Mindshare: a Collaborative Peer-to-Peer System for Small Groups

by

Gareth Charles Farrington

Bachelor of Science
in Computer Science
Florida Institute of Technology
2002

A thesis
submitted to the College of Engineering at
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Software Engineering

Melbourne, Florida
May, 2005

Technical Report
CS-2005-10

The undersigned committee,
having examined the attached thesis,
”Mindshare: a Collaborative Peer-to-Peer System for Small Groups”,
by
Gareth Charles Farrington

hereby indicates its unanimous approval.

____
William H. Allen, Ph. D
Assistant Professor, Computer Sciences
Major Advisor

____
Walter P. Bond, Ph. D
Associate Professor, Computer Sciences
Committee Member

____
Mohammad Shahsavari, Ph. D
Associate Professor, Computer Engineering
Committee Member

____
William Shoaff, Ph. D
Associate Professor, Computer Sciences
Department Head

Abstract

Title:

Mindshare: a Collaborative Peer-to-Peer System for Small Groups

Author:

Gareth Charles Farrington

Major Advisor:

William H. Allen, Ph.D.

We present Mindshare, a system for small group collaboration using Peer to Peer networking technology. This paper details the motivation behind its design, how it benefits users and details of its construction and operation. The solution focuses on the needs of small collaborating groups with limited computing experience and resources. Mindshare allows the group to share an unlimited number of files and visualize them in unified hierarchical file system. Mindshare synchronizes the files between users without user input. Its robust design allows files to be shared even when the owner is offline and allows users to work with files from the group while not connected.

Table of Contents

List of Figures

Acknowledgements

I would like to thank my parents, Charlie and Ann, without their unwavering love and support this work would not have been possible.

Thank you to my supervisor, Dr. William Allen, for keeping me focused and urging me to keep it simple. Without his aid I am sure I would never have finished.

I would also like to thank the members of the Open Source community whose tools, examples and support were vital contributions to this project.

Chapter 1 Introduction

Working in coordinated groups using computers is often more difficult than it should be, considering the high level of performance and connectivity that modern computers and networks provide. If a small group is attempting to produce a body of work that consists of many large files, the tools in widespread use today often fail to provide adequate support. The most widely used collaboration tool is e-mail. Many people who do not own a computer have an e-mail account and use it to send messages and images to family, friends and co-workers. Although e-mail is useful for exchanging information between individuals, it provides little assistance in organizing and managing a collaborative project. Updates must be transmitted manually and care must be taken to save the latest version of a document or photo that often arrives with the same filename as the previous versions. E-mail accounts usually have a limit on the maximum size of file that can be transmitted (often 5 to 10 Megabytes [FEPWC98]) and many word processing or presentation (PowerPoint) documents exceed that size when they contain graphics. Even a small group of people (say 5 to 10) may generate dozens of updates daily and, if e-mail is employed, each person must spend a significant amount of time managing this manual transfer of information.

While commercial and open source solutions to this problem exist (see further discussion in Section 2.2), most require centralized account management and a dedicated server. However, many small group collaborations are constructed in an ad hoc manner and lack the infrastructure support needed to make use of current workgroup support systems. In this thesis, we propose an automated, peer-to-peer collaboration support system, called Mindshare, which will provide the organizational and file transfer features needed for small group collaboration.

1.1 Collaboration Scenarios

To illustrate the scope of the small group collaboration problem, we will describe three scenarios where a tool such as Mindshare would be useful. Initially, we will discuss the most common solutions available today, assuming that the participants must make use of universally-available tools, such as e-mail or removable storage.

1.1.1 A Group Photo Album Scenario

You and several other individuals have taken a significant number of digital photographs and want to create a group photo album. Your group may be composed of co-workers, fellow students or family members, but this is not considered to be a work-related project and must be completed using the resources shared by the group. Each camera has at least a 256MB memory card that holds 100-200 images (depending on the camera's resolution), thus the group could have produced 500-1000 images which require well over one Giga Byte (GB) of storage space. The group members agree that they want to produce the highest quality photo album possible and that many of the photos must be edited or “enhanced” to meet that goal. However, although all members of the group have access to a networked computer, not all possess the expertise and/or software to perform the required editing. To further complicate matters, some of the group members do not want to exchange their memory cards.

Considering the number and size of image files, e-mail cannot provide the support needed to transfer and organize the data for this project. While images could be exchanged via removable media (CD-ROM, Flash cards, etc.) or a local area network (LAN), the task of organizing the files and dealing with duplicate filenames is daunting.

Each group member could copy all of their image files to a CD-ROM or other removable storage medium and deliver it to one location. The group could then work together to select the images to include in the final album and determine what editing or enhancements were needed. Once the selecting and editing are completed, a CD could be made of the finished product and five copies could be distributed to the group. Of course, with up to 1000 images, the editing and selection process could take a considerable amount of time. Alternatively, each person could carry copies to all of the other group members and they could discuss the editing and selection via e-mail. However, edited images would have to be distributed back to all group members by hand before final selection could be made. Needless to say, the difficulty of coordinating these activities keeps most users from using this solution for projects of any size.

If the group members were all in the same office or University campus and their PC’s were connected on a common network they could eliminate the physical transfer of images by using remote file access. This solution allows the group members to access and transfer files between machines at very high speeds. This solution is more attractive but it presumes the existence of significant infrastructure and poses potential security risks. It also does not provide automated transfer or organization of the image files and could lead to the accidental erasure of files on another group member's computer.

1.1.2 A Group Project Scenario

Let’s consider another problem that occurs frequently in a academic setting. A class is broken into groups and asked to write a paper and produce a presentation. One copy of each final document must be submitted to the instructor. The paper and presentation must include images, diagrams and perhaps other media.

Groups of this type, particularly in college, use e-mail to solve this problem. E-mail is ubiquitous and easy to use. Its behavior is simple and predictable for novice users. Groups using e-mail generally find that it serves them well for projects with a limited scope. However, there are several common problems that groups exchanging files via e-mail will experience.

Presentations in particular tend to be very large documents, particularly when they include images, sound and movie clips. These files are often significantly larger than 5 Megabytes (MB). But, as a general rule, 5 MB is the largest attachment that many e-mail systems can reliably deliver [FEPWC98]. Files larger than this have to be compressed or partitioned by the sender and re-assembled when received.

There can also be confusion over file versions. If each member of the group is contributing individual slides, a single user must be in charge of integrating these slides into a final presentation. They must then send the final document back to everyone for approval. This tends to generate lots of duplicate files. It can be difficult to tell what version of a file or slide you are looking at, particularly if you aren’t looking at it in the context of the e-mail message that it arrived in.

This process also wastes space. Old versions of files are not automatically deleted from your mailbox when new ones arrive. Users often reply to messages containing files and include the file in the reply. This creates yet another unnecessary copy. As the number of duplicates climbs it becomes harder to determine which version is the most recent.

When sending multiple attachments in a single e-mail message there is no folder structure. The sender's original file organization is lost. The sender could archive the files before transmission by using one of the many archival tools that are available; this adds complexity to the version tracking issues mentioned above. Unless there is an agreed-upon protocol and each group member has a certain level of familiarity with the archival software, files could be over-written or erased.

1.1.3 Small Team Programming Scenario

A small team of programmers has to produce a working program of some reasonable complexity. The program is too large for a single programmer so it is divided amongst several team members. The team uses and produces many of files including; specification documents, code, libraries, configuration files and documentation. Every member of the team needs to have these files in order to carry out their work. They all need the most current code to build and test their work. This is called Continuous Integration by extreme programmers [Beck99].

Large programming teams use specialized collaboration software called a Source Control System that includes functions for source code control. Source code is stored in text files and this allows two files to be compared and merged automatically. A popular package is Concurrent Versioning System (CVS) [CVS04] but many other packages exist (SourceSafe [SourceSafe04], Subversion [Sussman04]) that have similar capabilities.

Source control systems are designed to work from a dedicated server that runs the Source Control Software. Each user installs a client program to interact with the Source Control Server. This requirement is usually not a problem for teams in a corporate environment. They have the required resources; a server and someone to maintain that server required to use source control. However, teams in small companies or in open-source environments often do not have the necessary resources or expertise required to use server-based source control software.

The other barrier to continuous integration with a Source Control System is synchronizing resources across computers. The server stores the master copies of the resources (source code etc.) and manages changes that are initiated by the clients. When a resource is changed by a client, the other clients are not informed of this change. They have to initiate synchronization manually with the server to receive the update. In some cases this is desirable. However, if the project is large and synchronization is time consuming, the clients may only synchronize once a day. In smaller teams it is more likely that an update will affect another team member because the smaller working set puts the group members in closer proximity. Therefore, for smaller teams it is critical that updates reach the rest of the team in a timely manner.

1.2 Problem Summary

The scenarios presented above each describe a different collaboration problem, but could all be solved with a flexible, on-line system for small group collaboration. In each case there is a small group of individuals who are not strangers and who have no desire to share their work with people outside their group. The group is small enough to solve conflicts socially, rather than having a computer-based tool enforce a workflow or structure. Each member will be contributing in some way to the group, but often performing different tasks. Each user has a PC that is capable of performing the necessary work. In a computerized environment, information takes the form of files and productive work involves changing and organizing those files and creating new derivative works. In ad hoc small group collaboration scenarios, such as those described above, the problem lies in getting the right files to the right user in a timely manner so they can perform their work. For a truly collaborative environment many people must see the same information so they can all provide input on the work being done, even if they are not doing it themselves.

As described above, existing tools that support small group collaboration provide a range of different solutions, each with its own strengths and weaknesses. Figure 1.1 shows the relationship between the solution provided by Mindshare and existing tools such as e-mail and CVS. Mindshare provides some features from each of these systems but also introduces new capabilities that better support small workgroup collaboration.

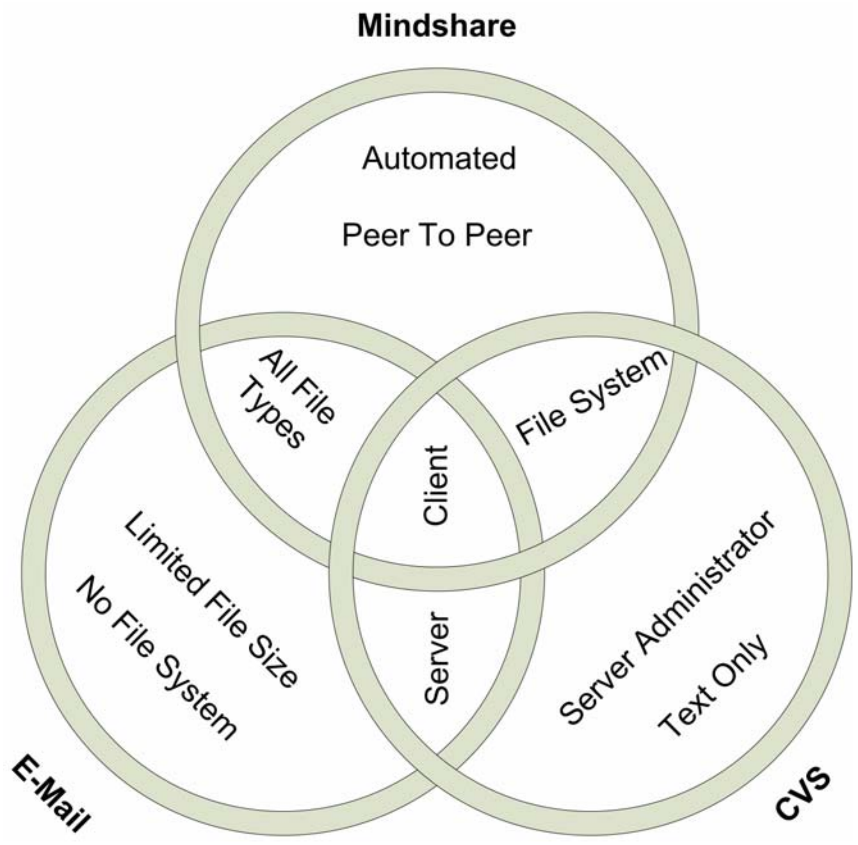


Figure 1.1 How Mindshare relates to other collaboration solutions

1.3 Proposed Solution

In this thesis we propose a solution to the problem of small group collaboration that does not require a dedicated server and supports the formation of small ad hoc workgroups. We will discuss the design of Mindshare, a Java-based prototype that demonstrates the proposed system design. Mindshare is a Peer-to-Peer (P2P) system to aid small groups of users in performing tasks similar to those described in Section 1.2. Our solution is simple to use and easy to deploy and is based on Java so that it runs on many platforms. It requires no dedicated server hardware; the client runs on the PCs the users already have. No special expertise is needed to set up or configure the software. Mindshare allows the groups members to share an unlimited number of files of unrestricted size and to visualize and use these files in a single unified file system structure. Mindshare performs all of the necessary file replication tasks in real time without requiring user-intervention.

The remainder of this paper is divided into four chapters that cover the design and implementation of the prototype: Chapter 2 encompasses the requirements and goals that drove the design of the prototype. Chapter 3 provides specific details on the prototype design and implementation. Chapter 4 covers the Software Engineering challenges specific to the prototype. General challenges in developing P2P software are also discussed along with a description of specific tools and libraries that are already available for P2P applications. We will also comment on the difficulties involved in testing a network-based P2P application. Finally, in Chapter 5, we present ideas for future enhancement of the prototype and our conclusions.

Chapter 2 Computer-Supported Collaboration

In this chapter we discuss current collaborative systems and how they relate to our solution. We will also briefly describe goals and requirements for the Mindshare system.

2.1 The Potential of Peer-to-Peer Systems

In this section, we discuss the benefits provided by peer-to-peer (P2P) systems and technologies, along with comments on the equally significant potential for abuse.

[Briedenbach01] discusses the growing potential of P2P systems for distributed processing, file sharing, collaboration and content distribution, but also points out the security threats and copyright issues that come with decentralized systems. [Thompson05] discusses the growth of P2P file sharing and how it has been used for software and content piracy. The impact of the BitTorrent [Cohen04b] file distribution protocol on file sharing is also discussed, along with a projection of future trends in file sharing. Kant, et al., [Kant02] provide a detailed taxonomy of P2P technologies and usage models.

Some research has provided a more detailed look at specific systems and technologies. For example, [Ripeanu01] takes an in-depth look at one peer-to-peer architecture, Gnutella, and [Vaughan03] discusses the importance of maintaining real-time presence as an aid to collaboration systems.

In [Biddle03], the authors discuss the future of legal and illegal content distribution. This paper, written before the release of BitTorrent, predicted that the restriction or elimination of server-based or semi-centralized illegal file sharing (such as provided by Napster [Stern00] or Gnutella [Ripeanu01]) will not prevent an increase in the illicit distribution of protected content over "darknets", i.e., unregulated on-line distribution channels[Biddle03]. They state: "There seem to be no technical impediments to darknet-based peer-to-peer file sharing technologies growing in convenience, aggregate bandwidth and efficiency."

Clearly, peer-to-peer systems provide technology that can be used for both positive and negative purposes. The possibility that Mindshare can be used for illicit activities has been considered, but we believe that its ability to solve collaboration issues for socially-connected small groups far outweighs the potential for misuse. We remain hopeful that positive social interaction between group members will provide the common goals and incentives that will exclude malicious or illicit motives from impacting the overall good of the group.

2.2 A Brief Overview of Existing Collaboration Systems and Environments

In this section, we will discuss several existing workgroup collaboration systems that could potentially be used to solve the problems presented in Chapter 1. We will describe why they may or may not provide a useful solution and will also compare their features and capabilities to the Mindshare system.

2.2.1 Local Area Networks

A Local Area Network (LAN) is the most widely used collaboration environment in today’s workplace. A LAN generally consists of one or more server computers that store data and shared software and many client computers that access that data. Clients collaborate through the server by altering files that are stored on the server. This clearly entails an investment in infrastructure and often requires (both for security and performance reasons) that the client computers be on the same physical network as the server. If the server is inoperative, the clients can no longer work together. One of Mindshare’s goals is to free users from these constraints by eliminating the centralized server and providing connectivity over any existing network connection that supports the TCP/IP protocol.

2.2.2 CVS

Concurrent Versioning System (CVS) [Cederqvist02, CVS04] is a widely used Source Control System. It allows a set of users to share and work on a corpus of text files with coordination and control provided by a central server. A variety of client applications exist to access files stored on the server, but CVS does not notify clients when a file on the server is changed. Mindshare does not require a server and it will automatically update clients that are online when a file is modified. Mindshare also supports binary files but forgoes the advanced processing of text files (that CVS provides) to do this.

2.2.3 Lotus Notes

Lotus Notes [Lotus05] is an e-mail system that is based on a database-driven network of ‘hub’ servers and Lotus Notes clients. Lotus Notes can support very large file attachments between clients over the hub network. For projects that consist of several files Lotus Notes does not escape any of the other problems with e-mail. Mindshare can keep many files organized both in terns of logical organization and keeping each peers copy up to date. Mindshare also does not waste space on archiving old versions of documents or make any unnecessary duplication of current files.

2.2.4 Waste and Grouper

Waste [Waste05] and Grouper [Grouper05] are two Peer-to-Peer applications that are very similar in their function. They allow a group to form a small world network and share files in a manner very similar to the tools predicted in [Biddle03]. Users can search for and download files from other peers in the network. This is a good design for groups where the material being shared is not of interest to everyone. Mindshare is also a small world network but it provides key benefits for groups whose main objective is creative collaboration. Mindshare automatically synchronizes files without requiring tedious user interaction.

2.2.5 Groove

Groove [Groove05] is a proprietary Peer-to-Peer application that can synchronize files between peers. It is very similar in functionality to Mindshare. Groove does not require a server for basic operation on a LAN, but a server is needed for Internet-based collaboration. Also, a relay server is required to synchronize information between peers that have gone offline. Mindshare can use other peers on the network to synchronize so that, in effect, each peer is a relay.

2.2.6 Speakeasy

Speakeasy [Edwards02] is an extensible Peer to Peer system that allows for a wide variety of collaborative features beyond file sharing. In addition to files, Speakeasy can share peripheral devices like cameras, printers and displays. These features are great for group collaboration when the individuals are in close physical proximity. Speakeasy does not appear to support automatic file synchronization or structured file shares.

2.3 The Mindshare Approach to Collaboration

One important design assumption for Mindshare is that the typical size of a group that would use this tool is less than twenty people. Much larger groups are not as likely to employ a peer-to-peer tool such as Mindshare because those groups usually are formed in situations where alternative groupware is a possibility. Enterprise users generally have a Local Area Network (LAN) provided for them. They also have dedicated infrastructure and staff to administer more complex solutions such as CVS or Lotus Notes. In larger groups, social and organizational issues would require a tool that imposed a stronger organizational structure than Mindshare attempts provide. Larger groups require more structure because the social dynamics in those groups increase, requiring a system that enforces strict rules and access controls.

We also assume that the individuals participating in computer-based groups will have a computer with an Internet connection because they already send e-mail or have digital devices that require a computer for support. It would also be safe to assume that most users of this type have broadband access. These connections are fast but are generally asymmetrical, providing greater download speed (1.5 Mbits) than upload speed. A Local Area Network would have links an order of magnitude faster (10 or 100 Mbits) and these links are symmetric.

The collaborators have a need to work with large files and/or frequently updated files. On a LAN such files can be accessed from another computer quickly so local duplication is unnecessary. The lack of reliable direct access to the other group member’s computers in a P2P environment means each user will need a local copy of all the files being shared. This will allow the user to work ‘off-line’, something they cannot do on a LAN. This will also eliminate the need for dedicated server hardware.

The system also targets users that do not have significant computer usage experience. They can use e-mail successfully but a more complicated system would make a formidable barrier to adoption of a new tool. E-mail works well on all popular computing platforms. Users already have an e-mail address so setting up a group to collaborate using e-mail simply involves exchanging e-mail addresses. If the system is significantly less reliable or more difficult to use than e-mail they may not be successful. Any system developed needs to meet the needs of this group of users.

2.3.1 Mindshare’s Design Philosophy

The design of the system was not driven solely by the deriving requirements from scenarios described in Chapter 1. We wanted to create a Peer-to-Peer (P2P) system to investigate P2P networking technologies as a useful tool for solving a real problem that would impact a large potential user set. P2P systems have this far been used primarily for the theft of intellectual property (e.g., Napster [Stern00] and Gnutella [Matei02, Ripeanu01]) or for academic computing applications (Cracking encryption [Distributed05, RSA05] and SETI [Anderson02, SETI05]) that don’t yield tangible benefit to the average home user. The lack of positive and widely visibly applications based on P2P technology has caused many, including much of the news media [Thompson05], to assume that any technology that is described as ‘P2P’ is a vehicle for theft. We had a desire to show that this view is incorrect. We believe that P2P technologies can find widespread use without the lure of potential theft of intellectual property.

Usability was a primary focus in all aspects of the design. To provide a useful solution to the small group collaboration problem, the proposed system must rival e-mail and Local Area Networks in ease of use. This meant giving users a more powerful tool than e-mail without making its use overly complicated or foreign to the typical user experience.

2.3.2 Mindshare's File System Organization

The first key area where Mindshare enhances usability and power is in the way it organizes and distributes the files themselves. Early in the design phase a decision was made to support a hierarchical file system over a flat system. This fits well with the way users are already comfortable with for organizing their files on disk or on a LAN. They use directories, sometimes called folders, to organize and segregate files. Most P2P systems to date have used a flat file system so there was no precedent for this in existing P2P tools.

Most P2P tools require the user to search for information on remote computers. Some users make available far more resources than other users are interested in. In a collaborative environment this is not very useful. We can assume the collaborators are sharing a set of files focused on a particular topic or project. When files are added, changed or removed this should be immediately visible to the other users. These files need to be available even when the peer that is sharing them is offline.

2.4 Functional Requirements

“Functional requirements describe the interaction between the system and its environment” [Pfleeger01].

2.4.1 Large and Numerous Files

The tool needs to support large files and large numbers of files. The only practical limit should be the size of the user’s hard disk. If possible, transferring these files should happen as fast or faster then e-mail can support.

2.4.2 Structured File System

The file system should have a hierarchical structure like a native file system. This structure of files and directories is familiar to users.

2.4.3 Offline Access to Files

To support shared viewing and editing, a user’s files should be available even when that user is not online. From within the tool it should be obvious if a file has been changed or a new file has been created even if the file has not been completely downloaded to the user's PC. This provides an accurate picture of the structure of the file system even if the data is not available.

2.5 Non Functional Requirements

Non functional requirements describe the constraints on the system that limit or guide choices in constructing a solution to the problem. [Pfleeger01]

2.5.1 Automation

The process for starting a new group (bootstrapping) needs to be simple enough to encourage users who have limited experience with computer-based collaboration. It should be as simple a process as possible, analogous to people swapping e-mail addresses.

However, the tool should also be at least as easy to use for file transfer as e-mail or users will prefer that method. Automatic replication of new and changed files would be a useful feature which would save users the effort required to create an e-mail message and attach the changed resource(s). These updates should also be accessible to all the group members immediately, synchronization should be automatic and require no user intervention. The files should reside on each local user’s hard disk in a structured manner, something e-mail does not support. It should also be possible to access the files from a PC in a lab or other ad hoc location where team meetings might take place.

2.5.2 Reliability

Typical home PCs are not on at all times. They are prone to power outages as few are protected by an interruptible power source. Users also turn off their equipment to save power or to reduce noise and heat. E-mail gets around these outages by holding mail in a mailbox at a central server until the user is online again. A collaboration support system should provide the same level of reliability as e-mail and should automatically deliver updates when the user comes back online.

Modern home computers are of adequate power and are equipped with more storage than most users need. A useful tool should leverage this extra storage to overcome the limitations of an unreliable network.

2.5.3 Security

E-mail users are besieged by spam. Any new system needs to allow for keeping out unwanted data and people. If a social group is formed, then the tool should reinforce the boundaries of this social network. Only those invited to the group by existing members should be allowed in. The group should be reasonably sure that someone they are inviting to join is not misrepresenting their identity.

To protect against eavesdropping and theft, the tool should also use cryptography to keep data safe while it is being transmitted. E-mail does not provide for transport security and most mail transfer agents don't support encryption. Some users hare resorted to encrypting the plain text sent inside messages. This requires the individuals that wish to exchange encrypted communications to exchange keys. E-mail does not support key exchange directly, though keys can be sent in an e-mail message. This is a manual process and deters most users from employing encryption.

2.5.4 Clients for Multiple Platforms

To approach the usability of e-mail any new collaboration software will have to be available on as many platforms as possible. The three key consumer platforms are Windows, Mac OSX and Linux. The Java virtual machine [Sun05] is available for all of these platforms and, potentially, for future developments as well. Java was chosen as the implementation language for Mindshare largely because of its portability.

Chapter 3 Prototype Design and Implementation

In this chapter, we will discuss the evolution of Mindshare's design and the prototype implementation. Major components and sub-systems will be described, along with details on their features and limitations. The various open-source libraries that are used for network connectivity and data transfer will also be described.

3.1 Design Issues

The Mindshare prototype was initially intended to be a distributed source control system. The idea was that the server could be replaced by a distributed locking mechanism. If a peer wanted to change a resource it would obtain a lock, make the change and release the lock. Work started under the pretense that this initial design could be made functional. Use cases and scenarios were developed to try and break this system. The initial use cases did not take the unstable nature of home PCs and Internet connections into account. Once this lack of reliability was considered, it was determined that the existing design was not adequate and subsequently, a new design was developed.

3.1.1 Concurrent Resource Modification

To understand why a fully distributed system with inconsistent node uptime can’t avoid conflict we need only look at a simple example. Consider a simple network with 2 peers, Bob and Alice and a single file shared between them. Bob has a laptop and in the morning he goes online and finds that Alice is not online. Bob spends the morning making many changes to the file and then packs up his laptop and heads to lunch. While Bob is out Alice goes online and joins the network. Because Bob was not online when Alice came online there is no way for her to know that changes have been made to the file they share. She then makes a minor change to the file while Bob is out to lunch. When Bob gets back there is a problem. Both Bob and Alice have made legitimate modifications to a file but software cannot be used to decide whose file is most current. Computer Scientists call this a Concurrent Modification. A single resource appears to have been modified by two entities at the same time.

Comparing modification timestamps would not be useful. Alice’s modification timestamp is newer than Bob's but her file only contains a single minor change. Bob’s file contains a whole morning’s work. Using timestamps to choose a file would cause all of Bobs work to be lost when the system updated his file with Alice’s minor change.

Comparing version numbers won’t work either. If each client keeps a version number you could use software to compare the version numbers. Bob’s version number would be higher because he made several changes. Alice’s version would be overwritten by Bob’s version.

Neither situation is satisfactory for both parties; in either case data loss is the result. This situation would not arise if Bob had left his laptop on and connected while he went to lunch. There are a number of other circumstances beyond Bob’s control where the same situation might arise even if he had left his laptop behind. His battery might die, the power might be cut or his network connection might go down all leading to the same result. The fact remains that consumers don’t want to leave their computers on at all times. Computers will be shut down for a variety of reasons and the software needs to cope with this scenario gracefully.

The other problem is detecting and resolving these conflicts. Using both a timestamp and a version number together could allow the software to recognize concurrent modification conditions. Detection involving more than two peers becomes increasingly complicated. Resolving these conflicts requires human intervention and coordination.

Normally a server, a centralized master computer, is used to solve this problem. It has centralized authority to prevent concurrent modifications from occurring. It enforces this by not allowing a resource to be modified while it is in use by someone else. In a pure Peer to Peer environment there is no server and hence no possibility of central authority. A different solution had to be found that avoided concurrent modification but still let Bob take his laptop on his lunch break.

3.1.2 Distributed Authority

Mindshare takes the approach of distributing authority amongst the peers to make concurrent modification impossible. Each file in the system has an owner. Only the owner of the file can modify the file. Ownership can be transferred from one owner to another but to accomplish this both peers have to be online at the same time.

Let's see how Alice and Bob would have managed their day under this new system. Assume that Bob has ownership of the file at the start of the day. He makes his modifications and goes to lunch. Alice doesn't own the file but she still wants to work on it so while Bob is at lunch she makes a copy of the file and modifies the copy. Bob arrives back from lunch and the system automatically synchronizes Bob's file to Alice's computer. Bob is the owner so his version is always considered the most current. Alice then sees Bob's work but still wants to make her small change. She asks Bob to give the file to her so she can change it. Bob complies, she makes her change and then she returns the file to Bob.

The file can be thought of as if it were a physical object. Only a single person can possess it at once. This gives users a very simple mental model to plan their work around. They don't have to concern themselves with Concurrent Modification. They can also turn off their computers when they wish.

3.1.3 How Distributed Authority is Implemented

Mindshare implements ownership using metadata. The metadata for all the files that a user owns is compiled into a single document. This document, called a Tree, is replicated by Mindshare to all the other connected peers in the group. The receiving peers take all of the individual Trees they receive and merge them to form an intermeshed Tree to display to the user. Imagine that each Tree was written on a transparency sheet and then layered one on top of another. The display is a sort of optical illusion. There is no centralized master data set but we can make the user believe this is so by composing the individual Tree together. Thus, the Tree is a visual representation of the merged Trees of all users, but that merger of Trees does not actually exist in the system.

Files in the display that do not belong to the user cannot be renamed, moved or deleted. This is not because of artificial checks in the software but because of the architecture. A user's edit operations are performed against the user's own Tree, not across the intermeshed view. After an edit the intermeshed Tree is recreated and the view is updated.

Files are the only resource that you can own in Mindshare; directories (folders) have no owner. If two or more users create a folder with the same path name, Mindshare will interpret this logically as being the same folder. Any files that each user might place into that folder will be displayed as if there were in the same logical folder, but they will actually reside in the owner's version of the folder and each owner retains control over the files.

3.2 Trees

To achieve the goal of distributed authority a document is needed to convey to other peers what is being shared. This document would include the metadata about the files that a peer is sharing. In the Mindshare system this document is called a ‘Tree’ because it represents a hierarchical set of files. The metadata includes details like the file’s path, name, size, last modification time and a hash calculated from the data in the file. Compared to the total size of the files a peer is sharing, the Tree document can be orders of magnitude smaller. A Tree’s small size allows it to be quickly transmitted between peers. Peers can receive a Tree and present the update to the user even before the files it represents are downloaded. This enhances usability in an unstable network. For example, if a peer goes offline before it can distribute a large file to the rest of the network, the other peers still know that the file exists because they have the metadata tree that shows it exists. Peers don’t have to receive a tree directly from the peer that published it either. Any peer can deliver a copy of the tree to other peers. This allows peers to distribute information on behalf of others that are currently offline.

The use of metadata Trees allows each peer to form its own strategy for downloading the files from other peers. A Mindshare peer will perform a comparison of each incoming Tree update against the files it already has. If any files have changed or new files have been created, they are put into a download queue. Mindshare uses the file size information so that the smallest files are downloaded first. This "shortest-file-first" approach does mean that very large files may not be distributed across the entire set of peers as quickly as smaller files. Although the possibility that some large files may never be distributed to all peers can be a drawback, it also discourages sharing large files that other peers may not want on their computers, such as large video files. This design decision makes Mindshare less useful for sharing illegal music or video files, but may be undesirable for groups that want to build a photo album from their own pictures. Some thought has been given to allowing users to assign a priority to files that they need updates to, overriding the default "shortest-file-first" algorithm. This approach lets the user decide what files should be brought to his or her own computer, rather than letting the person who adds the file force its distribution upon other peers.

Trees are also easily merged; this is a key factor in the display. The Mindshare client shows a file system composed of all the trees shared on the network merged together. Directory names that are common to two trees will appear as a single directory in the display. This allows the user to place their files in a folder that another user has created even though the files, in fact, exist in separate trees and have different owners. The display can also be recomposed to show only a single tree or subset of trees. This feature is useful for answering questions like "Where are Bob's files?" Showing only Bobs tree will hide unnecessary data and let the user quickly find what they are looking for.

Lastly, trees can be stored and used later. Mindshare saves the trees it receives from other peers. Even if a peer has not been online in weeks, other users can still see files from that peer in the combined display.

3.2.1 Uniform Resource Identifier Usage

Mindshare uses a Uniform Resource Indicators (URI) to identify resources. Each file can be identified individually by its URI. Each URI is unique and can be used as a primary key.

Mindshare URI's take the form:

```
mshare://userid@groupid/path/file.txt
```

This form fully qualifies a file with the Tree so that it can be identified if it is being processed outside of the context of the Tree it belongs to. The URI contains the user's unique identification, group identification, path to the file and name of the file. The format of the unique identifiers is discussed in the next section.

URI's are versatile and can be used to denote an individual user:

```
mshare://userid@groupid/
```

The group:

```
mshare://groupid/
```

Or a path within the group:

```
mshare://groupid/path/
```

You will notice that user identifiers are only valid within the context of a group and thus have group scoping. Paths ending in ‘/’ denote a directory.

3.2.2 Unique IDs

Each user and group needs to be identified by a unique ID to allow for possible collisions between user names and group names. This also allows for user and group names that contain characters that would be illegal in a URI. Because each user has a unique identifier and each user can only publish a single Tree it is logical to think of the user ID as the Tree ID.

The IDs need to be used in the user and host components of a URI there are a number of reserved characters that cannot be used for unique IDs in Mindshare. The raw randomly generated IDs are in binary format and must be encoded before they are suitable. The raw data is hex encoded to produce a 20 byte ID. For the rest of this paper references to ‘`userid`’ and ‘`groupid`’ refer to one of these 20 byte hex encoded identifiers.

3.2.3 Merging Trees

Trees are merged to be displayed in the user interface. This is the process of converting the flat list of URIs and metadata within each Tree into a hierarchy of objects that can be used as a model for the file browser table. Files that have matching paths are lumped together into a single list. This is achieved by processing every Tree and file in the system and recursively building a graph of folders that contains these lists of files.

3.2.4 Tree Synchronization

The goal of this process is to distribute new Trees throughout the entire network quickly and efficiently. While the size of a single Tree is theoretically unlimited, the average size is only a few kilobytes. Still, sending trees directly between peers without cause could use unnecessary bandwidth.

New Trees are announced over the Availability Channel (see Section 3.6) in the form of a Tree Availability Message. Only two pieces of information are needed to reference a particular Tree; the userid of the tree's owner and a version number. Together, these bits of information form a Tree descriptor. Because the descriptors are small they are perfect for advertising and requesting Trees.

Each peer keeps a list of current Tree Descriptors in memory. This list mirrors the actual Trees stored on disk. Each time an availability message is received it is compared to the list in memory. Receiving information about a new or updated Tree causes an event to be fired and initiates the tree synchronization process. The new Tree Descriptor is entered into a request queue. If the new Tree would duplicate a descriptor already in the queue, it is discarded. This is entirely possible as peers that have successfully downloaded the new Tree will be sending new availability information.

The queue is emptied by first choosing a peer that has the desired Tree and then downloading a copy from that peer. Once the new Tree has been downloaded and saved to disk then the internal Tree Descriptor list is updated. The new Tree is merged with the other trees and the user interface is updated. Finally the peer will send an availability message advertising the new Tree at this peer.

3.3 Network Technology

Mindshare uses JXTA [Oaks02] for all communication between peers. JXTA is a peer to peer overlay network. It allows peers to communicate from behind firewalls and Network Address Translation (NAT) routers. It transparently uses relay and routing peers to achieve this.

JXTA provides a facility for grouping peers together called a Peer Group. Members of a Peer Group are expected to communicate with a common set of protocols. JXTA also dynamically designates one or more super peers, called Rendezvous Peers, from within the group to route/relay traffic and maintain routs between peers. Any peer is also free to act as a Relay Peer to retransmit traffic on behalf of peers that cannot be directly connected. Such cases include peers behind Network Address Translation routers, and firewalls that restrict traffic. JXTA can operate over any suitable transport mechanism. The current Java implementation uses UDP, TCP and HTTP as transports. Peers located behind restrictive measures use the HTTP transport to communicate with other peers.

Every peer group is created as a child of another group. There is a single parent peer group, called the Net Peer Group of which all nodes are a member by default. From there peers create and join subgroups as needed. For Mindshare a two layer approach was chosen. A peer group called ‘Mindshare’ is created under the Net Peer Group. All Mindshare clients are deployed with the advertisement of this group. This group serves to segregate the application from the rest of the public JXTA network. The Mindshare group contains numerous user created subgroups. The client creates a new sub group for each group of users that wish to collaborate. This scopes the resources and traffic generated by each group of users. It also keeps the burden of sufficient resource provisioning within the group.

3.3.1 Protocol Stack

Mindshare uses both multicast and regular sockets for peer communication. JXTA provides primitive socket types to be used in place of the native Java Sockets. These Sockets resolve endpoints in the JXTA overlay network rather than in the physical TCP/IP network. Below these sockets is whatever physical transport (HTTP/TCP) that the user has configured JXTA to use.

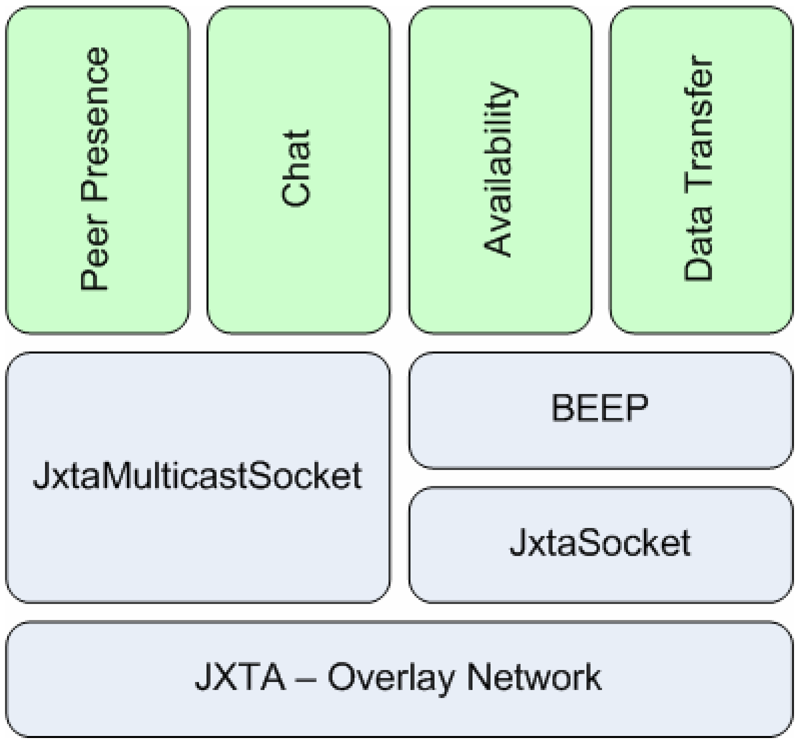


Figure 3.1 Mindshare Protocol Stack

A single common Multicast Socket address is used for Peer Presence. All peers transmit and listen on the same socket address. The address, known formally in JXTA parlance as a Pipe Advertisement, is generated from the Peer Groups Advertisement so it is unique to the group. Multicast traffic is unreliable and limited in size to single packets. Generally the limit for safe delivery is 16k of data payload. All Presence messages are well below this size. A separate but similar Multicast Socket is used for the group chat protocol.

To transfer large objects, like Trees and Files a reliable Socket capable of fragmentation and reassembly is used. To avoid having to open many sockets between each pair of peers the Blocks Extensible Exchange Protocol (BEEP) [BEEP05] is used. BEEP supports multiple channels of communication over a single socket. Each channel can operate a different protocol and the messages will be multiplexed over the socket automatically. In the prototype two channels are used; one for announcing the availability of resources and the other for File & Tree transfer. In the future this architecture will support more channels for one to one chat and the exchange of security information.

3.4 Peer Protocols

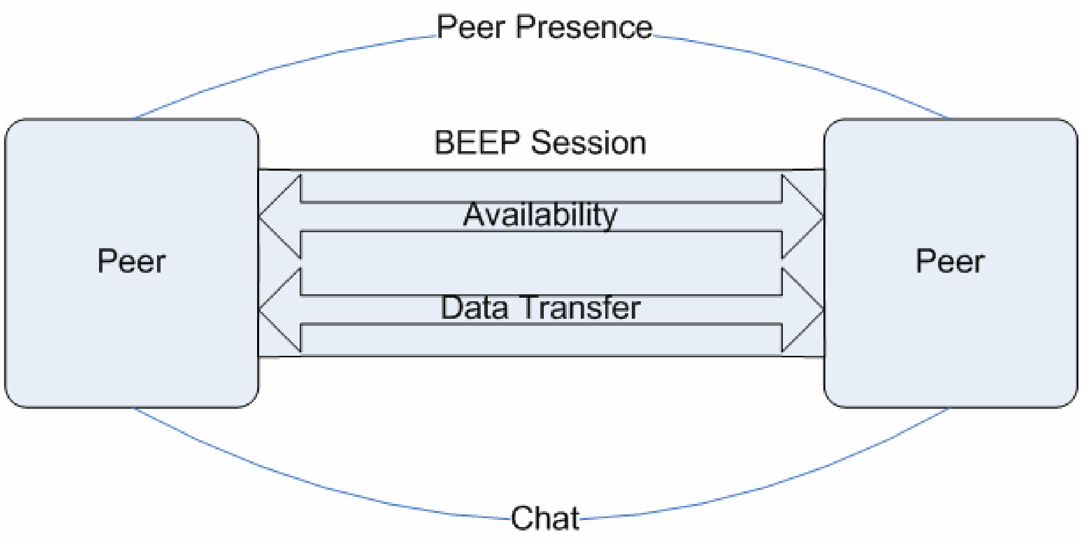


Figure 3.2 Mindshare Protocols and Connections

Figure 3.2 shows the typical connections between two peers. The BEEP session carries the Availability and Data Transfer channels while separate Multicast links are used for Peer Presence and Chat.

3.4.1 Message Encoding

Mindshare uses a single encoding scheme for all messages and data passed across the network. This scheme is called *BEncoding* and was created by Bram Cohen [Cohen04a] for the BitTorrent protocol. This encoding is attractive because it supports structured documents and allows for raw binary data to be included in those documents. XML [XML05] is currently a popular standard for data transfer but it does not support the transfer of binary data without wasteful encoding. For that reason we decided that XML was not a suitable encoding scheme for messages.

BEncoding supports strings, numbers, lists and dictionaries. Dictionaries are similar to hash tables and can be used to store structures of associated data elements. Strings can be of arbitrary length and can contain raw binary data. BEncoding uses a form of run-length encoding to record the length of strings so that a unique, reserved end-delimiter character is not needed. By design there are no reserved characters in a BEncoded document so that strings may contain raw binary data. BEncoded documents are not as readable as XML therefore, this paper simply describes message elements without providing examples.

Messages sent between peers are preceded by a single byte to denote the type of message. This makes it easier to route raw messages to the factory without reading the entire message.

3.4.2 Peer Presence

This protocol allows peers to quickly discover when the other peers in the same group are online. This is a basic service and uses multicast messages that reach the entire group. This service will be described in detail in section 3.5.

3.4.3 BEEP Session

When two peers within the same group intend to exchange data, they first use the presence service to discover each other and then establish a single BEEP session between them. BEEP requires that one party act as the initiator; opening the underlying socket connection and starting all channels. To determine which of the two peers will serve as the initiator each peer compares its unique PeerID to the other. The peer with the lower id acts as the initiator and must open the connection. This architecture implies that the network of peer-to-peer connections created is a fully connected graph. Although this technique will not scale well to large networks, our design assumes that networks will remain small (see section 2.3).

The initiating peer opens at least two channels within the BEEP session, one for exchanging availability information and the other for data transfer.

3.4.4 Availability Channel

The Availability channel transfers information about the Trees and file pieces available at a peer. Files are divided into 512 kilobyte pieces to simplify transfer. Two messages are supported, one for advertising a Tree and another for a single piece of a file. To save bandwidth the Tree availability message also includes a bit field encoded in a string element that represents the availability of every file piece in that tree. This allows peers to quickly get a picture of the network without exchanging thousands of messages.

Each message is sent as a BEEP ANS type response. The receiving peer is not required to give a response to the individual ANS messages. When the sending peer is shut down it sends a NUL message to terminate the exchange gracefully.

As the peer receives availability data it is checked and recorded in internal data structures. This allows each peer to builds an effective map of the resources that are available at each peer. The data in this map is then used to decide which pieces to download from other peers. Trees and file pieces are requested over the Data Transfer channel.

3.4.5 Data Transfer Channel

This channel is used for lengthy bulk data transfers. This includes both the Trees and files. This protocol has two messages that follow a request-response pattern; one message is used for requesting a tree and the other is used for requesting a file. There are corresponding response messages. Requests are queued on the client side and prioritized. Tree requests have a higher priority than all file requests and file requests are further sorted by the size of the file, smallest first. This causes updates to smaller files like office documents or code files to move to the head of the queue and makes the system more responsive.

Once a data item has successfully been transferred and stored, its availability is advertised through the Availability Channel.

3.4.6 Chat

This is a very simple multicast-based service. It allows the group to communicate in a fashion similar to an Instant Messenger. The chat service broadcasts and listens for simple BEncoded chat messages. This service does not support advanced features found in other chat protocols but it may be enhanced in the future. Chat is functionally separate from the rest of the system, and is not dependant on the presence service. As such, it will not be discussed in any greater detail in this paper.

3.5 Presence Service

Because JXTA uses a “super peer” architecture, network nodes in a group are not always directly connected. Without a guarantee of direct connectivity, it is necessary for each peer to discover which other peers are in the peer group before a direct connection can be established. A peer group in JXTA may have many RDV peers and this can result in queries that do not span the entire group. In order to have the software operate in as many environments as possible, it was necessary to avoid using RDV peers for presence information. Instead a protocol was created using Propagated Pipes [Oaks02]. Messages sent through a Propagated Pipe will reach all peers that are listening in a group.

In this case ‘listening’ is not so easily defined. For a peer to hear a propagated message its RDV peer must be able to ‘see’ the RDV peer that is propagating the message on behalf of the sender. Experiments show that in large groups this can sometimes be a problem. Nodes may require a significant amount of time to find each other on the network and traffic will not be delivered. However, once messages do start to arrive at a peer they continue to do so reliably.

JXTA Propagate Pipes are by definition unreliable, therefore message delivery is not guaranteed. This is partially due to the use of UDP multicast in LAN environments. The underlying transport is itself unreliable. Over the internet however UDP multicast is not available and JXTA ‘simulates’ multicasting behavior by using reliable TCP connections. This means that the design of the presence protocol was less affected by any lack of reliability inherit to the underlying transport protocol and more by the lack of reliability of JXTA itself.

Nodes that join the group at opposite ends of a large network of RDV peers will take a long time to find a route to each other, even though they have joined the same group. Also after a connection has been realized it is possible for a relay peer or RDV to go offline changing the route between peers. In the end there is no fixed timeout or positive condition that occurs when a route is found so the protocol has to take the dynamic nature of peer-to-peer systems into account.

3.5.1 Aims of a Presence Service

The presence service must reliably supply information about the peers that are currently online. To provide reasonable performance mindshare must know when peers join and leave the group or when peers fail. It should also be possible to convey arbitrary status information about a peer. The other network services rely on the presence service to know when to establish a connection between peers.

3.5.2 Pipe ID’s and Naming

Peers connect to the “PeerGroup:Presence” propagate pipe, where ‘PeerGroup’ is the name of the JXTA group. This is similar to the IP:Port naming scheme used in TCP/IP. The unique ID for the pipe is generated by applying a hash function to the groups ID. In JXTA parlance is this referred to as a ‘well known id’ because all peers that join the group have the necessary information to generate it directly.

3.5.3 Presence Messages

A presence message contains only 2 fields; a message type and a payload. The message type is a single byte that denotes the purpose of the message. The remainder of the message contains the payload and can include binary data. There are three recognized message types; CONNECT (0), UPDATE (1) and DISCONNECT (2).

When a peer joins the group it transmits a ‘CONNECT’ type message. Peers that receive a CONNECT are expected to reply by broadcasting an UPDATE message. In this manner, a peer that joins the group will quickly become aware of any peers that are already online.

When a peer is leaving a group it sends a DISCONNECT message. This notifies the other peers in the group that this peer is being shut down in an orderly fashion and further communication with this peer will not be possible.

3.5.4 Managing Failures & Network Issues

In a centralized P2P architecture a server would maintain presence information for the group. The server would have a direct socket connection to each peer. If a peer failed the resulting socket failure could be detected. This event can then be used to inform the remainder of the group that a specific node had failed. When using a JXTA Multicast Socket, there is no ‘connection’ to break, therefore the socket closure method cannot be used to detect failures. Instead a ‘keep-alive’ or ‘heartbeat’ system is used. Each peer transmits a single UPDATE message once every 30 seconds. It is quite possible that the first evidence that members of a group will see from a new peer will be these keep-alive messages. The initial connect messages may be lost because the new peer may not have found the rest of the group yet. This is common when a peer fails to find the groups RDV peer and promotes itself to RDV status. This is handled automatically by the JXTA platform. At any time, when a peer detects another peer that it has not seen before it will respond with an UPDATE message of its own.

If a peer fails to receive a keep-alive signal from another peer for twice the duration of the keep-alive interval it can be assumed that the peer has failed or that its location has been lost. When this timeout occurs, the peer acts as id the other peer had disconnected. Receipt of a subsequent UPDATE message from the lost peer is treated as a new connection attempt and the two peers repeat the initialization protocol.

Note that the choice of time interval between keep-alive messages is chosen as a tradeoff between network traffic volume and application responsiveness. Users have become accustomed to the quick response times of server-based protocols such as Instant Messaging. It is important to make every attempt to emulate server based performance without destroying network throughput. Also the aim here was not to build a perfect Presence Protocol but to build something robust and serviceable for the prototype. Other protocols may prove more serviceable in the future such as the new Session Initiation Protocol [Johnston04, SIP05].

3.5.5 User Presence

The Presence protocol, as described thus far, is an implementation of ‘Peer Presence’. The other important element in a collaborative application is ‘User Presence’. In Mindshare some additional information is needed to identify the user at each peer. User presence information is sent inside the payload block as a

BEncoded message. All messages, except for disconnect, can include a payload. The payload sent by Mindshare has three fields.

- The ‘userid’ field is the unique User ID this identifies the current user at the originating peer.
- The ‘name’ field is used to associate a friendly name or ‘handle’ with the User ID. This is used in the buddy list display and the ownership field in the File Browser.
- The ‘join’ field contains the time in milliseconds that the peer joined the network. This is the remote peer’s perception of the time it joined, taken from the start of Unix epoch. This is used to calculate and display the peer’s uptime, e.g. “Bill has been online for 4 hours, 20 minutes”. This can be displayed even if the observing peer joined the group after the peer that sent the message.

3.6 Availability

Ulike the classic client-server model where each client must request specific information from the server, Mindshare uses an approach that distributes the availability of information from peer to peer. Peers broadcast a list of the items they have “available” for download. When a peer receives a tree update it “discovers” that it must download new files. Rather than query each of the other peers in the group to determine where it may find the information it needs, the peer can look in the “availability” cache that is has received to determine which peer, if any, has the files it needs. If more than one peer has the needed information, the peer can request some portion of the files from each of those peers, distributing the load for fairly.

Availability information is exchanged by peers when a channel is opened and updated throughout the channel's lifetime. Peers use a BEEP message to send availability messages. The receiving peer replies with a '`NUL`' message indicating that the response is not important.

There are two message types that are sent over the availability channel: Tree Availability and File Piece Availability.

3.6.1 Tree Availability

The tree availability message has two purposes. First is to advertise the availability of a particular version of a tree. The second is to advertise the availability of file pieces from within that tree. The message body contains three elements:

- `uri` – A text field denoting the URI of the user in the form '`mshare://userid@groupid/`'
- `version` – An integer field denoting the version of the tree being advertised.

- `bits` - A string field containing a bitfield encoded as `1`'s and `0`'s, such as '`1011110010001`'.

The URI and version fields identify the tree. All trees belonging to the same user will have the same URI. The version field increments with each change that is made. Thus, when a peer receives a tree availability message describing a higher, and hence newer, version of a tree, it knows to download that tree.

The `bits` field contains a complete set of availability data for the files described within the tree. Each bit represents one 512 kilobyte chunk of a file. There is one MD5 hash for each 512k piece of each file and the pieces correspond to the bits in the `bits` field. The file entries in a tree are sorted first by their path and then by the file's name. The bits in the `bits` field follow this ordering so it is trivial to match up an availability bit with the MD5 hash for that piece.

When a peer receives one of these messages for a tree that it does not have, it wipes out all stored availability information pertaining to that tree from memory. It also stops all data requests for file pieces from that tree. It caches the information in the `bits` field until it can download the new tree and process it. The old information is thrown away because it is now stale. Changes in the tree may have changed the mapping between the `bits` field and the old tree.

3.6.2 File Piece Availability

Availability information about individual files continues to be transmitted while peers download pieces. These update messages are sent out after each piece is successfully downloaded, verified and stored. Peers use this information to spread the network load and to form an efficient download strategy. The messages contain fields similar to a Tree availability message:

- `uri` – A string field denoting URI of the file for which the piece is available in the form: ‘`mshare://userid@groupid/path/file.txt`’
- `version` – An integer field denoting the version of the tree that this file belongs to.
- `index` – An integer field containing the index of the piece within the file.

3.7 Data Transfer

The data Transfer Channel is responsible for carrying all bulk data traffic between peers. The two kinds of data are whole tree documents and pieces of files. Tree documents have a higher importance and they are queued to be requested before any file pieces are.

3.7.1 Tree Request Messages

The request response pair for a tree is very simple. The tree is requested with a message containing:

- `uri` – A string field denoting URI of the tree being requested in the form '`mshare://userid@groupid/`'
- `version` – An integer field denoting the version of the tree being requested.

The response is a BEncoded tree document with no additional elements. BEEP remembers an identifier for each request so there is no need to add identifying data to the response.

This is a very simple request-response protocol. Trees are transferred far less frequently than files so the optimization of this protocol is not of great concern.

3.7.2 File Piece Request Messages

File Piece requests contain three fields:

- `uri` – A string field denoting URI of the file being requested '`mshare://userid@groupid/path/file.txt`'
- `version` – An integer field denoting the version of the tree this file belongs to.

- `index` – An integer field containing the index of the piece within the requested file.

Again, the response is the raw bits within the specified piece. BEEP takes care of the message tracking. Once the piece is received, it is hashed and checked against the appropriate hash in the tree. While the hashing and checking proceed in one thread, the request thread can issue another request and start receiving a response.

3.7.3 Efficiency

Efficiency in file piece requests is the key to good performance. File data will dominate all other data transfer on the network. The basic problem is to efficiently disperse a large volume of data from a single peer to the other peers in the group. The bottleneck in this problem tends to be the outgoing bandwidth of the peer that is distributing the data. Broadband connections in use today tend to be asymmetrical and this restricts the amount of bandwidth a peer has available for sending data to other peers. A perfect algorithm would disperse the data to all peers in the group in the time it takes the originating peer to transmit exactly one copy of the data.

The BitTorrent Protocol [Cohen04a] appears to solve this problem breaking large files into smaller ‘pieces’ and dispersing 8k ‘blocks’ of those pieces to other nodes on the network. The group of receiving nodes then trades the pieces amongst themselves, returning to the originator only when a chunk cannot be found elsewhere. In the real world this algorithm has proven effective for distributing very large files and scales well to swarms of peers much larger than Mindshare was designed to support.

In BitTorrent, a hash value is calculated for each piece of a file and a file containing these hashes is distributed to the swarm via a web server. Special software is installed on the web server to track the peers in the swarm so they can find one another. Mindshare already has equivalent functionality for both of these functions. The hashes are distributed in the Tree and the Presence Service allows peers in the swarm to find each other. The Availability Channel is used to advertise availability of individual pieces of a file.

As discussed by BitTorrent's designer in [Cohen04b], peers in these swarms tend to act in a selfish manner. Mindshare has some advantage over BitTorrent in this case. Mindshare peers are more likely to stay online after they have downloaded a file because they are waiting for updates from other peers. This means that a Mindshare network would have a large percentage of ‘Seeding’ peers when compared to a Bit Torrent Network. When a peer that has been offline for some time rejoins the group it will find that every peer has a copy of the files that it needs. This means that the peer can spread its requests across all of the peers in the group and be updated quickly without significantly affecting the resources of any one peer.

BitTorrent type file transfer was not implemented in the prototype but is an important component of the future work described in chapter 5.

3.8 User Interface Design

Mindshare is a graphical application and relies on its user interface to present a large amount of information to its users. The UI uses components and constructs that are familiar to users but are overloaded to present the extra information that Mindshare needs to convey.

The UI contains two main areas; the Buddy List and the File Browser. The buddy list is a concept that will be instantly familiar to users that already use any sort of Instant Messenger application. The File Browser is typical of many operating system browsers and includes a location bar that shows the current path and a list of files and folders available at that path. Each of these displays has an associated toolbar that can perform common functions relevant to that display area.

3.8.1 Buddy List

Other users appear in the Buddy List (see Figure 3.3) in a Tree categorized by the group that they belong to. Selecting a group or a buddy in that group will cause the file browser display to change and display the files for the relevant group. This is the primary reason why the buddy list is situated on the left side of the screen. Users typically move from left to right as they recognize information, similar to how they read. All information in the right panel is relative to what is selected in the buddy list on the left. This arrangement allows the user' to orient themselves when they glance at the display without having to look back and forth across the screen.

Icon	Function
●	Represents a user who is online
○	Represents a user who is offline
👥	Represents a group in the buddy list

Figure 3.3: Buddy List Icons

Other users appear by name and have an icon that depicts the user's state. The currently supported states are online and offline. The users name also dims when they are offline. In the future this display may support more states as the presence support is enhanced. One example of this is an "away" state that lets the user know that someone is online but not physically available at their computer.

Icon	Function
➡	Enter the selected group
⬅	Leave the selected group
👤	Invite someone to join the group
👥	Create a new group

Figure 3.4: Buddy List Toolbar Icons and Functions

The toolbar (see Figure 3.4) attached to the buddy list provides three buttons. The Enter/Leave button allows the user to enter and leave individual groups, similar to entering a conference room. This allows the user to choose to end participation in an individual group without closing the program. The Invite User button allows the user to create an invitation to send to someone else who is not already a group member so they can join the group. The Create Group button is used to create a new group.

3.8.2 File Browser

The File Browser is intended to function as much like the user's native operating system file browser as possible. Thus, the File Browser uses the host operating system's icons to display each file's type correctly. The address bar shows the current folder. It also has icons (see Figure 3.5) for navigating up the folder Tree and to return to the root of the Tree.

Icon	Function
🏠	Navigate back to the root of the file system
⬆	Navigate upwards to the parent directory
📁	Create a new folder in the current path
✂	Cut the selected objects
📄	Copy the selected objects
📋	Paste the selected objects
❌	Delete the selected objects
🖍 ▼	Change the highlighting

Figure 3.5: File Browser Toolbar Icons & Functions

The toolbar at the top of the pane is duplicated as a context menu that can be accessed with a right click on the mouse. This gives two ways to perform the identical set of functions. In this case choice is not a bad thing. Most applications give the user this choice and users have individual preferences that need to be catered to. The functions on this toolbar include the familiar file manipulation commands; New Folder, Cut, Copy, Paste, Rename and Delete. The additional commands specific to Mindshare are Give and Highlight. The Give command initiates the process for changing ownership of a file from one user to another. This is covered later in the section on Trees. The Highlight command can be used to change the display so that files belonging to a specified user appear with bold text. This can assist the users in visualizing who owns the files they are looking at. To actually change what user us being actively highlighted you select the user from the buddy list. The other mode supported will highlight the difference between files that are available locally and those that are still currently being replicated.

One column in the file browser shows the name of the owner of each file. Folders with the same name that appear in multiple Trees are specially annotated to show a comma separated list of owners. This alerts the user as to whose files they will find in that folder. The other columns show information common in most file browsers, including the size and type of the file, date it was entered into the system, date it was last modified and version of the file. The version number tells the user the number of times a file has been changed.

Chapter 4 Software Engineering

In this chapter we will discuss the development of the prototype from a Software Engineering perspective. We will examine the development process that was employed and applicable stages of the software lifecycle from design to deployment. We will also detail the choices that were made regarding tools and libraries used during development.

4.1 Development Model Overview

The development model for this project was a mix of Extreme Programming and Iterative Development. From the Extreme Programming world we took the idea of incrementally designing, constructing and testing small batches of functionality to the program. We used Unit Testing and the JUnit [Niemeyer03] test framework to create and automate those tests. Each incremental piece was designed with a minimal understanding of its impact on the rest of the system. This is counter intuitive to the ideas suggested by the Waterfall model [Pfleeger01] but it proved advantageous on this project. Because the capabilities of the underlying network layer continues to change and mature, the design of Mindshare continued to evolve as the software was being built. A procedure or technique that was inefficient one week might become viable the next and cause the redesign of a component.

The continuous change caused many refactoring activities. We practiced ruthless refactoring. No part of the software remained completely unaffected or was immune to change. The most common refactoring performed was renaming of a Class, Method or data member followed by moving functionality from one package to another. This was supported by two key advantages. The first was the use of a refactoring development environment, Eclipse, which can automatically perform many common but tedious refactoring tasks. The second was the small team size, just one developer, affected by these changes. In a larger project sweeping changes might have broken the build or upset other developers. With a single developer these issues are not a problem so it made sense to take full advantage of this opportunity for agility. The final result is higher quality software that is easier to understand.

In the book “The Mythical Man Month” [Brooks72], the author suggests that you should “Plan to throw one away, you will anyhow”. This turned out to be very good advice and we threw away a significant amount of code on this project, although we actually kept every file that we threw away in case it was useful for later analysis. To date, the project consists of approximately 100 source files and we discarded 70 additional files, so over 1/3 of the files were eventually replaced. Some components, like Peer Presence, were re-written no less than three times.

4.2 Requirements

Thorough the project we maintained an overall guiding vision of the system's purpose and capabilities. We divided the overall design into small, easily manageable, chunks to plan in detail. More detailed requirements were created for each chunk and the chunk was then implemented. This was an iterative process such that we only planned and implemented one set of functionality at a time.

If we found that we lacked some knowledge necessary to implement the requirements we would often create a Spike Solution [Wake01] to explore the requirement in greater detail and to access its feasibility and associated risk. For example, there was an original requirement that the Group Share be viewable in a JTree widget to give the user their current location in the Tree. To be confident that we could meet this requirement we needed detailed knowledge of the Java Swing API [JavaAPI05] for using JTrees. We built a spike to test this and found it to be extremely difficult. This caused the requirement to change and resulted in the location being displayed in a browser-style address bar instead.

Another Spike Solution was created to answer a simple question; "Could we make BEEP work over JXTA?" We constructed a small program that retrofit the BEEP echo server/client to a JXTA application. The result found a bug in the close semantics of `JxtaSocket` that had to be fixed before work with BEEP could continue. Ultimately the answer was "Yes, we can use BEEP" but had we immediately integrated the code into the application spotting the bug in the API would have been much more difficult.

Requirements changed constantly throughout the project and we handled this very well. We minimized the amount of redesign by restricting our efforts to creating detailed requirements for a single component per iteration.

4.3 System Architecture and Design

In this section, we will discuss important design issues, such as Design Patterns [Gamma95] and major components.

4.3.1 Mediator Design Pattern

The System Component Diagram (Figure 4.1) shows the relationships between the major system components of the system. The system's global architecture uses the Mediator Design Pattern [Gamma95]. This pattern's goal is to promote low coupling between the individual components it mediates. What is not obvious from this diagram is that it also acts as a thread boundary, allowing events from different threads in different components to interact without deadlock.

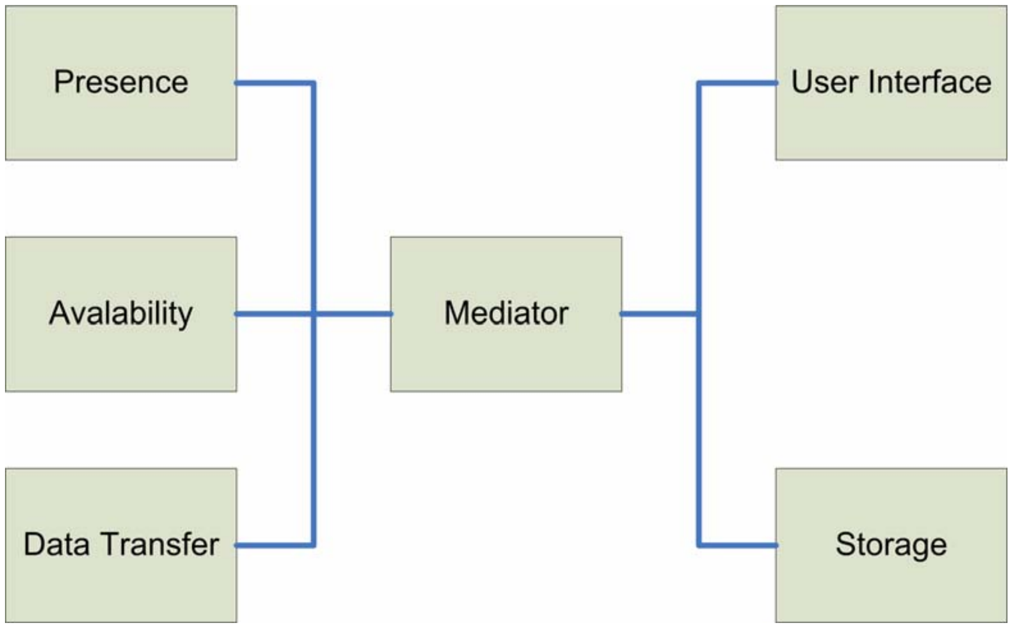


Figure 4.1 System Architecture Diagram

4.3.2 Network Components

The Presence, Availability and Data Transfer components are the components the system uses to interact with other peers. Each of these components sends and receives information. They receive information in an asynchronous manner and they deliver this information in the form of events to a listener interface. The mediator component implements these interfaces, allowing it to receive and process these events. This is known formally as the Observer Design Pattern [Gamma95]. This potentially allows several components to receive events from a single component and allows the Mediator to be broken into several objects that perform different tasks on the same event.

4.3.3 Model View Controller Pattern

The application makes use of the Model-View-Controller [Gamma95] pattern for the User Interface. This pattern separates the responsibility for displaying data from the actual data model. In our case the users, groups and trees are the model layer. Where these components meet the user interface there are controllers that interpret user interaction and call appropriate functions in the model. These controllers are all part of the UI package. The view is represented by Java Swing components rendered from SwixML [SwixML05] documents. This topic will be discussed further in section 4.5.3. In Mindshare, the model state and data can change due to events received across the network. This means that the Model must be able to update the view asynchronously. Many of the controllers extend a common Java class that gives them an event queue capability and allows them to process model events in a separate thread. This reduces the need for synchronization and makes the interface more responsive.

4.3.4 The Storage Component

The storage component uses the host file system to store Trees and files. This component is also responsible for hashing imported files and for copying files.

Two logical storage layouts could have been implemented. The easier to implement would have stored each user's files in a separate directory. This segregation allows the user to determine the owner of a file even when the program is not running. It also avoids problems with name collisions. This scheme does have one key disadvantage. Files from different users are not in the same folder, as they are depicted in the display, so they cannot be used together by external programs. This would be an issue for code compilation or photo slide shows. A better solution would be to build a single file hierarchy that mirrors the display in the File Browser.

Placing files from several users in the same folder makes it hard to tell what files belong to whom. To assist the user (and to prevent inadvertent changes) the files received from other users are marked as read only by setting the host file system's `read only` flag. The user can open these files but if they attempt to alter the files they will be warned that the files are *read-only* by the host operating system. If the user chose to override the warning and make a change, the Mindshare client would be forced to synchronize those files the next time it ran, possibly deleting the user's changes.

The other major problem is with file name collisions. The host file system will not allow two files to exist with the same name at the same path. Mindshare has to support this duplication because it cannot prevent the creation of duplicate file names on loosely coupled machines. From the programs, and the users, point of view the files are distinct because they belong to different people. The files may be created at different times and it may be hours or days before the collision is discovered. The software must handle this gracefully and without error.

The actual implementation in the prototype uses the first method discussed above. This method was chosen because it was the safest (minimal chance of implementation error) and quickest to construct. We acknowledge that the storage system is not ideal for a collaborative environment and is likely to evolve. To support this evolution the components details are shielded from the rest of the program by an interface. The Storage interface supports operations based on file and Tree URIs. The URI contains the three pieces of information necessary to store or access a file, the user id, the group and the path. This approach allows a Storage implementation module that can store the files in a way that is opaque to the user. This will allow for future improvement without changes to the rest of the program.

4.4 Risk Management

Working on any new software system carries some inherent risk and it is up to the designer to identify and mitigate those risks. True Peer to Peer systems and overlay networks are a relatively new field in Computer Science history and well understood examples of such systems are still developing. There were many sources of risk on this project, some obvious, some hidden. Identifying risk was an important task throughout the project.

The decision to allow Mindshare to rely on third party libraries to supply the underlying networking facilities was difficult because of the associated risk. The JXTA library in particular is under active development at the time of this writing. When the Mindshare project commenced, the JXTA library was very immature and there was a genuine risk that it would not perform as required. If there was a failure in JXTA the only vector for resolution, aside from abandoning it entirely, was to become involved in its development process, therefore we have become active in the JXTA developer community and contributed bug reports and bug replication code to the project.

The other risk management activities on the project focused around identifying potentially complex areas of the software and prioritizing their development. This resulted in multi-threading being an initially identified risk. Utility code was developed early in the project to allow for many threads to co-operate with the user interface to avoid performance issues in the User Interface.

4.5 Software Libraries

Many well-documented open-source libraries are available for developers but no single package provides the full range of features needed to build peer-to-peer applications. Mindshare's peer management and file transfer components are based on the JXTA [Oaks02] and BEEP [Rose01] libraries which we will describe in more detail below. Other libraries are needed to make a fully functional system. Smart choices in libraries helped to reduce the workload and indeed make the project possible for a single programmer.

4.5.1 JXTA

The main goal of this prototype was to prove that something useful (and legal) could be done with P2P software. Developing a full P2P infrastructure was not a main goal. Having to develop a sophisticated P2P toolkit was beyond the scope of the project. During the initial design phase a search began for an off-the-shelf library to perform key P2P functions. JXTA was the only candidate that showed much promise in this area.

JXTA fulfilled the core requirements of locating peers across the internet and grouping peers together. JXTA also brought some unexpected bonuses. JXTA can connect from behind NAT and firewalls using HTTP. Peers that would be unreachable otherwise can be reached by routing messages across other peers on the network. JXTA also uses XML for all messages it sends between peers which initially appeared to fit well with the goal of using XML for all messages in Mindshare. All resources in JXTA are described using XML documents called Advertisements. JXTA was not designed for any particular application but as a general P2P toolkit for connecting peers. JXTA is a “super peer” network, this means that for any group of peers there is at least one peer that acts as the coordinator. In JXTA the super peers are called Rendezvous Peers or simply RDV’s. JXTA allows for multiple super peers to spread load and for redundancy. Any peer can be an RDV and the network allows for peers to be promoted to RDV status dynamically as needed.

4.5.1.1 Peer & Group Organization

JXTA organizes all peers into a hierarchy of groups. All peers belong to a global group called the Net Peer Group. Peers can then join other groups that are logically below the Net Peer Group. It is possible to recursively form subgroups but JXTA does not support a mechanism for accurately specifying the path to a group below the second level. Each group has its own super peer, the Net Peer group is the largest group and it has several RDV peers. Each application is encouraged to arrange all of its peers in a single group below the Net Peer Group. If further subdivision of peers is needed then groups should be formed below the application level group. Mindshare follows this advice in creating groups. A single Group Advertisement cannot contain the exact path of a group three levels down in the group hierarchy. This makes it essential for every Mindshare peer to also have a copy of the application level group advertisement. It is interesting that a primary usage pattern is not directly supported by JXTA. This is not a failure of the implementation but of the actual design of the JXTA protocols.

Having to support a three-level group hierarchy requires a Mindshare peer to be prepared to become an RDV at both the application and individual group levels. Implementing this correctly involved much more code than joining a single group by using a single advertisement.

4.5.1.2 Locating Peers

JXTA provides a built-in mechanism for locating groups and peers called the Discovery Service. It is advised [Parker04] that before accessing a resource it first be ‘discovered’ using the Discovery Service to prove its existence. Group discovery was not very useful because information about groups is delivered to each Mindshare Peer in the form of an invitation. Mindshare peers know the ‘address’ of the groups they wish to join and so discovering these resources is not necessary.

One of the first development tasks was to test peer location. Before peers can begin to coordinate they must first know what other peers are available to coordinate with. In Mindshare’s case all peers are interconnected at all times. Initially the Discovery Service was used for locating peers. This worked well in testing on a single machine or over a LAN, but when testing was performed over the Internet with peers located in different networks, the results were poor. Discovery could take several minutes and often returned no results, even when it was known that peers were indeed active.

The Discovery service was not designed for Peer Presence. Discovering a peer only tells you that the peers advertisement is cached somewhere on the network. It cannot tell you if that peer is still online or when the peer goes offline. In groups that have more than one RDV peer, Discovery queries are not guaranteed to reach all peers. Queries only propagate over a single hop from their source. This leads to a lack of repeatability in testing. The group needs several minutes to organize and to find routes between all peers for proper query propagation. There is no way to tell if this point of proper organization has been reached. Ultimately the uncertainty of the Discovery Service makes it unsuitable for peer discovery. Propagated pipes were used to get around this limitation because propagated messages are not dropped like Discovery Queries. This required a major development investment. It also means that JXTA does not directly support one of Mindshare's major requirements. This was not apparent from the JXTA documentation.

4.5.1.3 Library Immaturity

JXTA is at version 2.0. This usually suggests that a program or library is mature software, has been thoroughly tested and found to be satisfactory. However, the JXTA libraries (particularly the Java implementation of JXTA) do not exhibit these characteristics in practice. JXTA's documentation is still being developed and frequent API changes mean that the tutorials are often out of sync with the current version of the software. The examples are simplistic, do not embody good usage patterns and mislead the novice developer.

During development we uncovered a bug in the `JxtaSocket` class that prevented a socket from closing in a timely manner even when connected through the loopback interface. Providing automated tests for JXTA is difficult not only because peers are difficult to configure and start but because the network cannot be controlled in a repeatable manner during the test. Many problems and potential bugs may be falsely ascribed to network difficulties. Reproducing an effect is difficult or impossible because the network topology in which the error occurred is not known and is not under the control of the tester.

It appears that JXTA has a long way to go before it will be a stable platform. In the Chapter 5 we will discuss alternatives to JXTA and what it an alternative might look like.

4.5.2 BEEP & BEEP Core Java

BEEP, the Blocks Extensible Exchange Protocol [Rose01], is defined by IETF draft RFC 3080. Its reason for existence is to make building application layer protocols easier by providing common functions and features that all application layer protocols need. BEEP makes it easy to construct request-response type protocols. It can automatically split messages that are too large to fit in a single packet and deliver them to a remote peer. It also supports the concept of 'channels' which are separate tunnels over a BEEP session. Multiple channels can be in use at the same time and their traffic will be multiplexed over a single socket. BEEP has been used in a number of projects and products including the Intrusion Alert Protocol [Betser01], Reliable Syslog [New01] and SubEthaEdit [SubEthaEdit05].

BEEP's ability to process large messages and to support multiple channels made it a very attractive basis for the application layer protocols in Mindshare.

Using channels eliminates the need to have multiple socket connections between peers. JXTA sockets (called pipes) require significant time to connect and so it was attractive to use a single socket to decrease the time it takes for one peer to connect to another.

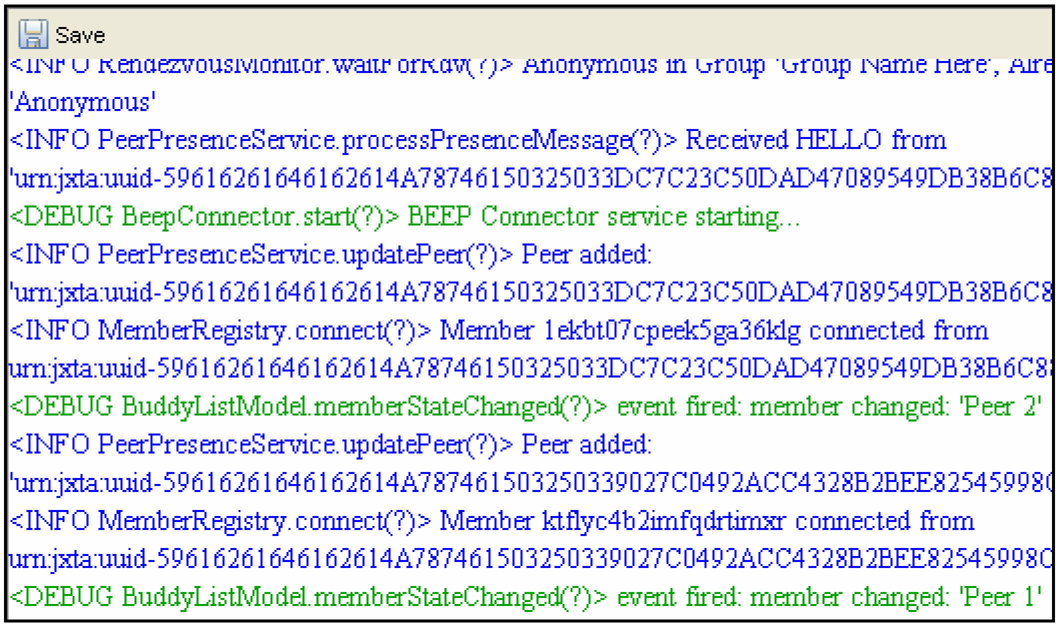
The Java BEEP Core library [JavaBEEP05] was easily adapted to work with the `JxtaSocket` class. From there, the application layer protocols for Mindshare can be designed using the BEEP connection. Using a standardized networking layer partially shields Mindshare from JXTA and could facilitate porting of the application to another overlay network in the future.

4.5.3 SwixML

SwixML [SwiXML05] is an XUL (XML User-interface Language) [Bullard01, XUL05] motor for Swing. SwixML uses XML documents to describe the Swing components and layout of a user interface. The SwixML motor then 'renders' these documents by creating Swing components at runtime. This approach significantly reduced the component creation and layout code that would have been written in a standard Swing application. The remaining GUI code had better organization and could be focused on user interaction and event handling. There are no unnecessary references to components that do not interact with the user, such as frames or layout managers, in the code. Functionality can be split across multiple, small, focused controller objects making the code more modular and adhering more closely to the Model View Controller Pattern.

SwixML also facilitated rapid prototyping of UI layout ideas. Changes to the design often did not involve any code changes. This ability to 'play' allowed for the refinement of usability and freed the developer to easily redesign bad interface design choices. Use of an XUL motor was a great advantage on this project and saved much time and effort, reduced complexity and increased quality.

4.5.4 Log4J



```
<INFO RendezvousMonitor.waitForRdv(?)> Anonymous in Group 'Group Name Here', Alre
'Anonymous'
<INFO PeerPresenceService.processPresenceMessage(?)> Received HELLO from
'urn:jxta:uuid-59616261646162614A78746150325033DC7C23C50DAD47089549DB38B6C8
<DEBUG BeepConnector.start(?)> BEEP Connector service starting...
<INFO PeerPresenceService.updatePeer(?)> Peer added:
'urn:jxta:uuid-59616261646162614A78746150325033DC7C23C50DAD47089549DB38B6C8
<INFO MemberRegistry.connect(?)> Member 1ekbt07cpeek5ga36klg connected from
urn:jxta:uuid-59616261646162614A78746150325033DC7C23C50DAD47089549DB38B6C8
<DEBUG BuddyListModel.memberStateChanged(?)> event fired: member changed: 'Peer 2'
<INFO PeerPresenceService.updatePeer(?)> Peer added:
'urn:jxta:uuid-59616261646162614A787461503250339027C0492ACC4328B2BEE82545998C
<INFO MemberRegistry.connect(?)> Member ktflyc4b2imfqdrtimxr connected from
urn:jxta:uuid-59616261646162614A787461503250339027C0492ACC4328B2BEE82545998C
<DEBUG BuddyListModel.memberStateChanged(?)> event fired: member changed: 'Peer 1'
```

Figure 4.2 Sample Log Output in Mindshare

Throughout the prototype we used the logging framework Log4J [Gulcu03] to provide logging facilities for the application. The application contains a log viewing window so that log output can be observed when the application is being run outside of the development environment. The log output uses colors for different types of events. More critical events appear in orange and red while informational and debugging events appear as blue and green respectively. Within the development environment the system error output is captured by Eclipse, which displays this logging output. The use of a configuration file compiled with the application allowed the developer to configure log message output from individual components and message priorities. This was an invaluable aid in debugging and was often the best source of valuable clues.

4.6 Testing

Testing activities for the prototype implementation focused on two program areas; network testing and non-network testing. Each type of testing presented its own challenges and required a very different approach.

4.6.1 Offline Testing

When testing the application off the network we used a variety of standard techniques. We used JUnit to write and run unit tests of as much of the software as possible. Eclipse has built-in support for JUnit and can discover new test classes and run all the tests in a single operation. Eclipse can link back to the code that caused a test to fail to aid in debugging.

We used exploratory testing and scenario testing to test UI interaction. Log4j output was useful in tracking the bugs found by exploratory testing.

4.6.2 Network Testing

Network Testing covers all activities that require the program to be connected to the JXTA network to facilitate testing. Working with JXTA to automate or at least accelerate testing proved to be very difficult.

JXTA is a dynamic network environment and Mindshare was intended to operate in a dynamic environment. Throughout development the public JXTA network exhibited instability. This instability is widely acknowledged by the developers and efforts are ongoing to improve network stability. The main problem was that any peer could become an RDV in the net peer group. These are the super peers that route traffic for the entire network. Some developers run tests on the public network that caused these super RDV peers to start and stop over the course of a few minutes. Each time these peers join the network there is general upheaval. Recently the option for peers to become super RDV’s has been removed and this has greatly improved network stability. This came too late to help with most of the testing however.

JXTA peers choose a random RDV peer at start up. This is one of the causes of non determinism in JXTA. If, by coincidence, all the peers in a test instance pick the same RDV it is much more likely that they will establish communication quickly. If they choose a wide dispersion of RDV peers it can take much longer for the peers to find each other and begin the test. It is even possible that a peer might connect to a short lived RDV that would terminate during the test. This sometimes caused peers to fail in a non-repeatable way.

Peers being tested on a LAN can use multicasting to locate each other reliably. This can aid and accelerate testing but can also lead to results that do not work outside of the LAN. In particular, timeout values for a LAN are much shorter than for peers that are physically dispersed. Choosing timeout values based on LAN testing will give poor results in real world conditions.

When Mindshare’s development started, JXTA required the user to interact with a GUI to configure each peer. This was a major obstacle to automated testing. Each peer needed to have slightly different configuration to allow several peers to run on the same computer. As a result, test automation focused on improving the performance of the build and deployment process. This reduced the time it takes to set up each test scenario. Testing was then carried out manually by the tester. This is not ideal because it makes test repeatability partially dependent on consistent execution by the tester. In the case of JXTA the lack of deterministic behavior negatively affects the results.

The time it takes to run a particular test is dominated by peer startup and connection times. Peers need to have long timeouts for connecting to each group. In practice it takes between 1 and 2 minutes for a peer to load and connect to its assigned group before a test can start. For most tests, coordination is required from all peers. They must be connected and locate each other before the feature can be tested. New techniques need to be devised to test groups of peers. Existing testing tool don’t work well at exposing bugs found in a non-deterministic environment.

4.7 Deployment

For the prototype we made use of the Ant [Ant05] build system to automate deployment. An Ant script can build the entire system and package all resources into a structured Zip file and transfer this file to a website for download. The Ant script also increments the build number so that each build is uniquely identifiable. The build number and version number are added as properties to the Java Archive (Jar) that contains the Mindshare executable. Mindshare picks up the version and build number from the Jar properties and displays them in its title bar. Mindshare can also be run from inside the IDE development environment where Jar properties are not available and it handles this situation silently.

The Zip package that Ant creates contains a folder structure necessary for Mindshare to run after the Zip file has been decompressed. When the application is first run the Configuration Subsystem stores required global variables in a configuration file in the root of the install directory, called “config.cfg”. New versions of the software can be installed directly over old installations. The Configuration System will analyze the existing configuration file and make any changes necessary to accommodate the new version while keeping any old configuration settings.

The software detects the path it is being executed from at startup and uses this path to locate resources, such as Icons and UI files, in folders relative to that path. This allows the installation to be moved on disk without adverse effects. The application can also be completely and removed by simply deleting a single folder.

4.8 Maintenance

This project will continue after this paper is published. The source will be released under an Open Source license to solicited contributions and improvements from others. We intend to follow the “Release early, release often” strategy of many successful Open Source projects. There are many ideas that were not practical to implement in the prototype and many opportunities for improvement. The source code includes comments that conform to the Java Doc specifications and also includes Unit Tests to aid in system evolution. We hope to see the system improve and gain widespread use with our target user base.

Chapter 5 Conclusion

In this paper we have presented the design of a system suitable for collaboration between a small group of people and given motivation for our specific design choices. We have detailed the design, and architecture of the system. In this section we look at the future of the system and what was learned during its construction.

5.1 Future Work

Development of the Mindshare system will continue long after this thesis is presented; the system described in this paper is merely a prototype. We feel that the requirements are applicable to the real world and that we will have the strong support of users to continue development and evolution of the system. To produce a prototype for a thesis requires a focus on new and unproven areas of computer science. Future development will fill in the gaps left behind and provide functionality that users already expect from mature software systems.

5.1.1 Security

Mindshare doesn’t have security implemented at the time of this writing. There are some fairly clear paths towards securing the system both from attack and interception. At the moment, two core security problems exist with the software. The first is keeping unauthorized users out of a group; the second is securing traffic between group members.

5.1.1.1 Membership Authentication

Currently anyone who obtains a copy of the group advertisement can use the client to join that group. Group advertisements can be found using the Discovery service and so it would be easy to infiltrate a group by searching for one of these advertisements.

A technique is needed to differentiate between members and non members in secure and distributed way. One way to implement this would be using public key encryption. Each member of the group would have a key pair. The public keys would be distributed to other group members by a distribution mechanism similar to tree synchronization.

To add a member to the group, an invitation would be generated by any current group member and encrypted using their private key. This invitation would include the details about the invited user, their user ID and an encoded lifespan during which the invitation is valid. The peer that generates the invitation is known as the ‘sponsor’. Any peer in the group that knows the sponsor’s public key can verify that the invitation is valid. They can also test that the invitation has not been used yet by checking for the embedded user id in their list of known members.

The invited peer can then initiate a secure transaction with any peer in the group to become a member. This transaction would involve a member authenticating the invitation. The new member would generate its own key pair and the authenticating peer would distribute the new member ID and public key to the rest of the group.

5.1.1.2 Tree Security

Trees would be distributed in a compressed and encrypted format. Zip compression could be used to significantly reduce the size of distributed trees because trees are XML documents. The zipped data would then be encrypted using the member’s private key. This would allow secure distribution of trees by third parties and protect tree data from attacks coming from within the group.

5.1.1.3 Transport Security

Securing transport of data between peers is also vital to protect the network from eavesdropping. Several opportunities exist to secure the point-to-point communications. JXTA provides for secure pipes using SSL. BEEP also provides for sessions using TLS (an SSL derivative) and challenge-response authentication. BEEP currently offers the most flexible and programmer friendly solution. Data that remains unencrypted at the `JxtaSocket` layer does not reveal anything about the payload other than size and source peer.

Multicast data is another problem. No lower level support exists for encrypting multicast communications. One solution is to avoid using multicast for anything sensitive. Mindshare already takes this approach. User ID’s and tree versions are sent via multicast but no tree data is sent. This data would only provide a potential attacker information on whose Trees are being updated. However, much of this information could also be inferred from traffic pattern analysis of a Mindshare network even if the communications were encrypted.

5.1.2 Distributed Download

Mindshare could improve its file download speed and resource usage efficiency by allowing peers to download different blocks of a file from several sources. BitTorrent [BitTorrent04] has already proven this to be a powerful approach. Minimal changes would be needed to allow Mindshare to use the BitTorrent method.

Trees would need to include block size information and a set of SHA1 hashes along with the existing MD5 hash and length information. The file transfer protocol GET operation would have to be parameterized to reference each specific block index to download. The ‘Tracker’ (program that tracks all peers downloading a file) is not needed in Mindshare because all peers are actively downloading all files. A more important requirement is that Mindshare clients must build a map of which peers have specific blocks from a given file. A command to get a block map for a specific file would then be necessary.

5.1.3 Transferring Ownership

A peer cannot directly edit the Trees that it receives from other peers and this limits some activities that the collaborating users might wish to perform. To work around this restriction, one user could give up ownership of a file to another. To transfer ownership both parties need to be online and have an up to date copy of the file in question. The current owner initiates the transaction based on a request from the peer that desires ownership. We have already included some UI elements in the prototype to support this interaction.

The transfer would be a multi-part exchange including stages for verifying that the receiving peer has a complete copy of the file(s). This transfer would occur over a separate BEEP channel. The receiving peer would use the `bits` field for the files being transferred to create new entries in its own tree with the same names as they had in the original owner’s tree. The peer that is granting ownership waits for acknowledgement of the transfer (which comes when the recipient transmits its updated tree) and then removes the file(s) from its own local Tree. Because the original owner does not delete its file(s) until the full transfer is acknowledged, interruption of this multi-part transaction will not result in a data loss.

Users can already accomplish this by copying files and importing them back into the program. This manual process is more cumbersome but no less effective.

5.1.4 Database Backed Storage

As we discussed in 4.3.4, the current design of the storage component is not ideal. The use of a database would be the best solution for storing files in a way that mirrors the display in the application. The database is necessary to resolve ownership and to track naming collisions. When a naming collision occurs it will be necessary to rename the file on disk and there may be no way to successfully ‘guess’ the owner of a file if this data is not recorded persistently somewhere.

5.1.5 Usability Testing

Usability was a major focus area for this project. Given the final state of the prototype it was not yet practical to deploy it for real-world use. As soon as the above enhancements are made the prototype could undergo usability testing. A good way to do this would be in an academic setting where a group project is required. They would use the software for their collaboration needs and report on their experience. Observations could also be made in a lab setting where user interactions and usage patterns are recorded. Based on those results, the user interface could be enhanced and features selected for future development could be prioritized.

5.2 Lessons Learned

5.2.1.1 Improved Testability

Testing the network interaction of a Peer to Peer application is not yet well-supported by standard testing tools. In developing the prototype we focused on two primary techniques. First we used unit testing to validate as many of the components as possible offline. This reduced the probable source of a fault to a limited number of components that interact with the network. Secondly we used logging facilities to produce accurate transcripts of the interaction at each peer those logs were consulted after test runs to aid in debugging.

Network testing was not an automated process and testing a protocol or peer interaction is a very time dependant process.

5.2.1.2 Alternative P2P Networking

JXTA adds a considerable amount of extra code to Mindshare and also negatively affects performance and reliability, thus its refinement or replacement is a top priority. JXTA is attractive because it allows peers to connect from common consumer connections, many of which are behind firewalls or Network Address Translation. This is because JXTA is an ‘overlay’ network which can route communications between peers over different connection types.

JXTA does not deliver on all of its promises though. Interestingly, a server is required to bootstrap peers. The server has a simple list of the running super peers that an edge peer uses to connect to the network. This bootstrap server could be enhanced to eliminate much of the JXTA core. If the bootstrap server had the addresses of every peer this would significantly cut down on startup time and enhance reliability. The amount of data the server would have to store would be small and the server could be written in several languages. A single server could scale to server thousands of clients.

The hierarchy of groups could be replaced by a flat group namespace using cryptographically generated group ID's. Group security could be assisted by storing keys and passwords on the server. This would also solve the problem of removing a group member. A central server could also hold invitations so that they could be claimed via e-mail.

Of course, increasing reliance on a server raises all the issues related to server based systems. The main attraction of this scheme is that it is simple to implement. The server does not store any of the files for a particular group and requires very little bandwidth. A single server could handle all of the peers on the Internet. An example of a similar system is the SHOUTcast [SHOUTcast05] list server at Winamp.com or the Bit Torrent Tracker. A single server lists all peers in the group without needing significant bandwidth or storage space.

Peers would connect to the server at startup to get the peer addresses of the other group members. From there connections are initiated using BEEP between the peers. Future development would allow for overlay communication between peers over HTTP. The HTTP relays would be published by the server.

5.3 Conclusion

In this thesis, we have presented the design and development of a prototype collaborative system for small groups based on peer to peer networking technology. We discussed the issues that arise in designing and developing P2P systems and present useful solutions to these problems. The prototype design is very robust, allowing peers to collaborate even when they are not continuously online. The prototype achieves these goals without the need for a centralized server, which s required by other collaboration solutions, such as e-mail. The Metadata Tree based approach is unique and provides for a fault tolerant system with version control in a fully distributed environment. The prototype highlights the need for usability in software targeted at this user segment. It has strong user interface design and deploys to any platform that runs the Java Virtual Machine.

References

[7Zip05] 7-Zip website, http://www.7-zip.org/

[Anderson02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, Dan Werthimer, "SETI@home: An Experiment in Public-Resource Computing", Communications of the ACM, Vol. 45 No. 11, November 2002, pp. 56-61

[Ant05] The Apache Ant Project, http://ant.apache.org/

[Beck99] Kent Beck, "Extreme Programming Explained: Embrace Change", Addison-Wesley, 1999

[Betser01] Betser, J., Walther, A., Erlinger, M., Buchheim, T., Feinstein, B., Matthews, G., Pollock, R., Levitt, K., "GlobalGuard: creating the IETF-IDWG Intrusion Alert Protocol (IAP)", in Proceedings DARPA Information Survivability Conference & Exposition, 2001

[Breidenbach01] Breidenbach, Susan, "Peer-to-Peer Potential", Network World, July 30, 2001

[Biddle03] Peter Biddle, Paul England, Marcus Peinado, et al. "The Darknet and the Future of Content Protection", Lecture Notes in Computer Science, Springer-Verlag Vol. 2696, pages 155-176, 2003

[Bullard01] Bullard, Vaughn, Kevin Smith and Michael C. Daconta, "Essential XUL Programming", Wiley Publishing, 2001

[Cederqvist02] Per Cederqvist, "Version Management With CVS", Network Theory Ltd. 2002

[Cohen04a] Cohen, Bram. “BitTorrent: Protocol Specification”, Nov 2004. "bittorrent.com/protocol.html"

[Cohen04b] Cohen, Bram. “Incentives Build Robustness in BitTorrent”, Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, May 2003.

[CVS04] Concurrent Versioning System. “CVS Product Overview”, "www.cvshome.org", Nov 2004.

[Distributed05] Distributed Encryption Cracking Tools, "www.distributed.net", 2005

[Dourish92] Dourish, Paul and Victoria Bellotti, "Awareness and Coordination in Shared Workspaces", in Proceedings, Conference on Computer Supported Cooperative Workgroups, 1992

[Ellison03] Ellison, Carl and Steve Dohrmann, "Public-Key Support for Group Collaboration", ACM Transactions on Information and System Security, Vol. 6, No. 4, pg. 547-565, November 2003

[Edwards02] Edwards, W. Keith, Mark W. Newman, Jana Z Sedivy, et al., "Using Speakeasy for Ad Hoc Peer-to-Peer Collaboration", in Proceedings, Conference on Computer Supported Cooperative Workgroups, 2002

[Farrell04] Farrell S., Ed., "RFC3767 Securely Available Credentials Protocol", June 2004

[FEPWC98] Federal Email Postmasters Working Committee. “Draft Policy on Permissible Email Attachment Size” August 1998.

[Gamma95] Gamma, Erick, Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns", Addison-Wesley Publishing, 1995

[Groove05] Groove Networks, Inc., website, "www.groove.net", 2005

[Grouper05] Grouper website, "www.grouper.com", 2005

[Gulcu03]Gulcu, Ceki, “The Complete Manual - log4j: The Reliable, Fast and Flexible Logging Framework for Java”, QoS.ch, 2003

[Halepovic02] Halepovic, E., Deters, R., "Building a P2P forum system with JXTA", in Proceedings, IEEE Conference on Peer-to-Peer Computing, 2002

[JavaBEEP05] Java BEEP Core Libraries web site, "sourceforge.net/projects/beepcore-java", 2005

[JavaAPI05] Java SDK Reference web site, "java.sun.com/reference/docs/index.html", 2005

[Johnston04] Johnston, Alan B., "SIP: Understanding the Session Initiation Protocol", Artech House, second edition, 2004

[Kaner99] Kaner, Cem, Jack Falk and Hung Q. Nguyen, "Testing Computer Software", 2nd Edition, Wiley Publishing, 1999

[Kaner01] Kaner, Cem, James Bach and Bret Pettichord, "Lessons Learned in Software Testing", Wiley Publishing, 2001

[Kant02] Kant, Krishna, Ravi Iyer and Vijay Tewari, "A Framework for Classifying Peer-to-Peer Technologies", in Proceedings, IEEE/ACM Symposium on Cluster Computing and the Grid, 2002

[Kim03] Kim, Yongdae, Daniele Mazzocchi and Gene Tsudik, "Admission Control in Peer Groups", in Proceedings, IEEE Symposium on Network Computing and Applications, 2003

[Klensin01] Klensin, J., "RFC 2821: Simple Mail Transfer Protocol", "www.rfc-editor.org", November 2004

[Larman02] Craig Larman, "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and design and the Unified Process [2nd ed.]", Prentice-Hall, 2002

[Lotus05] IBM Lotus Notes website, "www.lotus.com", 2005

[Matei02] Matei, R.; Iamnitchi, A.; Foster, P., "Mapping the Gnutella network", IEEE Internet Computing, Volume: 6 , Issue: 1 , Jan.-Feb. 2002

[Niemeyer03] Niemeyer, Glenn and Jeremy Poteet, " Extreme Programming with Ant: Building and Deploying Java Applications with JSP, EJB, XSLT, XDoclet, and JUnit", SAMS Publishing, 2003

[New01] New D., M. Rose, "RFC3195 Reliable Delivery for syslog", November 2001

[Oaks02] Scott Oaks, Bernard Traversat, Li Gong, "JXTA in a Nutshell", O'Reilly Publishing, 2002

[Parker04] Parker, D.C.; Collins, S.A.; Cleary, D.C., "Building near real-time P-2-P applications with JXTA", IEEE International Symposium on Cluster Computing and the Grid, 2004

[Pfleeger01] Shari Lawrence Pfleeger, "Software Engineering: Theory and Practice [2nd ed.]" Prentice-Hall, 2001

[Ripeanu01] Ripeanu, M., "Peer-to-peer architecture case study: Gnutella network", in Proceedings, 1st IEEE Conference on Peer-to-Peer Computing, 2001

[Rose01] M. Rose, "RFC 3080 The Blocks Extensible Exchange Protocol Core", www.rfc-editor.org, March 2001

[Roseman96] Roseman, Mark and Saul Greenberg, " Building real-time groupware with GroupKit, a groupware toolkit", ACM Transactions on Computer-Human Interaction, vol. 3, no. 1, 1996

[RSA05] RSA Labs website, "http://www.rsasecurity.com/rsalabs", 2005

[Saxene03] Saxena, Nitesh, Gene Tsudik and Jeong Hyun Yi, "Admission Control in Peer-to-Peer: Design and Performance Evaluation", in Proceedings, ACM Workshop on Security of Ad Hoc and Sensor Networks, 2003

[SETI05] SETI@home website, "setiathome.ssl.berkeley.edu/"

[SHOUTcast05] SHOUTcast MPEG Layer 3 Audio Streaming Technology, http://www.shoutcast.com/, 2005

[SIP05] Session Initiation Protocol, http://www.cs.columbia.edu/sip/

[SourceSafe04] Microsoft Visual SourceSafe, "msdn.microsoft.com/vstudio/previous/ssafe/, 2005"

[Stern00] Stern, R., "Napster: a walking copyright infringement?", IEEE Micro, Volume: 20 , Issue: 6 , Nov.-Dec. 2000

[SubEthaEdit05] The Coding Monkeys, "SubEthaEdit: a Collaborative Text Editing Environment", http://www.codingmonkeys.de/subethaedit/

[Sun05] Sun Microsystems website, Java Programming Language and Virtual Machine, http://java.sun.com

[Sussman04] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato, "Version Control with Subversion", O'Reilly Publishing, 2004

[SwiXML05] SwixML website, "www.swixml.org", 2005

[Thompson05] Thompson, Clive, "The BitTorrent Effect", Wired, vol. 13, no. 1, January, 2005

[Tichy04] Tichy, W.F., "Agile development: evaluation and experience" in Proceedings, 26th International Conference on Software Engineering, 2004

[Udell00] Udell, Jon, "How Ray Ozzie Got His Groove Back", openp2p.com, O'Reilly Media, Inc., 2000

[Vaughan03] Vaughan-Nichols, Steven, "Presence Technology: More Than Just Instant Messaging", IEEE Computer, Vol. 36, No. 10, October 2003

[Wake01] Wake, William, " Extreme Programming Explored", Addison-Wesley Publishing, 2001

[Waste05] Waste website, "waste.sourceforge.net", 2005

[XML05] World Wide Web Consortium website, Extensible Markup Language (XML) Specification, http://www.w3.org/XML/

[XUL05] Mozilla XUL documentation, http://www.mozilla.org/projects/xul/

[Yeager03] Yeager, B., "Enterprise strength security on a JXTA P2P network", in Proceedings, 3rd IEEE Conference on Peer-to-Peer Computing, 2003