then creates a BranchInformation object that is tied to the collection and keeps the client status, that way we can find branches of any collection and know what has changed since the collection was branched. The source server then replicates his collection to the client and sends back the result of the operation to the client, figure 3.6 shows the sequence diagram from the source server's perspective. Once the client receives a successful result, it creates a BrachInformation object but in this case it's used to keep a reference to the source server, which is used for further operations like update or request a pull, figure 3.5 displays the sequence diagram that the client server executes in this operation.

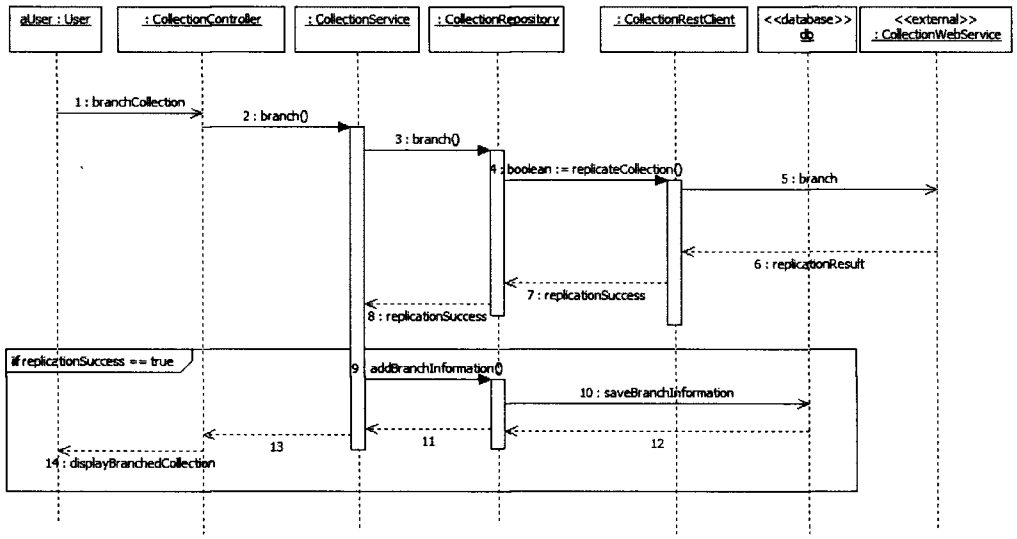

Figure 3.4: Sequence diagram of the client server branch operation.

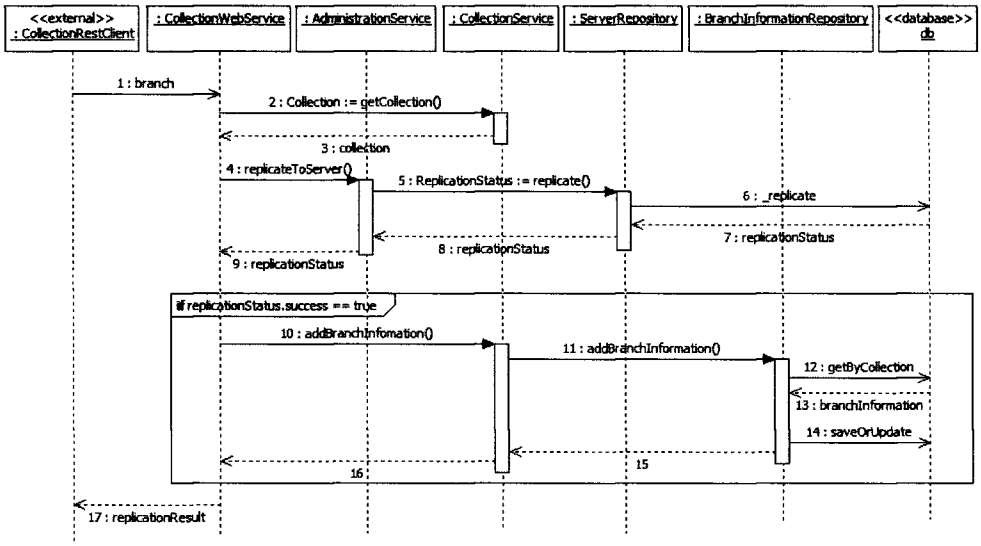

Figure 3.5: Sequence diagram of collection's source server branch operation.

Updating A branched collection doesn't affect the ability of the owner to keep working on it, in this situation is important that all of the users that branched the collection get the opportunity to get the latest changes from the master collection, this is the update operation.

The update command is available for any collection that has been branched, and it allows a user to fetch the latest changes from a collection and merge them with this work. The process, which can be seen in figure 3.7, starts with the client server getting the BranchInformation that contains the reference to the source server, once it has the server reference it can invoke the update command on the source server, this will issue a replication command on the desired collection and notify the results to the client. Upon a successful update, the client server will check if there are conflicts, in which case the conflict solving mechanism will be triggered, this will be explained on the next section.

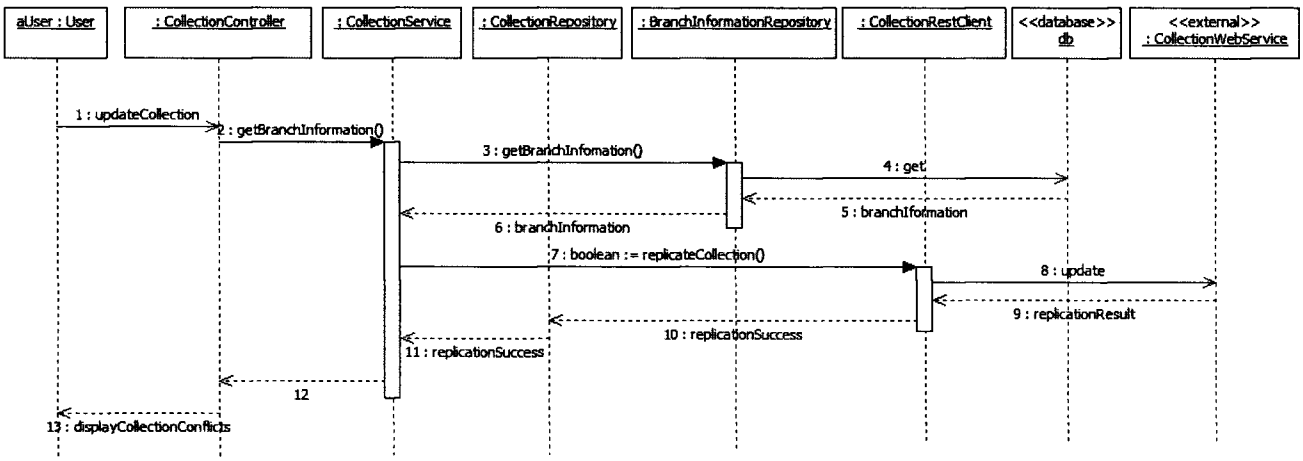

Figure 3.6: Sequence diagram of the client server update branch operation.

Merging Getting a copy of a remote collection and be able to work on it is good, but in order to really consider this project as a collaboration model, users need to have the ability to integrate their changes back into the master branch but at the same time, owners of the collection must have control over their files and be able to review any changes that are going to be made to his files. In order to support this, the current model proposes a pull request process. This operation consists of a couple of steps, which can be seen in figure 3.8, first, the user sends a pull request to the original server where a notification is made for the owner to review the request, if the client server has already sent a pull request, the new description will be appended to the previous request, so there can only be one request per client's branch at a time.

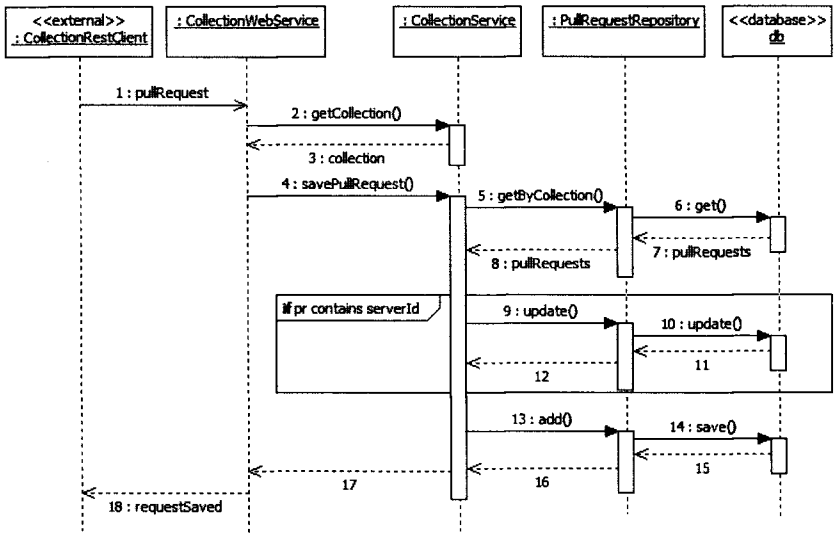

Figure 3.7: Sequence diagram of the collection's source server pull request operation.

The owner of the branch can review the proposed changes and either approve or reject them, he can individually go through each change and decide whether or not they want the change. This step is accomplished by using the sequenceId of the client server that's stored in the BranchInformation object of the collection, using the sequence Id the source server requests the changes from the client and presents the list of changes to the user. In order to display a side by side comparison of a specific document, the source server compares the change object received from the client to the current version in the server and displays a list of attributes with changes. Figure 3.9 displays the sequence diagram of how information is obtained and assembled in order to display a difference view.

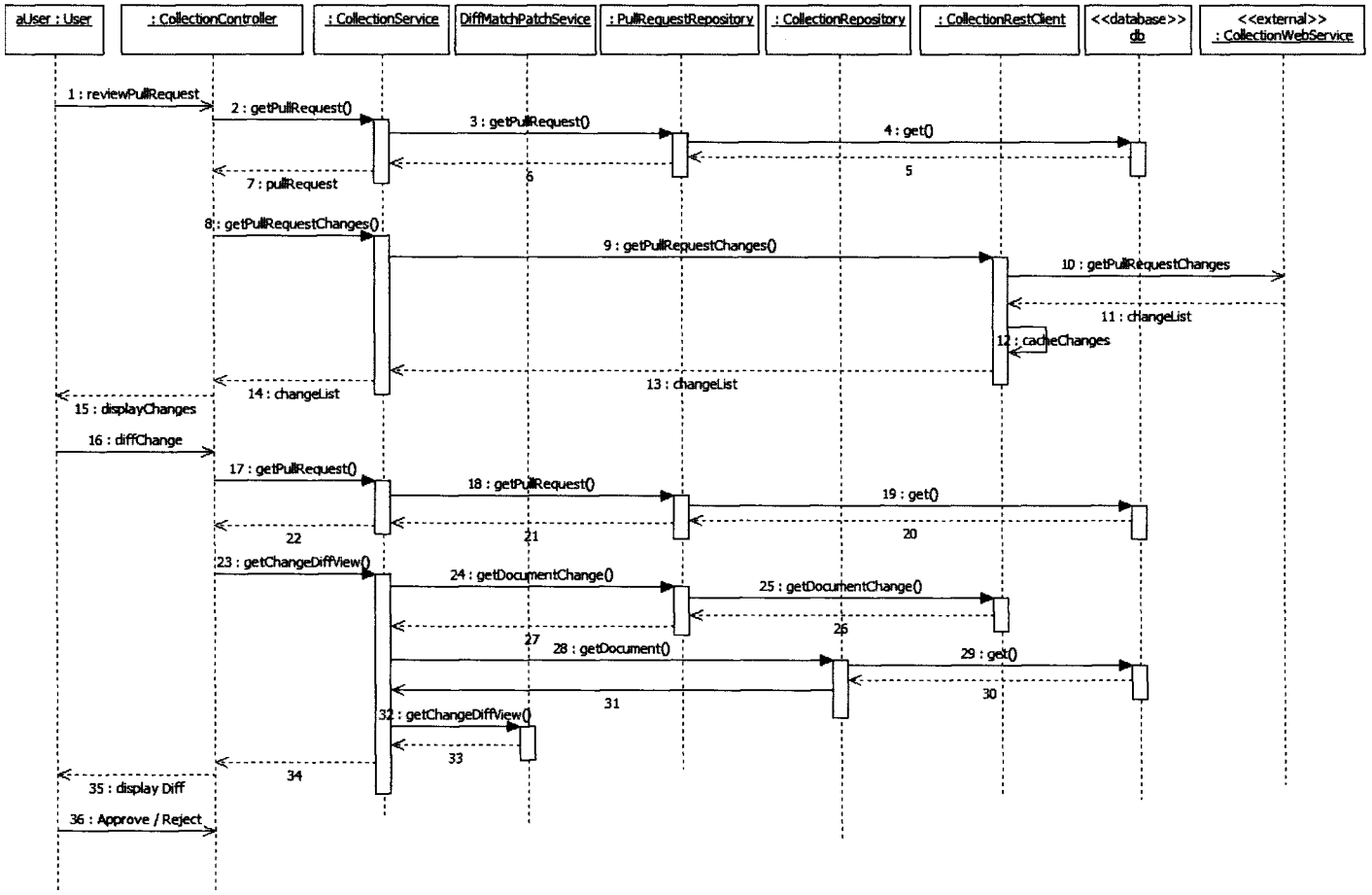

Figure 3.8: Sequence diagram of the collection's source server pull request review operation.

Once they have reviewed the changes, they get to chose whether they want the change set or not, if they decide to accept the change, the changes get copied to the branch and the conflict solving operation is triggered. Users are notified after the decision is made.

Solving conflicts It's possible that the user that branched the collection modified some of the same files that were changed on the master branch as well, this is what we call a conflict. Conflict resolution is a very wide and complex problem, that's why this research takes a simple approach to the problem by giving the user the ability to choose between the two change set properties.

Conflicts can arise in two situations, the first one is when users commit their work and the source server has modified one or more files that the commit is changing, the second is when the source server has

changed one or more files that the client server has changed as well and the client server issues an update command. Whenever one of this two operations is made, users are taken to the conflicts page, if there are no conflicts they are redirected back to the collection page, otherwise, they get a list of all of the documents and collections with conflicts. Users have to review each conflict individually, clicking on file will take them to the merge page.

Semantic resolution of conflicts It's possible to implement semantic resolution of files and metadata but it involves several changes to the rest of the operations in the model, and fell out of the scope of the current research, applying the following changes to the framework it would be easy to perform a semantic resolution of metadata and files:

Keeping track of the common ancestor The most important step for the implementation of this feature is maintaining a reference of the common ancestor in the client server, the reference can't be kept in the copies of the files because it would generate changes and conflicts when trying to commit a pull request. In order to do this we propose adding an extra step after branching the collection to go through all of the files in the collection and create a new ancestor reference document that describes the document, the current revision which becomes the ancestor revision after changes are made to the local copy, and a dirty flag that indicates whether the file has changed locally or not.

A new post processor callback needs to be added when updating documents locally, this callback updates the ancestor reference by marking the dirty flag as true. The reason we need this flag is because the client server might update his copy using the remote update operation, if files have not been changed locally, a new step needs to be performed post update in order to reset the ancestor reference revision to the new one obtained through the update.

Performing the difference and patching the target document The purpose of keeping the ancestor reference is to be able to apply only the changes that were made in the client to the possible new version of the target, this way the user doesn't have to choose between either but just merge the two versions. The procedure to diff and patch a document would be the following:

While viewing differences or solving a change, the ancestor revision reference can be fetched from the client and loaded from the local server. Three sets of data will be available then. The ancestor version of the metadata or file contents are compared against the new remote version, using the difference we can create a patch. The last step would be to apply the patch to the newest local version.

3.8 Copyright Management

Due to the highly collaborative environment suggested in the project, the problem of keeping track of authorship and avoiding plagiarism arises, one of the measures that were taken in this project was to keep track of the source of a collection. This means that each branched collection holds a reference to the original, and can be accessed by anyone.

This solution works well at the collection level point of view, but better mechanisms need to be designed to protect individual files and collections.

3.9 Conclusion

During this chapter we gave a thorough review at the proposed model and architecture including some UML diagrams to provide a better understanding of the operations of the system. The model uses and implements many of the concepts presented in the Background chapter and justifies some of the decisions made during the design of the system while providing the advantages of the implementation.

In the rest of the current work, we will describe and showcase an implementation of this model, some interesting use cases and tests and finally we will conclude and offer some hints about the future of this research.

Chapter 4

Results and Evaluation

4.1 Introduction

In the following chapter we will present an implementation of the model that was introduced in prior chapter, as we go through each of the application layers we will be explaining some of the most important implementation details, providing specific technology related information and contextualizing the theory in terms of the actual prototype.

After explaining the implementation details of the model, we will go through some use interesting use cases, showcasing the steps that a uses would need to do in order to execute them, for this we have created a controlled test environment which we will also describe in detail.

Finally we will present some test results and metrics from the application operations and give a small conclusion of the chapter.

4.2 Server Implementation

The prototype server was implemented as a Java Enterprise Edition using the Spring Framework, which means it can run on any Servlet Container Server like Tomcat, Glassfish or JBoss, these are open source and freely available to anyone and can run on any platform making it a very affordable option for organizations.

The different layers of the application were implemented using the following technologies:

- Presentation: Java Server Pages and JavaScript, views are composed using the Tiles library in order to reuse common functionality and prevent code duplication.
- Controller: The controller layer is composed by Spring 3 MVC controllers classes, each controller class is in charge of wiring the user's actions with the business service layer specific to an entity or

component.

- Domain: The domain is where the server logic and entities reside, the entities were implemented as Plain Java Objects annotated with information about persistence (Ektorp library) and serialization to other representation technologies like JSON (Jackson library) and XML (XStream library).
- Data-Mapping: Data is persisted in the database using an ORM (Object Relational Mapping) library for CouchDB called Ektorp. Ektorp implements the JPA (Java Persistence API) standard and encapsulates the underlying protocol from the application.
- Data Source: Data is stored in CouchDB using a document per instance record, CouchDB can run as a cluster or a standalone server so more instances can be added as needed.

4.3 REST Configuration

The REST interface is built on top of the Spring framework, by default it operates using JSON but it can respond to XML as well given the appropriate HTTP headers. In the application level, there's a series of controllers on a specific package that correspond to this layer, they use the same service components which guarantees consistency across the different interfaces of the system.

In order to serialize the domain object models that are the result of the invocation of the RESTful services, it's necessary to make use of two libraries to support JSON and XML. These libraries are Jackson and XStream, through the use of annotations in the domain classes, just like the ones in figure 4.1, it's possible to define which attributes are going to serialized to outside of the application.

```
@JsonSerialize(include=Inclusion.NON_NULL)
@JsonIgnoreProperties(ignoreUnknown=true)
@JsonTypeName(value="collection")
@JsonIgnore
```

Figure 4.1: Some examples of annotations used to configure domain classes JSON serialization.

4.4 Data Source Implementation

In the following sections we describe the technical details of the CouchDB implementation of the data source implementation:

4.4.1 Documents

CouchDB databases don't have schema or predefined structures, data is stored in documents that can have any number of attributes in spite of any other documents structure. In our prototype every entity's

```
[
{"_id":"Physics",   "path":["Physics"]},
{"_id":"Fluid Mechanics",  "path":["Physics","Fluid Mechanics"]},
{"_id":"Mechanics Demonstration", "path":["Physics","Fluid Mechanics",
"Mechanics Demonstration"]},
{"_id":"Course Notes", "path":["Physics","Fluid Mechanics", "Course Notes"]},
{"_id":"Electricity and Magnetism", "path":["Physics", "Electricity and Magnetism"]},
{"_id":"Electrostatics", "path":["Physics", "Electricity and Magnetism","Electrostatics"]},
{"_id":"Charge, field, and potential", "path":["Physics", "Electricity and Magnetism",
"Electrostatics", "Charge, field, and potential"]}
]
```

Figure 4.2: Simplified collection structure example using names instead of UUID.

instance is represented internally by a document in the database. Each document in CouchDB has an ID. This ID is unique per database, which allows servers to share data with other servers without conflicts, for the library system we use Java UUID (Universally unique identifier), these identifiers have such a low collision probability that everybody can make thousands of UUIDs a minute for millions of years without every creating a duplicate.

Documents in the system have another important concept which are the revisions, every time someone updates or deletes a document in the database, they are creating a new revision of the original document, which is kept along side with previous revisions of the document. The prototype uses this revisions to compare versions when conflicts arise.

In order to identify the type of a document, the current work has introduced a special field called 'type', which is used to determine which class gets used at runtime for an specific database document. There are some specific instances when we use a polymorphic relationship for which we had to include a 'className' field to specify the instance of the class when using an interface.

Attachments Documents in the database can have file attachments, these are used in the system to keep the actual document files that are part of the library collections. An attachment is identified by a name and a MIME type.

Library Logical Model One of the most important elements on the design of the system and the data was representing the hierarchical structure of the collections. Each collection holds an attribute with the full path to each node, this path is represented as an array of UUIDs, one for each parent in the hierarchy, this structure allows all of the required operations for the system, an example of this structure can be seen in figure 4.2, which are: getting the whole collection tree or subtree, the children of a node, deleting a tree or a node, and getting the parent of a node.

```
{
   "_id": "meta/dublincore",
   "name": "Dublin Core",
   "fields": [
       {
           "name": "identifier",
           "label": "Identifier",
           "description": "An unambiguous reference to the resource within a given context.",
           "order": 1,
           "type": "string",
           "inputType": "TEXT"
       },
       {
           "name": "title",
           "label": "Title",
           "description": "A name given to the resource.",
           "order": 2,
           "type": "string",
           "inputType": "TEXT"
       },
       {
           "name": "creator",
           "label": "Creator",
           "description": "An entity primarily responsible for making the resource.",
           "order": 3,
           "type": "string",
           "inputType": "TEXT"
       }, ...
```

Figure 4.3: Snippet of Dublin Core metadata set fields.

Another important piece of the design was metadata for documents. The prototype handles metadata in sets, each metadata set can have as many fields as it needs, and each field holds information about presentation, name, description and type, adopting the Dublin Core set as the default set, figure 4.3 shows a snippet of the definition for Dublin Core metadata set document.

4.4.2 Views

In order to obtain data CouchDB uses map and reduce functions in a style known as MapReduce [13]. These functions provide great flexibility because they can adapt to variations in document structure, and indexes for each document can be computed independently and in parallel. The combination of a map and a reduce function is called a view. View definitions are stored in a special type of documents called design

document.

Map functions are called once with each document as the argument. The function can choose to skip the document altogether or emit one or more view rows as key/value pairs. View results are stored as rows that are kept sorted by key, this makes retrieving data from a range of keys efficient even when there are thousands or millions of rows.

Views are used through the system to retrieve data in an efficient manner, they are defined in Repository classes using the @View annotation, when the server startups it checks if views have changed and updates the respective design document accordingly, updating the view's b-tree. In figure 4.4 we can see a sample map function of a view, the emit functions returns a key and a value, in this case the key is composed, which means that is not a single value.

```
function (doc) {
    if (doc.type == 'collection' || doc.type == 'document') {
        for (var i in doc.path) {
            emit([doc.path[i], doc.path], doc)
        }
    }
}
```

Figure 4.4: Subtree view example.

4.4.3 Replication

Replication is the core concept that's behind branching, updating and merging collections, a CouchDB replication is a one-off operation, an HTTP request is sent to CouchDB including the source and the target and changes will be copied from one instance to another.

Replication is a great mechanism and it works very well for many different scenarios, but in our case we wanted fine grained control over the documents we copied over, in order to achieve this kind of control we made use of replication filters and parameters. A replication filter is a function that controls whether a document should be copied over or not, and as a view function, it runs over every document in the DB. This function has access to the request object, which means it can receive parameters to make the check, a sample filter function can be found in figure 4.5.

```
function (doc, req) {
    if (doc.type && (doc.type == 'collection' || doc.type == 'document')) {
        for (var i in doc.path) {
            if (doc.path[i] == req.query.key) {
                return true;
            }
        }
    }
    return false;
}
```

Figure 4.5: Single collection replication filter example.

The branch operation is pretty straightforward but when it comes to updating or pulling content the system has to provide mechanisms for reviewing changes and conflict resolution. As explained on chapter three, conflicts can arise whenever two servers modified the same file before an update or pull request with each other, whenever this scenario occurs, the prototype presents the user with a conflicts screen that allows users to easily resolve them quickly before continuing to work or browse the library.

4.5 CouchDB Lucene

Searching is a very important operation in our model, it's the step that starts the collaboration process, in order to bring fast and reliable search operations to the users we adopted the Lucene library which is the market leader in indexing and searching in Java. The search engine is tied to the database using one the CouchDB Lucene, which runs as a daemon thread invoked by specific CouchDB hooks.

In order to index the DB documents, CouchDB Lucene requires one or more index functions, these are pretty similar to CouchDB views but instead returning a key, document value pair, they return a document index definition that contains the fields and other important metadata that's used to determine search results. Figure 4.6 shows an index function that indexes the whole document including its metadata fields and attachment files.

```
function (doc) {
    if (doc.type && doc.type == 'document') {
        var ret = new Document();
        ret.add(doc['public'], {
            'field': 'public'
        });
        ret.add(doc.ownerId, {
            'field': 'ownerId'
        });
        for (var metaField in doc.metadata) {
            var metaValue = doc.metadata[metaField];
            if (metaValue) {
                ret.add(metaValue, {
                    'field': metaField
                });
            }
        }
        for (var a in doc._attachments) {
            ret.attachment('default', a)
        }
        return ret;
    }
}
```

Figure 4.6: Sample full indexing function for documents.

4.6 Autodiscovery and network management with JGroups

In order to implement autodiscovery and automatic network organization we have implemented a JGroups network. On server startup a PostConstruct callback is created on the LibraryService bean. This callback is in charge of starting a connection the library channel, this channel has a known logical address that other servers can use to connect to the network. Right after creating the channel, a new MessageDispatcher block is created in order to communicate with other servers. Figure 4.7 shows the different states that a JGroup channel can have.

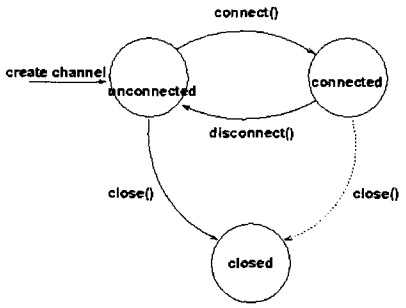

Figure 4.7: JGroups channel state diagram.

In order to implement the messages we have defined a LibraryMessage interface, this interface has methods to help the different instances communicate common messages like the initial synchronization on startup which is a CONNECT_SERVER message, or searching through the network which is the SEARCH message.

The CONNECT_SERVER message is sent on startup and it simply sends the server's LibraryAddress object that let's the other servers communicate with his REST API, every other server in the network will leverage the replication functionality explained on the previous section, using a LibraryServer replication filter to synchronize only their knowledge about the network and the rest of the peers, this way the newly introduced server gets the information it needs to communicate with each of the servers in the network.

4.6.1 Performance and Scalability Considerations

The proposed solution works well with small and medium size number of nodes, but in order to be able to scale it, some considerations and configuration options must be done. In the following text we refer to the combination of servers connected to a channel as a cluster.

Running many server instances on the same network: In this case, it's recommended that the library instances use the udplargecluster.xml configuration settings for JGroups, this settings file contains the following measures:

- Reducing the chattiness of the underlying protocol, when protocols are too chatty, too many messages are sent which can have a negative impact on the network.
- A discovery protocol is run at startup, to discover initial membership, and periodically by the merge protocol, to detect partitioned subclusters. When a multicast discovery request is sent to a large cluster, every node in the cluster might possibly reply with a discovery response sent back to the sender. In a cluster of 300 nodes, the discovery requester might be up to 299 responses. To reduce the large number of responses, a max_rank property can be set, the value defines which members are going to send a discovery response.
- Failure detection protocols determine when a member is unresponsive, and subsequently suspect it. Usually (FD, FD_ALL), messages(heartbeats) are used to determine the health of a member, but TCP connections (FD_SOCK) can also be used to connect to a member P, and suspect P when the connection is closed.

Create multiple small groups of servers on different networks using STOMP: STOMP is a JGroups protocol which implements the STOMP protocol, STOMP support means:

- Clients written in different languages can subscribe to destinations, send messages to destinations, and receive messages posted to (subscribed) destinations. This is similar to JMS topics.
- Clients don't need to join any cluster; this allows for light weight clients, and many of them can be run.
- Clients can access a cluster from a remote location (e.g. across a WAN).
- STOMP clients can send messages to cluster members, and vice versa.

The JGroups STOMP protocol can be used when we have clients, which are either not in the same network segment as the JGroups server nodes, or which don't want to become full-blown JGroups server nodes.

Bridging between remote clusters using RELAY: The RELAY protocol allows for bridging of remote clusters so that multicast messages sent in one group will be forwarded to the other and vice versa. The bridge between local clusters is essentially another cluster with the coordinators of the local clusters as members. The bridge typically uses TCP as transport, but any of the supported JGroups transports could be used (including UDP, if supported across a WAN, for instance).

4.7 Search

Searching efficiently through the network is possible using the JGroups infrastructure to send and receive responses from multicast requests. The search operation sends a LibraryMessage object using the MessageDispatcher block to everyone in the network including the sender of the message. Upon the reception of the message each server runs a local search on their own repository and reports back the results to the client server. On the client server, the message dispatcher can choose to implement several strategies for the reception of the results from other servers, for example it can wait until the majority of the servers have responded, all of the servers or just timeout after a period of time.

4.8 Collaboration

Branch Implementation After searching through the network, we are left with a list of results from many different servers, and these results have a reference to their source id, using the reference we can look up in the local library server repository to obtain the LibraryAddress from that server and allow the invocation of the branch operation on the source server. The source server then issues a replication using the single collection filter and then records the BranchingInformation as source. The client also creates a BranchInformation object to the source.

Update Implementation The update implementation just gets the BranchInformation object of a collection and uses the server reference to obtain the server's LibraryAddress and invoke the update command, that will in turn issue a replication command using the single collection filter.

Pull Request Implementation For the implementation of this operation it was decided that sending a pull request wouldn't copy changes to a temporary database on the target library, this would have opened the possibility for malicious users to misuse the system in many different ways and also depending on the number of requests and use of the system it could have had an impact on the application performance. Instead of this, we just store a small object with the description of the changes and how the library server should fetch it.

Once the owner of the library decides to review a pull request we need to fetch the changes from the client server, in order to do this we leverage our REST API and some functionality from CouchDB, more specifically the Changes API. When a pull request is reviewed, a library API call is made to get a list of DocumentChange objects, these objects are the result of invoking the changes API in CouchDB with the 'since' parameter set to the last sequence ID before branching or after committing a change to the source library, and using the SINGLE_COLLECTION filter function.

Reviewing Differences The list of changes from the client server brings a copy of the most recent version of the changed item, this version is compared on an attribute by attribute basis against the current version in the local server. In order to do this, we iterate through the properties defined in the metadata set of each version and only display the differences using the Diff Match Patch library by simply comparing the local version against the change.

Ignoring Single Changes During the review process, we give the user the possibility to ignore a remote change and keep his local version, in order to do this, we update the local document with a 'keep' flag that holds the current server ID, which prevents possible errors if another server were to send invalid data. The update of the local version effectively creates a conflict. Keep conflicts are automatically detected and resolved after the pull request has been approved.

Conflict Resolution In order to find conflicts on the library, a collection conflict view has been created, this view allows users to see which items in a collection are on a conflicted state and need to be resolved. After resolving the conflict on the merge screen, the application generates a new version of the document or collection that was merged, deletes the conflicted revision and updates the item with the merged version.

4.9 Use Case Scenarios

In order to demonstrate the system and it's possible uses, we set up a test environment consisting of a set of 4 personal computers running on the same network, that will be used to run some of the possible scenarios that might present on the application.

4.9.1 Test Environment Specification

The specification of the machines running on the network is as follows:

- MCASTELLANOS: Work Library
  Intel i5
  Windows 7 64 bits
  4GB RAM

- MARTINLAPTOP: Mobile Library
  Intel Centrino Duo
  Windows 7 34 bits
  2GB RAM

- MARTINNEWPC: Science Fiction Collector
  Intel i5 2500k
  Windows 7 64 bits
  8GB RAM

- MARTINPC: International Library
  Intel Core 2 Duo
  Windows 7 64 bits
  4GB RAM

4.9.2 Scenario 1: Contributing new material to a collection

A science fiction collector is compiling a collection of short stories, fan fiction, independent novels and public domain books about science fiction from very different sources all over the Internet. This person already has a fairly large collection but it has been having a hard time finding new sources to keep feeding his collection but he's using the library prototype and his collection has become quite known in his campus.

As his collection grows larger, several people has branched the collection for off-line browsing and personal reading, but there's a group of amateur writers that has started contributing their works to the library, these new contributions are always reviewed by the collection administrator in order to maintain the quality of the material in the library and in case the new material is not accepted they get valuable feed back from a

very good source. The prototype displays a list of results and adds a little arrow next to each external result as seen in figure 4.8.

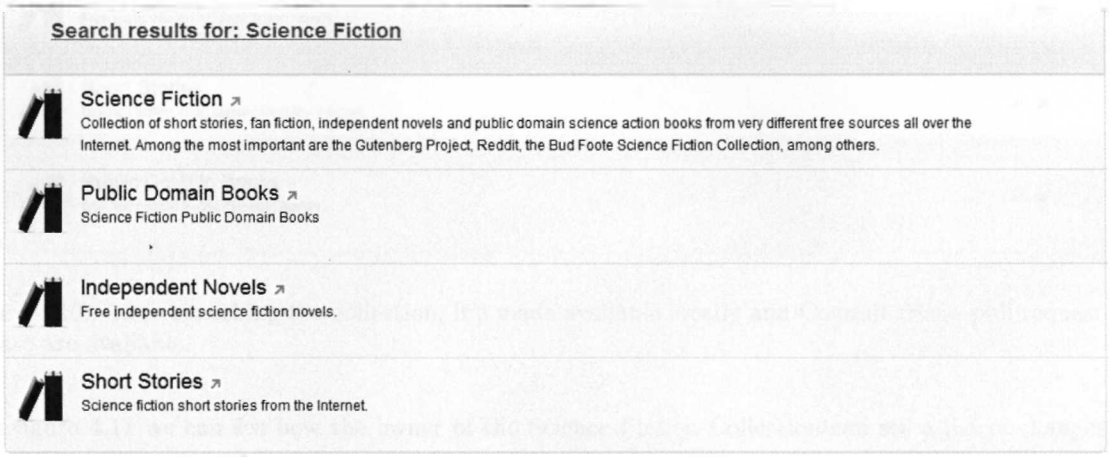

Figure 4.8: Search results from Science Fiction on the Mobile Server displays results from Science Fiction Collector server.

Once the user clicks on a result it can drill down and start navigating through the collection hierarchy even for external collections, this is possible thanks to the RESTful API, figure 4.9 shows an external collection being browsed and how the server allows that collection to be branched.

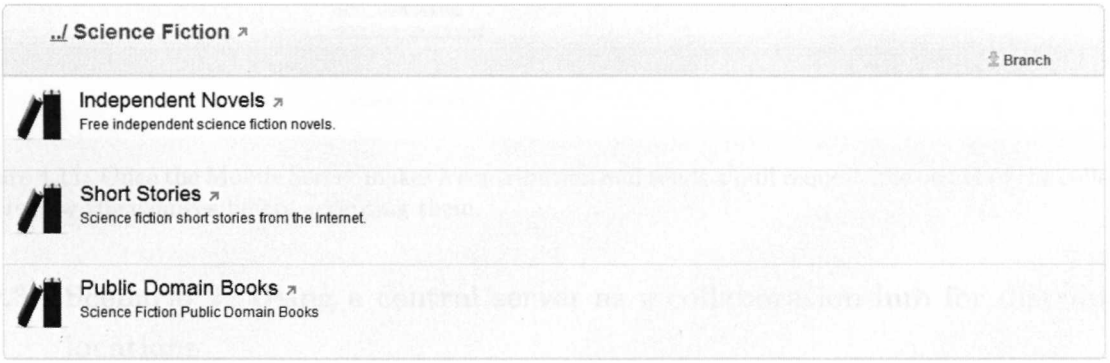

Figure 4.9: Browsing external collection from results, the Branch command is available.

Figure 4.10 shows how after the collection has been branched, the collection looks like a regular local collection but it displays two additional commands, one for sending pull requests and one for updating.

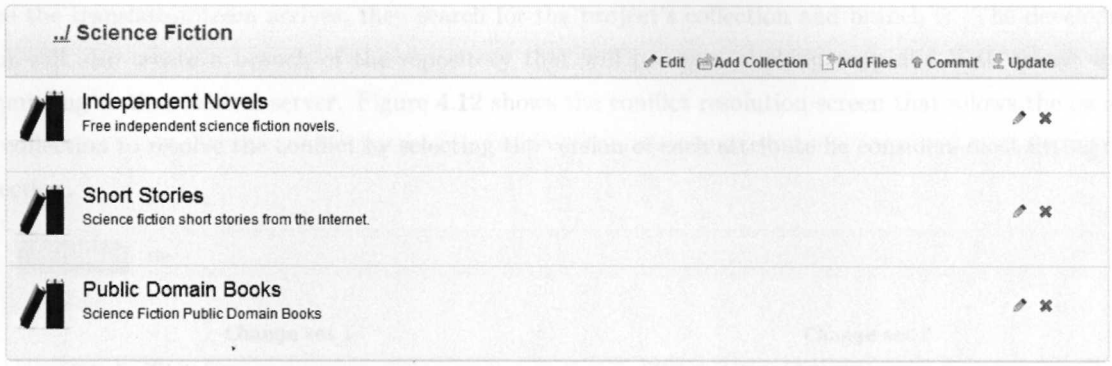

Figure 4.10: After branching the collection, it's made available locally and Commit (Send pull request) and update are available.

In figure 4.11 we can see how the owner of the Science Fiction Collection can see a list of changes that were made to his collection and can decide to approve or reject those changes.

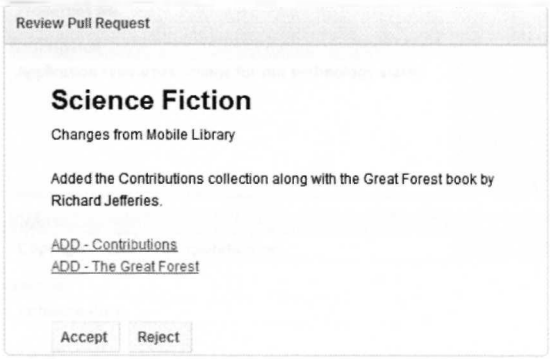

Figure 4.11: Once the Mobile Server makes a contribution and sends a pull request, the owner of the collection can review the changes before accepting them.

4.9.3 Scenario 2: Using a central server as a collaboration hub for distributed locations

A software development company is working on an application for one of its biggest customers and one of the main requirements is to support customers around the world by giving them a user interface in their native language. In order to fulfill this requirement, they have hired a translation company and they are going to have a visit from the translation team to teach them about their development process, planning and implementation strategy and set up a repository for these files.

Since the development team doesn't want to expose its customers code, they have set up a library server repository that's available from Internet that the translation team can use to promote their files and updates.

Once the translation team arrives, they search for the project's collection and branch it. The development team will also create a branch of the repository that will use as a working copy and both teams will be committing to that central server. Figure 4.12 shows the conflict resolution screen that allows the owner of the collection to resolve the conflict by selecting the version of each attribute he considers most fitting to his collection.

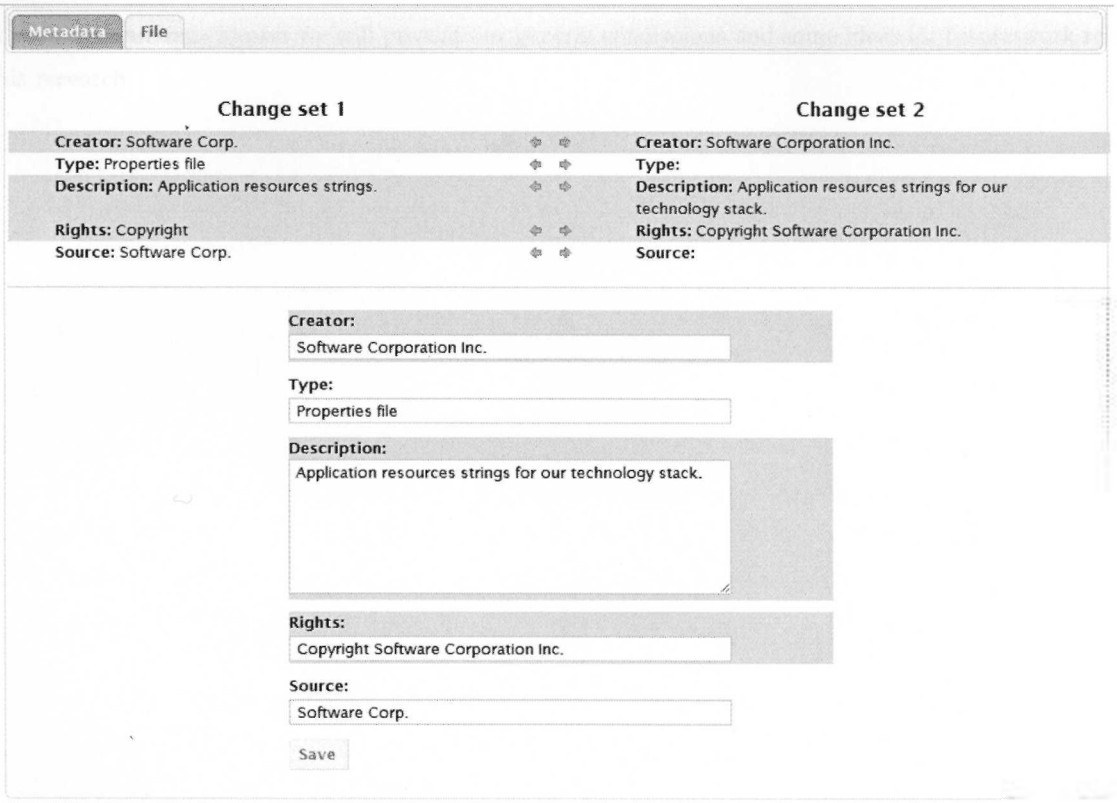

Figure 4.12: Both Intenational Library and Work Library changed the same file, the user merging the change can select the desired properties for each conflict or change.

4.10 Conclusion

In this chapter we have presented a working prototype of the model including implementation details and the technologies that made possible the model operations. The current prototype provides only the most basic library functionality required for the model to be showcased properly but it has been designed to be extended and given more advanced users targeted at the personal use of the digital library.

The prototype was then demonstrated using a set of use case scenarios that cover most of the important operations of the model and shows the ease of use that this approach brings to the users when compared with other mechanisms and the versatility the system offers in order to be able to handle different collaboration

scenarios between two or many library servers.

Finally, we presented some performance tests to compare the model against the regular approach, the results show an improvement to the conventional approach that would be much more significant on a busier network environment or with library servers on different geographic locations.

In the next and last chapter we will present our general conclusions and some ideas for future work related to this research.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Currently there are several digital library systems available, but so far very few of them have tried to solve the problems that arise when trying to work collaboratively among different digital library servers, as is a very common case with personal digital libraries, users are missing a great opportunity in terms of leveraging the knowledge and assets existing on the networks they belong to, by solving this problem they could proactively give or receive help or obtain high quality information from their peers at school, work or home environments and perhaps even help thrive the adoption of digital libraries from many more users.

Another problem with existing digital libraries is that they have not evolved with the same pace the software development industry has, modern software development is trending towards the creation of platforms and giving users the power to extend functionality of the software itself, to giving users better ways to work and collaborate and to having more reliable decentralized services and applications that can run or be accessed from any device.

The contribution of this research is the design and implementation of an interoperability and collaboration model that is capable of solving the collaboration problem in a decentralized manner while using a modern approach that goes along with the current trends of software development and defines a solid framework for others to develop new tools to handle more advanced or specific interactions between digital libraries.

The presented model consists of a set of operations and work flows that allows users to collaborate with each other, the most important of this operations being: Search, Branch, Update, Send Pull Request, Accept Pull Request and Resolve Conflicts, these were presented from a design and implementation perspective in chapter 3 and 4 respectively.

The implementation of the model heavily relies on CouchDB but it's perfectly possible to implement using a different technology stack, but given the amount of features that overlap with the goals of the model it made a lot of sense to use CouchDB for the prototype. In general it's a good match but unfortunately many of the advantages of using a non predefined schema data storage structure were not as useful as they could be be due to the static typing nature of Java. We feel that by using a more dynamic language it would be so much easier to integrate more naturally with the CouchDB storage and make development easier. Also, since CouchDB is a fairly new technology, there were some consequences in the development of the prototype, the first one was the lack of high quality documentation, some of the less popular features were not documented at all in the official sources, and the second being, new versions and features are frequently released but they aren't always backwards compatible, which means in order to upgrade, usually the application had to go through some maintenance.

As shown by the results obtained in chapter 4, this prototype is a big move forward for digital libraries, it brings many advantages to users in terms of availability, ease of access to information and extendability and we can see it being developed further and implemented on environments similar to the ones presented through this work, this is why on the next section we present some of the future work that might be expected to follow up this effort.

5.2 Future Work

The current work creates an excellent platform that could be used for many interesting projects, among the most relevant we would like to mention the creation of a native OS integration with the file system. The described program would map folders to collections and files to documents, and would provide users with seamless integration with the way they currently work, the implementation of such tool would ease the maintenance operations in the library and would increase even further the freedom from users to use the tools of their choice to handle their collections' assets.

Better copyright management and intellectual property protection of branched collections, we currently keep references to the source of documents and collections, but we feel that once branched documents are taken out of the library it's not possible to prevent abuse of the information, this problem could be solved by implementing some sort of DRM mechanism, possibly in combination with watermarking tools or digital signature of documents in the collections.

Finalize the implementation of the semantical merging of metadata information and text documents. In the current work we have presented a proposed solution and we have detailed the design of such solution but we didn't implement it on the final prototype.

Interoperability with other digital library systems, currently we have implemented a REST API that anyone can use to obtain information from the prototype library but it would be very important to implement some of the well known digital libraries protocols like OAI (Open Archive Initiative) or the METS protocol, this would allow users of the library leverage the content of other digital library systems available, this idea could even be extended to proposing a collaboration protocol for other libraries.

Automatically finding metadatada for local documents using the network of library servers. The current interoperability and collaboration mechanisms could be extended to automatically try to find similar documents within the network and complete missing metadata in local collections.

Bibliography

[1] Witold Abramowicz. *Knowledge-based information retrieval and filtering from the Web.* Springer, 2003.

[2] F. Alvarez-Cavazos, D. A. Garza-Salazar, and J. C. Lavariega-Jarquin. Pdlib: Personal digital libraries with universal access. In *5th ACM/IEEE Joint Conference on Digital Libraries*, pages 365–365, Denver, CO, 2005. Assoc Computing Machinery. 0 NEW YORK BCO70.

[3] William Y. Arms. *Digital Libraries.* MIT Press, 2001.

[4] Bela Ban. *Reliable Multicasting with the JGroups Toolkit.* Red Hat Inc, 1.15 edition, 2010.

[5] Jeffrey Beall. A wiki-like approach to metadata editing, September 2010.

[6] Brian W. Fitzpatrick Ben Collins-Sussman and C. Michael Pilato. *Version Control with Subversion.* O'Reilly Media, 2004.

[7] Bernadette G. Callery. *Collaborative access to virtual museum collection information: seeing through the walls, Volume 7.* Routledge, 2004.

[8] Priscilla Caplan. *Metadata fundamentals for all librarians.* American Library Association, 2003.

[9] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* Addison Wesley, 2003.

[10] George Fairbanks and David Garlan. *Just Enough Software Architecture: A Risk-Driven Approach.* Marshall & Brainerd, 2010.

[11] Martin Fowler. *Patterns of enterprise application architecture.* Addison-Wesley Professional, 2003.

[12] David Bainbridge Ian H. Witten and David M. Nichols. *How to build a digital library.* Morgan Kaufmann, 2009.

[13] Jan Lehnardt J. Chris Anderson and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax.* O'Reilly Media, 2010.

[14] William Janssen and Kris Popat. Uplib: A universal personal digital library system. Technical report, Palo Alto Research Center, 2003.

[15] Joe Lennon. *Beginning CouchDB.* Apress, 2009.

[16] Richard P. Smiraglia. *Metadata: a cataloger's primer.* Haworth Information Press, 2005.

[17] Shashank Tiwari. *Professional NoSQL.* John Wiley and Sons, 2011.