A Collaborative Document Development Environment Using XML and Mobile Agents

by

Timothy Donald Newell

B.C.S.H., Acadia University, 1996

Thesis
submitted in partial fulfillment of the requirements for
the Degree of Masters of Science (Computer Science)
Acadia University
Spring Convocation, 2000

© by Timothy Donald Newell, 2000

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-51999-6

Canada

Table of Contents

List of Tables v
List of Figures vi
Abstract vii
Acknowledgements viii
Chapter 1 – Background and Overview 1
1.1. Background 1
1.2. Overview 6
1.3. Architecture 7
1.4. Operations 13
1.5. Summary 17
Chapter 2 - Relevant Technologies 18
2.1. Voyager DXML 18
2.2. JSDT 18
2.3. JNDI 19
2.4. JAAS 19
2.5. JCE 20
2.6. JSSE 20
2.7. Java API for XML Parsing 20
2.8. GSS-API 21
2.9. SASL 21
2.10. XML Parser for Java 21
2.11. Log for Java 22
2.12. Xeena 22
2.13. XMLTreeDiff 23
2.14. ObjectSpace Voyager 23
2.15. XML Security Suite 23
Chapter 3 – Requirements and Basic Functionality 24
3.1. Design Goals 24
3.2. Supported Operations 26
3.3. Description of Components 30
3.4. Required Infrastructure 38
3.5. Usage Scenarios 39
3.6. Security Comments 45
3.7. Implemented Features 47
Chapter 4 – Design 48
4.1. General Design – Techniques Applied 48
4.2. Basic Components and Infrastructure 52
4.3. Specific Components 59
4.4. Design Summary 65
Chapter 5 - Implementation 69
5.1. General Implementation Considerations 69
5.2. Implementation Process 72
5.3. Issues Encountered 81
5.4. ObjectSpace Voyager – Concepts and Comments 88
5.5. Modifications to Original Plans 93
5.6. Required Infrastructure 95
Chapter 6 – Summary and Conclusions 97
6.1. Results 97
6.2. Application Improvements 98
6.3. Analysis and Measurement 99
6.4. Further Investigations 102
6.5. Conclusions 103
Bibliography 106

List of Tables

Table	Page
Table 1: XMLTreeDiff Testing	88
Table 2: Packages and Environment Variables	95

List of Figures

Table	Page
Figure 1: Component Overview	30
Figure 2: Directory Service	31
Figure 3: Core System Classes	67
Figure 4: Document Directory Classes	68
Figure 5: Document DTD	95
Figure 6: Main User Interface	98
Figure 7: Document Browser	98

Abstract

The purpose of this thesis is to explore the use of autonomous mobile agent and Extensible Markup Language (XML) technologies for collaborative document development. The application of these technologies for collaborative environments is investigated to support computers permanently connected to a network, as well as intermittently connected devices such as laptops. A prototype environment has been implemented as an experiment to determine the feasibility of this model and facilitate further investigation. Support is provided for disconnected operation, distributed development of documents, local execution of computation-intensive operations, and a "push" model of information sharing. The more traditional "pull" model of sharing is also implemented. There are a number of collaborative environments currently available, but it is believed that the merging of mobile agent and XML technology for such applications is a relatively new domain. The intent is to explore this topic as a new collaboration paradigm, and make observations concerning its usefulness.

Acknowledgements

I would like to take this opportunity to thank my wife Heidi for all of her love, patience, and support throughout this degree, and most especially during these last few months. This would not have been possible without her. I would also like to thank my children, Katie and Aidan, for putting up with Daddy having to work all the time instead of playing with them.

My supervisor, Dr. Tomasz Müldner, has been very accommodating throughout this process. He was always willing to work around my schedule and go above and beyond in supporting me, for which I am grateful.

I would like to thank my employer, xwave solutions, for their support and sponsorship of my pursuit of this degree.

Heather Laine graciously assisted me by proofreading and correcting the grammar of this document. Her comments and suggestions added greatly to the quality of the final paper.

Finally, I would like to thank Tim Beamish for his assistance during the prototype implementation.

Chapter 1 – Background and Overview

1.1. Background

The term "collaborative environment" signifies many different technologies to different people. These technologies range from simple messaging environments that use e-mail and similar applications to share information, through teleconferencing and related applications allowing the real-time exchange of audio, video, and data. Scheduling and contact management applications provide basic information sharing capabilities, and are the next step in complexity from messaging environments. Document-sharing and document-management applications extend the complexity of scheduling and contact management applications further, while remaining less comprehensive than teleconferencing systems.

Collaborative document environments are used to allow multiple users to share documents. Some environments allow simultaneous access to documents, while others simply provide a central document store which documents can be "checked out" from for modification. Revision tracking and control facilities are commonly provided by these applications, to allow participants in the development of the document to review previous iterations and track progress. Two general categories of collaborative document environments are synchronous and asynchronous systems. Synchronous systems propagate document changes to all participants immediately. This allows "real time" collaboration. Asynchronous systems do not immediately update all copies of the document being worked on, but provide some mechanism for periodically updating or synchronizing each copy. The approach to collaborative environments described in this document is based on an asynchronous collaborative document model. The implementation of this model uses mobile agents and the Extensible Markup Language (XML).

Both mobile agents and XML are relatively new technologies. These technologies offer significant and useful benefits, as well as some drawbacks to be considered. Active research is being

conducted to resolve many of these drawbacks. Others drawbacks arise from a lack of completed standardization efforts. The rest of this section will discuss some of the relevant benefits and weaknesses of both mobile agents and XML.

1.1.1. Mobile Agents

Mobile agents, or agents, can be thought of as threads or processes which migrate from machine to machine during execution. They are autonomous, in that they can interact with their environment and initiate actions independently. Agents typically travel to various hosts in the performance of some task, such as visiting a number of different travel agents to obtain the lowest price on a particular flight. They carry both their execution state and data along with them when they migrate. In most cases, they can carry their code along with them, enabling them to execute on hosts that did not already contain their classes.

Agents can continue to execute on remote machines even if their original host disconnects from the network or crashes. When the original host is available again, the agent can return to or contact it with the results of its operations. Agents are useful for offloading processing to more powerful servers. They can be launched from very simple client devices, such as Personal Digital Assistants (PDAs), which might not be capable of providing much functionality normally. They can distribute the processing of a problem among multiple machines (in parallel or in serial), utilize specialized services or hardware at a remote location to perform a particular operation, or travel to a meeting site to interact and collaborate with other agents.

One of the major benefits of agent technology is the ability to move the processing of an operation to the location of the required data, rather than transferring all of the required data to the local host for processing. This ability can yield very substantial gains when working with large data sets, i.e. when the overhead for transferring an agent to a remote site and returning its results is lower than the overhead of transferring all of the required data. Agents form the primary

transport mechanism for data in the prototype collaborative system. That is, they are responsible for distributing data to participating systems and carrying out operations on data locally.

There are many different applications quoted as examples of how mobile agents can provide benefits over existing methodologies. At the same time, "Mobile agent proponents will begrudgingly admit that the problems for which their systems are appropriate can also be solved with normal distributed computing technologies like RPCs and distributed objects." [Kiniry et al.1997] That is, there is no single major application enabled by mobile agents that could not be implemented using other technologies. Harrison, Chess, and Kershenbaum summarize the issue by stating that, "whereas each individual case can be addressed in some (ad hoc) manner without mobile agents, a mobile agent framework addresses all of them at once." [Harrison et al. 1995]

Mobile agents are subject to a number of limitations. These limitations may influence their effectiveness for this type of application. Agents typically operate within a restricted "sandbox" environment, similar to Java applets. Restrictions may be placed on the operations which agents are permitted to execute. These restrictions may require changes in the types of operations conducted by agents, or the manner in which they are performed. For example, an agent may be prevented from opening network sockets. This would prevent an agent from connecting to its source host to report its findings remotely. An alternate mechanism for returning results would have to be chosen. Fortunately, these restrictions are often configurable. However, this issue brings up one of the major weaknesses of current agent technologies, which is security.

Agents require different security measures than traditional computing models, both to protect themselves from malicious hosts or agents, as well as to protect hosts against agents. Although some of the issues surrounding security and agents have been resolved, problems still remain. Agent security is discussed further in the security section presented below.

Agents require that an agent execution environment be available on each host which they are to visit. The execution environment is typically a server process that sends and receives agents, as well as providing services to them. Provision of this facility requires some configuration effort on each destination system. It should be noted that this administrative overhead is similar to most traditional applications, which require that software be installed or similar steps taken in advance of using an application. Unlike most traditional applications, however, once a mobile agent execution environment is available on a host, many different types of agents and agent applications can share the same environment. Configuration of the execution environment then becomes a one-time effort for a number of applications, rather than a recurring task for each new application or upgrade.

As agents remain a relatively new technology, there are few general standards for agents. This leads to difficulties when trying to get agents from different vendors or which were written in different languages to interoperate, or even share execution environments.

1.1.2. XML

XML is a subset of SGML, the Standard Generalized Markup Language. It is a "meta" language that is used as a tool for specifying new languages. These new languages are generally domain-specific grammars. XML might be used, for example, to create a standardized language defining data formats for filing insurance claims with various insurance carriers. XML tools can compare a document with the Document Type Definition (DTD) defining the proper format of compliant documents and verify that the document's format is valid. Thus, document formats can be standardized and easily verified.

XML is a markup language. Elements of documents are identified using special tags. Tags are enclosed in angle brackets and are used to mark the start and end of a particular element. Tags can be nested. The possible tags for a document type, as well as their relation to other tags, are

specified as part of the DTD. When viewing the contents of an XML document, the tags and formatting typically look very much like the HyperText Markup Language (HTML). One important difference is that XML is concerned with data content and the relationships among elements, while HTML is just a presentation language. Other tools, such as the XML Style Language (XSL), are used to control the presentation of XML data. A single XML document can be presented in many different formats through the use of different XSL style sheets. These style sheets can control how much of the data is displayed, such as when presenting a summary of a document.

XML parsers are applications which process and manipulate XML documents. Parsers can validate that documents are formed correctly based on the document's DTD. Documents can be edited and individual document elements accessed using parsers. There are two widely used standards for XML processors, the Simple API for XML (SAX) and the Document Object Model (DOM). SAX uses an event-driven model in which applications receive notification of events as documents are parsed. Examples of events include reaching the start of a new element or the end of the current one. SAX is an efficient mechanism, especially for scanning documents for specific fields. DOM is an alternate specification that processes XML documents by creating a tree of objects in memory representing a complete document. DOM is well suited for operations based on entire subtrees of the model, such as moving a section from one part of a document to another.

The use of tags to identify different data elements provides a number of benefits for automated processing of documents. Tags add context to data. Specific types of tags can be searched for rather than doing simple text string matches such as those done for HTML data. Since tools can be written which are generic to XML, much of the work required to utilize such a rich and flexible language can be avoided by using standard, off-the-shelf utilities. Finally, using XML documents as a standardized language for communication may help facilitate the interaction of different types of agents from multiple vendors, as well as processing by more traditional applications.

SGML, on which XML is based, is a technology that has been in use for many years. However, some of the more advanced operations on XML documents are relatively new facilities still undergoing standardization and finalization. As a result, some operations are proprietary to certain tools, or not well supported yet. In addition, the various tags used by an XML document can add extra bulk to a document, beyond what the raw text of the document would require. (This tends to vary based on the DTD and purpose of the document.) Finally, there can be significant overhead in processing and validating XML documents. This overhead may be higher in some cases than that required for special-purpose formats, but is mitigated somewhat by factors such as the productivity and reliability gains to be found from using standardized tools and utilities.

1.2. Overview

A sample implementation providing operations for both offline and networked functionality forms a part of this investigation. Both "push" and "pull" models of communication are supported. Users can choose to select and download documents and updates as well as automatically receive pushed document updates. The current implementation provides the ability for documents to be shared among a group of users, each of whom can add to or modify the original document as desired. Changes are propagated to other users at the modifier's discretion. Modifications from other users can be merged into the local copy of the document. An interface is provided to support the use of external tools for creating and modifying XML documents.

Support for style sheets and the Extensible Stylesheet Language (XSL) was not included as part of this work. Such features are used to control the presentation of XML documents. Since these features do not affect the core parameters being examined, and there will soon be a variety of third-party tools available to easily provide such functionality, they were not implemented.

The current implementation uses a standard XML Document Type Definition (DTD) for all collaborative documents. The initial DTD support was kept simple, as the focus of this

investigation is on the use of agents and XML together. A minimal DTD is all that was required for such purposes. In implementing a more feature-rich system, the ability to employ arbitrary DTDs would be useful. This ability could include run-time negotiation of DTDs to use, as well as conversion between different DTDs using technologies such as XSL Transformations (XSLT) if desired.

1.3. Architecture

The basic components of this system include a user interface, a local system manager at each node, mobile agents, and a directory server. The user interface provides facilities for the user to invoke the various operations of the system; edit, manage and distribute documents; and control the various configuration parameters of the system. The local system manager serves as the coordinator of operations. It implements most of the core system functionality, and interfaces among the user interface, mobile agents, and underlying operating system. Mobile agents are used to implement all remote operations. They are responsible for distributing and requesting updates, publishing documents, and so on. Finally, the directory server is a key piece of infrastructure which allows users to locate other users, determine what documents are available, and control access to their own documents. The rest of this section will provide a more detailed discussion of these key components, as well as other relevant architectural considerations.

1.3.1. Graphical User Interface

A simple Graphical User Interface (GUI) is provided to allow the user to select from the various operations. The GUI allows the user to access functions to create, modify, search for, and share documents. A list of available local documents can be browsed, including information about the document such as author, version, and modification date. Documents that are ready to be distributed can be published to the directory service, allowing other users to discover them.

Permissions and access control lists for documents can be manipulated. Operations such as checking for updated copies of documents can also be managed from the GUI.

1.3.2. Local System Manager

Each node's execution environment includes a local system manager. This system manager provides services to mobile agents, and essentially serves as the interface to the host system for mobile agents. New mobile agents are created to carry out any remote operations required by the manager. The local system manager also serves as the mechanism by which a user's commands are carried out. The user interface invokes operations on this component. The local system manager then services the request itself if appropriate or works with mobile agents as necessary.

1.3.3. Mobile Agents

As described above, mobile agents are used to perform remote operations. These agents travel to the workstation of one or more other users involved in a collaboration. Upon arrival, they interact with the destination system's stationary agent. Interactions can involve: requests for document information; identifying differences between documents; pushing updates to the destination system; and carrying the results of operations back to the originating client. For example, a client would dispatch an agent in order to distribute a new section for a shared document. The agent would travel to each user involved with the system's computer. On arrival, it would invoke an operation on the local manager to notify it that a new section was available, and pass it the document update. The stationary agent would then take responsibility for actually adding the changes to its existing copy of the document. Once it had delivered its data, the mobile agent could proceed to the next host or return to its point of origin with the results of its actions.

Although this approach may seem to be a simple variation on standard Remote Procedure Call (RPC) or Remote Method Invocation (RMI) client/server programming, there are several benefits. All dialogs between the mobile agent and stationary agent at the target are local. Complex interactions can be encapsulated within a single agent, which is capable of making a series of different calls to the stationary agent and taking independent action based on the results. Such operations could include a mobile agent collaborating with the local system manager to traverse the DOM tree of an XML document to compare or merge differences, involving a significant amount of processing within the mobile agent. The client machine can crash or disconnect from the network as soon as the agent has been sent, and the operation can still continue.

1.3.4. Directory Service, Users, and Groups

A directory service provides information to locate users, list their available documents, and manage access permissions for these documents. Operations could be supported to locate a user's computer, search for documents by name or author, or check for access rights to a particular document. Directory entries can also be added, deleted, or modified. Users must be the owner of an entry to delete or modify it. Identification of users is done through the use of a unique user name and cryptographic signature.

The directory service is also used to maintain group membership lists identifying the users collaborating on each document. Group membership can change during the life of a collaborative effort, as participants join or leave the group. Groups can be associated with more than one document at a time. More than one group can collaborate on a document. Groups contain one or more users. Groups may be nested; that is, one group may contain another group.

Access permissions can be specified for each entity associated with a document, whether that entity is a single individual or a group. Access permissions specified for a group apply to all members of that group. If a user is identified individually and as part of a group, the group

permissions may be consulted for permission to operate on the document if the individual's own permissions are insufficient. Only the owner of a document may delete the document, or change the document's access permissions. The specific user who creates a document initially owns it. The existence of the document is published to the directory service. Other users may indicate to the document owner that they are interested in collaborating on the document. The document owner may then decide whether to allow the user to participate or not, and under what initial access permissions. The initial owner of a document may choose to give ownership of the document to another user or group. If a group is specified as the new owner of a document, all users in that group have owner privileges on the document.

1.3.5. Document Management and Storage

Documents are stored in a set of directories. These directories are used to identify documents to be pushed to other users, older versions of documents, and to separate documents belonging to different groups. Copies of different versions of documents are maintained with older versions of documents being stored in an archival directory. Storage of old document versions is done both for backup purposes, as well as to simplify the identification of document changes. Documents are stored as flat text files containing XML markup.

An interface is provided which allows the user to browse a list of the various documents available. This interface provides details for each document such as its owner, size, version, and date and time of last modification. Further information, such as access permissions and a list of participating group members can be obtained by querying the directory service using the file browser interface. The interface can also be queried to show the locally available versions of a particular document. Document editing and access control operations can be invoked from the browser interface.

1.3.6. Security

Any collaborative system faces a number of different security concerns and possible attacks. Such systems must take steps to protect against the subversion of remote commands by malicious parties. User identities must be protected against masquerading attacks. Denial of service attacks, such as deleting data or shutting down services, must be considered. The integrity and confidentiality of important data, both while in storage and during transmission, must be protected. Some of these concerns are generic security concerns which any significant application must deal with while others are inherent to networked and distributed systems.

Any application that interacts with a computer network is inherently exposed to greater risks than one existing on an isolated system. Networked applications are subject to remote attacks. Remote attacks are more difficult to defend against, as they may come from any point within the network (subject to various barriers such as firewalls). Such attacks may originate from systems outside the control of the owners of the local application, making it easier for a malicious party to stage an incident.

The situation is further complicated by the fact that although it may not be feasible to compromise an application's host directly, it may be possible to successfully attack a system indirectly. Gaining control of a host trusted by the application's system would be the first step of such a process. The trust relationship between the two hosts might then be exploited to attack the application host. Essentially, a system is only as secure as the least secure system it trusts. Issues such as these, and solutions for many of them, are well known and described in much of the network and security literature. These issues will not be discussed further except to note that they serve as important information to consider when evaluating the security risks of any new system.

The mobile agent paradigm introduces a number of new security issues. These issues are discussed in detail by [Farmer et al. 1996]. The main problems are associated with the issues of

allowing unknown (and thus untrusted) agents to safely utilize a host for some activity, and for an agent to use an untrusted host for performing its work. Allowing untrusted agents to utilize a computer system may allow them to steal information or otherwise abuse their access. Hosts have complete control over the execution of agents. Malicious hosts may steal or alter data carried by agents. For the purposes of this investigation we will make the assumption that agents and their hosts are known to each other. The set of users participating in a collaborative effort are a set of known entities whose identities can be accurately determined through the application of cryptographic techniques such as digital signatures. Users are associated with hosts (stationary agents) and mobile agents. It is thus possible to determine whether any agent or host is associated with a trusted user. Access to the system can then be restricted to the set of known and properly authorized users. As long as the communication mechanisms and user identification process are properly protected, a reasonable level of security can be maintained. Since the system uses cryptographic techniques to protect data and identify users within a closed user base, most of the major security issues that arise when considering mobile agent technology are mitigated.

Security concerns are addressed in two ways. Permissions of physical files can be set to provide basic protection while in storage, and protect the system from other components of the system. In addition, encryption and/ or digital signatures can be used to provide enhanced security for stored documents if desired. Digital signatures are a simpler form of protection used to detect tampering (protect document integrity), while encryption can be used to ensure confidentiality of documents. Both techniques are used during communication and transmission in order to protect against snooping and detect any attempts at altering the agents or their data. Two common methods of securing transmissions can be used. Encrypted transport tunnels, such as Secure Sockets Layer (SSL), can be used to protect communications between execution environments. Alternately, public key encryption can be used to encrypt agents and data with the destination's public key.

In general, the collaboration system adopts a conservative, fail-closed approach to security. It is a common security requirement that in the event of a failure associated with a service, that service should refuse access. For example, if a directory service is unavailable to verify a user's credentials, the user should be denied access to the service. Firewalls are another good example. In the event that a firewall fails, the default system state should be such that access to the resources being protected by the firewall is denied, so that an attacker cannot gain access to the resources simply by crashing the firewall. Although a simple and obvious principle, it is common to see many systems designed with a fail-open approach that is generally more convenient to work with. Such systems allow access unless given a reason to refuse it.

1.3.7. Administration

It can be argued that any significant system requires an administrative interface of some sort. Control over advanced configuration options is only given to users with administrative access. Such users can also override access controls in certain situations. Facilities to control the directory service, including user and group maintenance, should be available to administrators.

1.4. Operations

There are several classes of operations supported by this system. Local, or disconnected, operations can be performed using a system not currently connected to the network. Local operations include viewing or modifying existing documents, creating new documents, preparing mobile agents for tasks to be initiated at a later time (such as when reconnecting to the network), and controlling the access permissions of documents. Remote, or connected, functions require that a valid network connection be available in order to communicate with a separate system. They include searching for (retrieving) remote documents, publishing the availability of a new document, and pushing updates to interested parties. Administrative functions such as managing users and groups can also be provided. Where appropriate, functionality is provided by

interfacing with or extending existing third party products, such as those available from IBM's AlphaWorks web site (http://www.alphaworks.ibm.com).

1.4.1. Local Operations

Local operations are those functions that do not require interaction with another system, and thus do not need a network connection. Note that although a network connection is not required, the presence of such a facility does not prevent local operations from functioning. Local operations generally allow a user to continue to develop documents independently, even when not connected to the network. Specific useful operations include:

1. Browse Documents – The document browsing interface described above is available for use even when offline. Certain functions, such as those relying on the directory service, are unavailable while not connected to the network. Further local operations can be invoked from the document browser.
2. View – Existing documents could be viewed using a standard text editor or third party XML document viewer. Advanced features such as formatted display using style sheets are possible through such third party applications. In some cases, document viewing operations may feature more advanced display options than editing tools, or may even involve converting the XML content into a more easily displayed format, such as HTML. Access permissions may require that a user only be given read access to a document, and not be permitted to modify it. View mode can be useful in such situations.
3. Create / Edit – Documents can be created or updated while offline. Notification of updates will not be available to other group members until the system is reconnected to the network, however. As with viewing documents, editing operations can be performed using a text editor or third party tool such as an XML editor.
4. Control Access Permissions – Users can manipulate the permissions of documents which they control while disconnected. Any changes to permissions can be propagated to the directory service on reconnection.

5. Schedule Remote Operations – Operations that require network connectivity to complete can be prepared and scheduled while offline. Basic scheduling functionality might include the ability to initiate scheduled operations automatically on reconnection to the network. More advanced capabilities would feature a combination of time-based scheduling and checking of connectivity status.

1.4.2. Remote Operations

Remote operations require network access in order to perform their task. Mobile agents dispatched from the requestor's system carry them out. The mobile agent then travels to one or more remote hosts to perform its task. Remote operations could be scheduled for later execution in the event that a network connection is not immediately available. Possible remote operations include:

1. Push Updates – Updates made locally to documents can be pushed to other participants in the group. Documents to be pushed are identified using a file selector dialog. The update will be pushed to all members who have registered for membership in a group having access to the document. The directory service is also notified of changes so that the summary information for the document can be updated.
2. Publish Document Availability to Directory – When new documents are created, their existence can be published to the directory. By default, only the document creator has access to the document. Other users can access the directory server and register an interest in the document. The owner will be notified and given the opportunity to accept or reject the membership application. Information published to the directory includes the document title, owner, last modification date, and group associations.
3. Search for Document – A mobile agent may be dispatched to visit the participants in a group in order to search for a document or document fragment. In some cases, a search might be done to explicitly obtain a new or modified document section. It may also be used to obtain a complete copy of a document listed in the directory service. When the mobile agent arrives at

the remote destination, it interacts with the stationary agent at the location to obtain a copy of the requested document or document fragment.

4. Merge Differences – Two different versions of a document can be merged. These differences may have been identified by a previous operation to identify the specific changes, or that operation can be invoked automatically as part of the merge. If the set of differences is already available from an earlier identification operation, it is possible to merge them into the local document with no further remote interaction. In either case, once the differences are identified, the rest of the merge can be completed offline.
5. Extended Document Browsing – Although basic operations to review the locally available documents are available while offline, extended operations require network connectivity. Online browsing operations include the ability to identify changes to the participant list for a document. A check can be performed to determine whether updates are available which have not been received yet. The user can browse a list of available documents from the directory service and identify which are available locally. Operations requiring the directory service, such as joining or leaving a group, can be performed.

1.4.3. Administrative Operations

A set of basic administrative operations should be provided for system maintenance. These operations include the ability to manage user and group lists. Users could be added and deleted, and their properties updated as required. Groups could be created, deleted, or modified. Directory service data and configuration parameters could be updated. In general, control of basic system parameters and overrides for standard restrictions should be accessible through a protected administration facility.

1.5. Summary

This chapter has identified the intent and scope of this thesis. The intent of the project is to develop a prototype system to support collaborative development of documents using XML and mobile agents as underlying technologies. Information was presented which describes the justification and technological background for the investigation. An overview of the prototype system, including its basic architecture and operations, was provided. The scope of the initial implementation has been restricted to the core functionality necessary to demonstrate the concepts being explored. As much as possible, the design presented in this document includes expanded capabilities expected of a fully functional system.

The next chapter presents the general design of the system. The supporting infrastructure, both for development and operation, is described. The tools used and the reasons they were chosen are discussed. A high level design of the various major system components is followed by a presentation of the detailed design decisions made and implementation process. The final chapter discusses the conclusions reached from the implementation of the system, including detailed comments describing potential future investigations.

Chapter 2 - Relevant Technologies

There were a number of potential technologies investigated during the research phase of this thesis. Each of these technologies is discussed in turn, including their strengths and weaknesses. Some of the products and libraries discussed are included or planned for the implementation of this thesis, while others were determined to be unsuitable. The reasons for these decisions are also discussed.

2.1. Voyager DXML

ObjectSpace Voyager now offers a facility to manipulate and process XML documents called ObjectSpace Dynamic XML (DXML). The documentation for this technology promotes its ease of use. While interface appears simple to use, the actual functionality supported by Voyager's DXML is still rather limited. At the current time, the parser support is non-validating only. It seems to be in a very early state, perhaps even an early access or beta stage, based on the scope of functionality provided. Future releases will likely expand the capabilities of this product significantly. Once the product has further matured, it will likely be very useful.

One other drawback observed is that the API provided does not appear to be standards-based. Most XML parsers tend to support either the DOM or SAX standards, if not both. ObjectSpace's DXML appears to be a replacement interface for these technologies. Although it appears to be simpler to perform common tasks using DXML, the fact that it is not a standard interface is something of a concern, as code developed using it will be incompatible with other XML parsers.

2.2. JSDT

The Java Shared Data Toolkit is a Java API providing services for sharing data among multiple participants in a collaboration. Support for various communications mechanisms for distributing

shared content among multiple participants using a standard interface is provided. The emphasis of this toolkit is on supporting communications protocols to simplify the sharing of data among members of a group. Group membership operations are also supported. Although group-based communication and membership operations are important parts of the system discussed, the JSDT was not used in this implementation. The facilities provided by the JSDT offer a more traditional distributed computing model of group communication, rather than an agent-based approach.

2.3. JNDI

The Java Naming and Directory Interface is one of the Java 2 Enterprise Edition technologies. It provides an object-oriented and consistent interface to various naming and directory services. Support is included for Lightweight Directory Application Protocol Version 3 (LDAPv3) directory servers, DNS, and other standard services. ObjectSpace Voyager provides its own set of wrappers for standard naming and directory services, as well as JNDI support in the Voyager Pro package. JNDI appears likely to become a well-accepted Java standard. One of its advantages is that it provides transparent support for various directory service products. The same code can be used to interface with a variety of different products. JNDI is a useful technology for implementing the interface to the directory service discussed in this document.

2.4. JAAS

The Java Authentication and Authorization Service (JAAS) provides extensions to the standard Java security model to support the concept of user-based security protections rather than code-based. Abstractions for security concepts such as principals and various types of credentials are supported. Access control mechanisms based on user identity are also provided. Although Java is touted as a secure programming language, the previous lack of user-based security mechanisms was a real weakness. JAAS helps to fill this gap in a flexible and generic manner.

The facilities provided by JAAS are considered key to the planned support for user-based access controls and authentication.

2.5. JCE

The Java Cryptography Extension (JCE) is a standard Java extension providing support for cryptographic operations to the Java environment. Facilities provided by this extension include encryption operations, key management and manipulation facilities, and message digest calculations. It forms a core part of the security infrastructure of the Java 2 platform. The services of the JCE and related security products for Java would provide support for the protection mechanisms designed for this thesis.

2.6. JSSE

The Java Secure Sockets Extension (JSSE) is a standard extension of the Java 2 platform. JSSE provides support for encrypted communications channels using Secure Sockets Layer (SSL) and Transport Layer Security (TSL) technologies. JSSE provides a useful tool for encrypting communications between systems. as well as authentication of communication endpoints. This tool could be useful for providing SSL services to the collaborative document system, such as encrypted channels through which agents can travel.

2.7. Java API for XML Parsing

The Java API for XML Parsing (JAXP) is an early access Java technology providing a standard Java interface through which to access the functionality of different XML parsers. Both SAX 1.0 and DOM Level 1 Core are supported. Various XML parsers can be plugged into the architecture. This API is a technology likely to prove very valuable as it matures. At the current time it is not

being planned for use due to its status as an early access technology and lack of support for some of the more advanced DOM Level 2 operations.

2.8. GSS-API

The Generic Security Services API, Version 2 (GSS-API), is an interface defining mechanisms to access standard security services. It is a standard developed under the Internet Engineering Task Force (IETF). GSS-API offers services similar to JAAS and other Java security interfaces. Of interest is the fact that it is being proposed as an Internet standard, and as such is not specific to a particular language. It is a standard that appears to be useful. Due to the better implementation support for JAAS, however, GSS-API was not selected for the design of this system.

2.9. SASL

The Simple Authentication and Security Layer (SASL) is another Internet standard "... for adding authentication support to connection-based protocols." [SASL 1997] As such, it relates to both JAAS and GSS-API. SASL is not specific to a particular implementation language, and supports standard authentication mechanisms. Like GSS-API, this technology overlaps with the facilities provided by JAAS. JAAS was selected to provide authentication services rather than SASL due to its current implementation and documentation support.

2.10. XML Parser for Java

The XML Parser for Java is an XML parser implemented in Java by IBM. It is a popular XML parser and offers support for advanced XML features. This parser is a fairly mature product, and has been favorably received by the industry. Support for features such as DOM Level 2, SAX 2.0, and XML Schema is being developed and is available through an early access release of the

software (some of these features are still in an incomplete state). This tool has been selected as the XML parser to be used.

2.11. Log for Java

Log for Java is a logging facility for Java developed by IBM and available through its AlphaWorks program. Powerful support for logging operations is provided. Messages can be categorized and prioritized. Control over the types of log messages output is accomplished at run time via configuration files. The logging facility has been optimized to incur very little overhead for processing log statements whose output to disk is not currently enabled by the logging configuration. This tool could be used to provide logging support to the document development environment.

2.12. Xeena

Xeena is a Java-based XML document editor from IBM's AlphaWorks program. It parses the DTD being used and provides syntax checking during editing operations. Xeena provides a list of currently valid element choices during editing based on the rules in the DTD for the current part of the document. Different modes of operating on documents are supported. Although some customization of the editor is available, it would be useful to be able to more fully control the user interface of the tool, including which operations are enabled. It is distributed as a command-line binary tool. Programmatic access to Xeena would allow improved integration with the collaborative document development environment. Despite these minor shortcomings, Xeena has been selected as the document editor for use in this thesis.

2.13. XMLTreeDiff

XMLTreeDiff is a tool produced by the AlphaWorks program at IBM. It provides facilities to compare XML documents and merge the differences identified. DOMHASH and some advanced comparison techniques are used in the implementation of XML. Differences are identified using DOM trees, rather than direct comparison of file contents. This technique provides much more accurate results since the same DOM tree can be represented by a number of different text representations (whitespace, entity references, etc. would not be processed properly by doing a simple text comparison). XUL, the XML Update Language, is used to describe the differences between documents. Both command-line and programmatic access to XMLTreeDiff is provided. This tool was selected to provide support for comparing and updating documents.

2.14. ObjectSpace Voyager

ObjectSpace Voyager is a Java framework for distributed and agent-based computing. It supports traditional distributed systems such as CORBA, DCOM, and RMI, as well as mobile agents. It is a system being actively developed and promoted, with new features being released frequently. Voyager is used to provide the mobile agent infrastructure used by this system.

2.15. XML Security Suite

The XML Security Suite is a set of XML security-related components from IBM's AlphaWorks program. It is intended to provide facilities such as element-wise encryption of XML documents and document fragments, digital signatures of XML documents, access control functions, and an implementation of the DOMHASH specification. Currently, DOMHASH is supported but the other functions are largely incomplete. XMLTreeDiff utilizes the XML Security Suite's DOMHASH functionality. This product is used to perform DOMHASH calculations by the collaborative system. Some of its facilities that are under development are very interesting.

Chapter 3 – Requirements and Basic Functionality

3.1. Design Goals

There were a number of goals that shaped the design of the collaborative environment prototype. The general intent was to design a prototype system that could serve as a platform for further investigation of the collaboration paradigm discussed in the previous chapter. This prototype could serve as a proof of concept by demonstrating the feasibility of the proposed technologies. The intention is that the design and implementation of this system should provide a solid infrastructure upon which to base future research. The design must therefore be both flexible and extensible to support the easy integration of new features and refinements.

The architecture and design discussions presented in this chapter include some advanced features not forming part of the actual implementation. They are included to highlight useful features, which could be added at a later time. Those features implemented as part of the proof of concept are outlined later in this paper. Features that could be anticipated for future revisions should be included in the initial design, even if they were not immediately implemented. Such features could include those expected of commercial production-quality software or desirable enhancements. An emphasis has been placed on identifying and discussing areas likely to benefit from further investigation and development.

One of the benefits of using mobile agent technology is the ability to minimize the requirements placed on the underlying communications infrastructure, both in terms of bandwidth and connection reliability. These features are certainly desirable in a collaborative environment. A design goal of this project was to use agents to minimize both the frequency of interaction among components, as well as the amount of bandwidth consumed by interactions. It should be feasible to utilize the system over relatively low-bandwidth communications channels or channels which are not very reliable. Such a goal helps to facilitate disconnected operations. It also tends to yield

"well-behaved" applications, in the sense that they do not overuse relatively expensive network resources.

The final goal of the design is in two parts. The first part is that existing tools, libraries, and standards should be leveraged as much as possible. Tools and components are readily available to implement many common operations or provide the basic infrastructure to utilize advanced technologies. Little is to be gained or learned by redeveloping common infrastructure or complex tools. Most of these utilities and libraries have been tested and reviewed by many knowledgeable individuals, and are likely more complete and robust than those which might be quickly developed in support of a larger project. The cost of using "home-grown" components in terms of future maintenance requirements should not be underestimated. The use of external components can thus save time and effort both in terms of an initial development as well as during the maintenance portion of the software lifecycle.

The second part of this final design goal is that tools and libraries requiring proprietary interfaces should be avoided. Wherever possible, existing standards should be adhered to. Interfaces supporting open standards and protocols are to be preferred. Use of such standards makes the substitution of alternative implementations of a particular feature simpler. Information about, and people with, experience using published standards are typically easier to obtain than when proprietary products are used. The life span of standard interfaces is generally longer than that of nonstandard interfaces, as there are usually alternative implementations available as well as wider industry support.

3.2. Supported Operations

3.2.1. Agent Creation and Dispatch

Agent creation and dispatch operations provide functionality to create the various supported types of mobiie agent and assign tasks to them.

A specialized agent class is available for each task to be performed. Each agent supports a method used to initialize it with its assigned task. Task information is generally expressed as an ordered collection of URLs to visit, and the callback function to invoke at each destination. Additional parameters required to define a particular type of task vary according to the type of agent.

Once created and initialized with a task, agents can be dispatched to begin their work. Upon completion, agents return to their original host and invoke a callback function to return their results. Functionality to allow an agent's owner to check its status and generally monitor it is also useful.

3.2.2. Agent Registration and Monitoring

Agent registration and monitoring operations encompass the functionality required to allow administrative and monitoring components to track the activities of agents active in the system. These facilities provide valuable information and statistics when analyzing the performance and general behavior of the system.

Local agents are those created within the local node. These agents register their existence with the local system manager prior to departure, and deregister upon their return. Agents originating from remote systems and currently performing some task within the execution environment of the local system are referred to as visiting agents. Visiting agents register on arrival and deregister on

departure. The times of these events can be useful for monitoring system performance. Other useful statistics include: the number of agents created; number of visiting agents; number and identities of documents requested; and which operations are commonly performed. The ability to query agent status on demand is useful. Active monitoring of the status of agents, perhaps using a heartbeat-monitoring algorithm, is also beneficial.

All of the information discussed should be available to users or administrators via an interface. Much of it should also be available to other components of the system.

3.2.3. Document Management and Storage

The core of the system is the creation, modification, and distribution of documents. Operations must be provided to create, edit, and store documents. Support for downloading remote documents is required. Version management facilities are needed in the form of tools to associate version information with documents, as well as compare and merge different document versions.

3.2.4. Logging

Logging facilities are an important part of any significant system, whether for debugging program behavior, collecting statistics, or monitoring system status in production. Analysis of log data is generally facilitated through the use of some mechanism to consolidate logs from various system components. Flexible logging which supports the prioritization and categorization of log messages is also highly desirable. The amount of logging performed should be configurable at run time.

3.2.5. Configuration

In general, parameters that control the run time behavior of applications should be placed in configuration files. These parameters include those that might vary from environment to environment, as well as those which enable and disable various features. Parameters subject to change should not be hard-coded within application code. Such facilities help to increase application flexibility and general usability. It is useful to be able to reconfigure applications during execution, especially long running programs such as network daemons.

3.2.6. Permission Management

Documents should be protected from unauthorized access. Facilities must be supplied to manage access to documents. Permissions should be based on the identity of the end user requesting access and any groups in which they are members. The document owner is given control over who is allowed access to the document (subject to administrative override). The document owner or other authorized party should be able to assign or revoke privileges for individual users or groups of users. Permissions include the ability to read, modify, change the ownership of, or delete documents. The ability to participate in a collaborative development effort and receive pushed updates should also be controlled. Some users may be allowed to download static copies of documents but not receive updates automatically.

It should be possible to assess whether a user requesting a document has access to it even if the document owner is not connected. Verification of such permissions requires that the access permissions for documents be stored separately from the document owner's system. This information could be shared among all participants in a collaborative effort, stored in a central server, or both. For now, usage of a central server will be assumed, but further development efforts may wish to investigate the reliability benefits of using a supplemental peer-based distribution of access permissions.

3.2.7. Publishing and Collaboration

Publishing and collaboration operations are those used to distribute documents and document updates, as well as to control participation in a collaborative effort. These operations include publishing information about new documents to interested parties. Other users must be able to register their interest in document updates or simply request a copy of the document without receiving updates. These other users should be able to specify whether they want the ability to modify the document, or just view it. Facilities to push updates to registered users must be provided. Support should be provided to request any outstanding updates, or even specific document portions.

3.2.8. Searching

Facilities allowing users to search for particular documents, document versions, or document fragments (sections or chapters, for example) should be provided. A flexible mechanism for specifying search criteria should be employed, to allow complex queries using combinations of properties such as author, title, and version. A directory service is recommended to enable this type of functionality, as useful search facilities can be complicated and time consuming to implement. Once a set of matching documents and their locations have been identified, mobile agents can be used to gather and return them.

3.3. Description of Components

This section presents the various components of the collaborative environment. Figure 1, below, illustrates the various core components of the system. Significant interactions among components are identified using arrows.

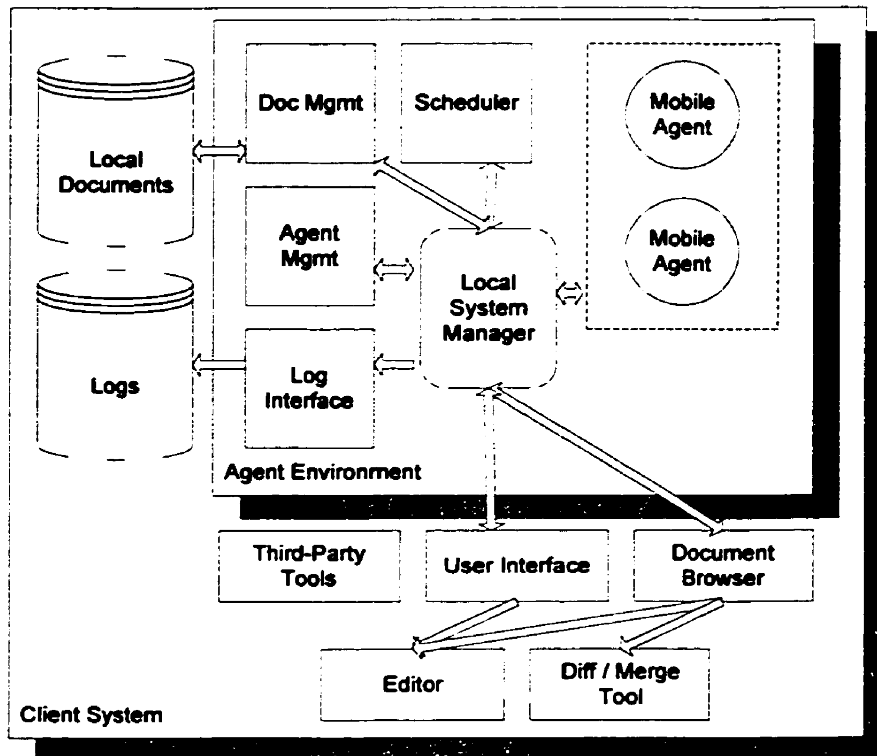

Figure 1 - Component Overview

Figure 2 shows the basic structure and elements of the directory service.

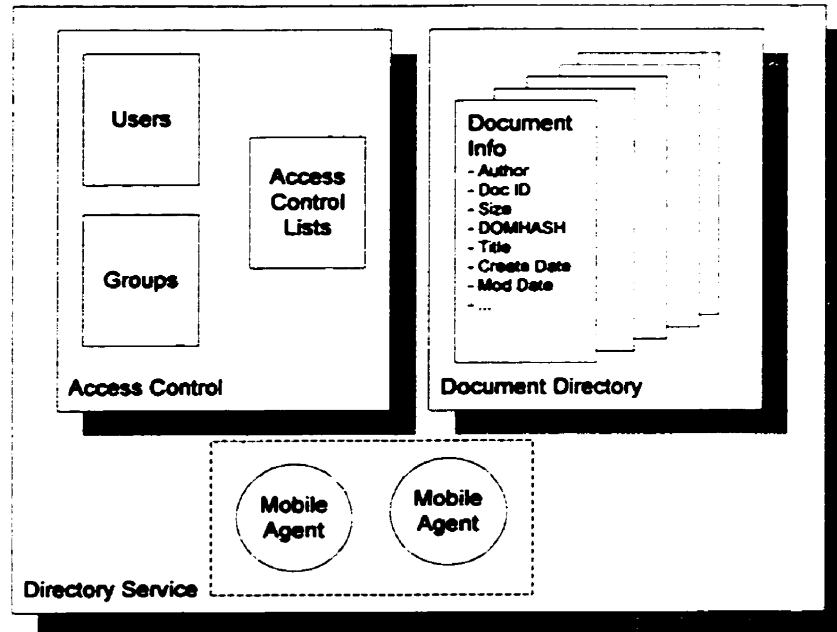

Figure 2 - Directory Service

3.3.1. User Interface and Menu

The User Interface and Menu presents the user with a means to utilize the various facilities of the system. Use of the User Interface requires that a successful user authentication occur. Once authenticated all actions performed by the system are done using the credentials of the current user. This component forms the initial system interface seen by the user. Using the menus provided, the user can access the document browser as well as directly invoke various other operations. It is desirable to use a standard representation for the various operations available to the user in order to keep the programming of the interface generic and easily extend it to include new functionality.

3.3.2. Document Browser

The Document Browser is the heart of the user interface to the system. It provides users with access to the various document development and distribution operations. Existing documents,

both locally available and on remote systems, can be listed and their properties reviewed. New documents can be created and published. Existing documents can be edited and the changes made distributed to other participants in the collaborative effort. Updates can be requested, received and processed.

A common class called docInfo is shared among the directory service, document browser, and other system components to encapsulate the useful properties of documents. The information in this class includes the document author, unique identifier, DOMHASH digest value, date and time of last modification, size, DTD, title, version, whether it has been published yet or not, and filename if the document is available on the local system. The unique identifier might be created by concatenating information such the author's user identifier, URL of the system creating the document, and date/time of creation. The contents of these docInfo objects are stored in the directory service to provide information about published documents. The document browser uses the docInfo class to display summary information about documents, as well as detailed properties on request. Many other system components and operations use the docInfo class to identify documents or document properties.

The Document Browser provides a graphical interface that summarizes information about documents for users. Essentially, it obtains a collection of docInfo objects describing all of the local documents. These docInfo objects can be assembled dynamically by scanning the available files or may be created statically and kept in persistent storage. This information can also be requested from the directory service to obtain a list of published documents. Search criteria can be used to restrict the set of documents returned from the directory service to those of interest.

Functionality supported by the document browser includes the ability to:

- List local and remote documents.
- Edit or view the selected local document.
- Obtain the selected remote document.

- Create a new document.
- Publish a new document to the directory service.
- Request updates for the selected local document.
- Obtain a summary of the differences between two documents.
- Apply a set of differences to an existing document in order to update it to a new version.
- Obtain a list of the detailed properties of the selected local or remote document.
- Review history information for the selected local document.

3.3.3. Document Editor

The Document Editor provides a convenient environment within which to create, view, and edit documents. A third-party application is used to provide this functionality. Use of an abstraction layer to separate the system from the details of any particular editor implementation minimizes dependence on specific third-party products.

3.3.4. Document Comparison and Merge

A third-party utility is used to provide document comparison and merge functionality. As with the document editor, an abstraction layer is employed to isolate the system from the details of the tool in use. This component provides facilities to identify the differences between documents, as well as to apply a set of differences to a selected document in order to update it to match the other document. Difference summaries can be calculated locally and sent to a remote node, which can use the set of differences to update its copy of a document to the current version. Use of these summaries provides a means of efficiently distributing document updates.

3.3.5. Documents

A Document in this system is an XML document and its associated DTD. The initial system implementation uses a single DTD. Future expansion will likely add the ability to utilize a variety of different document types. Therefore, the design and implementation should take this into account. Document fragments, or partial documents, generally exist within the system as document difference summaries. These difference summaries are expressed using the XML Update Language (XUL), an XML language used by the XMLTreeDiff tool.

3.3.6. Local System Manager

The Local System Manager is the coordinator of the system. It provides services to the various other components. It is involved in the active management of a local node in the system, and provides the point of contact for remote nodes. Agent operations and coordination, including services to visiting agents, are provided through the local system manager. General administrative tasks are performed through this component as well.

3.3.7. Mobile Agents

There are a number of specialized mobile agents used to carry out various remote tasks. Update agents push document updates to all those registered to receive them. Search agents locate and retrieve documents according to user-specified criteria. Directory agents are used to publish and retrieve document summary information from the directory service. Permission requestor agents are used to request access to a particular document from the directory service. Requestor proxy agents are used by the directory service system to request access to a document from the document owner on behalf of another user.

3.3.8. Third-Party Components

Several third-party components are used by the system to provide useful functionality. It is important to keep the implementation of the system flexible, so as not to couple the implementation too tightly to a specific tool. In general, the system should not need to care what tool is being used to implement an operation. This separation is typically obtained by employing an abstraction layer for operations using isolating the system from the tool in use. The system operates in terms of the abstract operations. The abstraction layer handles the details of how to use the tool to accomplish the operation. By abstracting from the specific tool, it becomes possible to easily add and remove different components which perform the same types of task.

3.3.9. Document Storage and Management

The document storage and management interface of the system is used to manage the local storage of documents. Functionality required includes managing the versioning of documents, archiving old versions of documents, and coordination of document update fragments. The interface must also be able to manage updates to local documents which have not been processed yet, as well as those which require manual intervention to resolve versioning conflicts.

In general, document versioning is accomplished using the DOMHASH [DOMHASH 1999] value of the document. It is also useful to associate symbolic version tags with a particular document state. Copies of old versions of documents should be stored in an archive for later use in preparing and applying document updates.

Although some consideration has been given to creating a directory hierarchy for each document in the system to accommodate the various history and update files, it seems more reasonable to use a set of shared repositories for these things. Repositories can be created for pending updates, document archives, outgoing updates, and updates which require manual intervention to apply.

3.3.10. Users and Groups

Permissions within the system are based on user identity. Support for permissions and the Java Authentication and Authorization Service (JAAS), an extension to the Java security model, can provide user identities. Traditional Java security was based on the author of the code being executed, rather than the identity of the user running the code. JAAS extends this model to include both types of security. It also provides a standard way to access a variety of standard security mechanisms. The concepts of Principals and Groups, as well as credentials and access controls, are provided by JAAS. Principals represent a name associated with a user (the concept being that a single user can have multiple user names associated with them, and a Principal represents a single user name or its equivalent). Credentials are verification of a user's identity, and can include X.509 certificates and public keys, among others. Support for groups of users is also provided. These facilities can be used to authenticate and authorize users of the system. User credentials are delegated to mobile agents who perform remote actions on their owner's behalf. This delegation allows remote systems to make access decisions based on the user controlling an agent.

3.3.11. Administration

The administration component supports management of the directory service and general system configuration. It can provide access to monitoring information, such as agent status and activities. Historical information can also be provided, including statistics and logs. The administration interface can also be used to override default system behavior when necessary. Users must have administrative access rights to utilize this interface.

3.3.12. Logging

A centralized logging facility is an important feature. This facility should be accessible to local components and visiting agents through the local system manager. A standard logging package is useful to provide flexible and configurable logging. Individual components may also use the logging facilities independently for debugging purposes.

3.3.13. Directory Service

The directory service is used to share information about published documents, as well as to store access permission data for documents. Document information is stored based on document identifier with the various document properties contained in the docInfo class stored as attributes of the entry. Standard JNDI searching facilities may be useful to employ when searching for documents. Initially, the directory structure is a simple list of documents and their properties. Future investigation may explore more sophisticated structuring, such as an arrangement based on document author and author's group, or the document subject.

Access permissions for documents are also stored with the directory service. By default, only the owner of a document has the authority to allow other users to have access to the document. This default constraint can be changed when publishing a document or at a later time by specifying the additional access permissions to allow. Permissions include the type of access (read only or read and write permissions) as well as whether registration for updates is allowed. Only users given write and update registration permission are allowed to publish updates to a document. If permission is not given to register for updates, simple downloads of the published document are all that is allowed. The user will not receive updates, and cannot publish their own updates. It should be noted that users can modify documents locally regardless of whether they have write access, but they will not be allowed to push their updates to others. Group membership is evaluated when assessing a user's access permissions. If permissions allowing a user access to a document have not been explicitly granted, a request may be sent to the document owner to

explicitly request access. The document owner may then accept or reject the request. If the application is accepted, the requestor's access rights are updated in the directory service and the user is then free to download the document.

One drawback of using a directory service as a centralized repository for document and access control information is that it represents a single point of failure in the system. This issue is a common criticism of directory services. To alleviate such problems, many directory services support replication, in which a set of directory services duplicate each other's information and automatically propagate updates made to one directory service to the others. With the addition of automatic load balancing technologies, directory service replication can allow a cluster of different directory services to appear to be a single service. The members of the cluster can often be geographically dispersed to further improve reliability. It is expected that the use of replication and load balancing techniques could be used to alleviate potential problems with the use of a centralized directory service in this system.

3.4. Required Infrastructure

Development of this system is based on the Java 2 Platform (JDK 1.2). Java was chosen as the implementation language for several reasons. Many of the available packages for working with XML documents are written in Java. Standardized Java packages are available to support much of the required infrastructure of a distributed system and work with standard protocols and algorithms. Java is also one of the most common languages used for mobile agent development. A Java 2 run-time environment is required in order to use the prototype system. ObjectSpace Voyager 3.2 is used to provide mobile agent support. A number of XML tools from IBM are also used. The XML Parser for Java Version 2.0.15 is used for XML parsing support. Xeena 1.1 is used as the document editor. XMLTreeDiff is used for document difference calculations and merging updates. A working TCP/IP stack and network connectivity are required for remote operations.

3.5. Usage Scenarios

This section describes a series of common tasks and how they are implemented by the system. The intent is to provide a clear description of how these tasks are performed.

3.5.1. Create a new document

1. Initiated from document browser or user interface.
2. User is prompted for document name and directory if necessary.
3. The document management interface is invoked with the document name, location, and DTD to use.
4. The document management interface creates the new (empty) file and a docInfo object to represent the document.
5. The Xeena editor is then invoked on the specified file.
6. The user edits and saves the file using Xeena.
7. Once Xeena is closed, the docInfo object for the document is updated.
8. If the document is empty (nothing done to it by the user), then the file is deleted and the docInfo object is destroyed.

3.5.2. Edit a local document

1. Initiated from document browser.
2. The docInfo object for the document to be operated on is provided by the document browser when initiating the operation.
3. The document management interface is used to archive the current (existing) version and update the docInfo object.
4. Xeena is invoked on the specified file.

5. The user edits and saves the file using Xeena.
6. Once Xeena is closed, the docInfo object for the document is updated.

3.5.3. Publish a new document to the directory service

1. Initiated from document browser, which provides the docInfo object for the document.
2. A publisher agent is created and tasked with publishing the document.
3. The agent is dispatched to the directory service.
4. The document information is transferred from the docInfo object carried by the agent into the directory service if the user has permission to publish documents. If the user is not allowed to publish documents, the request is refused.
5. The agent returns to its host with the results of the operation.
6. The status information in the local docInfo object is updated as necessary.

3.5.4. Publish a document update

1. Obtain the docInfo object for the document that has been updated.
2. Store the updated version of the document in the archive.
3. Retrieve the docInfo object for the last version published.
4. Use XMLTreeDiff to calculate the changes from the last published version of the document to the new version.
5. Package the differences and other relevant information.
6. Create an update agent to update the directory service with information concerning the new update (modification date and time, new size, etc.) and distribute the update package to all participants.
7. Dispatch the update agent to the directory service to publish the new information. If the user does not have permission to publish updates, the request will be refused.
8. Obtain a list of participants in the collaboration who have registered to receive updates.

9. Once the update agent retrieves the list of participants from the directory service, it visits each one in turn to provide the document updates.
10. Wait for the agent to return.

3.5.5. Browse local documents

1. All appropriate docInfo objects are collected, either from a repository or dynamically.
2. The collection of objects is passed to the graphical interface.
3. The selected attributes of each docInfo object are added to a selection list. This initializes the interface.
4. Buttons are provided to:
   - Edit or view the selected document.
   - Create a new document.
   - Browse the locally available documents.
   - Obtain the detailed properties for the selected document.
   - Publish the selected document or distribute local changes to others in the collaboration.
   - Check for updates to the selected document.
   - Compare the selected document with another.
   - Manage access permissions for the selected document.

3.5.6. Browse remote documents

1. All appropriate docInfo objects are collected from the directory service.
2. The collection of objects is passed to the graphical interface.
3. The selected attributes of each docInfo object are added to a selection list. This initializes the interface.
4. Buttons are provided to:

- Register for collaboration on a document (i.e. register to receive and / or produce updates).
- Download the selected document.
- Obtain the detailed properties for the selected document.
- Manage access permissions for the selected document.

3.5.7. Request access to a document

1. Get the docInfo object for the document of interest.
2. Create a requestor agent.
3. Give the requestor agent the docInfo object and requested access permissions.
4. Dispatch the requestor agent to the directory service.
5. Once the agent arrives at the directory service and submits the request, the document's access permissions are consulted.
6. If the document permissions allow the requested access, the requestor is registered with the document's collaboration list as appropriate. The agent then returns to its host with the results.
7. If the document permissions do not allow the requested access, a request proxy agent may be created and dispatched from the directory service to the document owner to explicitly request the desired access.
8. When the request proxy agent arrives at the owner's system, the owner is notified of the request (either immediately or by suspending the agent in a holding area until the owner checks for requests).
9. The owner eventually approves or rejects the request.
10. The request proxy agent returns the response to the directory service.
11. The directory service updates the requestor's status as appropriate.
12. The response is returned to the requestor.

3.5.8. Receive a document update

1. Initiated by the arrival of an update agent. The update agent carries an update package which includes docInfo objects describing the "old" (previous) and "new" versions of the document being updated, as well as an XUL file describing the changes to be made.
2. Verify the credentials of the agent.
3. Check if the "old" version of the document update package is available locally.
4. If the current local version is the "old" version, check the local update policy.
   a) If the local update policy allows updates to be applied automatically, apply the update package.
   b) Otherwise, put the update in the holding area and optionally signal the user that a new update is available.
5. Else (the local version is not the "old" version),
   a) If the "old" document is available in the archive, apply the update to the archived version, yielding a full copy of the "new" document.
   b) Else request a full copy of the "new" document from the originator of the update.
   c) Calculate the differences between the "new" document and the local document.
   d) Either merge the two documents together to form a copy of the local document with the updates, or place the new set of updates in the holding directory (depending on local update policy).

3.5.9. Request and receive a new document

1. Identify the document to receive (obtain its docInfo object)
2. Request access to the specified document (see above).
3. Once access is approved, create a document download agent.
4. Task the agent with obtaining the document (along with any criteria, such as a version to request) and give it the URL of the document owner as its primary destination. Also provide

URLs for other collaborators on document if available in case document owner is not available.

5. Dispatch the requestor agent to get the document. The requestor will first try to contact the document owner. If the document owner is not available, traverse any alternate sources (other collaborators on the document) until the document is found or no options remain. If none remain, return failure.
6. Requestor returns with document attached on success.
7. Install the new document locally. (Register docInfo object locally, archive initial version, etc.) The following steps could be used to accomplish this task and reuse existing functions:
   a) Request that the document management interface create a new empty doc with the specified properties.
   b) Perform an update with the full document text.
8. Notify the user that the document is available.

3.5.10. Compare two local documents

1. Identify the two documents to compare by providing their docInfo objects.
2. Retrieve the filenames of the two documents from the docInfo objects.
3. Invoke XMLTreeDiff on the two files to generate a difference summary.
4. The difference summary can be used to update one of the documents, or used to prepare an update package for distribution.
5. An update package consists primarily of the two docInfo objects (used to identify the original and modified documents) and the difference summary (expressed in XUL by XMLTreeDiff).

3.5.11. Manage access permissions for a document

1. Receive a permission request proxy agent.
2. Assemble relevant information about the requestor.

3. Signal the user or place the request in a holding area until the user checks for new requests.
4. User can review available information and choose to accept or reject the request.
5. The result is returned to the directory service.

3.6. Security Comments

The basic responsibilities of the security infrastructure of this system are to:

1. Ensure that documents can only be access by authorized users.
2. Protect the integrity of documents and information.
3. Protect the confidentiality of documents and information.

The first point implies support for authentication of users and an access control mechanism to authorize access is in place. The second point implies that digital signatures be employed to detect changes made to documents. The third point generally involves employing encryption and file system permissions to ensure that only authorized parties can access the data.

The Java Authentication and Authorization Service (JAAS) can be used to provide facilities to authenticate and authorize users using standard cryptographic tools, such as certificates and public key cryptography. By combining the services of JAAS with a centralized policy server (located with the directory service), users can be identified and their access rights checked from any of the nodes in the system. The Java Cryptography Extensions (and possibly Java Secure Sockets Extension) can be used to provide encryption services, including message signatures, encryption of files, and encrypted communications channels.

The Java 2 Platform allows developers to create customized access permissions. Applications can require that their associated user possess the specified permissions for the operation to complete successfully. These facilities can be used to create new permissions governing:

- Read access to documents.
- Write access to documents.

- Ability to receive document updates.
- Ability to create and distribute document updates.
- Who has the ability to manage the membership in a collaborative effort.
- Who has the ability to change document access permissions.

These new permissions can be used to provide effective and efficient access control restrictions.

One of the important benefits of using some of the standard Java security interfaces is the ability to keep the security infrastructure flexible and easily adapt it to changing requirements. In addition, the security interfaces are generally designed to allow new security mechanisms and providers to be added to the system easily.

The document directory could be extended to include tracking user identities and associated credentials, such as certificates. Access rights possessed by the user for each document they are collaborating on could be tracked as part of the list of document participants. Agents would carry the certificate or other credentials of the user they represent as they carry out operations, providing accountability and a way to track their origins. Access decisions can be based on an agent's credentials, and thus the identity of the end user making the request. Fine-grained access control decisions can be made using customized security permissions, as described above.

Data carried by agents can be encrypted for the end recipient using the recipient's public key, retrieved from the agent's certificate. This would help to prevent the data from being viewed by others during transit. Further protection using IPSec- or SSL-based encrypted tunnels for agents to travel through could be provided. This would protect the agents themselves from tampering, in addition to the data. Although such a step would render encryption of the data payload somewhat redundant, it should be kept in mind that an agent may travel to a number of other hosts before returning to its point of origin. In many cases, it may be prudent to ensure that none of the intermediate hosts can access the agent's data, while still protecting the agent from third parties using encrypted tunnels for travel.

In some environments, it may be considered unnecessary or too computationally expensive to make such extensive use of encryption. In such situations, digital signatures can be used to verify that data has not been tampered with. Sensitive document portions could be encrypted using the facilities of the XML Security Suite if protection of the entire document was unnecessary. With the exception of IPSec, all of these security mechanisms are supported directly by various Java packages, as described in the previous chapter. IPSec is a lower-level facility that requires a supporting network protocol stack.

3.7. Implemented Features

The functionality presented is intended to describe a reasonably complete implementation of a collaborative document development system. Some of the features described are not implemented for the initial proof of concept system. Specifically, security features such as encryption, authentication, and access control do not form part of the initial development effort. Searching operations are not implemented either. The intention is to implement sufficient functionality to support the usage scenarios described earlier in this chapter (with the exception of security and possibly searching functionality). It is felt that this level of functionality should suffice to demonstrate the concepts being explored. Tools to implement the security and searching operations have been identified and described in this chapter, and should prove relatively simple to implement at a later time if desired.

Chapter 4 – Design

4.1. General Design – Techniques Applied

Three general design techniques were applied when implementing the prototype collaborative document environment. These design approaches are described in greater detail by design patterns texts, such as [Gamma et al., 1994]. The core Java libraries utilize a number of these patterns, and they were found to be beneficial when applied to this design.

4.1.1. Model / View / Controller

The Model / View / Controller (MVC) pattern is used to separate the formatting and display of data from the underlying data model, and the operations performed upon it. Java's Swing libraries support the use of MVC in many of its classes. Of particular note is the use of the JTable class. This class provides a way to display a table of data, and was used as the basis of the document browser component.

The JTable class can be used to display a simple data set by loading the data to be displayed directly. Of more benefit is the ability to specify a separate data model class which is responsible for maintaining the data set being displayed. The model class implements an interface used by the JTable class to obtain the data to display. This interface allows the model to selectively display only certain parts of the total data set.

In the case of the document browser, the data model stores a Vector of docInfo objects. By implementing the data model interface, this class can cause the table to display data directly from the underlying docInfo objects without requiring that a separate set of data be maintained for display. The model has complete control over what fields are provided for display. This control allows summary information from each docInfo object to be displayed. When the contents of the

model are updated, the graphical JTable object can automatically update itself to reflect the changes. The model controls the display of table headers and advanced capabilities such as displaying images to represent data can also be utilized.

To summarize, the JTable class provides a window for viewing a custom data structures. Fields within the data structure can be selectively shown or hidden. The underlying model controls how each field is displayed. The data model is not concerned with the display of data, except for implementing the basic model interface. Data can be changed in the model, and the changes will be propagated to the display. This class and design pattern in general can add a great deal of flexibility to the display of custom data, while greatly simplifying the mechanics of providing such a display.

4.1.2. Publish / Subscribe

The Publish / Subscribe model allows for flexible interactions among loosely coupled components. The basic concept is useful when a particular object must notify one or more other objects of particular events. Each subscriber must implement a common interface, which specifies callback methods used to receive the event or events of interest. Subscribers invoke a registration method in the publisher object. This method adds them to a list of event listeners, which are notified when a new event occurs by invoking the appropriate callback method. Different types of events can be signaled using either different callbacks or different arguments to a standard function. The same basic facilities and techniques are used for the different events, and the number of subscribers being supported does not affect how the events are communicated.

Publishers and subscribers are only loosely coupled through the use of a standard interface. Many different types of objects can be used as subscribers. The publisher object does not have to be aware of all of the different types of subscriber, as the only communication is through the

subscriber interface. Typically, the subscriber interface is relatively simple. Therefore, it is easy to add support for a particular publisher to an object. The original publisher does not have to be modified to support the new subscriber's underlying type.

This design pattern is useful for flexibly supporting varying numbers of clients who are interested in particular events, especially if the clients are of various otherwise unrelated types. Publish / Subscribe can be used to avoid cyclic references and tight coupling of objects even when there is only a single subscriber.

The document manager and data model classes utilize a publisher / subscriber relationship to communicate. The document manager serves as a publisher that announces events such as the creation of new documents or updates to existing ones. The data model subscribes to these events, updating its data structures to reflect the changes. The data model then notifies the JTable object of changes that need to be displayed, using a similar mechanism, but in the role of publisher. Events are reused by different parts of the document manager implementation. For example, the creation of a new document and the download of a remote document both cause a new document event to be generated. The same infrastructure in both the document manager and data model is employed to create and process the event. Edits, updates, and publishing of a document all generate a document change event, which is part of the same subscriber interface.

4.1.3. Interfaces and Wrappers

The third technique employed in the design of this system was the use of interfaces and wrappers to decouple the system from the implementation details of particular components. These techniques are also useful when it may be desirable to substitute different implementations of a particular solution during the life cycle of a program, or even at run time. An interface is a mechanism specifying the operations supported by its implementers. It is a "lightweight" mechanism, in that no particular implementation is associated with an operation. By interacting

with objects using only interfaces, any class implementing that particular interface may be used without requiring changes to the code utilizing the interface.

Wrappers are very similar to interfaces. Interfaces may be used in the implementation of wrappers. A wrapper allows a component or tool to be accessed using an alternate mechanism from the normal one. For example, a wrapper may provide a set of methods implemented using the facilities of the wrapper component. Typically, the processing performed by a wrapper is minimal. The basic intent is to translate calls to the wrapper into the appropriate underlying invocations on the underlying object. The wrapper acts a façade, providing an alternate interface to a component which can be used by the rest of the system. Wrappers are typically employed to adapt and reuse existing tools.

Several instances of the application of interfaces to add flexibility to this design have been described above. In addition, a combination of interfaces and wrappers are used to good effect in order to interface with third party tools. These tools are used to provide enhanced functionality to the rest of the system. It is anticipated that there may be a requirement to also support other tools in the future, either as replacements or as alternatives to the original selections. In order to shield the rest of the system from the details of which utility is being used, and to adapt existing tools to the purpose, a combination of standard interfaces and wrappers is employed.

Third party tools are employed for editing documents and for preparing or applying updates to documents. It is expected that alternate tools might be desired for these functions. It should be possible to reconfigure the system to use these new tools with minimal impact, and ideally at run time. In order to provide this functionality, simple interfaces were developed which specify how edit and update operations are to be invoked. All access to this functionality by the system is carried out using these interfaces. Wrappers are then employed to implement the interfaces and invoke the underlying tools in the appropriate manner. For example, the wrapper created for the Xeena editor implements the "edit" interface. The wrapper simply extracts the required

information from the method invocation, and runs Xeena with the appropriate parameters. Other editors could be substituted by simply providing and registering an appropriate implementation of the interface.

4.2. Basic Components and Infrastructure

4.2.1. Document Repositories

Document repositories are used to store and retrieve files as part of a collection. These files can be documents or update packages. Basic operations supported include storing, retrieving, removing, and listing documents. Multiple revisions of each file can be accommodated and tracked. Files in a document repository are identified by the document identifier of the document or update package they represent. Different revisions of a particular document are identified by their DOMHASH digest value.

The general concept of document repositories lends itself to several possible implementations, including both collections of simple files as well as databases. This design employs collections of files in a single directory for each repository instance. Files are stored using random filenames, in order to accommodate different revisions of the same file, or files with the same name but from different directories. Mappings are maintained to track which repository files are associated with each document. A database implementation could be accommodated with little impact on the rest of the system.

The document repository design consists of four main components. A base class, basicRepository, provides the main infrastructure for the repository implementation. Operations are performed on XML documents and docInfo objects. The basicRepository class relies on a helper class called repDirectory, which maintains the mappings of files within a repository directory to documents. Operations to provide flexible searching and retrieval capabilities are

implemented by these two classes. These capabilities include identifying all revisions of a particular document, as well as the oldest or most recent revision. Exact matches for a particular revision are also supported.

Document mappings managed by the repDirectory class are stored in a hash table. These mappings are persisted to disk using Java's object serialization capabilities. Each entry in the hash table represents the data for a particular document identifier. Entries consist of a vector of docInfo objects, each of which describes a specific revision of a document. The local filename of each docInfo object is set to its associated file in the repository, allowing efficient retrieval of specific document revisions by searching the repository mappings. Much of the searching capabilities of the document repositories are based around the facilities of the repDirectory class.

Two specializations of basicRepository are used to provide task-specific operations based on the facilities of their superclass. These are the classes historyRepository and updateRepository. Class historyRepository provides functions for tracking revisions of a document. Complete XML documents are stored as flat files. In many ways, the functionality provided is similar to that of a revision control system, although storing the complete text of each document is somewhat inefficient for that purpose. The main intent of this class is to store snapshots of each published revision of a particular document, for use in preparing and applying updates. It is also used for tracking local edit histories for documents, or unpublished document revisions. The use of this class for the latter purpose has not been fully exploited at this time. The operations supported by this class include storing documents, as well as retrieving the oldest, most recent, or specified revision of a particular document.

The sibling of historyRepository is updateRepository, which stores and manages update packages. This class is typically used to queue update packages waiting to be processed. Updates packages are broken into two files each. One file is a flat file containing the update document itself. The second file stores the update package header as serialized objects. The

header file is named the same as the update document file, but with an extension of ".hdr". Operations to retrieve the oldest, most recent, or all pending updates for a particular document are provided. New updates can be stored, and processed updates removed. Updates are typically processed one at a time, beginning with the oldest.

4.2.2. Document Directory

The document directory provides a centralized facility to track published documents. In addition to tracking documents, membership lists identifying all users who are collaborating on each document are maintained. An LDAP directory service could be used to provide much of the basic functionality required. It was decided to instead implement a customized document directory. There were several reasons for this decision. There would be a requirement for document-specific processing to interpret the data in a directory service. Although an LDAP directory could form part of the underlying implementation of a document directory, custom processing and communication with agents would still be required. The installation and configuration of an LDAP directory service was felt to be a rather time-consuming activity of little direct benefit to the design of this system. The data storage and tracking requirements for this system's directory were of a simple nature compared to that which LDAP directory services typically manage. The core functionality required to track documents exists within the repDirectory class. Reuse of the repDirectory class provided the core of a document-aware directory service, or document directory. Some basic capabilities such as agent processing and tracking of participants in a collaboration were added to complete the required functionality.

The basic purpose of the document directory is to track all published documents, updates to these documents, and those interested in each document. Operations required to fully implement these functions include: publishing a new document; listing users involved with a document; identifying the owner of a document; registering for a collaboration; and listing all documents.

Other options such as removing a user from the collaboration list for a document and retracting a previously published document are also desirable, but do not form part of the prototype design.

The document directory maintains document details and collaboration membership information using two separate data structures. The two types of data are logically separate, and existing classes could be reused with no modification to provide the document detail tracking capabilities. It was felt that the benefits of integrating all data in one structure were outweighed by the minimization of impact on existing components, as well as the advantages of reusing existing classes. Document owner and membership data are basically static, while document information can change relatively frequently as documents evolve. Separation of the two types of data helps to minimize duplication or complexity which would arise from merging all functionality into a single representation. It should be noted that this separation is not visible to clients of the document directory, as the directory class managing the two data stores presents a unified interface to them. The details of manipulating the two classes which manage the data, repDirectory and collabTable, are encapsulated by the main directory class, docDirectory.

Users who participate in a document collaboration are identified in the document directory using the URL of their computer. Integration of the Java Authentication and Authorization Service would allow for more accurate tracking of users in this case. The collabTable class is quite similar in basic design to repDirectory. A hash table is used to store membership data, with an entry in the table for each document. A vector of participant URLs is maintained for each document. Object serialization is used to maintain data in a persistent manner. The docDirectory class ensures that the serialized copies of both the repDirectory and collabTable objects are updated following any changes. Registration for a collaboration is implicitly carried out when a client requests a document for download or publishes one. The document owner is identified as the first user in the collaboration list. Agents used to push updates to participants update the document information stored in the directory service. When providing a list of published documents to clients, the most recent revision of each document is presented. The document directory object is exported to a

standard URL using Voyager in order to allow agents to travel to it and communicate locally. Remote communications are also possible.

4.2.3. Agent Infrastructure

Voyager provides very flexible and easy to use facilities for developing both distributed and agent-based systems using Java. In order to promote reuse of code and further simplify the development of different types of document-related agents, a common base class and several helper classes were designed for this project. Class docAgent serves as the agent base class. It provides scheduling and travel control logic which can be leveraged by subclasses, which are then responsible for application logic, rather than agent mechanics.

Mitsubishi's Concordia agent system uses the concept of agent "itineraries." [Walsh et al. 1997] Itineraries are essentially tables of destinations to which an agent is to travel, along with the callback function to invoke at each point. As Voyager, like most Java-based agent systems, is also based on callbacks, it was felt that the itinerary abstraction was a useful tool to use in developing this system. Two classes were designed which provide basic itinerary functionality. These classes, itinerary and itineraryElem, are used by the docAgent class and its subclasses to specify destinations and operations to be performed. The itinerary class contains a collection of itineraryElem objects. Each itineraryElem object specifies a destination as a URL, a callback function to invoke, whether the callback function requires any arguments, and an Object array containing any arguments to be passed to the callback.

The docAgent class contains an itinerary object specifying the tasks to be performed. Subclasses of docAgent typically populate the itinerary with tasks during construction. One of the interesting abilities of agents is their ability to control and change their activities in response to events that occur. To that end, agents may adapt their itinerary at run time. This ability is used in a number of situations by this system. For example, when distributing updates, an agent's initial task list is

simply to visit the document directory, and then return home. While at the document directory, the agent obtains a list of participating users. The agent then updates its itinerary to visit each of the users to deliver the update, prior to returning home. More advanced uses of this algorithm could include resequencing the order in which clients are visited, in order to exploit locality of client clusters and current network conditions. Advanced error handling and recovery techniques could also leverage this capability. These decisions can all be made without requiring the agent to contact its home system, optimizing use of bandwidth and facilitating weak (low bandwidth or poor quality) network connections for the host from which the agent originated.

The basic mechanics of processing itinerary elements and invoking agent travel methods from Voyager are encapsulated within the docAgent class. This encapsulation serves to simplify agent subclasses, as well as helping to isolate them from Voyager-specific processing. Once the itinerary for an agent is set, it can simply invoke the travel() method of docAgent to process the next itinerary element and travel to a new destination.

A specific design decision to note at this point is the use of URLs to identify destinations. Voyager supports the use of both string URLs as well as actual destination objects when specifying where to travel. The use of destination objects, obtained through naming service lookups, is often more convenient for processing. When URLs are used, agents must typically then invoke naming service lookups upon arrival in order to access the local object which it to be operated on. Object-based travel does suffer from some drawbacks, however. Specification of URLs as destinations is typically easier to understand and is more generic to other agent systems. In order to effectively use objects as destinations, casting from the generic Proxy class used by Voyager to the specific object type being manipulated must be performed. It was felt that this casting, required with any naming lookup, should be localized to the end object which is actually manipulating it. This casting requirement was felt to conflict with using a common base class for encapsulating travel and scheduling activities, or at least add more risk to the approach, as there was an increased chance of performing incorrect casts during the life cycle of the system. Although a name lookup

is required using either approach, by deferring the lookup until the agent had actually arrived at the end system, there were some performance and bandwidth conservation gains to be obtained. Finally, some additional flexibility is realized by using URLs instead of objects for specifying destinations, as the agent can select a different object to communicate with at the site, or change the order in which objects are used, without requiring changes to the underlying schedule of activities. Only the application logic requires adaptation in this case.

4.2.4. Core Data Elements

There are three basic data elements utilized in this system. All other data structures and operations are based on these three elements or their manipulation. The first of these elements is the docInfo class. This class represents the summary information for a particular document. This information includes the filename, size, author, title, unique identifier, DOMHASH digest value, modification time, and version tag information for a specific document. This class is used in most operations. It is used to identify actual documents, and thus serves as a convenient "key" to documents. A docInfo object identifies a specific version and copy of a document. Class docInfo is basically a passive object, in that it serves as a container for data, but performs very little processing on that data. Aside from accessor functions, the only other operations supported are a comparison function to determine whether another docInfo object describes the same document and a deep copy method to replace a docInfo object's contents with the data contained in another object.

Generation of docInfo objects, as well as the manipulation of docInfo object contents, is normally controlled entirely by the document management system component. Part of this responsibility includes ensuring that document identifiers are unique within the scope of the entire distributed system. The mechanism must be capable of functioning while disconnected from the network in order to allow users to create new documents while offline. The initial mechanism for constructing these identifiers consists of combining a random string with the local host identifier at time of

creation. Although not guaranteed to be unique in all cases, chances of a duplicate identifier are sufficiently remote for this prototype. Use of public key cryptography might provide a more robust mechanism for uniquely identifying documents, by combining a user's keys with time and possibly host information.

The second element is the Document class. This class is actually a standard Document Object Model (DOM) interface which defines an XML document. Specific XML parsers, such as the XML Parser for Java, provide implementations of this interface and its operations. Processing is normally done using the standard Document interface, however. The Document interface is used by this system to manipulate XML documents in memory. Documents are processed using the functionality of the XML Parser for Java. These operations include constructing docInfo objects to represent documents dynamically, as well as storing, retrieving, and updating documents.

The final core element of the three is the updatePkg class. Although it is in reality an encapsulation of docInfo and Document objects, this class is fundamental to much of the operation of the system and so is worth discussing here. The purpose of class updatePkg is to encapsulate the differences between two versions of a particular document. To accomplish this purpose, updatePkg contains two docInfo objects and an update Document. The two docInfo objects form the header of the package, identifying the new and previous versions of the document to which the update pertains. The update Document contains the differences between the two and can be used to update a copy of the old version with the new changes. All document updates are stored and communicated as updatePkg objects. The update tool is responsible for generating these objects, as well as for processing them.

4.3. Specific Components

The local manager is represented by class localMgr. This class is responsible for the basic operation of the system. It allocates and initializes the other key system components such as the

user interface, document manager, and agent manager. It is responsible for general housekeeping and maintenance operations. The initial system design requires a fairly minimal implementation of localMgr that processes and applies basic configuration parameters, initializes the rest of the system, and establishes relationships among the other components. When the application exits, localMgr performs basic cleanup tasks such as shutting down the Voyager environment.

Class MainMenu provides the initial user interface into the system. It consists of a simple window with buttons to activate other components of the system. Operations are supported to create new documents, access the document browser, invoke administrative options, or exit the system. The administrative options are intended for a future update to the system. MainMenu is the launching point for the rest of the user interface, which is primarily provided by the document browser. Initialization of the document browser, including providing a reference to the document manager object to interact with, is the responsibility of class MainMenu.

The docBrowser class implements the document browser component. The core of this class is a table of document summary information, obtained from a collection of docInfo objects. Menus and buttons to perform various operations on documents are provided. These operations are typically carried out through the browser's associated document manager object. The document browser is tightly coupled to the document management object. It is also loosely coupled to the document data model, which interacts with the JTable object encapsulated within the document browser.

The document browser operates in two modes. Local mode is used for operations on documents available locally on the system. These operations, although based on local documents, may involve the use of agents to perform remote functions involving these documents, such as pushing local updates to others or publishing new documents. The other supported option is remote mode. Remote mode is used for operations on documents not necessarily available on the local system. Typically, local mode is used to display the documents available in a directory

on the local system, while remote mode displays the collection of documents accessible via the document directory. Operations which do not require an existing document to operate on, or which can be applied to both types of document, are available in both modes. Other operations are enabled or disabled based on the current state. Operations supported in both modes include creating new documents, viewing detailed document properties, accessing a list of remote documents from the document directory, and browsing a new local directory. Downloading a selected document is the only additional operation available in remote view. Local mode does not allow documents to be downloaded, but instead allows the user to edit documents, publish new documents, push updates, request any updates not received, and apply updates.

The data model object is implemented by class docInfoTableModel. This class extends the basic implementation of table interface methods provided by class AbstractTableModel. The data model serves as the main run time data collection of the system. The core of docInfoTableModel consists of a vector of docInfo objects. The class implements the document event listener interface to receive and process document events from the document manager. Interactions with the JTable object consist of providing table headers and individual cell contents to the table on request. A vector of docInfo objects is indexed into in order to provide the specific field requested by the table in order to identify cell contents. The data model publishes events to notify the JTable display of changes that occur. This pushing of events has the conceptual effect of forwarding event notifications from the document manager to the user interface.

The document manager is responsible for controlling all document processing. This responsibility includes managing the repositories, handling and application of updates, implementing operations invoked from the document browser, and interfacing with the agent manager. It is the core of the document system. It is used by virtually all of the other components, or uses them to perform its own operations. Class docMgmt encapsulates two instances of historyRepository, and one of updateRepository. The first historyRepository object is used to track all revisions to documents, regardless of whether they are published or not. The second historyRepository stores snapshots

of each document as it is published or as updates are applied to it. It maintains a history of each revision of every document circulated a among the collaboration group. The updateRepository is used as queue for incoming update packages awaiting processing. The document manager is the only interface between the main document processing system and the agent manager. This interface is a bi-directional relationship. The document manager utilizes the services of the agent manager to dispatch agents to perform remote tasks. The agent manager invokes methods in the document manager to return the results of operations, as well as to service requests from visiting agents. The docMgmt class is responsible for creating and manipulating docInfo objects, as well as update packages. The edit and update tools are accessed through the document manager.

Services to create and edit documents are provided by the document manager. These services automatically update the edit history repository and notify components such as the data model of document events when invoked. Unique document identifiers are allocated to documents during creation. The document manager will update any document following editing to ensure that the system generated document identifier is not changed by the end user, as it is the basis for all document identification. Tools are invoked as required. The document manager can scan a specified directory for existing documents, based on file extension. Documents located are summarized with new docInfo objects, which are communicated to the data model, which adds them to the documents displayed. Lists of documents available from the directory service are assembled in a similar manner, except that the docInfo objects are obtained through the services of the agent manager, which dispatches an agent to the document directory to assemble the docInfo collection. Other operations implemented by docMgmt include: publishing documents to the document directory; creating, distributing, receiving, and applying document updates; downloading specified documents; and providing documents to visiting agents requesting them for download.

The agent manager controls all agent operations. It is the only part of the core system which directly works with agents. The agentMgr class is responsible for creating, dispatching, receiving,

and providing services to agents. These operations include both agents which local and visiting agents. In order to carry out an operation, the agent manager creates a new instance of the appropriate agent type, providing it with information about its task during construction. The agent is then dispatched and returns later with the results of its operations. Class agentMgr serves as the external interface into the local system from the network. This class functions as an intermediary between the document manager and the agents themselves. It provides a task-oriented interface to the document manager and translates these requests into specific operations using agents. Since agentMgr is the only part of the system using agents and uses a relatively high-level interface, alternate mechanisms for providing remote operations could easily be substituted in this class without impacting the rest of the system. The agent manager also accesses the document manager's methods to report the results of earlier requests in an asynchronous manner and service requests from visiting agents, such as receiving update packages or providing requested documents for download.

It is important to note that the interaction between the document and agent managers is asynchronous in nature, due to the use of callbacks for agent operation. Since remote agent operations may be time consuming, it is best not to block while waiting for the results of an operation. Even if blocking for results was desired, it is somewhat complicated to implement, since agent invocations are essentially one-way methods which return immediately, without waiting for the action to complete. Agents require a separate method which can be invoked to return their results. Returning results to the next point in the calling function is not an option. Use of an intermediary object that blocks until the returning agent invokes a notification method to resume processing could simulate this behavior if required. This intermediary would add a fair amount of complexity to the model, and does not fit well with the typical design style used in both agent and user interface programming. The basic paradigm used is that the document manager may invoke an operation using the agent manager, and then resume normal processing. At some later time, the agent manager invokes a separate method in the document manager to cause the results to be processed. Operations such as publishing a document are implemented by the

document manager as two separate methods. The first method initiates the publishing action. The agent manager calls the second when the publishing agent returns. This second method then performs actions such as updating the published status of the document in question and notifying the rest of the system using the document event interface.

The agent manager supports two types of operation. These two types are often the mirror images of each other. Outgoing operations are those which are initiated by agentMgr, and include distributing updates, requesting updates, requesting document lists, publishing documents, and initiating document downloads. Incoming operations are often the returning half of a previously initiated operation. Other incoming operations are involved with servicing the requests of visiting agents. Incoming services include receiving document updates, requested downloads, document lists, update requests, and status results from returning agents.

Specializations of the base docAgent class are used for each type of operation. Each subclass is designed to carry out a particular type of task, and utilizes the facilities of docAgent to manage the mechanics of travel. Download agents, implemented as class dlAgent, are provided with a docInfo object and file name during creation. The docInfo object identifies a document to be obtained by the agent, while the file name is the name of the file the user wants the document stored in on return. The agent travels to the document directory to locate the owner of the document. Once the owner's address is obtained, the agent travels to the owner's system and requests the document. It then returns the requested data to its original agent manager.

Publish agents are responsible for publishing new documents to the directory. Implemented by class pubAgent, they are given a docInfo object describing the document on creation. They convey this information to the directory, where it is stored in the list of available documents. The agent's home URL is stored as the document owner's location. Once its task is completed, the agent returns the results of the operation to its home agent manager.

Class updtAgent is used to distribute update packages. Update agents are initially dispatched to the document directory. Upon arrival, they obtain a list of all participants in the document collaboration, and update the published information for the document. They then visit each participant's agent manager in turn, delivering the new update package. Once all participants have been visited, the agent returns home to notify its agent manager of whether it was successful and how many participants received the update.

Update request agents are used to check if there are any updates which have not been received by the local system. This functionality is intended largely for use when the local system has been disconnected from the network for a period of time and has just reconnected. Class upreqAgent travels to the document directory with a copy of the docInfo object for the last published version of the specified document available on the system. It then determines if the current document information available from the document directory has been modified more recently than the local document. If the information is more recent, the agent obtains the document owner's URL and requests an update package from the document owner's agent manager. By providing the docInfo object for the most current local copy, the document owner's system can provide a specific update package to patch the local document to the new revision directly. If the user requesting the updates is the document owner, an alternate member of the collaboration should be queried for the update package instead.

The last type of agent is the directory agent, or dirAgent. This class is responsible for traveling to the document directory and requesting a list of all available documents. It then returns this collection of docInfo objects to its home agent manager.

4.4. Design Summary

A number of useful design patterns and techniques have been employed to create this system. Application of techniques to prevent tight coupling of components would benefit the current

interaction between the document and agent managers in particular, as they are currently quite tightly coupled.

One design decision that was made to simplify the development of the prototype was the integration of the user interface with the agent manager and other system components. Although effective and useful for demonstrating the collaboration paradigm being explored, a more robust solution would be to separate the user interface from the core infrastructure. This separation would allow the core infrastructure to execute as a daemon process which could run in the background on the host system at all times. A daemon process could help to keep documents up to date and help to ensure that visiting agents could obtain documents when desired, rather than being unable to communicate with the local system. With the interface part of the same program, this cannot be supported unless the user wishes to have the interface always running. If the interface was separated, a client-server model could be employed within the local system for communication between the two parts of the system. The current communication model between the components would require some updating to accommodate this. As mentioned above, this separation is not a requirement to demonstrate and explore the technology, but may be a usability and robustness issue if putting this system into more general use.

A class diagram illustrating the relationships among the core system classes is presented as Figure 3, below. Some relationships have been simplified to avoid cluttering the diagram unnecessarily. For example, most classes use the docInfo class. Only those classes having significant interactions with class docInfo show an association with it, such as those responsible for creation or display of docInfo objects.

Figure 3 – Core System Classes

Figure 4 shows the classes associated with the document directory. Classes forming part of the document directory are shown, as well as the agent classes which interact with the directory.

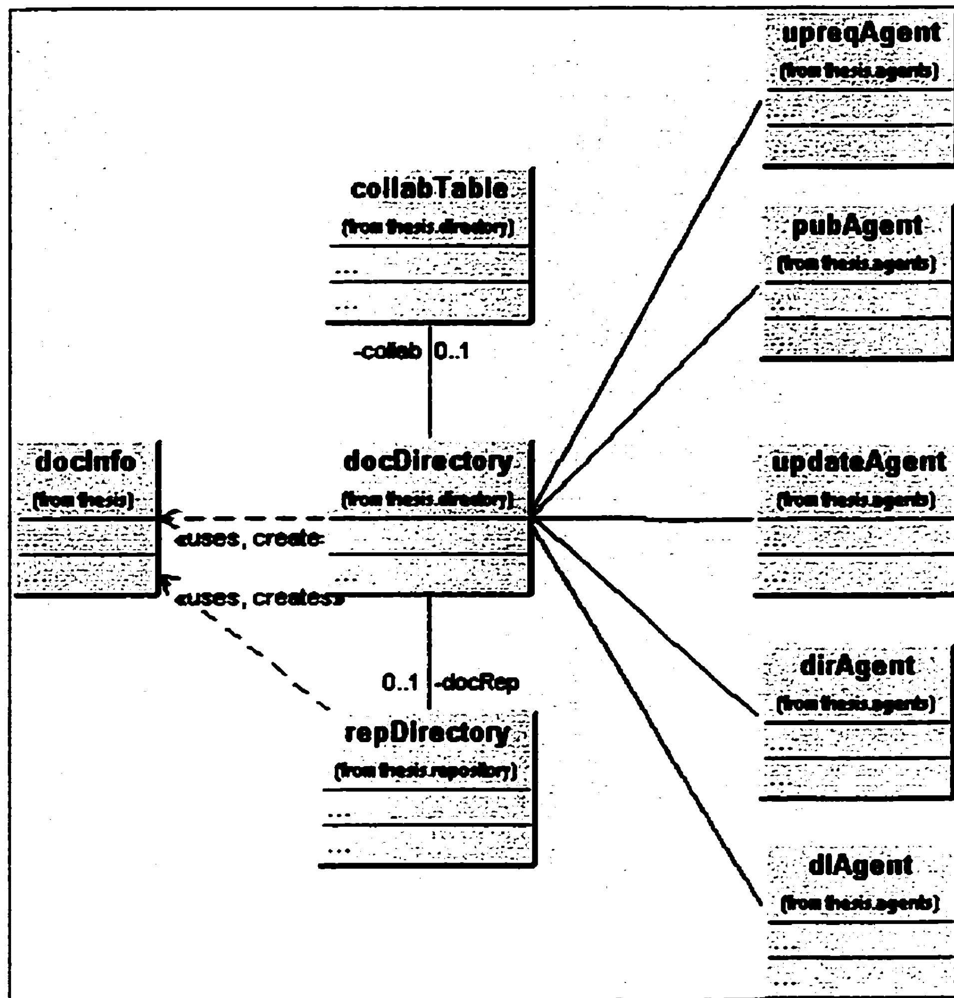

Figure 4 – Document Directory Classes

Chapter 5 - Implementation

5.1. General Implementation Considerations

An iterative development methodology was employed in the development of the collaborative system, following initial requirements definition and high-level design phases. The spiral life cycle model, as described in [Ghezzi et al. 1991], and the star model [Preece et al. 1994] are examples of this development philosophy. In both of these approaches, the basic principle is that requirements definition, design, prototyping, implementation, and testing are activities that recur throughout the life cycle of a system. These approaches contrast with the traditional waterfall model, in which development is conducted in a monolithic manner. Each phase in the waterfall model is conducted in a linear sequence and is not repeated. Although reasonable for relatively simple systems which can be grasped in their entirety from the outset, the waterfall approach is not well suited for high-risk, poorly defined, or experimental systems.

Iterative models attempt to minimize risk and develop systems in a phased approach. Although requirements, design, prototyping, development, and testing phases are generally followed in logical sequence, the scope of each phase is restricted to a subset of the total problem. By iterating through the basic development process, the system can be refined a step at a time. Since the scope of each iteration is restricted to a subset of the overall problem (often a specific high-risk issue which must be dealt with), complexity is minimized. In many cases, a specific feature or component may be developed during an iteration. Subsequent iterations build on the previous steps. This approach allows a complete system to be developed gradually, with the ability to produce a functional, though limited, system at the completion of any particular iteration. It is not necessary to complete the entire development cycle before a working system can be shown, or testing and verification can begin. Generally, the output of any particular iteration can be tested as a piece of the whole system, with new pieces added as iterations progress. This approach helps to avoid situations where eighty percent of the development is complete, but

there is nothing functional to show. Development can be phased to focus on core requirements, exploration of alternatives, or refinement and resolution of particular issues.

One of the drawbacks with iterative development is that if there is not a clear vision of the desired result, at least in high-level terms, it is easy to become sidetracked and allow the system to evolve in an uncontrolled manner. For this reason, the methodology followed after the initial research and investigation was completed began with a phase devoted to defining the purpose of this system. Requirements for a useful and functional prototype were identified next. The system architecture and high-level design was produced. Once these activities were complete, a more iterative approach was followed. This approach put a guiding structure in place, while allowing flexibility in refining the detailed design and exploring issues and alternatives. As there were a number of different technologies and tools to investigate and integrate, the iterative model allowed each system component to be examined in phases. Often the initial iteration for each component might only produce a simple mock-up, with operations defined as simple skeletons. The general interactions could then be examined and explored without an extensive investment of development time. Rapid prototyping could be employed to quickly refine the model to be implemented. Successive iterations then resulted in the refinement of the initial skeletons. This approach allowed a great deal of flexibility to adapt as a greater understanding of the system requirements, issues, and general infrastructure emerged.

Several issues arose during and after the development of the prototype related to design tradeoffs and areas in which the system design could be improved. Optimization of the system was not a specific objective of this investigation. However, revisions to the design to optimize the reading and writing of documents would be quite beneficial. Parsing XML documents is a relatively expensive task. Many of the current operations are based on docInfo objects. In some cases, the full document must be read and parsed in order to complete the operation. As other operations may already have been performed on the same document, analyzing these linkages and passing in-memory documents to operations where possible could enhance the efficiency of

the system. This analysis is not as trivial as it may appear, as there may not be a clear connection between two operations at first examination. An in-memory caching mechanism might be employed, but memory consumption for large documents would have to be taken into account. Another approach might be to develop variations of a number of common operations that accept a parsed document when available, or read in the document themselves when necessary.

An effort was made to keep processing functionality separate from the user interface, and specifically the document browser. It often becomes very difficult to manage modifications and enhancements to user interfaces if the interface code is intermixed with processing functionality. This intent was successful to a large extent, by relying on functionality in the document manager. Some additional benefits could be obtained by further separating the user interface from processing. There is some basic preprocessing performed by the document browser. By employing an intermediary object to perform this basic preprocessing, the code within the document browser module itself could be reduced to simple user interface operations and calls to the intermediary. This technique is similar to the mechanism used by the JTable class and its underlying data model.

Design patterns and other techniques were employed during the design to minimize coupling between some of the major components. The maintainability of the document and agent manager components could be improved by reducing the current amount of direct interaction between them. This separation may prove difficult, but one technique that might be employed would be to register callbacks as part of each operation. The current design requires both a service invocation methods and a result returning method for most actions. By registering the result method to use as part of the invocation, the number of concrete method invocations between the two components could be reduced. Separation of the document manager into smaller subcomponents would also be helpful, as it is currently somewhat monolithic. Application of the Singleton design pattern [Gamma et al. 1994] might be also appropriate for both the document and agent managers.

5.2. Implementation Process

An important side benefit of this investigation was the opportunity to refine and employ design and implementation skills using advanced technologies. Details concerning the development process used to implement systems are often difficult to obtain. In an effort to provide a concrete example of such development, a discussion of the process by which the collaborative document environment was created following the initial design follows. Issues and difficulties encountered are discussed in greater detail in the next sections, including changes made to the original system plans. There were mistakes made during this process. It is not intended that this should be a perfect example of systems development, but rather a practical discussion that might be helpful for others.

5.2.1. Infrastructure and User Interface

The first step to implementing the system was to develop greater familiarity with the Java language and tools being employed. Some basic classes such as docInfo were created, as well as some simple screens to form the basis of the user interface and document browser. This approach provided an initial iteration that allowed for exploration of Java's Swing user interface classes, with which the author had been unfamiliar. Concentrating on such simple tasks was a very practical starting point for the development effort, and provided some core functionality upon which to base further development. The JTable class was identified during this phase. The next iteration built on the basic Graphical User Interface (GUI) by implementing the docInfoTableModel class and using it to populate the document browser with some trivial docInfo objects. These two initial iterations provided the basic user interface and the ability to display docInfo objects, as well as an opportunity to become familiar with the development environment.

A skeleton for the document manager was designed and implemented next. The user interface was then updated to access the skeleton methods, which typically output a simple message showing that they had been invoked. This mock-up allowed the basic interactions and required operations to be identified and explored. It also served as a useful framework for tracking further development requirements. Functions could be filled in and rapidly tested, without requiring the complete development of the component. The intent of these first few phases was to focus on developing the core infrastructure required by the overall system. More advanced functionality, such as XML processing and agents, was saved for the later phases. This approach was useful in limiting the scope of each iteration, so that there was a limited set of new technologies to learn at each step. It also simplified the addition and testing of the XML and agent components, as they could quickly be plugged into the waiting infrastructure following initial prototyping.

The final step in implementing the core infrastructure was to develop the document repository classes. Although these classes are heavily involved with XML Document objects, the core operations were defined, including the repDirectory class. These classes could then be tested using docInfo objects and simulating Documents with simple strings.

5.2.2. XML Support

Once the basic infrastructure was in place, support for using the XML Parser for Java was added. The initial goal was to add the ability to read and write simple XML documents. This facility was added quite easily, especially with the very clear descriptions of using the XML Parser for Java found in [Maruyama et al. 1999]. With the ability to read and write XML documents, the repository implementation could be completed.

The initial intent of the basicRepository class was to provide the basic mechanics of storing arbitrary files in a repository. It was responsible for naming and tracking files, regardless of the file content. It was intended that the historyRepository would use these basic capabilities to store

Documents, while updateRepository would store update packages. Each of the subclasses would format their data in buffers, which were passed off to basicRepository for storage. Although this technique worked for handling simple Document objects, update packages proved more difficult. Serializing the update headers and then writing out the contents of the XML Document as text to the same buffer proved difficult. This model also seemed wasteful, as data was first written to a buffer in memory, then passed off to the base class to be written to disk. The work of writing documents was being duplicated in the two specializations of basicRepository.

These issues led to two decisions. The first was to store the update package headers separately from the update document. The same base filename was used, but the file to which the header objects were serialized used a different extension from the update document text. By separating the update document, it could be processed in the same way that the history repository stored basic documents. The second decision was to update basicRepository class to continue to provide generic naming and lookup operations, but store and retrieve XML Documents rather than simple buffers. Most of the functionality required by historyRepository was then provided within its parent class, and it could be used simply to offer a task-based wrapper around the basic storage facilities. Class updateRepository was also simplified, as it only needed to add processing of update package headers to its inherited capabilities.

Interfaces for the edit and update tools were designed next. Implementation of an edit wrapper for the Xeena XML editor proved simple, though inefficient. Xeena was selected as it seemed to be a stable, well-supported graphical environment for creating and editing XML documents based on user-specified DTDs. The intent of including Xeena was to provide an easy to use document editor that provided validation of document formats and guidance in manipulating document elements. One factor that was not given adequate consideration during the initial selection process was the fact that programmatic access to Xeena is not available, nor is source code. It is thus not possible to simply run Xeena as part of the calling program. Instead, Java's Runtime class must be used to exec() Xeena as a separate process. This separation means that a

separate Java Virtual Machine must be executed. The result is very high processor and memory requirements, and significant performance degradation. Correspondence with Xeena's authors indicates that programmatic access is planned for an upcoming release.

Access to the Xeena Application Programming Interface (API) might also have allowed greater customization of Xeena's behavior. This access would be of use in restricting options such as the "Save As..." feature, which might allow a user to save the document in a location that the collaborative environment would not be aware of, and thus unable to track immediately. An interesting idiosyncrasy of Xeena and the XML Parser for Java is that Xeena will allow element attributes with default values to remain null. The XML Parser silently inserts these attributes when processing documents, even if the document was created with Xeena and did not include the attributes originally. This difference caused some confusion when copies of the same document started showing different sizes in the document browser. Upon examination, copies processed by the parser contained the default attributes from the DTD, while those only manipulated using Xeena did not.

Interfacing with Xeena was inefficient, but did not pose any implementation problems. The update tool selected, XMLTreeDiff from IBM, did. XMLTreeDiff does support programmatic access and seems to be ideal for creating and applying "patches," or descriptions of differences between documents. It uses an XML-based language for describing differences, and so patches are expressed as XML documents themselves. This format provides a reasonably compact description of document differences requiring no new infrastructure beyond the XMLTreeDiff libraries themselves. Since the tool works at the XML level, textual differences between documents that generate equivalent XML structures can be accounted for, which is one of the main shortcomings of standard text-based comparison tools.

The complication that arose when attempting to integrate XMLTreeDiff was in many ways the opposite of what happened with Xeena. Xeena had been tested, but its usage had not been

examined as carefully as it should have. XMLTreeDiff turned out to be incompatible with the other tools being used. It was thought that the tool had been tested before selection, but if so, it was tested using older versions of the other components of the collaborative environment which were no longer options to use. The details of the conditions encountered and tested are provided in the next section. The end result was that there was no alternative tool available. In order to preserve the semantics of processing and applying updates, and provide some means of sharing changes to documents, a simple implementation of the update tool interface was developed. Since an interface was used, the rest of the system performed operations using only the interface. By simply using a different construction command, the new implementation could be deployed. This simplified update class, called fakeDiff, simply replaces the existing document with the new version. The processing model remains the same, with the drawback that any local changes made since the last published version of the document are lost when the update is applied. There is also a loss of bandwidth efficiency since entire documents are transmitted rather than just differentials. This approach does serve to simulate the required processing. When a compatible version of XMLTreeDiff or a similar tool is released, it should prove trivial to deploy using the standard interface and should not disrupt the rest of the system.

Creation of docInfo objects to represent documents had used default values and very simple calculations for most data elements. With the basic ability to create, parse, save, and edit XML documents in place, the infrastructure was available to begin populating docInfo objects with properties extracted directly from the XML documents using the parser. The basic requirement was to read some of the basic document fields such as author, title, document identifier, DTD, and version. Most of these properties were readily obtained. Retrieving the DTD turned out to be complicated. Although the name of the DTD can be obtained using standard DOM operations, support within the DOM specification for obtaining the name of the external DTD being used [XML4J]. As the external DTD must be specified to Xeena at startup, it became necessary to hard-code the DTD being used within the docInfo generation code.

The current implementation of dynamic docInfo generation is somewhat proprietary to the DTD used during development. Fields such as <author>, <documentID>, <versionTag> and <title> are required. This requirement is not a major restriction, however, as these are largely common concepts in most documents, and the exact fields being examined can be updated relatively easily in the code. A more flexible approach might involve maintaining mappings of supported DTDs and the appropriate fields to consult in each. Another improvement that could be made would be to employ SAX for generating docInfo objects. The current method uses DOM parsing, which reads and parses the entire document before the data can be accessed. SAX allows processing of document elements as they are encountered within the data. As there are only a small number of fields near the start of the document which need to be accessed, SAX might prove more efficient. This approach was not taken in the prototype in order to avoid introducing another technology to integrate, especially as SAX appears more complicated to utilize than DOM. A second improvement would be the automatic creation of an initial document skeleton by the system, incorporating the system generated document identifier and user-selected DTD to employ. This automation would provide control over the document identifier and DTD. A final option to consider would be the use of processing instructions or comments, rather than full elements, for important system information such as the document identifier.

Once the ability to generate docInfo objects for documents dynamically was implemented, scanning of directories for documents to be displayed by the document browser was added. The basic capabilities of the Java File class, combined with a filename filter which uses the extension ".xdoc" to identify supported XML documents, was used to provide this capability. The document browser uses this feature to select documents to display when operating in local mode. The last part of dynamic docInfo operations required was the calculation of DOMHASH digest values. The XML Security Suite is used to implement this functionality. DOMHASH values can be used to identify documents in an essentially unique manner, and are similar to standard message digests except for the fact that they are calculated on processed DOM trees rather than plain text.

Document identifiers are used to identify a document, while DOMHASH digests are used by the system to uniquely identify an exact document revision.

Completion of the core update processing implementation was the last piece of functionality required for the basic infrastructure prior to the implementation of mobile agent support. The basic logic had been mapped out previously, while developing the basic document manager functionality. Completion of this functionality involved applying the update interface to generate and apply patches as required, as well as utilizing the various repositories. For testing, receipt of new updates was simulated by causing the update push function store the update packages it generated in the update repository as incoming packages. By changing the document being used between generating the update and applying it, testing could be performed.

5.2.3. Mobile Agent Implementation

The final set of iterations was devoted to implementing mobile agent operations. This phase began with an initial exploration of the Voyager example programs. Once the correct operation of these programs was confirmed and the basic mechanisms employed understood, a design cycle began. This design phase focused on the detailed design of the agent manager. The specifics of how it would interface with the document manager were determined, and required operations were identified. Specific agent requirements could then be derived from the supported operations, as well as the basic behavior and general processing performed by each type of agent. In order to effectively implement agent operations, the document directory service was required. The data to be stored and managed was determined. The basic design was determined, including specific operations and methods.

At this point, some reorganization of the main routine and system startup processing took place. Initially, the main routine was part of the MainMenu class. MainMenu simply initialized the document browser and manager, then displayed itself. The local manager was developed

instead. The functionality of class localMgr is more limited than the initial architecture had intended. Its responsibilities include configuring the basic system parameters, initializing the Voyager service, and creating the core system components such as the user interface, document manager, and agent manager. It exports the agent manager on a URL accessible to other nodes in the system as part of its system initialization responsibilities. Once the system is running, the local manager waits for the user interface to close and then terminates the Voyager service.

In order to provide infrastructure and support for further agent development, the document agent base class and supporting itinerary representations were designed and developed next. The publish agent was the first actual agent developed. Its task of registering a new docInfo object with the document directory and returning the results was the simplest of the agent operations. In addition, published documents were required for all of the other agent operations. The initial implementation was nothing more than a shell which used the docAgent facilities and output some debug messages to signal where specific processing steps would occur. This shell was used to test the docAgent infrastructure and general agent operations.

Implementation of the document directory followed the initial agent experiments. The core operations were implemented, with some refinement occurring later in development. Further refinement of the features of this service could include more sophisticated up-to-date checks when update requests are received, as well more sophisticated collaboration tracking.

Once the document directory was available, the functionality of the publish agent could be completed and tested against the directory. A simple test program was employed to exploit this functionality. Once verified, the agent manager skeleton was implemented. The required operations were simulated, with only the document publishing functions implemented initially. The publish agent and agent manager were then integrated with the main system for testing. Although the initial agent framework had appeared to function properly, further testing with the completed agent and document directory revealed that only the initial hop of the agent was executing

properly. A discussion of this situation, the reasons it was not noticed immediately, and general comments concerning Voyager are presented later in the chapter.

Development of the directory agent was the logical extension of the functionality provided by the publish agent. Its responsibilities were also simple, and provided a means of confirming that the publish agent and document directory were functioning correctly. A local list of the available documents was also required in order to select a document for download. It should be noted that development of each agent type was performed as a complete iteration of the development cycle. The specific functionality and functions required for the document manager, agent manager, document directory, and agent were determined first. Development of the required methods, or refinement of existing ones, was carried out. End-to-end testing of the completed functionality was then possible. The order in which agents were developed was chosen both to minimize the additional complexity at each step, develop required infrastructure for the next iteration, and to allow this phased testing approach.

The download agent added the ability to obtain documents which had been first published by the publish agent and then listed by the directory agent. This agent was the first requiring the addition of support in the agent manager for visiting agents. The agent manager was responsible for providing copies of documents to visiting agents trying to perform downloads. A second new capability introduced by this agent was that of changing a task itinerary while at a remote site. Previous agents had only been required to set up a simple itinerary to travel to the document directory and back, which was done at initialization. The download agent was required to determine where to obtain the document from once it had contacted the directory, then travel to the remote site and return the document. The update agent was developed using similar facilities as the download agent. Distribution of updates also required visiting agent support in the agent manager and itinerary updates while at remote sites. In this case, the itinerary changes tend to be more extensive in order to visit all of the participants in the collaboration, rather than just the document owner. This agent provides some of the core functionality of the collaborative system.

The final piece of the system developed was the update request agent. It required more advanced processing and decision-making capabilities than the other agents, as well as all of the previous functionality. The update request agent is responsible for determining whether a local document is up to date. If not, it must locate a participant in the collaboration who can provide an update package to resynchronize the local document. The initial implementation of these capabilities is somewhat simplistic, as described in the design chapter.

5.3. Issues Encountered

Some of the difficulties and issues that arose during development have been mentioned briefly above. The intent is to discuss these issues in more depth. Topics discussed include development tools used, Java pass by reference semantics for method parameters, and XMLTreeDiff. Experiences with ObjectSpace Voyager are presented in a separate section.

5.3.1. Development Tools

Sun's Forte Java development environment was selected as the main development tool to be used. It is a full-featured commercial development environment, formerly known as NetBeans, which was acquired by Sun and is now available for use under Sun's community license arrangement. Forte includes support for editing, compiling, executing, debugging, and browsing code. It also features a GUI editor that can be used to create Advanced Windowing Toolkit (AWT) and Swing forms. Templates for common types of source files are provided. The source editor offers advanced features such as automatic completion of method names, based on a list of alternatives valid for the object type being used. Output from program execution is captured and can be reviewed later. Forte's feature and functional completeness are very similar to environments such as Microsoft's Visual Studio. Forte is written entirely in Java, and so the development environment itself is portable to essentially any platform supporting Java.

Unfortunately, Forte requires a huge amount of overhead. Memory and processor requirements are very steep. Although performance is typically poor at the start of a session, the usability of the system rapidly worsens as editing, compilation and test execution cycle are conducted. This problem appears to be a memory usage issue, probably related to Java's garbage collection. Forte executes a separate Java Virtual Machine when executing user code, which appears to be a large part of the overhead problem. Capture of output during execution is particularly slow. It is often necessary to wait for significant periods of time just for output to appear. Running multiple tools within environment, such as the GUI editor components as well as the basic source editor, also degrades performance. Unfortunately, closing these tools does not seem to recover any resources. It is thus necessary to periodically restart the Forte environment, and sometimes even the computer, just to bring the environment back to a usable level of responsiveness.

Since a new beta candidate of Forte was being used, an attempt was made to go back to the older, stable release of the environment. Unfortunately, this release proved to not be an option. One of the benefits for which Forte was originally selected was its GUI editor. It was felt that this would help minimize the complexity of learning Swing, allowing more focus on development of core functionality. Not surprisingly, in order to apply the GUI editor effectively, the mechanics of Swing still had to be learned. This learning requirement was expected to a certain extent. What was not anticipated was that once the basics of Swing were understood, there was still another learning curve in order to discover how to use the GUI editor to achieve the desired behavior. With the fairly plentiful sample code available for Swing, it might have been easier to develop the user interface manually. Forte embeds internal information in special comments within the GUI code. It does not allow direct editing of any interface code. A separate form data file is maintained which appears to be used as the basis of code generation. Manual changes to the interface code made outside the environment are simply replaced with code generated from the form data file the next time a change is made within Forte. The generated code is standard Java, and does not require any special facilities from Forte to run. However, if any changes to the user interface are

made outside Forte, either the changes must be abandoned or Forte cannot be used for any further interface development. Unfortunately, the form data file used by the new version of Forte is not backward compatible with the previous release. This lack of compatibility made going back to the older version much less attractive, especially as there was no guarantee that there would be much improvement in the performance of the environment.

Despite its shortcomings, Forte was used for much of the development cycle. Once the basic user interface was developed and it became necessary to start using external tools such as Voyager's igen interface generation utility, it was decided to develop batch files to automate these tools as well as compilation of the source code. With batch files to assist with compilation and interface generation, and basic GUI development complete, it was decided to abandon Forte and use a different editor. The poor performance of Forte had been reaching the point of being a significant hindrance to rapid development and testing.

An effort was made to identify an alternate, more efficient development tool. One promising environment was Kawa. Kawa is a Java development environment which, although not as advanced as Forte, is much more efficient. Unfortunately, only an evaluation copy of Kawa was available. It had been installed earlier in the development cycle but not used. Within a day of beginning to use it, the evaluation period expired and the tool disabled itself. Attempts to install a newer version were unsuccessful, and it was felt that the time to contact Kawa's producer could not be afforded. The FreeJava tool was briefly evaluated. Although adequate, it seemed to have some minor quirks. The final tool selected was Ultra-Edit. Although not a Java-specific tool, Ultra-Edit supports Java syntax highlighting and is a very easy to use and flexible editing environment. It was also one of the most efficient tools used. Combined with the batch compilation scripts and basic DOS windows for testing and compilation, an effective and rapid development environment was created.

5.3.2. Java Method Argument Semantics

One issue that was not well understood at the beginning of development was the exact semantics of Java's pass-by-reference model for method arguments. Method parameters in Java are references to objects, rather than objects themselves. This approach allows objects passed as arguments to be updated directly during the execution of the method. The alternative to this model is to use pass-by-value, in which a copy of the object parameter is passed. The object can then be updated locally, but the updates do not affect the original object and are lost on return from the function. What was not recognized originally is that, although the object being referenced within a method can be updated, the reference itself is passed by value. Thus, changes to the reference are lost when the method making the change completes.

The impact of this behavior is that a new object cannot be assigned to the reference from within a method. Although there are ways around this in Java, they are rather awkward. In C or C++, the same situation exists when passing pointers as arguments to functions. The variable pointed at can be updated within a function. If a change to the pointer itself is desired, a pointer to the pointer, or double pointer, must be employed. New objects can be created and returned from the function, but this only allows a single new reference to be provided. In some of the methods developed, it was desirable to return two objects. Specifically, the repository methods typically returned both a Document and the docInfo object describing that specific revision of the Document.

In order to accommodate this requirement without developing a new data structure to store both objects on return, a result docInfo object must be allocated and passed into the method for population with the appropriate data. The object is filled in by copying the contents of the docInfo object in the repository into the method argument. The original intent was to simply set a reference to the repository's copy. This technique works, but requires a bit more care when invoking the method and is somewhat wasteful of memory. The biggest drawback is that it took some time to diagnose the exact cause of the problem and correct it.

5.3.3. XMLTreeDiff

XMLTreeDiff is frequently referenced in Internet discussion groups as the tool to use when comparison of different versions of XML documents, or related activities, is required. It appears to be a very powerful and useful tool. In the context of this application, it is ideally suited for the generation and application of XML document updates. It allows the creation of patch files and their application as separate activities. Patch files can be distributed separately from the base document, which is very useful for sharing updates to published documents. The update format itself appears to be reasonably compact. The problem with using XMLTreeDiff for this development effort is that it is not compatible with the other tools employed.

Incompatibility of a core tool with the rest of the system would appear to be a glaring oversight from a project's investigation phase. Although this accusation is likely valid, there are a number of different factors that come into play in this case. There were a wide variety of tools examined during the research phase. Documentation was reviewed and brief testing conducted with most of them. Many of the core Java and XML tools have undergone significant changes recently, and there is a confusing array of options and versions to choose from in some cases. The XML specifications appear to be evolving at a fairly rapid rate. Sun released JDK 1.2 shortly before this project was begun. Three different major releases of the XML Parser for Java were available while this system was being investigated and developed, not including minor revisions and patches. Some of these parser changes were reflections of updates to XML standards, while others arose from the introduction of new features.

The documentation describing XMLTreeDiff's features and usage was examined quite carefully during the investigation phase and again during design. It was not until an attempt was made to actually integrate and use the tool that the incompatibilities were discovered. At this point, an in-depth review of the documentation and discussion group archives at IBM's AlphaWorks web site

was conducted. It became apparent from the discussion group that a number of other users had encountered the same problems. Unpacking and updating the archive corrected one or two problems, such as the incorrect case of certain filenames in the XMLTreeDiff JAR file. Unfortunately, only the binary class files were available. Many of the tools available from the AlphaWorks web site are distributed in binary class form only, XMLTreeDiff included. An attempt to contact the author of the tool for an update or access to the source was unsuccessful. These difficulties serve as a good example of why the open source model is very useful for infrastructure and basic tools, as most of the issues could probably be corrected fairly easily. Searches for alternate tools were basically unsuccessful. Most suggestions found for this type of utility pointed back to XMLTreeDiff, or papers describing the theory behind it.

One alternative identified during the research period was another tool from the AlphaWorks site, called the Compare and Merge Tool for XML. This utility was an interactive tool, however, which could not be run without user involvement. It also does not generate patch files, but instead merges two documents which are provided as input. Although inferior for the purposes of this project, the Compare and Merge Tool was reconsidered. Unfortunately, it suffers from the same incompatibilities that XMLTreeDiff does. There are indications that this tool may be under more active development than XMLTreeDiff and that an updated release may be compatible with the newer tools in use currently, as well as supporting features such as automated execution. These indications have not been updated for some time, however, and so the current development state of both tools is questionable.

While trying to determine the problem with XMLTreeDiff, two different versions of the package were obtained. The newer release, which was being used, only advertises support for the XML Parser for Java version 1.x and JDK 1.1.5 or higher. Many of the tools specify support for a particular release such as these, or higher, which was the assumption until closer examination of the requirements. In this case, release 2 of the parser is specifically not supported. The older release was retrieved from the CDROM accompanying [Maruyama et al. 1999]. This version

specified that it required the XML Parser for Java version 1.1.4, JDK 1.1.5 or higher, and Swing 0.7. It did not indicate the specific lack of support for release 2 of the XML Parser for Java. As updated versions of most of the tools used were obtained before beginning the actual development phase of this thesis, it is possible that it was this older release of XMLTreeDiff which was assessed. The other factor that might have contributed was that an upgrade from JDK 1.1.6 to JDK 1.2.2 occurred on the development system at some point near the initial research phase. The older JDK might have still been installed when XMLTreeDiff was first examined.

The Swing classes appear to be used in some capacity by this tool, but Swing was moved from the com.sun class hierarchy to javax.swing as part of the JDK 1.2 release. Other issues may also be present with the new Java changes. The major problem is the rapid evolution of the XML Parser for Java. Significant changes occurred between releases 1 and 2 of this tool, and the API's do not appear compatible. Although the newer version of XMLTreeDiff states that version 2 of the XML Parser for Java is not supported, it does appear to rely on new features of the parser not present in the at least some of the release 1 family of parsers. Perhaps it is a late revision of release 1 of the parser which is required, although this requirement is not indicated by the documentation.

Although reverting to JDK 1.1 might have allowed the use of XMLTreeDiff, it was felt that this change might prevent the use of a number of other tools, and possibly require significant changes to the existing implementation. Other combinations of tools were tested to try to find a combination that would allow all of the components to work together. Although release 1 of the XML Parser for Java was no longer available from the AlphaWorks web site when this problem was discovered, [Maruyama et al. 1999] also included a copy of release 1.1.9 of this tool. XML for Java 2.0.15 was the revision initially being used. It had been selected as a reasonably current version that would be more stable than the brand new third release. The following table illustrates the combinations of tools which were tested using the sample XMLTreeDiff programs and the outcome of each test:

	XML Parser for Java 1.1.9	XML Parser for Java 2.0.15
Older XMLTreeDiff	Swing error related to renaming of package in JDK 1.2	Swing error related to renaming of package in JDK 1.2
New XMLTreeDiff	Error message that method Child.getUserData() was not supported. This function appears to be a new feature of the XML Parser for Java, version 2.	Error message concerning invalid XML file, null pointer in sample.java

Table 1 – XMLTreeDiff Testing

It is very disappointing that support for XMLTreeDiff appears to have been dropped. It appears to be a very useful and widely respected tool which fills an important gap in the suite of XML tools readily available at the present time. Hopefully support will be renewed as the XML specifications solidify and the XML Parser for Java stabilizes. In the meantime, a simple substitution mechanism was employed to simulate the processing of updates using the update wrapper. This implementation should be simple to replace at a later date if XMLTreeDiff is updated or a suitable replacement found.

5.4. ObjectSpace Voyager – Concepts and Comments

ObjectSpace Voyager is a useful and powerful tool. Application of this tool is intended to be extremely simple and intuitive. Although the basics are quite simple, there are a several specific concepts which are not necessarily intuitive.

5.4.1. General Issues

Several general issues were encountered while using Voyager. Some of these are comments or criticisms, while others could be considered bugs. The main issue identified was the fact that the

Voyager utilities do not appear to deal well with mixed case CLASSPATH environment variables. The java and javac tools are insensitive to the case of CLASSPATH specifications under Windows 98. Mixed case, uppercase, and lowercase are all accepted as equivalent, which is how the underlying operating system behaves. This behavior is consistent regardless of whether long filenames or abbreviated eight character filenames are used. Voyager seems to require lower case paths and filenames. Although not a big issue on the surface, this detail was somewhat difficult to identify, as all of the other tools used did not behave in this way. No mention of this requirement was observed in the documentation. Voyager also requires that the Java Runtime Environment (JRE) JAR file be explicitly specified in the CLASSPATH variable. This is also inconsistent with other tools. A second issue noticed is that error reporting is very poor when callback methods cannot be found or executed in some cases. Situations in which an exception, or at least an error message, would be expected appeared to fail silently, even with detailed logging enabled.

Other concerns noted were related to documentation and logging facilities. Early versions of Voyager placed a significant amount of emphasis on its support for mobile agents. As time has passed and new versions of Voyager have been released, the initial hype over autonomous agent technologies has faded somewhat. Very little emphasis is now placed on support for agents, although some very useful facilities for their development are provided. This lack of emphasis leads to the next point, which is that the documentation for Voyager in general is quite poor. Agent support is documented even less than most. Although the manual for Voyager is quite thick, much of this bulk is empty space and formatting. This style is useful in many manuals for simplifying digestion of the material. In this case, there is little material to absorb, beyond general or simplistic examples. Very little in-depth discussion, explanation, background, or advanced examples are available. The agent documentation is literally a few pages, with a single example in the appendix. The example agent, although a useful starting point, does not demonstrate any advanced functionality, or even show how to contact an existing object within a new environment. It simply creates a remote object, interacts remotely with the object, and then moves to the object

to perform local interactions. Any other operations must be inferred from other parts of the manual. A more complete discussion of how to apply Voyager and use its facilities would be very useful, as well as some more advanced examples.

A final note is in regard to the Voyager's logging. Voyager uses some logging facilities internally. By default, no logging output is displayed to the user. This behavior can be changed to output exceptions or verbose details. Although useful on the surface, the Voyager appears to use exceptions on a regular basis when operating. These do not appear to be errors, but rather a way of determining what actions to take. The result of this technique is that enabling output of exceptions generates large amounts of stack trace output which appear to be meaningless for identifying sources of error. Enabling verbose output, which is a superset of exception output, adds only limited additional data, which is obscured by all of the exception messages and stack traces. A mechanism to disable stack traces would make the output much more useful. Another helpful change would be to expose the logging functions for application access. The logging facilities seem to be a capability only accessible internally to Voyager. Applications must then employ their own separate logging mechanism.

5.4.2. Special Requirements

There are several specific requirements that must be followed when using Voyager. These are largely due to its advanced usage of Java facilities such as serialization and reflection. Although mentioned at different points in the Voyager documentation, they were found to be easily overlooked, and so are highlighted here.

Voyager makes extensive use of factory methods and similar facilities. Many of these operate based on the name of the class used, which is expressed as a string. It is important to make sure that these class names are fully qualified; that is, they must be expressed with their full package prefix. This requirement includes both user-created classes and standard Java classes. A second

requirement is that primitive arguments to messages and callback functions must be wrapped in their Object equivalents. For example, an int variable must be passed to these functions as an Integer object. Interfaces and proxies are used to access most objects which are managed by Voyager. This requirement applies particularly to mobile agents and exported local services.

It is important that proxies are used consistently. In one case during development, the agent manager object was being exported directly, rather than exporting a proxy to it. When returning agents tried to access the agent manager, they correctly attempted to obtain a proxy to the exported object, based on the agent manager's interface. Since what had been exported was not the interface, but was instead the object itself, a "Class Not Found" exception was generated. The final requirement is related to the moveTo() method used for object and agent mobility. The documentation indicates that only exception-handling code should follow calls to moveTo(). What is not clear from the brief mention of this point is that any non-exception code which might happen to follow will still be executed normally, which may lead to some significant confusion, especially if this basic behavior is not clearly understood.

5.4.3. Mobile Agent Concepts

The points discussed earlier in this section highlight some of the difficulties and minor issues which were encountered while developing mobile agents with Voyager. The moveTo() method, which is Voyager's function for causing a mobile object or agent to change location, has some unusual semantics which can cause confusion. The moveTo() function should be thought of as a one-way method invocation. It is not a synchronous call causing a move to occur, a callback to be processed, and then returns to the next statement. It is more of a scheduling mechanism used to signal that a move should take place at the completion of the current method. There is no way to return from a callback to the exact point from which the moveTo() was issued, which results in the requirement discussed in the design chapter for the results of operations to be returned using a separate chain of method invocations from the sequence which initiated the move. If there is

additional code following the section in which the moveTo() is invoked, it is important that an explicit return statement be issued following the move instruction and any related exception handling. As mentioned above, any such statements would be executed without an explicit return, which is not always the desired effect.

The initial intent of the docAgent base class was to provide a task scheduling mechanism which handled the core agent movement requirements of specific agent types. The original concept was that an agent would populate an itinerary, then invoke a travelToAll() method. The intention of travelToAll() was to loop through the entries in the itinerary, traveling to each site and invoking the appropriate callback at each. Although seemingly a valid approach, the semantics of moveTo() do not permit this approach, since the move does not occur immediately, perform its task, and return control to the caller.

This issue was not completely understood during the initial development. This problem was made more difficult to detect due to the fact that there was debugging code executed after the moveTo(), as part of the processing loop, which seemed to indicate that all of the moves were happening. The lack of verbose error reporting from Voyager further complicated the situation. The observable behavior of the system made it appear that an agent was being dispatched to the initial remote destination, where it performed its task. The agent was then scheduled to return home to report its results. Messages showing the result return method's execution were displayed in the home location, although the debug output generated during an actual move did not appear. The results returned were always just the default results set in the constructor (normally false, although changing the constructor to set the result to true resulted in success being reported). The actual outcome at the remote site was never reported correctly, even when the result value was checked immediately before returning from the remote site. The actual behavior was discovered after a significant amount of debugging, research, testing, and code modification.

It would now appear that the successive moveTo() calls in the processing loop likely resulting in multiple moves being scheduled. Separate copies of the agent were likely created and dispatched in parallel. One agent went to perform its remote task, while the other copy immediately invoked the results method with the default results. Since the results were returned to the Voyager instance from which the agent originated, no travel was involved, and so the debug output from the travel functions was not observed. The remote agent never returned since it did not explicitly invoke a moveTo() operation, and its itinerary had been marked as complete due to the processing loop.

To correct the problem, several changes were made. The travelToAll() method was dropped. Each callback function within an application agent was responsible for invoking the travel() method of docAgent, which retrieves a single itinerary element and invokes moveTo() once. In other words, the travel() method had to be called at the end of each callback function. This requirement included early termination of a callback, such as when an error occurred. Rather than just issuing a return, the agent must call travel() to go to its next destination, or update its itinerary to handle the error. In either case, simply issuing a return would essentially terminate the agent, or at least become unrecoverable and inactive. Calls to travel(), especially when processing errors in the middle of a callback, must be followed by an explicit return statement to ensure that further processing is discontinued.

5.5. Modifications to Original Plans

The following list summarizes the changes in functionality or behavior that arose during the implementation phase. This list does not include features discussed in chapter two which were not intended for implementation.

- Automatic processing of new document updates was intended as an option. This option is difficult to apply utilizing the current implementation. Although automatic updates could be applied to the most recent update in the edit repository, there is no reverse

association maintained in the repository to identify the actual file which the user normally uses. Although tracking of both the repository and external files could be implemented, the intent of the system is to provide an environment that does not restrict the user to one particular system. Users should be free to create and edit documents using external programs, and then utilize the collaborative environment for sharing and distribution. Storage of user filenames with the repository entries would not guarantee that the user would not modify the file externally or even move the document entirely. This could cause errors and even undesired behavior. The approach of manually initiating the application of updates is considered to be a more flexible technique which facilitates the use of alternate tools for basic document processing.

- Replacement of existing documents with complete copies of updated documents was used to simulate update processing, rather than using XMLTreeDiff to create and apply differentials.
- Support for an application-wide logging facility based on the log4j logging package for Java was originally planned. Time constraints and other priorities prevented these facilities from being implemented.
- Some hard-coding was used, the main instance of which is the DTD employed. The Xeena wrapper hard-codes the basic command to be run. All other hard-coding is for configuration parameters managed by the localMgr class. It is intended that these parameters be moved to a separate configuration file in the future.
- The travelToAll() method which was intended to cause the docAgent class to initiate and control all agent travel and processing had to be abandoned. Explicit travel() calls are now required at the end of each callback method in application agents.

5.6. Required Infrastructure

Table 2 summarizes the software packages used, including their installation locations. The values of environment variables such as PATH and CLASSPATH are also presented as used during development.

Package	Location / Value
Java 2 Platform, Version 1.2.2	C:\jdk1.2.2
XML Parser for Java, Version 2.0.15	C:\IBM\xml4j_2_0_15
Xeena 1.1	C:\IBM\Xeena
XML Security Suite for Java	C:\IBM\xss4j
ObjectSpace Voyager 3.2	C:\voyager
PATH	%PATH%;c:\jdk12~1.2\bin;c:\voyager\bin;
CLASSPATH	c:\jdk12~1.2\jre\lib\rt.jar;c:\jdk12~1.2\lib;c:\ibm\xss4j\xss4j.jar;c:\voyager\lib\voyager.jar;c:\voyager\lib\jgl310~1.jar;C:\tools\log4j-v0.4;c:\ibm\xml4j_~1\xml4j.jar;.;d:\thesis\thesis;c:\voyager
JAVA_HOME	C:\jdk12~1.2

Table 2 – Packages and Environment Variables

The contents of the simple Document Type Definition used for this project are shown in Figure 5:

```
<!ELEMENT doc (title, author, documentID, versionTag, toc?, chapter+,
index?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT documentID (#PCDATA)>
<!ELEMENT versionTag (#PCDATA)>
<!ELEMENT toc (#PCDATA)>
<!ELEMENT chapter (section+)>
<!ATTLIST chapter
```

```
      title CDATA "chapter title">
<!ELEMENT section (#PCDATA)>
<!ATTLIST section
      title CDATA "section title">
<!ELEMENT index (#PCDATA)>
```

Figure 5 – Document DTD

Chapter 6 – Summary and Conclusions

6.1. Results

The implementation of the prototype collaborative document development environment has been successful. A functioning system has been created which supports the creation and editing of XML documents. Local documents can be listed with summary information and operated upon. Documents can be published to a central document directory. Available documents can be listed and downloaded. Document updates can be pushed and requested, with some limitations on the functionality of the actual update mechanism. The application demonstrates that the implementation of collaborative environments using XML and mobile agent technologies is feasible. Experience with the technologies used and this paradigm in general has been obtained and presented in this paper. A framework has been produced which should prove useful for further development, experimentation, and analysis of the paradigm.

Refinement of the prototype environment would allow the system to be used in a number of different contexts. The system could support development and review of documents from geographically dispersed locations. This might involve staff from different offices of a large corporation assembling and reviewing a proposal, research teams from different institutions cooperating in an investigation, or individuals collaborating on an open source project, for example. Distribution of course materials and updates could be performed by the system in an educational setting. This could be controlled to only allow the professor to push materials and updates to students. Alternately, students might be allowed to share their class notes and comments on materials with other individual students, the class, their study group, or just the professor.

Figures 6 and 7 show the main user interface and document browser, respectively:

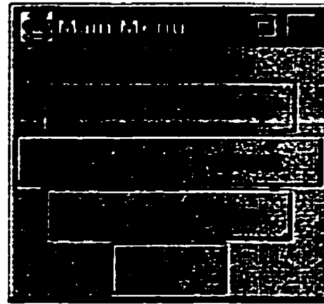

Figure 6 – Main User Interface

title1	author1	361	Mon Mar 13 23:47:00 ...
title2	author2	288	Fri Mar 03 23:11:48 A...
title3	author3	353	Sat Mar 04 14:54:50 A...
title4	auth4	350	Sun Mar 05 12:31:30 ...
doc5	auth5	283	Mon Mar 13 11:02:58 ...
title1	author1	361	Mon Mar 13 23:18:48 ...
title1	author1	397	Mon Mar 13 20:52:04 ...

Figure 7 – Document Browser

6.2. Application Improvements

There are a number of improvements that could be made to the existing prototype. These include both new features and enhancements to existing ones. Techniques based on thin client technology might be useful in separating the user interface from the core processing engine, in order to facilitate operation of the manager components as background processes. Such a separation would also serve to insulate the main system from changes to the user interface.

Implementation of a full directory service, perhaps in combination with the existing document directory, could be used to provide advanced searching capabilities based on author, title, or other document properties. Another improvement would be to integrate a functional update tool. This utility could be an updated version of one of the tools reviewed, an alternate tool, or a basic tool developed for this system. A tool that identified new nodes added to a document and produced update packages based on these new nodes would be a reasonable application to develop.

The repository tools developed offer significant potential for offering advanced application features. Current use of the repositories is limited to internal tracking of documents, with no visibility to users. Although this transparency simplifies the system, several features could be expanded to allow users to benefit from this functionality. The document browser currently supports local and remote modes for browsing documents. A third mode could be added to support browsing the repositories. For example, this mode could be used to review previous versions or recover from inadvertent changes. Much of the functionality of the repositories is comparable to that of revision control systems used in code development. By adding facilities to compare revisions of documents and apply symbolic tags, repositories could be more actively used as document revision tracking systems. Repositories could also be extended to serve as request queues between the document and agent managers. By monitoring a pair of queues for incoming and outgoing requests, the two components could be isolated from each other. The existing repositories handle most of the data required. The addition of an operation identified would be the main addition needed to implement this functionality.

6.3. Analysis and Measurement

The focus of this thesis has been on the initial investigation and prototyping of a document sharing paradigm based on the use of XML and mobile agents. One important aspect of further investigation in this area is the objective and subjective analysis of this paradigm, as well as

extensions to the basic model developed. Development of facilities to monitor and track operations would be essential for this type of investigation. Timing of operations should be performed, and statistics describing how the system is used should be collected. Useful parameters to track include: time required for repository and update operations; number of agents active at different times and locations; frequency and types of operations performed; documents typically requested; use of network bandwidth; and disk space utilization. Behavior of the system under different types of load would be useful to analyze. Load could consist of large numbers of users, documents, agents, or other parameters. Careful tracking of timing and resource parameters during such experiments can help to identify bottlenecks in the system and general performance degradation profiles.

Specific tests that would provide useful information include:

- Developing a functionally equivalent environment using more traditional distributed system technologies such as CORBA or RMI. Remote operations using the same data sets (same document, document lists, number of users, system and network load) could be conducted and timed using the two systems. The time to perform each operation, amount of load placed on the system, and network bandwidth utilized by each could be measured. Specific operations to analyze include downloading documents and document lists, distributing document updates, and publishing new documents. These measures would provide objective statistics to compare the performance and requirements of the two approaches.
- A network packet sniffer could be employed to determine the bandwidth and number of network connections required to perform various remote operations. The number of connections required to complete an operation would give an indication of how the system would likely behave in unreliable network environments. Applications requiring large numbers of connections, or interactions, to complete an operation would probably be more heavily impacted by unreliable network links than software not requiring as many exchanges. Measurement of bandwidth consumption is useful to formulate a profile on

the network impact of using different sizes of documents, as well as the overhead involving in using agents.

- Timing and profiling could be used to measure the execution time and general performance characteristics of operations. The creation and application of different types and sizes of document updates should be analyzed in this way. The overhead of parsing and extracting header information from documents could be examined, as well as the time to read, parse, and store different files of different sizes. These metrics would identify which operations are the most computationally intensive. Performance bottlenecks would be revealed and could be reviewed for potential optimizations.
- General usage tracking statistics to monitor which operations were performed most frequently would be useful. These statistics could also be used to identify patterns in sequences of operations. Counters could be employed to track method invocations, possibly with "session" keys that identify related operations carried out to complete higher-level tasks. Analysis of this data would provide information which could be used to make common operations easier to use. Related operations could be grouped or even replaced with convenience functions to simplify interactions. The data could also provide a guide indicating which operations are executed most frequently, and thus should be studied for possible optimization.

There are a number of variations on the basic model implemented that would be interesting to analyze. Different methods of formatting documents for local storage (such as storing a history of differentials, rather than complete documents) could be explored. The cost of converting from one format to another could be examined. A comparison of the costs of transferring raw XML files versus their logical representations (possibly including alternate logical representations) could be made. The cost of creating the logical representation of the document from a raw data file, or of creating the raw data file from a logical structure as part of the operation, could also be considered. The analysis could be applied to a variety of document sizes to identify trends in the cost of the mechanism. Such metrics could be used to identify when there might be benefits to

using one approach instead of another, and optimization of the various operations performed by applying these results based on run time criteria.

Future work may investigate the implications of shifting more of the computational burden from stationary services such as the document manager onto the mobile agent. This would be useful to compare the overhead of the additional code size of the agent in such situations against the benefits of simplifying the stationary services and increased mobile agent flexibility. The performance impact due to the additional class loading activity during initial transfer of the agent's supporting classes would also be interesting to analyze, especially compared to the overhead of subsequent visits by the same type of agent. One could speculate that the first visit by an agent from a particular client host would have a relatively high amount of overhead for class loading, while subsequent agents using the same classes and originating from the same host would require significantly less overhead. Measuring the actual difference might prove useful, particularly if large libraries or low bandwidth networks were involved.

6.4. Further Investigations

Throughout this effort, a number of areas for further research and development were identified. Discussion of these areas has been presented as a separate section because either the scope of the required development effort is beyond basic enhancements to the existing system or which investigate changes to the basic architecture of the model. One area to be pursued is the design and implementation of a security architecture for the system. Although general requirements for this infrastructure have been presented in the first chapters of this document, there remains a significant amount of investigation and implementation. Abstractions representing users are required. Access controls based on user identities, hosts, and credentials should be integrated into the system. The application of cryptographic protections for agents and the data they carry should be investigated. Similar protections for local data could also be pursued. A flexible mechanism for specifying and applying policies defining how requests and data from specified

users should be processed could be developed. Such a mechanism could be used to specify that data from unknown users be placed in a holding area for manual review before processing, or that updates from a research associate be immediately merged with the local document.

One of the advantages of agents is their flexibility and ability to adapt to changing conditions. Exploration of the application of these properties to develop a system resilient to network changes, failures, and congestion could be very interesting. Adaptation of travel to minimize usage of bandwidth and optimize travel patterns could also be studied. The elimination of a central document server in favor of pure peer-to-peer collaboration could be investigated. A parallel environment developed using RMI or more traditional distributed technologies could be used for both subjective and objective comparisons of the two techniques. Subjective analysis could be performed to analyze factors such as ease of use, ease of comprehension, simplicity of design, and so on could be done. Objective parameters such as the overhead of various operations could also be compared.

Finally, variations on the collaborative model discussed here which exploit some of the other benefits of agents could be explored. One example of such a variation would be an environment in which a large document storage environment was created. The focus of the collaborative environment could shift to the collection and tracking of documents at the central sites. Simple facilities could be provided to agents which traveled to the document site to perform complex searches and other operations locally. Much of this work could be built on the existing infrastructure of this system.

6.5. Conclusions

This experiment has successfully explored a new paradigm for collaborative systems development. Autonomous mobile agents have been used to develop a collaborative system that is well suited for use by intermittently connected clients, or environments where bandwidth

utilization and flexibility are important. The flexibility afforded to the system through the use of agents cannot be achieved using standard technologies such as Remote Procedure Calls without a significant increase in the frequency of interactions between clients and the rest of the system. The approach employed in this project may require additional processing resources over a more traditional model, which is an issue which could be explored using a parallel system and detailed analysis, as described earlier in this chapter. Agent technologies introduce new security issues not present in standard distributed systems. These issues are much less of a problem in environments having a defined user group, which is where a collaborative environment would typically be deployed.

The design goals stated at the outset included development of a framework from which the collaborative paradigm could be further explored; minimization of bandwidth consumption and remote interactions; existing tools and infrastructure should be used where possible; and proprietary interfaces and tools should be avoided. All of these goals have been successfully met. A functional prototype has been implemented with consideration given for future expansion throughout the investigation and development process. Client operations require only a single agent transmission to initiate an operation, followed by receipt of the returning agent to retrieve results, regardless of the number of different systems utilized in completing the task. Existing tools which utilize published standards, such as the XML Parser for Java, have been utilized as much as possible. Care has been taken to ensure that new tools and infrastructure can be easily added to the architecture. An initial assessment of the collaborative system indicates that it should prove very beneficial. The particular environment and model of usage being considered may dictate whether this system is appropriate or not.

Preparation of this thesis has been a valuable learning experience. It has provided an opportunity to explore and utilize a number of interesting new technologies. Practical and theoretical knowledge of Java, XML, and mobile agent has been gained, including numerous related tools. Research skills have been improved. The implications of decisions made during preliminary

investigations as well as system design have been experienced. Insight into the value and purpose of several useful design patterns and methodologies has been gained. In addition to the benefits experienced by the author, it is hoped that this investigation will be useful in facilitating further exploration of this topic.

Bibliography

[DOMHASH 1999] Maruyama, Hiroshi, Kent Tamura, and Naohiko Uramoto. "Digest Values for DOM (DOMHASH)," Internet Draft, available at http://www.ietf.org 1999.

[Farmer et al. 1996] Farmer, William M, Joshua D. Guttman, and Vipin Swarup. "Security for Mobile Agents: Issues and Requirements." In: Proceedings of the 19th National Information Systems Security Conference October 1996.

[Gamma et al. 1994] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. (Addison Wesley Professional Computing Series, 1994).

[Ghezzi et al., 1991] Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandrioli. Fundamentals of Software Engineering. (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991).

[Harrison et at. 1995] Harrison, Colin, David Chess and Aaron Kershenbaum. "Mobile Agents: Are they a good idea?" New York: T.J. Watson Research Center, IBM Research Division. March 1995.

[Kiniry et al. 1997] Joseph Kiniry and Daniel Zimmerman, "A Hands-On Look at Java Mobile Agents," IEEE Internet Computing, Volume 1, Issue 4. July-August 1997: 29.

[Maruyama, 1999] Maruyama, Hiroshi, Kent Tamura, and Naohiko Uramoto. XML and Java, Developing Web Applications. (Reading, MA: Addison Wesley Longman, Inc., 1999).

[Preece et al. 1994] Preece, Jenny, et al. Human-Computer Interaction. (Addison Wesley Publishing Company, 1994).

[SASL 1997] Myers, J. "Simple Authentication and Security Layer (SASL)," Internet Request for Comments RFC 2222, available at http://www.ietf.org 1997.

[Walsh et al. 1997] Walsh, Tom, et al. "Concordia: An Infrastructure for Collaborating Mobile Agents." First International Workshop on Mobile Agents (MA'97) April 1997.

[XML4J] XML Parser for Java 2.0.15 API Documentation, available at http://www.alphaworks.ibm.com