MASTER THESIS IN COMPUTER SCIENCE

Casual Resource Sharing with Shared Virtual Folders

Siri Birgitte Uldal

June 15th, 2007

FACULTY OF SCIENCE
Department of Computer Science
University of Tromsø, N-9037 Tromsø

Casual Resource Sharing with Shared Virtual Folders

Siri Birgitte Uldal

June 15th, 2007

University of Tromsø
2007

Abstract

Proliferation of wireless networks has been a major trigger behind increased mobility of computing devices. Along with increased mobility come requests for ad-hoc exchange of resources between computing devices as an extension of humans interacting. We termed it casual resource sharing where resources in this thesis have been narrowed down to files only.

We have named our casual resource sharing model for shared virtual folders (SVF). SVFs can be looked upon as a common repository much in the same way as the tuplespace model. The SVF members perceive the repository similarly to a common file directory on a server, while in reality all participating devices stores their own contribution of files. All types of files could be added to the repository and shared. To become a SVF member one needs to be invited by another member or initiate a SVF oneself. All members are free to withdraw their SVF membership whenever they wish. They are also free to log on to the SVF and log out as they please. The SVF cease to exist when the last member has drawn his membership. The SVF implements a simple versioning detection system to alert members when a file has been modified by another member.

Feasibility of the model is demonstrated in a prototype implementation based on Java and the JXTA middleware, a peer-to-peer (P2P) infrastructure middleware supporting the Internet protocol. The implementation functions with any underlying network supporting the IP protocol, both LAN or WAN. The interacting devices could be running on any operating system. The SVF itself is created with focus on simplicity and requires no more than software installation before use.

The model and implementation is discussed and contrasted with other existing approaches to casual resource sharing.

Keywords: Peer-to-peer, JXTA, ad-hoc

Preface

The master thesis was carried out during 10 ½ month in 2006 and 2007 and is a collaboration project between the Department of computer science at the University of Tromsø and Norut IT.

The master thesis started out as a supplement to Njål Borch's PhD thesis at Norut IT which focused on socialized ad-hoc networks based on peer-to-peer technologies.

I would like to thank my advisors Randi Karlsen at the Department of computer science, University of Tromsø together with Njål Borch for their help and efforts through the process of finishing my thesis.

I am also grateful to the Norwegian Polar Institute, especially head of Environmental Data Section Stein Tronstad, for allowing me to finish my master thesis. Also thank to the people at Norut IT for allowing me to stay there during work with the thesis.

Finally I would like to thank friends and family for their support and bearing with my absence.

Table of contents

1. Introduction

The first chapter starts with a problem description in 1.1 and the goal of the thesis in 1.2. In 1.3 we have included some necessary limitations and assumptions about the thesis. 1.4 describes the methods used and 1.5 contains a short summary of the research contribution of the thesis. The chapter closes with an outline of the remaining thesis chapters in 1.6.

1.1 Problem description

Proliferation of wireless networks has been a major trigger behind increased mobility of devices. Along with increased mobility come requests for ad-hoc exchange of resources between computing devices as an extension of human interaction. For some interaction between groups, there would be servers available. Typically, these groups could be employees in an office environment, or students working on a common university project. To other groups or in other situations, it is less obvious how resources could be shared. For example students that have a server for university work, may not be allowed to use disk space for game playing. A group of craftsmen may not have a server available at work at all, at least not for sharing holiday pictures. A neighbour watch may also lack a server, should they need an archive for logging interaction with the police.

We have termed this type of interaction casual resource sharing. Possible resources exchanged could be web pages, text documents and images, Internet chatting, audio-streaming, video-conferencing, game play interaction, common access to resources like printers, large display walls etc. Due to time constrains, we have found it necessary to narrow resources down to file exchange only.

Traditionally, servers have been used as resource storages between clients. But as outlined in the resource sharing examples above, servers are not always available. There could also be decentralized solutions to resource sharing typically based on specific applications, often configuring one of the devices to act as a server towards the others.

Usually, these solutions are restricted to local area network (LAN) either due to communication range or security restraints.

Another approach which do scale well to wide area networks (WANs) are based on a peer-to-peer (P2P) architecture where all parties may both serve and fetch. These networks differ in functionality, but they often lack a group concept to restrict outsiders from resource access. A closed group concept is desirable to avoid unwanted resources filling up the disk space and removal of requested resources. For example, if a large number of peers actively participate in a network, often less popular files can be difficult or impossible to find as they have been removed. Another issue as networks grow is motivating peers to offer resources and do routing for other peers.

While solutions are in use today and has found their market, we would like to focus on ad-hoc gathering of groups collaborating over some time without depending on access to servers. We will choose a decentralized approach using P2P to avoid dependency upon servers. The application should be device and network agnostic and not require explicit technical skills to set up or use. It should employ a group concept to avoid unrestricted access to resources. Group limitations can also give members increased incentive to provide and route resources as well as downloading. The collaborators will usually know of each other prior to resource exchange.

When users collaborate, they often not only want to exchange files, but also collaborate by updating common documents. Usually the operating system will carry out some simple versioning detection like changing modification dates when a file is updated locally. However, this will not be detected when the repository is located on several devices. Thus, the application should also contain simple versioning detection of the common repository so that group members do not have to worry about where the resources are located in the network or whether anyone has overwritten a file they wanted to keep.

1.2 Goal and objectives

The thesis goal is to develop a concept for casual resource sharing using P2P models. As an approach to archive casual resource sharing, we will define a concept called shared virtual folders (SVF).

Furthermore, the objectives of the goal are:

- Describe an architecture for the shared virtual folders using the P2P model rather than the client-server model.

- Ensure that all peers can both upload and download resources within a closed group concept.
- The SVFs should not be dependent on accessing servers at any level or require Internet access. It should be device and network agnostic to the largest degree possible as well as allow initiation and use by persons without specific technical skills. Devices could be connected and disconnected ad-hoc.
- The SVF should contain some form of versioning detection to inform users when a file has been updated by others in the group.
- Proof of concept should be demonstrated through prototype implementation, based on available middleware for P2P networks.

1.3 Limitations and assumptions

Many issues could be included in the application described above. However, due to time limitations some tasks have been omitted:

- Exchange of resources will be limited to files only. Thus Internet chatting, audio-streaming, video-conferencing, game playing interaction, common access to resources like printers, large display walls etc will not be considered.
- Hybrid P2P networks. If servers are available, they are looked upon as peers in the network as well as any other devices.
- Static computer networks. We will assume that highly dynamic networks represent more challenging environments than static networks. Thus focus will be on devices connecting and disconnecting ad-hoc rather than devices with a high uptime.
- Device discovery and routing. A lot of work has been carried out in this field already and the thesis will rely on work and programming code available [1, 2].
- Lower layer protocols for interaction. How resource transferral and interoperability between devices on a low protocol level are carried out will not have focus in the thesis but rely on work already carried out in these fields.
- Security. Although a vital part of ad-hoc networking and a closed group concept, due to time constraints the thesis will not look specifically into security related issues.
- Legal issues. File sharing is closely linked with issues of legislation and copyrights, which will not be discussed in the thesis.

Furthermore, the thesis relies on some assumptions:

- All participants have some network connection access based on the IP protocol.
- All peers are autonomous.

1.4 Method

There are three major paradigms in the discipline of computing [3]:

- Theory: This paradigm is rooted in mathematics following in the development of a coherent, valid theory.
- Abstraction (modelling): The paradigm is rooted in the experimental scientific method following the investigation of a phenomenon.
- Design: The paradigm is rooted in engineering followed in the construction of a system (or device) to solve a given problem.

For computing as a discipline and thus for the thesis, the design approach will be most appropriate. The design approach model is divided into four steps:

1) State requirements.
2) State specification.
3) Design and system implementation.
4) Test the system.

Usually engineers iterate these steps (e.g., when tests reveal that the latest version of the system does not satisfactorily meet the requirements).

The National Research Council in the USA reported on three primary purposes in experimental computer science and engineering [4, 5]:

- Proof of existence: The demonstration of a fundamentally new computing phenomena.
- Proof of concept: The demonstration of a particular configuration of ideas or an approach that achieves its objectives.
- Proof of performance: The demonstration of seeking performance or seeking improvement and enhancement of prior implementations.

Research in this thesis will focus on the proof of concept; devices and software functioning already will be put together to demonstrate a particular configuration of ideas.

1.5 Research contribution

The research contributions of the thesis are considered to be:

- The concept of shared virtual folders where also versioning detection is included.
- Proof of concept of a simple, ad-hoc file sharing application that combines the concept of shared virtual folders with versioning detection.

1.6 Outline

The rest of the thesis is organized as follows:

Chapter 2 presents background material for casual resource sharing together with descriptions of P2P networking and of P2P middleware.

Chapter 3 defines casual resource sharing and elaborates on the concept of SVF.

Chapter 4 explains design issues related with an SVF implementation. The chapter includes architecture and choice of middleware.

Chapter 5 describes implementation issues in detail.

Chapter 6 describes how testing was carried out showing some results, contains a discussion of results and findings, suggests changes and extensions to the model and implementation and gives directions for future work.

Chapter 7 contains the thesis conclusion.

2 Background

This chapter present background literature on subjects related to the concept of casual resource sharing. Section 2.1 gives a brief background and overview of peer-to-peer (P2P) networks sometimes contrasted with the client-server model. In 2.2 we describe four main resource sharing concepts closely related to our definition of casual resource sharing.

A model for casual resource sharing contains some properties of importance. In 2.3 some of the issues related to organizations of a P2P network are discussed. These includes handling message flow in the network avoiding message loss or network congestion, looking into incentives for peers to do routing, how persistent information is in the network and forwarding information to others and using a profile to obtain better performance.

A resource sharing model would typically need to handle updates of resources as users collaborate. For example, if two users share a document, it would be of interest to them if another peer had updated it by adding or removing information. Thus, in 2.4 we give a background on how decentralized versioning detection could function.

In order to carry out a proof of concept, we will need a model implementation. As implementations can be time consuming to develop, we will need to base ours on a fitting middleware infrastructure. Section 2.5 motivates for the use of middleware as well as describing four implementations that could be employed.

Section 2.6 presents related works to our resource sharing concept.

2.1 Peer-to-peer networks

Peer-to-peer (P2P) networks are often contrasted with the client-server model, but P2P networks also have other abilities. In 2.1.1, we present some definitions of the P2P network and choose the definition most relevant for our concept. In 2.1.2 we describe different types of P2P networks and finally subsection 2.1.3 outlines some advantages and disadvantages of a pure P2P network in contrast with networks based on the client-server model.

2.1.1 Definition of a peer-to-peer network

According to Schoder and Fischbach [6], the term peer-to-peer (P2P) refers to "technology that enables two or more peers to collaborate in a network of equals (peers) by using appropriate information and communication systems, without the necessity for central coordination".

Thus when two or more computers spontaneously collaborate it should be without dependency on already available servers in the network. In fact some sources, like Barkai [7], prefer to contrast P2P with the client-server model: "In a client-server model, the client makes requests of the server to which it is networked. The server, typically as unattended system in a back room, responds to, and acts on, the requests. The idea behind P2P computing is that each peer, i.e., each participating computer, can act both as a client and as a server in the context of some application."

Androutsellis-Theokokis and Spinellis [2] propose a very extensive definition: "P2P systems are distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the intermediation or support of a global centralized server or authority."

In the last definition P2P is defined beyond the contrast with the client/server model to focus on the sharing capabilities, of encountering a larger task by merging the resources of each peer. In addition, it is stated that a P2P network could function as an ad-hoc network with peers connecting and disconnecting continuously without the collapse of any remaining peers nor the network itself. Also considered is the elimination of the single-point-of-failure that a server may represent.

Because the focus of resource sharing is central to our approach, we will choose the definition of Androutsellis-Theokokis and Spinellis. But Schoder and Fischbach et al [8] give some interesting characteristics of a P2P network:

- Sharing of distributed resources and services: In a P2P network each node can act both as a client and server, both providing and requesting a service.

- Decentralization: There is no central coordinating authority for the organization of the network or the use of resources and communication between peers in the network. Frequently, a distinction is made between pure and hybrid P2P networks. Because there are no centralized services in a pure P2P network, this network represent the reference type of P2P design. In hybrid P2P networks, selected functions, such as indexing or routing, are allocated to a subset of nodes that assume the role of a coordinating entity. This type of architecture combines P2P and client-server principles.
- Autonomy: Each node in a P2P network can autonomously determine when and to what extent it makes its resources available to other entities.

2.1.2 An overview of P2P networks

Decreasing costs together with increasing availability of processor cycles, bandwidth, and storage, together with the Internet growth have created new fields of applications where P2P networks fit well. Development and proliferation of wireless networks has spurred growth of smaller, lighter mobile devices such as the personal digital assistants. The P2P concept fit well in these surroundings, perhaps because of it has focus on mobility and thus could be server independent.

Together with the increase in P2P networks, the networks themselves have become increasingly specialized, offering different tasks and different abstraction levels. Schoder and Fischbach et al [8] have chosen to divide P2P networks into a three level model shown in Figure 2-1.

Level 1 represents the lower layers in a communication model, typically addressing issues such as device and service routing and discovery. *Level 2* represents a variety of applications which particularly at this level have become diverse as P2P systems matures. *Level 3* deals with human implications of the new technology like how people react having their device slowing down as it must route information to others, how communities and interest groups develops and so on. At this level, the term peer is interpreted as a person, and not as a person's computing device.

As mentioned, *level 1* represents P2P infrastructures. P2P infrastructures are positioned above existing telecommunication networks, and acts as a foundation for all levels. P2P infrastructures provide communication, integration, and translation functions between IT

components. It provides services that assist in locating and communicating with peers in the network and identifying, using, and exchanging resources, as well as initiating security processes such as authentication and authorization. This infrastructure acts as a "P2P Service Platform" with standardized APIs and middleware which in principle can be used by any application.

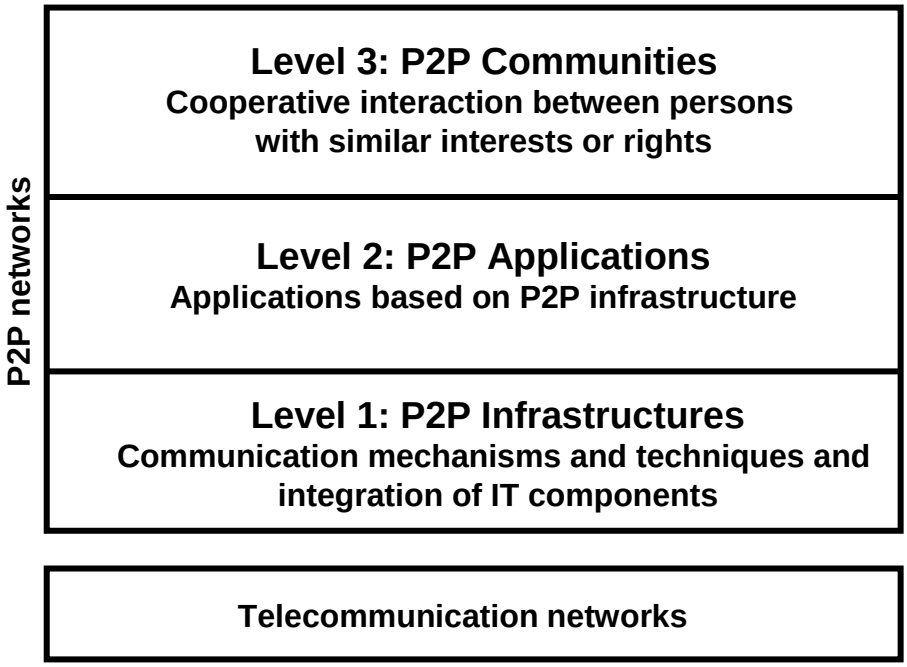

Figure 2-1 Levels of P2P networks [8].

Level 2 consists of P2P applications that use services of P2P infrastructures. They are geared towards enabling communication and collaboration of entities in the absence of central control. P2P applications are often classified according to the categories of instant messaging, file sharing, grid computing and collaboration. The categories of this classification have developed and now start to overlap. Instead Schoder, Fischbach et al suggest application classification through *resource coordination of information, files, bandwidth, storage and processor cycles.*

Information sharing has been divided into presence information, document management and collaboration. Presence information represents peer discovery and knowledge of which peers exist in a network. Applications can independently recognize which peers

and resources are currently available to them in a query search. In the thesis, we have chosen to let the P2P middleware handle presence information.

Document management permit *shared storage, management and use of data*. In a pure P2P network, the document index must be stored at each peer rather than centrally. Thus, indexing and categorization of data could be carried out on the basis of individually selected criteria for example based on interests. An example here is the OpenCola project [8].

Collaboration is permitted by P2P groupware at the level of closed *working groups*. Particularly beneficial here is the omission of server administration, which better supports ad-hoc collaboration. Microsoft Office Groove [9] (see also subsection 2.6.3) is a fitting example.

Characteristics of *file sharing* is given by peers that have downloaded the files in the role of a client subsequently make them available to other peers in the role of a server. A central problem for the P2P networks in general, and in particular for file sharing, is locating resources (lookup problem). In the context of file sharing systems, three algorithms have been developed: the flooded request model, the centralized directory model and the document routing model [10]. Gnutella (see subsection 2.6.7), Napster [11] and Freenet [12] are examples of these models.

Since demands on network transmission capacities are continuously rising, *effective use of bandwidth* is becoming more important. P2P-based approaches achieve increased load balancing by taking advantage of transmission routes that are not being fully exploited. For example, in hybrid networks only the initial requests for files are served centrally, while the download itself is carried out between the node actually storing the file and the requester. Moreover, P2P designs can accelerate the downloading and transport of big files that are simultaneously requested by different entities. Usually these files are split into smaller blocks by use of swarming protocols and then downloaded by the requesters. An implementation utilizing this principle is BitTorrent [13] (see also subsection 2.3.1).

Increased connectivity and availability of bandwidth enable alternative forms of managing *storage*. Within P2P storage networks, it is generally assumed that only a portion of the disk space available on a desktop PC is used. Thus a cluster of computers could replace expensive backup servers in a network. Yet, in P2P networks, this could

result in settings where no peer is available where the file is being requested. Thus, increasing the number of replicates stored at various geographic locations can enhance the probability that at least one peer will be available in the network. OceanStore [14] is perhaps the best known example of a P2P storage network.

Sharing processor cycles on a P2P network came as a result of the recognition that available computing power of the networked computers often was unused. At the same time the growing demand for high-performance computing, especially in the field of bio-informatics, logistics and the financial sector, was increasing. The approaches to the coordinated release and shared use of distributed computing resources in dynamic virtual organizations, is called grid computing. One of the most well known projects is SETI@home [15], an initiative to search for extraterrestrial life forms. The central SETI server divides the data into smaller units and sends these units to the computers made available by the volunteers who have registered to participate in the project. The SETI clients carry out the calculations during idle processor cycles of the participant's computers and then send the results back.

Level 3 focuses on social interaction, in particular, the formation of communities and the dynamics within them. Thus, whereas in level 1 and 2 the term peer essentially refers to technical entities, in level 3 the term peer is interpreted as a person. Schoder and Fischbach et al indicate that "they will be communities not of common location but of common interest". Grid projects such as those interested in finding a cure for AIDS [16] or users of file sharing network like Gnutella (see subsection 2.6.7) and FastTrack [17] who wish to exchange music for example, confirms the suggestion.

Important level 3 issues concerns free riding and accountability (see also 2.3.4). Individual maximization of usage in P2P communities would lead to collectively desirable results. This is because after a file is downloaded, a replicate is added to the file collection of the file sharing community. Free riders threaten collective desirable results by denying access to the downloaded file or moving the file immediately after downloading so that the collective file collection does not increase. Free-riding peers use the resources available in the P2P network, but do not make any resources available [18]. One of the most successful approaches to avoid free riding is the tit-for-tat algorithm used by BitTorrent (see subsection 2.3.1). Files are downloaded in pieces from several peers ensuring improved download capacity. A downloading peer eventually gets

chocked by their fellow peer downloaders if it does not provide server capabilities to the other downloaders once a piece is downloaded.

Another possible solution is accountability [19]. It consists of protocolling and assignment of used resources and the implementation of negative or positive incentives. For example the Socialized.Net use such mechanisms [1] (see subsection 2.5.7).

2.1.3 Decentralized versus centralized P2P networks

Central to the P2P definitions mentioned in section 2.1.1, is the ability to carry out collaboration without having access to or involving servers. As mentioned in the problem description (see section 1.1), there are situations where servers likely will not be available. Thus P2P networks have the potential to expand to new situations and markets and become more ad-hoc or casual in nature which we find desirable. On the other hand, many P2P networks have abandoned the fully decentralized approach and put some servers back into the network, often in response to slow or insufficient discovery and routing.

This section briefly summarizes some advantages and disadvantages of a pure P2P network as compared to hybrid or centralized networks. During construction of P2P networks these advantages and disadvantages must be considered.

Some advantages of the decentralized P2P network are:

- A pure P2P network will likely support ad-hoc networking better than a centralized structure because it is not dependent on the accessibility of a server.
- Usually a centralized service is more expensive than a decentralized, due to both investment cost and maintenance [8].
- Single-point-of-failures could be minimized or avoided when there are no servers in the network.
- There is no need for dedicated staff to maintain the service.
- Since the users handle the data themselves, they do not have to trust a third party’s security routines. Also, there is no centralized log where all activity could be traced.
- If proper algorithms are developed, resources from all participating computers can be exploited instead of just resources of a few servers.

- The network could be optimised for ad-hoc interactions since security routines like establishment of user accounts and security routines does not have to be settled in advance.

However, decentralization also comes at a cost:

- Algorithms such as routing and service discovery can get cumbersome, slow and even fail if the network structure is changing fast and/or the networks are large.
- It requires the peers themselves also to run routing and server capabilities. These routines use computer resources so there must be incentives for contribution and routing in the network.
- Maintenance is left to the users, depending on their technical skills.
- Untrustworthy participants can more easily corrupt or destroy the network or network content as there is no single administrator responsible.
- Since there are no centralized services, there is a greater risk of flooding the network limiting scalability.
- Usually access to services is faster and more reliable with a centralized approach than a decentralized.
- Asymmetric bandwidth access can be an obstacle to decentralized services.

2.2 Resource sharing concepts

This section looks into models used for decentralized P2P networks, where the different models considered are from a overview written by Johanson and Fox [20]. Of these models, we have extracted four which we considered to have the most relevance to our concept. The first two models are well-known middleware concepts. Section 2.2.1 describe the remote procedure call (RPC) followed by the description of an asynchronous version of the RPC called message oriented-middleware (MOM) in 2.2.2. These two models form a base for two other important models at an even higher abstraction level, the publish-subscribe model described in subsection 2.2.3 and the tuplespace model in subsection 2.2.4.

2.2.1 Remote procedure call (RPC)

Remote procedure call (RPC) was the first type of middleware (see section 2.5), used as a way to transparently call procedures located on other machines.

RPC is analogous to a function call. When a RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure 2-2 shows the flow of activity that takes place during an RPC call between two networked devices [21]. The client makes a procedure call that sends a request to the server and waits. The client thread is blocked from processing until either a reply is received, or it times out. When the request arrives at the server, it calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues.

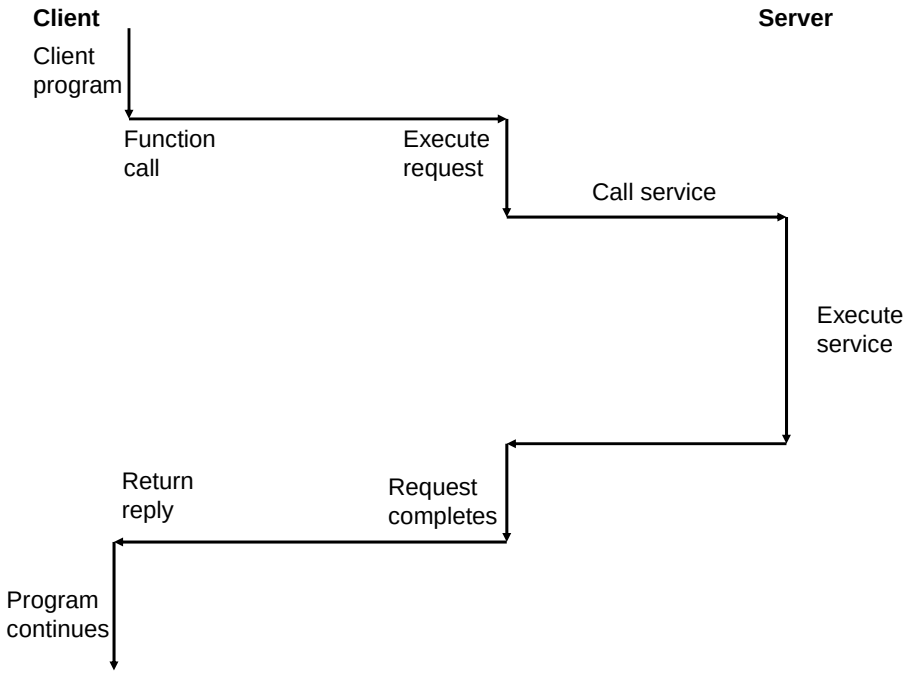

Figure 2-2. RPC activity flow [21].

RPC establishes a notion of client (the programme that calls a remote procedure) and a server (the program that implements the remote procedure invoked) [22]. Calls to remote computers will be hidden and make it appear as a local call. Thus it offers transparency to the users, typically location transparency (requests from an application need not know about physical component locations) and access transparency (interfaces for local and

remote communication are the same). The RPC mechanisms will also seek to hide heterogenity between computers, which is also desirable.

2.2.2 Message oriented middleware (MOM)

Using synchronous RPC demands that the server always will be available, serving upon request. As this is unlikely in many situations, synchronous communication is not always optimal. As a result, the message-oriented middleware (MOM) was developed.

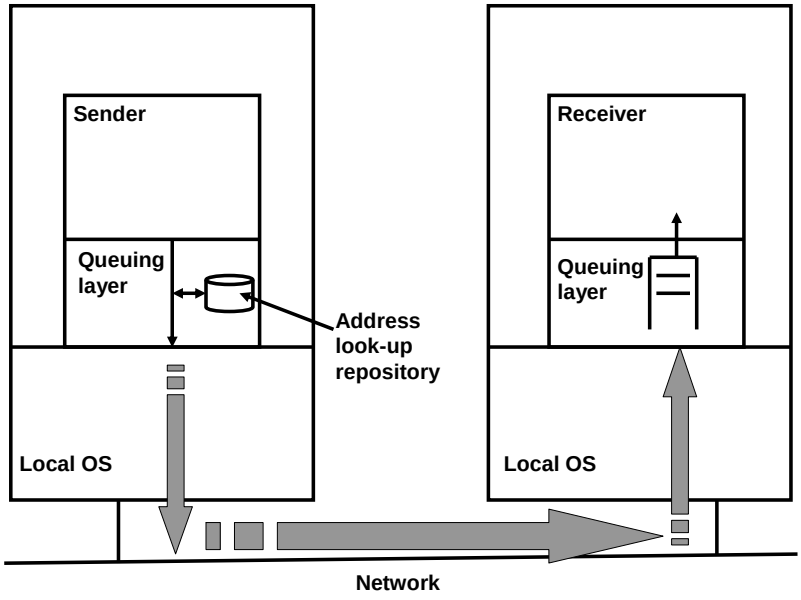

Figure 2-3 MOM functionality [23].

MOM supports message-based interoperability as shown in Figure 2-3 [23]. A sender puts a message into the local outbound queue. MOM uses a repository to map the address of the content in the queues to the correct destination address. Thereafter the message is forwarded to the queuing layer of the receiver where it waits until the receiving application can process it.

MOM allows the requester to continue as soon as the middleware has taken responsibility of the message and thus supports asynchronous message delivery. Eventually the provider will send a response message including the result, and the requester can fetch

the message at an appropriate time from the inbound queue. The result is a de-coupling between the requester and the provider which leads to more scalable systems since programs do not have to wait. For pure message-oriented interactions, the client-server model is no longer fitting because all objects on the devices both send and receive messages. MOMs can also support multicast messages and deliver it transparently to the receivers.

The primary disadvantage of MOM is requiring an extra component in the architecture, the message transfer agent. The message transfer agent is needed if the sender and the receiver are not directly connected to one another. As with any other systems, adding an extra component may lead to reductions in performance and reliability [24].

2.2.3 The publish-subscribe model

A model that supports MOM but on a higher abstraction level is the publish-subscribe model. A publish-subscribe service conveys published notifications from any providers to all interested subscribers with a matching subscription set [25]. The subscribers only need to subscribe to a service, and if there is a match with the publishers, the subscriber will get the message. Figure 2-1 shows three examples of how publish-subscribe works: In a) the subscriber registers for a certain type of message. In b) a publisher generates a non-matching message, so it is not delivered. In c) a matching message is generated and thus delivered to the subscriber.

Clients do not need to use source/destination identifiers or addresses. The model scales well because subscribers can be follow-up through multicast messages. Flexibility is achieved by varieties of subscription criteria. Oki et al [26] describe the publish-subscribe model as appealing for event-based applications because of the strong decoupling of participants in 1) time (participants do not have to be up at the same time), 2) space (participants do not have to know each others address) and 3) flow (data reception/sending does not block participants).

To achieve decoupling, consumers subscribe to specific kinds of event notifications. The most flexible selection criteria for notifications are realized by *content-based selection*. In the publish-subscribe model, notification messages are filtered according to content. Event notifications propagate from a provider to interested subscribers through a network of filters. Filters are typically boolean functions to help content selection.

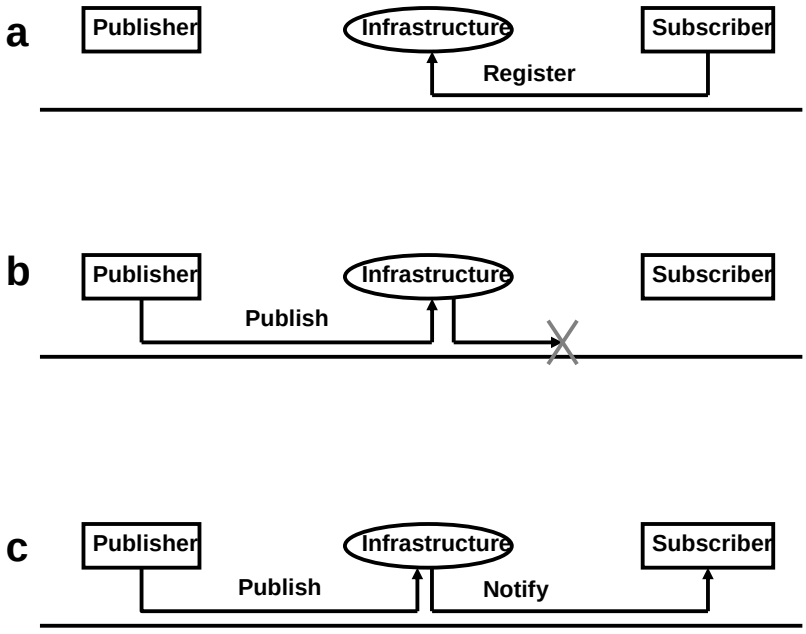

Figure 2-4 The publish-subscribe model.

Type-based publish-subscribe is an object-oriented variant of the content-based selection [27]. Here events are considered to be objects, i.e. instances of native types in an object-oriented programming language. The subscriber in a type-based publish-subscribe service will only receive instances of a particular type of objects and its subtypes. Subscriber-specified content filters may also be applied to further limit the events delivered to the subscriber. Content filters are specified in the native language based on the events' public attributes and methods.

The publish-subscribe model was first developed for static networks where subscriptions may change dynamically with the interest of the clients, while network routing remained fairly unchanged. Porting the model to P2P systems required dynamic routing algorithms, but has been demonstrated for several P2P networks through multicast message use [25, 28].

However, the publish-subscribe model provides no temporal decoupling [20]. An application must be running and subscribed at the time of message generation to receive a copy of the message. It is hard to account for this drawback since a message transfer agent could have temporarily stored the messages, but has to reside on one or more of the

nodes themselves in a decentralized P2P network. As all devices can be removed from the network ad-hoc, this approach will have drawbacks.

Johnson et al [20] also points out that the publish-subscribe model is not a general purpose coordination system since it is designed primarily for broadcast and multicast. Example given of anycast, where each destination address identifies a set of receiver endpoints, but only one of them is chosen at any given time to receive information from any given sender. Typically data is routed to the "nearest" or "best" destination. When the first receiver receives the message, there is no way to remove the message once one of the receivers acknowledges the receipt.

Gnutella (see subsection 2.6.7) implements the publish-subscribe model.

2.2.4 Tuple spaces

An alternative to the publish-subscribe model is the tuple space model. The concept of tuple spaces first rose within the discipline of parallel programming. In the eighties Carriero and Gelernter [29] published work on Linda, a model for process creation and coordination. If two processes needed to communicate, they did not need to exchange messages or share variables. Instead, the data producing process could generate a new data object called a tuple and set it adrift through a region called a tuple space. The receiver process could then access the tuple through the tuple space. In Linda, communication and process creation were considered two facets of the same operation. The result in both cases was that a new object was added to the tuple space, where any interested party could access it, taking the data object tuple out or copying it.

The senders in Linda did not know anything about receivers and vice versa. When a Linda process generated a new result that was of interest to other processes, it dumped the new data into the tuple space.

A tuple existed independently of the process that created it. Collectively the data structure from all the tuples formed the tuple space. A tuple itself was a series of typed fields, for example (“a string”, 15.01, 17, “another string”) or (0,1). There were four statements that could be used. To put data into the tuple space the command *out* was used, which would cause the tuple to be generated and added to the tuple space. An *in* or *rd* statement specified a template for matching: any values included in the *in* or *rd* needed

to be matched identically to one or several tuples residing in the tuple space. *In* removed the tuple from the tuple space as if it was taken out. The *rd* command specified how to get a copy of the tuple previously put out. There was also an *eval* statement which would start a process, which would return its value by becoming a normal tuple.

In the parallel programming model of Linda it was not important on what machine and in what memory the tuple actually resided. When a process requested a tuple it was delivered because the machines shared the same memory, distributed over several machines.

The concept of tuple spaces has later on been applied to a well-known pilot called the Interactive Workspaces project at the Stanford University [20, 30]. Initiated in 1999, the project focused on investigation of human interaction with large high-resolution displays. The main user setting was the open participatory meetings in the same location. In this setting, a group of 2 to 15 people worked to accomplish a task. People came to the meeting with relevant materials saved on laptops or file servers. During the meeting, the shared focus attention was on a large display surface, to which users could apply content from their computing devices. The goal was to facilitate ease of interaction among participants.

The project recognized that many different applications could be utilized as users brought along various computing devices when collaborating and during interaction with the high-resolution displays. Work was put into understanding the concept of interactive workspaces along with development of a software infrastructure called iROS (interactive Room Operating System). It was a higher layer meta-operating system that was tied together with devices through their own operating systems.

iROS had three subsystems, the Data Heap, iCrafter and the Event Heap. They were responsible for moving data, moving control and dynamic application coordination. Figure 2-5 shows the basic architecture. The only device demand was to support the Event Heap. Furthermore, decoupling of applications through the underlying coordination mechanism was emphasised. Through decoupling the system parts would be less dependent on each other and faults in one application would be limited as much as possible.

The Event Heap stored and forwarded messages known as events. It provided a central repository much like a tuple space to which all applications in an interactive workspace could post events. An application could selectively access events on the basis of pattern matching fields and values. Interface actions in one application could trigger actions in another running on any of the machines in the workspace. As an extension to tuple spaces, unconsumed events would automatically be removed and provided support for soft-state though interval signalling. The applications had interfaces which could interact with the Event Heap through several APIs like Web, Java, and C++.

The Data Heap facilitated data movement by allowing any application to place data into a store associated with the local environment. The data was stored with an arbitrary number of attributes that characterized it. The system received location independence through attribute use instead of naming the physical file system that stored the data. The Data Heap stored the format information, and transformed the data to the best format supported by the retrieving applications.

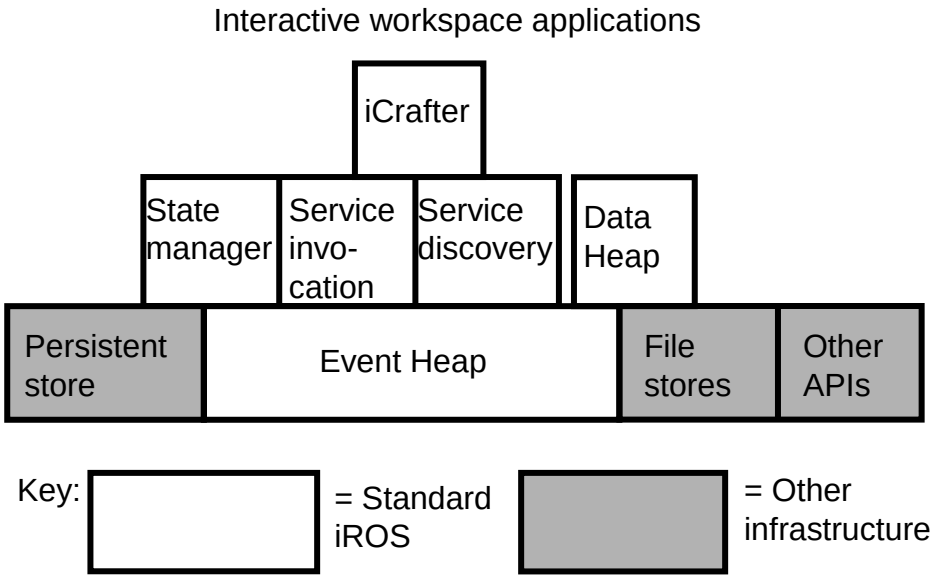

Figure 2-5 The iROS component software [30]

The iCrafter system provided a system for service advertisement and invocation, along with a user interface generator for services. iCrafter services were similar to other

middleware directory services, except that invocations happened through the Event Heap. If a custom-designed user interface was available, the iCrafter system would use it. Otherwise, a more generic generator rendered the user interface into the highest quality type supported on the device.

The Interactive Workspaces project only allowed interaction between devices within the bounds of the local physical space (typically between devices located in the same room). Therefore, software infrastructure for a particular room should only support the device within the room unless explicitly over-ridden by users to do otherwise. Vice versa, coordination with applications and devices outside of the space would not occur unless a user specifically requested it.

2.3 Properties of resource sharing models

As described in the previous section, choosing an appropriate resource sharing model is of vital importance to get a successful P2P application model. Within the chosen model, there again would be properties also in need of assessment. This section elaborates on some of the issues related to model properties.

For example, as devices come and leave ad-hoc, communication between devices could easily get lost, for example as a result of network congestion. Thus minimizing the effects of lost communication will be of importance as outlined in subsection 2.3.1.

A subject also related to communication is that of whether push or pull should be used as transfer mechanisms in a network, as presented in subsection 2.3.2. The two represent very different approaches to network flow, but can also be combined together.

Subsection 2.3.3 discuss a vastly different property of a casual resource sharing model; whether a peer application should make use of a configurable profile to specify needs, or whether all or most decisions should be implemented directly in the software coding, leaving the user with little or no alternatives.

In the absence of servers in P2P networks, the peers would have to route and also maybe upload for other devices themselves. Most people are not willing to do these tasks on behalf of others, as their device may work slower. Thus rewards and punishments may be

necessary properties of a casual resource sharing model. These issues are briefly discussed in 2.3.4.

Finally, also due to lack of servers, we have to trust other peers with information in the network. Since all peers usually act in their own interest, it can be difficult to obtain the same extent of persistence as a server in a network can. Subsection 2.3.5 looks into properties improving persistence in a P2P network.

2.3.1 Interruption handling during data transfer

In P2P networks, devices may connect and disconnect at any time. Errors may arise if a device is disconnected during resource transferral. These situations must be accounted for in order to avoid information loss or even damage to the network itself.

The transferral time is dependent on the connection time, the bandwidth and the channel contention. Channel contention depends upon the total amount of data transferred and the interval between data transferred.

Using compression techniques could be way of reducing the amount of data transferred, but over narrow bandwidths, it may not be sufficient as the only technique. Another technique is to try to avoid peers with little capacity. JXTA for example uses the notion of minimal edge peer (see section 2.5.6) for peers that should relieved for additional burdens like routing.

Another approach is to make priorities on the data a device should look for. Typically it is carried out by means of a device profile, where the user may specify his or her interests at the moment (see also subsection 2.3.3). For example a tourist looking for somewhere to eat, could receive advertisements from restaurant while walking down a city street [31].

Proactive caching could be carried out by means of user profiles or information weighting. If using a profile, registered preferences may be used as means to select areas of interest (see subsection 2.3.3 below on profile content). Instead of waiting for the user to explicitly request downloading of data, the device could start caching as soon as it discovers new data of potential interest [31]. Xu and Wolfson present an approach where one peer poses queries. The peer serving ranks the answers based an algorithm of

multiple attributes with individual weights, and start transferring the most important data first [32]. This approach requires that the data to download are small in size compared to the available bandwidth or that other measures are taken also to avoid uncontrolled device disconnection.

Another approach could be to propagate notifications before the actual data is sent to ensure at least arrival of meta-information. Later on, the resource itself can be downloaded if relevant. Tanenbaum and Steen [23] describe a similar approach carried out by invalidation protocols, where devices holding document copies are informed that an update has taken place and that the data the device has is no longer valid. An advantage of the approach is that notification messages itself are small and can thus be transmitted even at low bandwidth connections or during very short connection periods. The disadvantage is that the messages themselves take up bandwidth and must be planned carefully not to congest the network.

As an alternative to transferring the whole updated document, one can instead tell each replica which update operation it should perform [23]. This approach assumes each replica is represented by a process capable of “actively” keeping its associated data fresh by performing operations. The advantage of the approach is little bandwidth use, while it requires more processing power per replica. Another disadvantage is ensuring total ordering of all the different versions of the updates.

If a device is taken down in a controlled manner, it is possible to display a message informing about the remaining transferral time before download and give the user a possibility to cancel the transfer. In the UPnP middleware, it is possible to issue SSDP bye-bye messages and un-register the device before removal from the network (see subsection 2.5.5). Yet devices could be removed uncontrollably, for example through lost network connections in wireless networks.

A very interesting approach is swarming. Swarming is a P2P content delivery mechanism that utilizes parallel download among a mesh of cooperating peers. Instead of overloading the peer delivery content, swarming is initiated by giving peers now acting as clients only a block of the desired content, along with a list of other peers that can provide the other blocks of the same item [33], as seen in Figure 2-6. In a) several peers are downloading content from a peer. The swarming peer downloading will also exchange information with their peers in order to progressively find other peers with the

content they need. Moreover, as peers acting as clients discover suitable peers, they begin to download the content from them in parallel as shown in Figure 2-6, b. Overall, the more requests the peer with a resource get, the less content it serves directly and the more it redirects to other peers which already has downloaded blocks of content. During heavy load, the system may generate swarms of peers that cooperatively download content in parallel from each other and from the peer serving. Thus, if the peer serving is restricted by means of bandwidth, a swarming protocol can do a better load-balancing.

The most interesting example of a swarming protocol is perhaps BitTorrent, an application for file sharing which uses the tit-for-tat strategy to optimise fast downloading [13]. The clients have an incentive to participate in the swarming because they will receive the help of other peers in return. If a peer refuses to help other peers downloading, the other peers will gossip and soon the peer refusing will find itself down prioritised when needing to download from others.

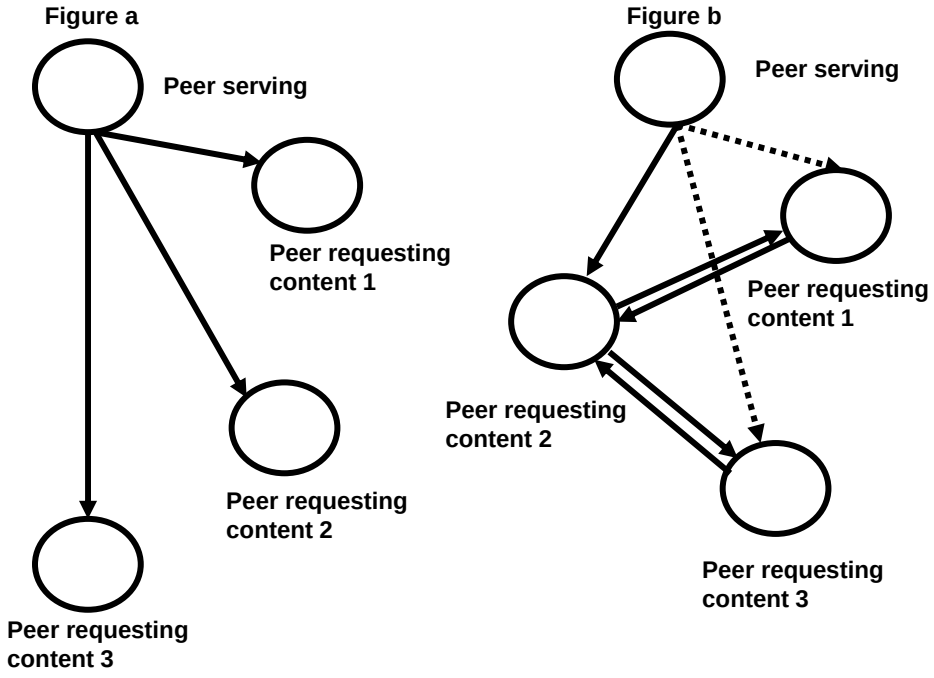

Figure 2-6. Swarming protocols [33].

2.3.2 Data push versus pull

Another issue related to network flow is whether data should be pushed or pulled to the client. This issue is related to minimizing message loss in a network. In a *push-based approach*, updates are propagated to the receiver without the receiver asking for it. This type of model is often related to the publish-subscribe model described in 2.2.3. Typically a peer subscribes to a service from another peer that pushes information every time new items are generated. Where data is requested simultaneously, broadcast or multicast could be carried out, typically on a local area network (LAN). P2P protocols usually present an overlay network that makes broadcast/multicast possible even across different network segments (WAN).

Data pushing is sometimes not desirable if the peer receiving has little capacity available, or is soon about to disconnect. Moreover, if the subscribing device is not connected, the pushed content will be lost. Even worse the providing device could be unavailable if removed ad-hoc from the network. Although outside of the thesis scope, if a subscriber is allowed to push information into a closed network, security would be violated [23].

In a *pull-based approach*, the receiver requests a sender to send the data contained at the moment. It is a preferred approach in many P2P networks [31, 32], due to both security and bandwidth limitations, but also due to ad-hoc connection and disconnection of devices and because the peers may select the files to receive on an individual basis.

A trade-off between the push and the pull model could be leases [23]. A lease is a promise by the peer serving that it will push updates to the receiving peer for a specified time. When a lease expires, the receiver is forced to poll the sender for updates and pull in the new or modified data. Alternatively the receiver can ask for release renewal. JXTA (see subsection 2.5.6) has specified the Rendezvous protocol for peers wishing to lease rendezvous peers for query propagation to others [34].

2.3.3 Configuration

A vastly different issue related to properties of casual resource sharing is whether a peer application should make use of a configurable profile to specify needs, or simplify user decisions by implementing logic directly in software coding, leaving the user with no alternatives. While at one hand experienced users may request a large degree of freedom,

we may want to keep the threshold for use as low as possible so inexperienced users could participate as well.

Self-configuration was one of the issues discussed at Ubisys 2003 and 2004 [35]. Most people agreed that a system should be able to configure itself. However, a self-configurative system does not mean that the system can predict user requests.

One approach to a self-configuring system, was initiated by the Aura programme at Carnegie Mellon where the person's scheduler was used as an indication of where a person is heading [36]. Also, it may provide an indicator for of how long the person's computer will be available on the network. This could work well provided that the person uses the scheduler for all appointments and keeps it updated all the time. As such, the solution is quite similar to a profile.

When deciding what device to route through or what device to download from Verma [37] suggests a system where a property manager located on each peer keeps track of the uptime cycle of the computer. Also bandwidth and available free storage can be extracted to suggest whether the user should participate in backing up files or not.

Yet although some self-configuration is possible, most often user needs are provided through a profile where the user may specify her intentions and desires [31]. If the user has enough incentive to provide this information, for example through specifications like a shopping list, this approach may work. Otherwise, if the profile is difficult to configure or has to be configured often, it may be an obstacle to application use.

2.3.4 Incentive mechanisms and accountability

In a P2P network, a peer may function as a client, a server or a router. By selfish behaviour most peers would like to download resources rather than to route or provide them. Thus these behaviours must be accounted for by system properties if deemed required.

For many P2P networks, employ mechanisms to provide incentives and stimulate cooperative behaviour between users are of importance, as well as some notion of accountability for actions performed [2]. An example of uncooperative behaviour is so-called free-riding; users that only consume resources without contributing any.

Androutsellis-Theotokis and Spinellis [2] divide incentive and mechanisms into two categories:

- Trust-based incentive mechanisms. Trust is a straightforward incentive for cooperation, in which a node participates in a transaction based on whether he/she trusts the other party. Reputation mechanisms are considered belonging to this category.
- Trade-based incentive mechanisms. In trade-based mechanisms, a node offering some service to another is explicitly remunerated, either directly or indirectly. This category is mainly represented by various micropayment mechanisms and resource trading schemes.

An example of a trust-based incentive mechanism could be through a preference and reputation systems like the one used in The Socialized.Net [38] (see subsection 2.5.7). The preference is a locally determined rating from neighbour nodes. It is based on gathered statistics and possibly user inputs. It is modelled after the human notion of impressions; the “like” and “dislike” of others. The preference is represented with a scale where 0 is neutral, a number between 0 and +5 gives a positive preference and a number between 0 and -5 gives negative preference. The extremes in either directions are given by user interaction, either blocking or high priority. A number of automatically generated ratios decide the middle part of the scale.

One of the most successful examples of trade-based incentive mechanisms also seeking to avoid free-riding is the tit-for-tat algorithm used by BitTorrent [13] (see also subsection 2.3.1). BitTorrent is a swarming protocol, where the nodes receive a block of the file which later on is offered to other peers. If two peers are both getting poor reciprocation for some of the uploading they are providing, they can start uploading to each other instead and both get a better download rate than they had before. Thus, peers that provide poor upload while doing download will get chocked. Choking is a temporarily refusal to upload; it stops uploading while downloading can still happen and the connection doesn’t need to be renegotiated when choking stops. BitTorrent peers recalculate who they want to choke every ten seconds, and then leave the situation as is until the next ten second period is up. Ten seconds is considered a long enough time to increase new transfers to their full capacity.

2.3.5 Persistence and search guarantees

By selfish behaviour most peers would also prefer not to store information on behalf of other users. Thus one cannot expect the guarantees in a P2P network to be equally good as for server storage. Lower persistence and search guarantees must be accounted for during design of a casual resource sharing network.

The ACID model is one of the oldest and most important concepts of database theory. It sets forward four goals that a database or information system should strive to achieve: atomicity, consistency, isolation and durability. *Persistence guarantees* are closely connected with durability. Durability demands that changes applied to a database or information system by a committed transaction must persist. The changes must not be lost because of any failure [39].

Regarding persistence guarantees given for a decentralized P2P network Kubiatowicz states that P2P systems must deal with an unreliable and distrusted infrastructure [40]. He defines “unreliable” as systems not professionally managed that may crash or fail at any time. By “distrusted” he refers to participants that could be adversarial, attempting to exploit vulnerabilities, compromise privacy, or damage the system.

While it is possible to give persistence guarantees in a read-only P2P system, systems with read-write capabilities cannot be guaranteed without active, well-behaving components [40]. In read-write systems unreliable peers may manipulate the stored resources. To some extent, Byzantine Agreement can provide a mechanism for cooperative decision making in spite of malicious elements. A Byzantine Agreement allows a set of peers to come to a unified decision about something even if some of them (less than one-third) are actively attempting to compromise the process.

Search or lookup guarantees is connected with the ability to locate a resource in a network. If a resource is sure to be somewhere in the network, search guarantees would return the resource. Deterministic search guarantees can be given in structured networks, for example by means of techniques like distributed hash tables [41]. For unstructured networks which lacks global routing guarantees, it has been shown that probabilistic search guarantees can be given [42].

Giving persistence and search guarantees for some P2P network can be a challenge. Thus, usually these systems work on a best effort basis. For example, if a tourist walks down the street looking for a Chinese restaurant, he may get within network range of one. If so, the restaurant's issued advertising message may be registered with the personal device of the tourist. Yet, there is no guarantee that the tourist will be in the vicinity of a Chinese restaurant, even though the city may contain several of the restaurants. The MoGATU project works on in this manner, assuming that two devices may never communicate amongst each others again [31]. As a consequence, the device will also offer to make a reservation as well at the same time so that all communication may be settled during the same session.

For a network that uses semantic query routing such as the Socialized.Net [1] (see subsection 2.5.7), a device can delay a search for another peer if the device is not available at the moment. The requester in this type of network may expect that the two devices will meet again, thus he does not download any files during the first encounter. Yet, two devices may never reconnect, loosing the opportunity to download a resource. In this type of network one cannot give more than best effort guarantees on both search and persistence. This is acceptable, as long as the users do not expect more. Networks based on semantic query routing are not meant to provide exhaustive searches, but rather to provide an answer according to the user's preferences if existing and available.

Other P2P networks may use a combination of both probabilistic and deterministic guarantees on search lookup. Yet it demands that the requested resource exists in the network, thus persistence guarantees must be given. For example, in a large P2P system, a probabilistic algorithm may search for the resource near the querying device. If the resource cannot be found, a full deterministic search can be carried out which eventually will locate the resource. The OceanStore project [14] carry out search in this manner, mainly because resources in the network are likely to be located near where they are being used.

A positive ability about a P2P network, is that by adding peers, together they may provide more stable capabilities even when individual peers vary in behaviour [40]. For instance, when requesting a document, one may gain faster response by issuing requests to several peers serving than just one. Thus, for a network storing information to achieve a 1,000–year data persistence guarantees, peers serving must continuously collect, regenerate, and redistribute fragments as individual disks have a life expectancy of only

five years. Kubiatowicz [40] recognizes three key elements to achieve stable persistence and search capabilities:

- Redundancy. More resources must be utilized than the “bare minimum” required for operation.
- Replacement. Some technique must be present to recognize failure and switch from faulty resources to functioning ones.
- Restoration. Some process must act to continuously repair data and routing.

Kubiatowicz also suggests that P2P systems will become more stable as they grow larger.

One assumption that often permeates large-scale systems is the belief that peers will fail independently [40]. If this assumption does not hold, persistence guarantees in the network may not hold. For instance, replica placement schemes do not protect data when peer holding the replicas both fail. Simultaneous failing may arise if peers share the same subnet, owner, software release, operating system or geographic location. Thus, simultaneous failing represent an obstacle to achieve good persistence guarantees in a P2P network.

2.4 Versioning detection and control

An important part of resource sharing involves resource collaboration. Thus a common resource could be modified by one user and then handed to another for further updates. Usually the operating system will carry out some simple versioning detection like changing modification dates when a file is updated locally. However, as we will use P2P systems, we cannot rely on the operating system of a common server. Thus, our resource sharing model needs to be able to inform the users when they have created different versions of a resource.

Revision control, versioning control, source control or software configuration management (SCM) is the management of multiple revisions of the same unit of information. The models we will use for versioning detection will not involve changing the resources to avoid inconsistency nor do we need as elaborate models as used within SCM. However, often models for control and detection can be hard to separate since they may use the same mechanisms, thus we will focus on versioning detection, but also touch upon versioning control when it is intertwined with detection models. An overview of

version models definitions are given in 2.4.1. Thereafter, in 2.4.2 we describe how detection of new versions could be carried out. While SCM is too elaborate a model to use directly within our casual resource sharing, it forms a base for simpler versioning models as well [43]. Thus we describe overall SCM terminology and principles in 2.4.2. Also in this subsection, we will focus on versioning detection.

2.4.1 Versioning models

If two developers try to change the same file at the same time, the developers may end up overwriting each other's work. We thus need to implement a versioning model. Conradi and Westfechtel [43] define a *version model* as the items to be versioned, the common properties shared by all versions of an item, and the deltas, which is the differences between them. Furthermore, versioning models determines the way version sets are organized and provides operations for retrieving old versions and constructing new versions.

The term *item* or *object* defines anything that may be put under version control, including all kinds of resources like for example text documents or software. A *versioned item* is an item that is put under versioning control. Thus for a versioned item more than one state should be maintained, in contrast to unversioned items where changes are performed by overwriting. Also, there must be some way of deciding whether two versions belong to the same item. An identifier can thus be defined to help identifing an item. For a versioned item, each version must be identified by a version identifier, for example a number.

Versions differ with respect to specific properties. The difference between them is called a *delta.* The term suggests that the differences would be small compared to the files themselves. Sometimes this does not hold, as a file could have all its content changed in the next version. Thus the common properties may become smaller and smaller the more versions are created. But it is necessary to define some common properties, as there otherwise would be no reason to group versions. Sometimes multilevel versions are introduced, where each version has versions themselves.

Typically, there are two ways to define a set of V versions from a versioned item, either extensional or intensional. *Extensional versioning* is defined by by enumerating its members, like giving each version a consecutive number. All new versions are explicit

and will be registered. The user interacting with the system base retrieves some version v_i, performs changes on the retrieved version, and finally submits the changed version back to the system base as version v_{i+1}. To ensure safe retrieval of previously constructed versions, versions can be made immutable. In some systems all versions are made immutable when they are checked into the system base, in others explicit operations are provided to freeze mutable versions.

In contrast, *intensional versioning* is applied when flexibility is needed in development of new versions. Typically a specific version v is constructed in response to a bug or a demand from a customer. In this case, many new combinations could be constructed on demand. These version sets are defined by a predicate, which defines all constraints that must be satisfied by all members of V. Thus a specific version v is described intentionally by its properties. For example conditional compilation as supported with the C programming language use intensional versioning. The preprocessor used for conditional compilation constructs any source file based on the values of preprocessor variables. Fragments of the source file whose conditions evaluate to false are excluded.

2.4.2 Versioning detection

Having defined some criteria of what a new version could be compared to older ones, we also need to know how new versions could be detected. Collins-Sussman and Fitzpatrick et al [44] states that there are two main versioning models. One solution, named *file locking* prevent concurrent access problems by locking files so that only one person at a time has write access to the central repository copies of those files. However, there are some problems related to the model:

- Administrative problems. If a process holds on to a lock for a long time, others will not be able to access the resource. It may also lead to deadlock if two processes each hold on to a file, at the same time trying to access each other's file.
- Unnecessary serialization. Suppose process A holds on to a file, modifying only the beginning of the file, while process B waits for the lock, but only wants to make changes at the end. Ideally they could have accessed the file simultaneously.
- Create a false sense of security. Process A locks and edits file A, while process B simultaneously locks and edits file B. But if file A and B are dependent on one another, the changes made to each could be incompatible. The locking system

may create a false sense of security.

A second solution is called *version merging* (or *collision detection*) and is used by Concurrent Versions System (CVS) among others. The model allows each process to create a working copy from the main file at the repository, update this file, and provide facilities to merge changes later. If process A finish the file before process B, A's changes is written back to the repository. When process B attempts to save it's updates later on, the repository informs that the file is out-of-date. Process B can ask the repository to merge any new changes from the repository into B's version of the file. If the changes do not overlap each other, this is done.

If the two copies cannot be merged, it is called a conflict. B's version of the file is usually flagged as being in a state of conflict which cannot be solved automatically. Often the conflict is resolved by the users who must look at changes in both copies and decide which one to keep.

Within P2P systems, there are distributed version control systems that uses both preventive approaches and collision detecting methods.

2.4.3 Software configuration management (SCM)

The most elaborate forms of versioning control usually happens within software development. As software is developed and deployed, it is common for developers to be working simultaneously with updates on different software versions. Bugs and other issues are often only present in certain versions because of the fixing of some problems and the introduction of others as the program develops. For the purposes of locating and fixing bugs, it is important to be able to retrieve and run different versions of the software to determine in which versions the problem occured.

Within SCM, each developer operates in a local workspace that contains the versions created and used.Traditionally, cooperation policies from a common server regulate when versions are exported from or imported into a workspace [43]. Typically, the common server holds a *repository*. The repository has a collection of system configuration files and history files including file control meta data, latest source codes, comments and revision records [45]. The repository also include a sequence of *directory trees* [44]. A

directory tree is a snapshot of how the files and directories versioned in the repository looked at some point in time. These snapshots are created as a result of operations of a developer, and are called *revisions.* Each revision is a new version intended to supersede its predecessor.

A version can be made a local working copy from the repository, through *check-out.* During check out a local copy of the requested directory tree is made from the repository. The directory tree contains the requested collection of files. The developer can edit the files and compile them if they are source code files. After changes has been made to the files in the workspace and it is verified that they work properly, the files can be transferred back to the server, called a *check-in* or *commit.* Here the working copy is written or merged back into the repository at the server. An *atomic commit* (or check-in) allows committing changes in multiple files. The set of changes is called a *changelist, change set* or *patch* and identifies changes made in a single commit and offering guarantees that all files get fully uploaded and merged [46].

Suppose two collaborators A and B, checked out the same working copy simultaneously. When A commits changes to a file back to the repository, B's working copy will be left unchanged. To bring B's project up to date, B can request the server to *update (*or *sync)* her working copy. It will incorporate A's changes into her working copy, as well as any others that have been committed since B checked it out. During an update the local workspace first builds a temporary transaction tree that mirrors the state of a working copy. The repository then compares that transaction tree with the requested revision tree (usually the most recently created tree), and sends back information that informs the client about what changes are needed to transform their working copy into a replica of that revision tree. After the update completes, the temporary transaction is deleted.

A *merge* or *integration* brings together (merges) concurrent changes into a unified revision. Merging files can be difficult, especially if repeatedly merge changes from one branch to another, one may accidentally merge the same change twice. It may still work, but if the already-existing change has been modified in any way, one might get a conflict.

Every revision starts as a *transaction tree*. When doing a commit, a client builds a transaction that mirrors their local changes plus any additional changes that might have been made to the repository since the beginning of the local client's commit process, and then instructs the repository to store that tree as the next snapshot in the sequence. If the

commit succeeds, the transaction is effectively made into a new revision tree and assigned an *identifier* or *tag*. The tag is an identifier for a revision to uniquely define it. If the commit fails for some reason, the transaction is destroyed and the client is informed of the failure.

A *trunk* is the main line of development. During an ongoing product development, there may be a need for a release of bug fixes. A *branch* (see Figure 2-1) can be copied from the trunk, moving on generating its own history, but previously sharing the same history as the trunk release. Thus a branch can serve as the bug fix baseline for that release. Patches that have to be made for this release in the future will be developed on this branch. The main trunk can be used for ongoing product development.

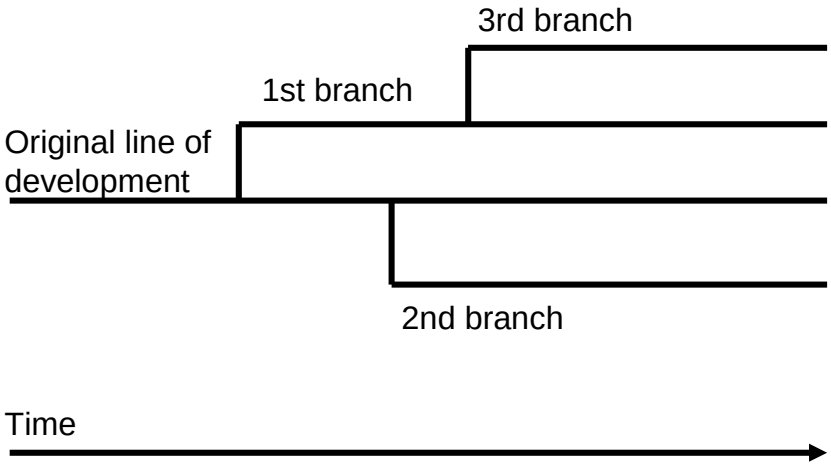

Figure 2-7. Branching [44].

Traditionally, SCM have used a centralized model, where all the revision control functions are performed on a shared server. *Distributed versioning control* allows multiple simultaneous editing by using a P2P approach to versioning control. Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is an independent repository. Synchronization is conducted by exchanging patches (change-sets) from peer to peer. Usually there is a global (world wide) name-

space for lines of development and revisions. Every branch is effectively a working copy and vice versa, with branch merges conducted by ordinary patch exchange, from branch to branch. New peers can join without applying for access to a server. It also allows developers to work without a network connection. GRAM, described in subsection 2.6.7 in an example of a distributed SCM.

2.5 Middleware for P2P networks

To demonstrate a proof of concept of our resource sharing model, we will build a pilot implementation. As there are many infrastructure functionality like routing, discovery and querying involved in a P2P network, we will need to employ a middleware with our application. Middleware is necessary to avoid application development becoming too time consuming. Thus, in the next two subsections, 2.5.1 and 2.5.2, we give a brief introduction to why we should use middleware and summarizes the most widespread middleware types. In 2.5.3 we consider middleware for decentralized P2P networks in contrast to other middleware. The four remaining sections thereafter describe different types of middleware implementations that are available for download. In section 4.3 we will choose the middleware best suited for our needs to use in our proof of concept.

The selection of middleware was done amongst vendors/institutions where support for the middleware was available and the middleware itself was accessible without any cost. By support we mean where it is possible to find papers, books, or other material where interaction with an implementation application programming interface (API) is described. Bonjour (2.5.4) is an Apple implementation of the ZeroConf protocol and is geared towards discovery of new devices to simplify configuration and service exchange. The UPnP middleware in 2.5.5 is used for much of the same purposes. The JXTA middleware from Sun in 2.5.6 is used for closed peer groups, to gather peers with similar interests, and to carry out surveillance of devices. The Socialized.Net developed at Norut IT (subsection 2.5.7) uses semantic query routing to gather peers with similar interests.

2.5.1 Why middleware

Middleware started to develop at the beginning of the 1980s together with the widespread of distributed systems. For application developers it was tedious and error prone to

convert application calls from the client to the server and ensure server response. Middleware simplified application development because [47]:

- Communication demanded complex parameters such as records, character strings or arrays to be transmitted from one device to another. These needed to be converted to lower level protocol implementations, usually to byte sequences.
- Two devices could have different encodings of data types in memory if they were not deployed on the same hardware platform and software not written in the same programming language. Application developers would have to map the types to each other.
- Parameters and return values might refer to components located on other devices. Application developers would have to implement object references in according to Internet domain names, port numbers and additional addressing information.
- Application developers would have to implement activations of a server component as a response to a client component request.
- An operation requested by a client had to ensure that the response from the server side is always carried out and that the parameters sent from the client match the server side parameters (type safety).
- After the request was sent, the client needed to wait for the result to return. Implementation of synchronization between two devices was non-trivial.
- Sometimes qualities of service were required that could not be guaranteed at the network level. For example, it might be required for different client requests to be implemented atomically, either completely or not at all (transaction support).

Middleware offered a layer that “glued together” applications across heterogenous platforms and offered abstractions simplifying application development. Putting as much as possible of the standard functionality into the middleware layer gave the application developers freedom to concentrate on more specific application functionality. They could now focus on the higher level programming, and omit time consuming low level details of object transferral between devices. Middleware produced separations of concerns.

2.5.2 Types of middleware

The remote procedure call (RPC) was the first type of middleware, used as a way to transparently call procedures located on other machines. The concept of middleware became popular and led to a number of different middleware types. Amongst the most widespread types are [22]:

- *RPC-based systems.* RPC provides the infrastructure necessary to transform procedure calls into remote procedure calls in a uniform and transparent manner. RPC systems are used as a foundation for almost all other forms of middleware. We described RPC briefly in subsection 2.2.1.
- *Transaction processing (TP) monitors.* TP monitors are likely the best-known form of middleware. Very simplified, TP monitors can be seen as RPC with transactional capabilities.
- *Object brokers.* RPC was designed and developed at a time when the predominant programming languages were imperative languages. When object-oriented platforms took over, platforms were developed to support invocation of remote objects, thereby leading to object brokers. These platforms were more advanced in their specification than most RPC systems, but they did not significantly differ from them in terms of implementation. In practice, most of them used RPC as the underlying mechanism to implement remote object calls. The most popular class of object are those based on the Common Object Request Broker Architecture (CORBA).
- *Object monitors.* When object brokers tried to specify and standardize the functionality of middleware platforms, it became apparent that much of this functionality was already available from TP monitors. At the same time, TP monitors had been extended to fit with object-oriented languages. The result of these two trends was convergence between TP monitors and object brokers that resulted in a hybrid system called object monitors. Object monitors can roughly be described as TP monitors extended with object-oriented interfaces.
- *Message-oriented middleware.* The earliest versions of RPC middleware had acknowledged that synchronous communication was not always optimal. Initially this was solved using asynchronous RPC. Later on, TP monitors extended this support with persistent message queuing systems. After a while these systems became middleware platforms on their own under the general name of message-oriented middleware (MOM). We described MOM in subsection 2.2.2.
- *Message brokers.* Message brokers are message-oriented middleware that has the capacity of transforming and filtering messages as they move through the queues. They can also dynamically select message recipients based on message content. In terms of basic infrastructure, message brokers are just queuing systems. The only difference is that application logic can be attached to the queues, allowing de-signers to implement more sophisticated interactions in an asynchronous

manner.

Of these, the RPC is of particular interest because it was the first type of middleware and thus explains the principles of middleware well. Moreover, the message-oriented middleware (MOM) is also important to our thesis because it forms the base for most more advance P2P middleware.

2.5.3 Characteristics of decentralized P2P middleware

P2P middleware differs from the middleware types described in the previous section. Obviously, as RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure has returned, it is not fitting. For a P2P network such tight-coupling between devices is undesirable as devices may be added and removed ad-hoc. The architecture represented by MOM where each device acts both as sender and receiver and interact through asynchronous transmission is far more appropriate. Thus P2P middleware are usually abstractions built on a basic MOM architecture.

Following from the client-server model is also that software usually has a centralized approach, entirely dependent on the server. Depending on one or a few devices being accessible, is undesirable in a fully decentralized network.

Even with MOM architecture, the ad-hoc nature of the peers raises issues about how to find devices and information in these types of networks. Typically a P2P middleware could offer:

- device and service discovery
- message routing
- query and search utilities
- caching/storage (related to the above mentioned issues).

New models for device and service discovery are needed because the network cannot be dependent on a server carrying out mapping between logical and physical addresses. Usually this is solved by trying to contact a bootstrapping node which gives the joining peer the IP-address of one or more existing peers, making the newcomer a part of the network. Each peer will usually only have information about its neighbours, which are peers that are directly connected to it in network [48].

The peer needs to locate information in the changing network. Thus, traditional routing protocols using pre-defined data access structures will not be fitting [32], and routing of queries becomes another issue. Often queries are flooded on the LAN to all the neighbour peers to find information. A side-issue within query routing and update messages is to avoid network congestion as messages are flooded in the network. Good routing protocols use additional techniques to keep message transmissions to a minimum.

As devices are added and removed, knowledge of where to find information must be further developed. Joseph divides search into three parts [49]:

1. identify what you want
2. work out where it is
3. download it

One example is semantic query routing described in 2.5.7, where a node's previous knowledge about another node's interests is used during search. Another example is JXTA's use of rendezvous peers (see subsection 2.5.6) or Gnutella's use of ultrapeers (see subsection 2.6.7).

In addition, choosing a decentralized approach, each device or node must itself be responsible for carrying some information on discovery, routing, search and querying in the network. The nodes need to retrieve this information and store it locally, usually in the cache, or more persistently on disk. Storing the information as devices comes and leaves leads to the possibility of storing stale information. Often this problem is simply dealt with by letting the cache empty itself as the network changes and new information continuously are added replacing the old.

Ding, Nutanong et al [48] describes two properties of a P2P network which also P2P middleware should abide:

- Scalability: There should be no algorithm or technical limitation to the size of the system, i.e. the P2P network should not be dependent upon the number of nodes participating.
- Reliability: The malfunction of any given node should not affect the whole system (or even any other nodes).

This is more or less the same statements as Kubiatowicz [40], as described in subsection 2.3.5.

The rest of the section describes four different decentralized P2P types of middleware that have freely available implementations. The subsections also describe how these implementation have solved issues related to P2P networks.

2.5.4 Bonjour

Bonjour, formerly Rendezvous, is Apple's trade name for its implementation of the Internet Engineering Task Force (IETF) Zeroconf protocol [50].

Bonjour provides automatic IP configuration if a dynamic host configuration protocol (DHCP) service is missing [51]. It also provides a service discovery replacement for domain name service (DNS), provided the peers are on the same subnet. It takes advantage of protocols that already exists like Advanced Function Printing (AFP), Server Message Block (SMB), Internet Printing Protocol (IPP) and HTTP in communication once the service on the devices has been discovered.

When a new computer or device is added to the network, Bonjour configures the device using a technique called link-local addressing (If a DHCP server is available, Bonjour uses the assigned IP address). Using local-link addressing, the device randomly selects an IP address from a predefined range of IP addresses set aside by the Internet Assigned Numbers Authority (IANA). Addresses are in the range 169.254.XXX.XXX. Afterwards, the computer sends a message out on the network to determine whether another device is already using the address. If the address is in use, the device randomly selects another address until it finds one that is available. When the device has assigned itself an IP address, it is ready to send and receive IP traffic on the network.

Once a device has been automatically configured to work on the network, it needs to discover services being offered by other devices on the network, as well as a way to tell other devices what services it offers. To share services, a device must create a unique name for each of its services and let the other devices on the network know of their existence. To do this, Bonjour uses DNS, which offers translation between human-friendly names and numbered IP addresses. To perform name services, Bonjour uses a variant of DNS called Multicast DNS-Service Discovery (mDNS-SD). An mDNS-SD

notification is query driven and retrieves the type of service (such as IPP printing), the name of the service (such as “Copy room printer”), IP and port addresses, and other optional information (such as the correct file format, for example PPD). Each device on the network receives the notification and stores the information. Applications running on the computer can use the information to create a list of services in their custom interface for the user to choose a service. But without a special DNS configuration, Bonjour only works on a LAN [50].

If a device is added to the network, it may query the network about a certain type of service. For example, the device may want to know what printers are available so it can create a list of printers for the user. The device queries the network for devices offering printing services. It receives responses from the devices that can print using the specified printing protocol and uses that information to create a list of printers for the user.

To minimize network traffic, Bonjour uses a range of techniques. For example, the multicast protocol is designed to reduce network traffic by issuing only one packet on the network that can be received by all devices. When a device queries the network for information, and the other devices on the network respond, all the devices receives all the responses. Since each device caches the information, the device does not need to query again. Furthermore, a device does not query before a service is requested by the user.

The source code of Bonjour is open source and freely available under the Apple Public Source Licence including software for UNIX, Linux, BSD, Solaris, Windows, Windows CE and Pocket PC [51]. Bonjour is used for many different services like finding shared music (iTunes), find shared photos (iPhoto), to find other users on the same subnet (iChat AV and Skype), to find digital video recorders (TiVo Desktop), and to find document collaborators (SubEthaEdit). Apple’s web browser Safari uses it to find local web servers and configuration pages for local devices and it is also used to advertise telephone services and configuration parameters to voice over IP (VoIP) phones and diallers [52].

An alternative to Bonjour is the Avahi project developed on Linux and other Unix-like desktops. Avahi is published under the less controversial Lesser General Public License (LGPL). Avahi is consided the default Zeroconf implementation on all Linux distributions and has also been ported to Apple's own Mac OS operating system.

2.5.5 Universal Plug and Play (UPnP)

Late 1999, an association of more than 340 vendors formed the Universal Plug and Play Forum. The forum defined the UPnP Device and Service Descriptions (originally called Device Control Protocols) after a common device architecture from Microsoft [53]. A number of companies today offer UPnP kits for implementations [54].

The basic building blocks of an UPnP network are *devices*, *services* and *control points*. The middleware is optimized for discovery and controlling surrounding devices and services.

A device is a container of services and nested devices. An example of a device can be a VCR which has a tape transport service, a tuner service and a clock service. Services will thus differ from device to device, and is documented in an XML *device description* document that the device must have. The device description includes vendor-specific, manufacturer information including the model and number, serial number, manufacturer name, URLs to vendor-specific Web sites etc. There is also a list of any embedded devices or services as well as URLs for control, eventing, and presentation. In addition, the device description also includes a list of properties.

The smallest unit of control in an UPnP network is a service. A service exposes actions and models its state with state variables. For instance, a clock service could be modelled as having a state variable, current_time, which defines the state of the clock, and two actions, set_time and get_time, which allow you to control the service. Similar to device description, this information is part of an XML *service description* standardized by the UPnP forum. An URL pointer to these service descriptions is contained within the device description document. Devices may contain multiple services.

A service in an UPnP device consists of a *state table*, a *control server* and an *event server*. The state table models the state of the service through state variables and updates them when the state changes. The control server receives action requests (such as set_time), executes them, updates the state table and returns responses. The event server publishes events to interested subscribers anytime the state of the service changes. For instance, the fire alarm service would send an event to interested subscribers when its state changes to “ringing”.

A control point is a controller capable of discovering and controlling other devices. Typially, in a home the controller point could be a PC controlling all other devices like VCR, DVD, washing machine etc. After the initial discovery, a control point could retrieve device description, service description and invoke actions to control services. It could also subscribe to a service event source, which sends an event anytime the state of service changes.

Figure 2-8 describes the UPnP specific protocols. UPnP vendors, UPnP Forum Working Committees and the UPnP Device Architecture document define the highest layer protocols used to implement UPnP. Based on the device architecture, the working committees define specifications according to device type such as VCR, dish washers and other appliances.

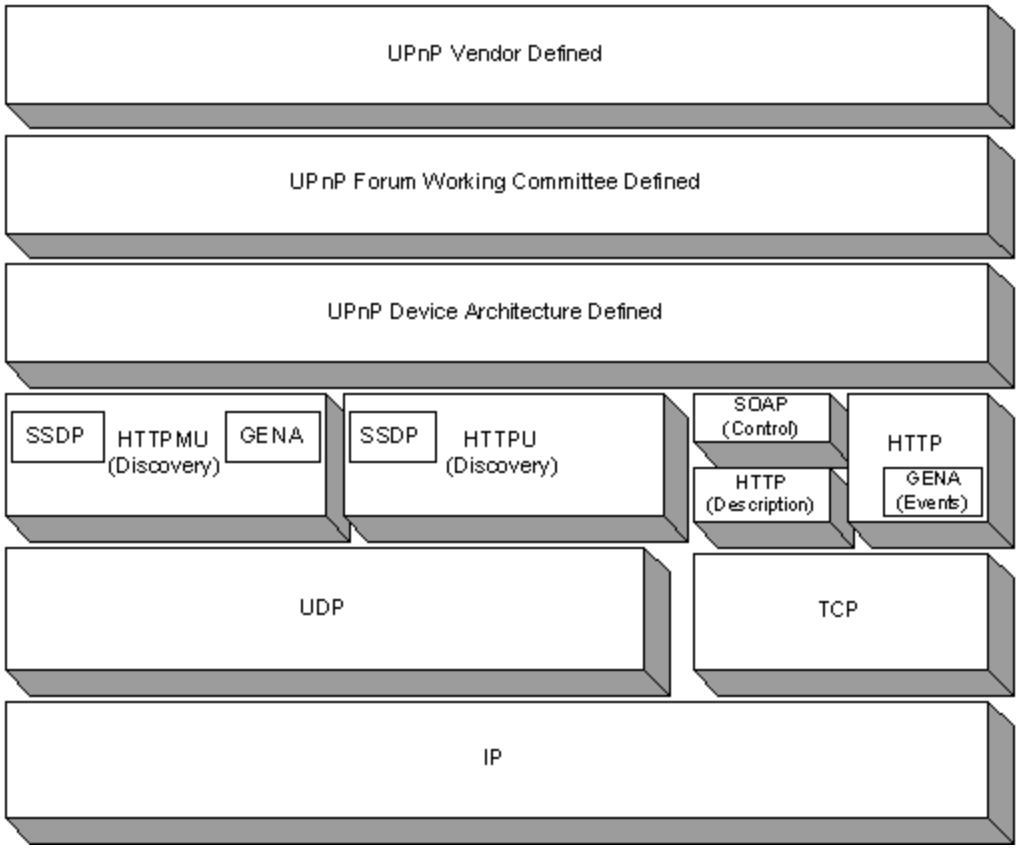

Figure 2-8 The UPnP protocol stack [53].

Two important protocols in the UPnP protocol stack are Simple Service Discovery Protocol (SSDP) and Generic Event Notification Architecture (GENA). SSDP defines

how network services can be discovered on the network. SSDP is built on two UDP variants of HTTP, Hypertext Transport Protocol Unicast (HTTPU) and Hypertext Transport Protocol Multicast (HTTPMU). SSDP defines methods both for a control point to locate resources of interest on the network (search), and for devices to announce their availability on the network (presence). Both control points and devices use SSDP.

In addition to discovery capabilities, SSDP also provides a way for a device and associated services to gracefully leave the network (through a bye-bye notification) and includes cache timeouts to purge stale information.

GENA was defined to provide the ability to send and receive notifications using HTTP and HTTPMU. GENA formats are used to create presence announcements sent via SSDP and to provide the ability to signal changes in service state for UPnP eventing. A control point interested in receiving event notifications will subscribe to an event source by sending a request that includes the service of interest, a location to send the events to and a subscription time for the event notification. Thus, GENA also defines the concepts of subscribers and publishers.

The protocol stack is used together with the following services:

- *Addressing:* If there is no DHCP server in the network, UPnP allows use of Auto-IP. With Auto-IP the device will choose an IP address in the 169.254.XXX.XXX range. After address selection, the address is tested on the network to see if it is already in use. If occupied, the device will randomly choose another address to test.

- *Discovery, advertisement:* Upon booting, a control point can send a multicast SSDP search request to discover devices and services that are available on the network. The receiving device examines the search criteria to determine if there is a match. If a match is found, a unicast SSDP is sent to the control point. The GENA format is used for the advertisements. The discovery message contains a few essential device specifications and its services, like its type, identifier, and a pointer to its XML device description document.

- *Discovery, search:* A search request is similar to advertisement, a device or control point sends an SSDP search request to search for services. The receiving device examines the search criteria to determine if there is a match. If a match is

found, a unicast SSDP is sent to the control point.

- *Description:* After a control point has discovered a device, the control point knows very little about it. For the control point to learn about the device or interact, the control point must retrieve the device's description from the URL provided by the device in the discovery message. The description service is carried out by means of HTTP over TCP.

- *Presentation:* Upon connecting, a device will send out multiple SSDP presence announcements advertising the services it supports. However, the capabilities of the presentation page are completely specified by the UPnP vendor. To implement a presentation page, an UPnP vendor may wish to use UPnP mechanisms for control and/or events, leveraging the device's existing capabilities.

- *Control:* Typically, control is carried out if for example a user has a PC and would like to control other devices from it. The PC then acts as a control point. To control a device, a control point sends an action request to a device's service. This means that a suitable control message is sent to the control URL for the service.

  In response to the control message, the service returns action specific values or fault codes. The information is encapsulated in UPnP specific formats and formatted using SOAP/XML, then transmitted using HTTP. A device must respond to control requests within 30 seconds.

- *Eventing:* An UPnP description for a service includes a list of actions the service responds to and a list of variables that model the state of the service at run time. The service publishes updates when these variables change, and a control point may subscribe to receive this information.

  The service publishes updates by sending event messages. Event messages contain the names of one or more state variables and the current value of those variables. These messages are also expressed in XML over HTTP and formatted using GENA.

All control points on a network that register for events receive the notifications. The state variables described in a service description can be evented. The service publishes updates when these variables change. A control point may subscribe to receive this information by sending a subscription message. The publisher of the event can accept this subscription and respond with a duration for the subscription. The subscriber can renew its subscription or cancel subscription when no longer interested.

First time a control point subscribes, an event message is sent that contains the names and values for all event variables and allows the subscriber to initialize its model of the state of the service. The event message is sent to all subscribers.

2.5.6 JXTA

JXTA (Juxtapose) is an open source P2P platform created by Sun Microsystems in 2001 [55] available in Java and C. A JXTA network is an ad hoc, multi-hop and adaptive network composed of connected peers. Peers may join or leave the network at any time, and network routes may change frequently.

Unique IDs are used for identification in the network. There are six types of JXTA entities which use JXTA IDs: peers, peer groups, pipes, contents, module classes and module specifications. A JXTA ID is defined by a Uniform Resource Name, URN, a form of URL that is intended to serve as a persistent, location-independent, resource identifier.

Together a group of peers form a peer group. Peers self-organize themselves into peer groups, where each group has agreed upon a common set of services. There are three motivations for creating a peer group: 1) To create a secure environment within the group, 2) to locate others with similar interests, like a document or a CPU sharing network and 3) to monitor the other peers for any special purpose (e.g. heartbeats, traffic introspection or accountability).

Peers can be four different types:

- A *minimal edge peer* can send and receive messages, but does not cache advertisements or route messages. Typically devices with limited resources like PDAs and cell phones would be minimal edge peers.

- A *full-featured edge peer* has the same functions as a minimal edge peer, but also it may cache advertisements and reply to discovery requests using its cache information.
- A *rendezvous peer* is like a full-featured edge peer, in addition it also forwards discovery requests. Edge peers send search and discovery requests to rendezvous peers which in turn forward all requests that they cannot answer themselves to other rendezvous peers. To avoid messages travelling many routing hops (any thus take too long time to get response) there is a “hop limit”, so called time-to-live (TTL), of seven hops. Loopbacks are prevented by maintaining the list of peers along the message path. Only rendezvous peers maintain a list of other known rendezvous peers and also the peers that are using it as a rendezvous. This structure significantly reduces the number of peers involved in the search for an advertisement. If a new device joins the network, it first contacts the rendezvous peer; if no such peer exists, the new peer automatically becomes the rendezvous peer itself.
- A *relay peer* maintains information about the routes to other peers and route messages to peers. Relay peers can forward messages on behalf of peers that cannot directly address another peer (e.g. NAT environments), bridging different physical and/or logical networks. The relay and rendezvous services can be implemented on the same peer.

Each rendezvous peer maintains its own list of known rendezvous peers in the peer group. A rendezvous peer may retrieve rendezvous information from a pre-defined set of bootstrapping, or seeding, rendezvous. Sun provides some servers for the purpose (see also subsection 5.3.3). They periodically select a given random number of rendezvous peers and send them a random list of their known rendezvous. Rendezvous peers may also periodically purge non-responding rendezvous peers. Thus, a loosely consistent network of known rendezvous peers is maintained.

When a peer publishes a new advertisement, the advertisement is indexed by the shared resource distributed index (SRDI) service using keys such as the advertisement name or ID. Only the indices of the advertisement are pushed to the rendezvous by SRDI, minimizing the amount of data that needs to be stored on the rendezvous peer. The rendezvous peer also pushes the index to additional rendezvous peers.

In order to send messages, the peers employ pipes. Pipes are the core mechanism for

exchanging messages between JXTA applications or services. Pipes can be either point-to-point, multicast (transmission to a group) or secure unicast pipes (a reliable point-to-point pipe).

A message is an object that is sent between JXTA peers; it is the basic unit of data exchange between peers. All JXTA network resources (such as peers, peer groups, pipes and services) are represented by advertisements. Advertisements are meta-data represented as XML documents. The JXTA protocols use advertisements to describe and publish a peer resource.

JXTA is based on six protocols:

- *Peer Discovery Protocol* is used by peers to publish their own advertisements or discover advertisements from other peers. If a peer group does not have its own discovery service, the Peer Discovery Protocol is used to probe peers for advertisements.
- *Peer Information Protocol* is used by peers to obtain status information (uptime, state, recent traffic etc) from other peers.
- *Peer Resolver Protocol* enables peers to send a generic query to one or more peers and receive one of more responses to the query. Unlike Peer Discovery Protocol and Peer Information Protocol which are used to query specific pre-defined information, this protocol allows peer services to define and exchange any arbitrary information they need.
- *Pipe Binding Protocol* is used to establish a pipe between one or more peers, connecting two or more pipe endpoints. A pipe can be viewed as an abstract named message queue; supporting create, binding, unbinding, delete, send and receive operations.
- *Endpoint Routing Protocol* defines a set of request/query messages that are used to find routing information. Path information includes an ordered sequence of relay peer IDs and time-to-live (TTL) that can be used to transmit a message to the destination.
- *Rendezvous Protocol* is a mechanism by which peers can subscribe or be a subscriber to a propagation (multicast) service. Rendezvous Protocol is responsible for propagating messages within a peer group and is used by the Peer Resolver Protocol and the Pipe Binding Protocol to propagate the messages. Moreover, the protocol controls the propagation of a message (TTL, loopback detection etc).

All JXTA protocols are asynchronous and based on a query/response model. JXTA peers are not required to implement all six protocols, only the ones that they use.

2.5.7 The Socialized.Net

Some P2P networks use a routing technique called semantic query routing. Semantic query routing focus more on the nature of the query to be routed than on the network topology in general. By evaluating the query answers, nodes that give fitting information as answers are prioritized rather than nodes giving less important information. This means that nodes with similar interests are grouped together.

Search is done by forwarding queries to a subset of nodes that is believed to possess matches to the search query [49]. For example a search for keyword A will make the routing node look up in a priority list the nodes associated with keyword A. The routing node would then choose a number of associated nodes holding the highest priority score. The querying node also establishes a direct link to the remote node, adding to the system's existing connections, leading to a gradual increase in connectivity. The effect is that all nodes gradually get more knowledge about the others, leading to a gradual increase in connectivity. An analogy used is to think of the nodes as humans that request a friend about something. If the friend does not know the answer, he can suggest another person that could help, which again might suggest another one and so on.

An issue in such networks is to handle malicious nodes spreading bogus information. The Socialized.Net (TSN) uses semantic query routing but extends it with the use of *preferences* and *reputations*, giving nodes a rudimentary social network to avoid unusable information. [38]. Nodes monitor their neighbour's replies, and would notice if they spread bogus information. Based on these observations together with possible user interaction a node calculates preferences. Nodes also spread their knowledge about another node's reputation (gossiping), making it possible to learn from other nodes' experiences. Querying is done by asking the neighbours with the highest scoring points.

Search is carried out in two steps: First a local filtering of incoming resource descriptions is set up based on the given query. Resource announcements and incoming replies are processed by the filter. A second optional step is to send the query to other nodes, which in turn will forward the query to more nodes. TSN can automatically resend the query at

a given interval, allowing the daemon to actively keep searching. It can be sent to different nodes every time. The search will not terminate until the user explicitly removes it from the daemon. A “copy-to” field allows application designers to specify recipients regardless of how the daemon itself routes the message. TSN is however not able to give guarantees that all possible resources are found, as only a “best effort” subset of nodes are queried. Also changes in the node infrastructure may affect the search.

TSN allows multiple addresses for each node [1]. If none of the addresses works, the node is assumed to be temporarily unavailable. Each address can be active, inactive or stale. When in an inactive state, the address can still be used, but active probing of the neighbour might be triggered. If a neighbour fails to respond, it will be set to stale. The address can still be tried, but ranks lowest. Last address is used if all addresses show the same response.

An address will receive points when in use. Also, it will receive more points when providing answers as opposed to routing only. The optimal path is sought for querying. TSN will monitor its own activity in order to decide its own interest. Thus, when having too many neighbours, only the neighbours with the highest scores are kept. Only addresses that have been active during a given time period will be kept. Very active, stable and well connected nodes can receive many points for their connectivity. Also, the local user can have an opinion about certain nodes, either due to excellence or disliked nodes.

An overview of the protocol layers are shown in Figure 2-9. The Socialized.Net is based on UDP communication over IP. Routing is carried out either directly or via other nodes. All nodes that route queries, will also cache routed information and thus at the same time update their semantic knowledge of the senders. With this structure, it is also possible to go beyond a network translation address (NAT) configuration and connect in WANs. To avoid very long routes with many hops, TSN uses Time To Live (TTL) counters. TTL gives a limit for the number of hops allowed before the search is terminated.

Each participating node is running a TSN daemon. Local applications can connect to various interfaces of the daemon. The http interface is a web based user interface, giving the user the possibility of directly access to the daemon. The Instant Messaging and XML-RPC interface allow applications to integrate with the TSN.

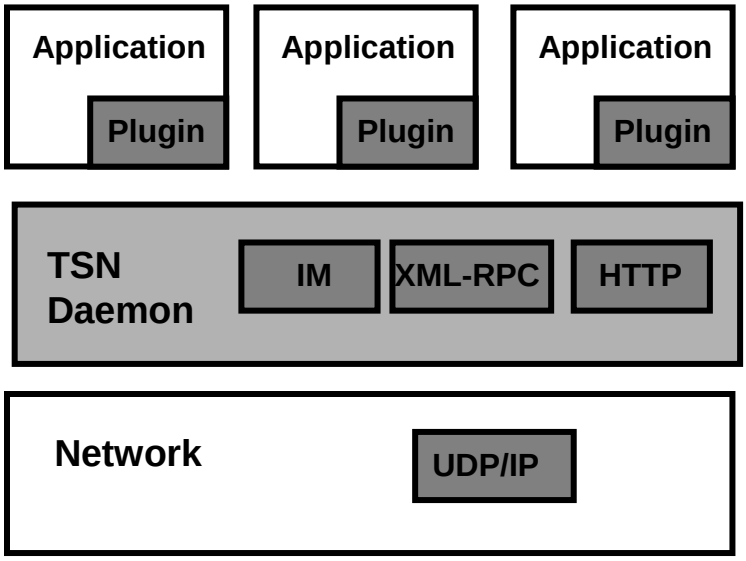

Figure 2-9 The Socialized.Net communication infrastructure [56].

Application developers can specify their own meta-data structures and matching policies for the meta-data. There are also policies describing how resource descriptions and queries are handled. For example, it is possible to limit caching of resource descriptions, or the scope of messages and more.

The TSN software has been developed at Norut-IT and is written in the Python programming language. The middleware is downloadable from the Internet [57].

2.6 Related works

A lot of work has already been carried out in the field of resource sharing. This section describes various related approaches to contrast and complement our resource sharing concept. We have not tried to make the section exhaustive, as there are numerous initiatives targeting resource or file exchange between devices and a wide variety of P2P networks whose purpose is file exchange. Instead we have focused on a few of the most profiled and relevant projects to show the variety of approaches and those working close to our approach.

Subsection 2.6.1 describes a shared resources approach by simple file transfer using Bluetooth, OBEX, ftp or others. In 2.6.2 we describe file sharing by Microsoft Shared Folders/SAMBA. In 2.6.3 and 2.6.4 we describe technologies that resemble each other in architecture, Microsoft Office Groove and iFolder. Google Docs & Spreadsheets are briefly described in 2.6.5. Using the JXTA platform, architecture of myJXTA is presented in 2.6.6. A JXTA version management system called GRAM is described in subsection 2.6.7. Finally in 2.6.8 we describe a decentralized representative for the P2P file sharing systems, Gnutella.

All approaches are compared to our model for casual resource sharing in subsection 3.6, Comparison to related works.

2.6.1 Bluetooth/OBEX/FTP

On close ranges, like people meeting ad-hoc, it is possible to transfer files one-by-one using Bluetooth or IrDA OBject EXchange (OBEX) protocol with assisting application protocols depending on device OS. While these solutions work well for personal area network (PANs) distances, they are not suitable for WAN. Moreover, there is no group concept, search possibilities, repositiory and other functionality beside offering file transferral.

Protocols like file transfer protocol (FTP) will in the same manner allow exchange of files between two peers via TCP/IP over the Internet. FTP is based on the client-server model, thus an FTP server can offer a secure repository similarly to any other file server. Other protocols like SSH, HTTP etc will function in the same manners. The downsides to using a server is typically additional routines such as registration for server access (filling out forms/contacting administrators explaining why you need access etc) and the need for network access between the client and the server. In contrast, servers also have benefits such as getting backup provided and a high uptime of the server itself.

2.6.2 Microsoft Shared Folders/SAMBA

With Microsoft (MS) shared folders it is possible to share out a part of a device’s harddisk to collaborators. MS shared folders can best be described as similar to disk server access, but without many of the additional routines related to servers themselves

since it is one of the client's devices that are usually shared. MS shared folders is thus based on the client-server paradigm, rather than P2P.

MS is relatively easy to configure which can explain some of the popularity. Moreover, it is shipped with the popular MS Windows operating systems so there is no need for additional software installations. MS shared folders use the same access control mechanisms including directory services as the shared device itself. It supports a multi-user environment, thus collaborating working groups can be organized. It includes concurrency control or locking of a remote file while a user is editing it [58].

MS shared folders runs on top of an application-level network protocol called server message block (SMB/CIFS). Depending on operating system and access mechanisms, MS shared folders uses transport protocols TCP, NetBIOS over TCP/IP or UDP/IP, NetBEUI or other NetBIOS transports [58]. Traditionally the shared folder access has only been available on LAN, and file access is normally prohibited by firewalls from devices outside the LAN. By means of virtual private networks it is possible to access the service from outside.

A drawback is that users are restricted to MS Windows operating systems, thus MS shared folders do not allow operating system heterogenity. However, Samba, a free software re-implementation of the SMB/CIFS networking protocol, offers shared folder integration with most Unix and Unix-like systems, also including Apple's Mac OS X Server. Samba is standard on nearly all distributions of Linux [59].

The SMB protocol also supports access to a number of other shared resources, like printers, scanners and serial ports.

2.6.3 Microsoft Office Groove

Groove was founded in 1997 by Ray Ozzie, the developer of Lotus Notes. Originally developed by Groove Networks, it is now a proprietary licensed product owned and developed by Microsoft as a component of the Office 2007 Enterprise suite [60]. Groove's main goal was to allow users to communicate directly with other users without relaying on a server. Others important goals were security and privacy, and flexibility [61]. Groove is a project management application based on the client-server model and

integrates chat, file sharing, calendar, discussion, picture sharing, and also allow third party tools to be integrated to improve the functionality.

A Groove user creates a workspace and then invites other people into it. Each person who responds to an invitation becomes a member of that workspace and is sent a copy of the workspace that is installed on his or her hard drive. All data is encrypted both on disk and over the network, with each workspace having a unique set of cryptographic keys. Groove also includes firewall/NAT transparency. Thus, a workspace is the private virtual location where users who are members interact and collaborate. After the initial connection, Groove synchronizes all copies through central servers via the Internet. When a member makes a change to the space, that change is sent to all copies for update. If that member is offline at the time the change is made, the change is queued and synchronized to other workspace members when the member comes back online.

Groove offers ad-hoc group formation. It uses a protocol Simple Symmetrical Transmission Protocol (SSTP), a small application-layer protocol designed to allow two programs to engage in bidirectional, asynchronous communication over both TCP and UDP protocols. In addition, Groove version 3.x also supports Extensible Messaging and Presence Protocol (XMPP, an XML communications technology) protocol for sending instant messages to users on an XMPP network.
MS Office Groove is linked with the MS operating systems and other MS programs.

2.6.4 iFolder

iFolder allows people to share folders of files of any type with each other. Currently, iFolder has support for Windows, Novell Linux Desktop and Mac OS X [62]. The iFolder client runs in two operating modes, *enterprise sharing* and *workgroup sharing*. In enterprise sharing, the iFolder Enterprise Server is used. The iFolder client first synchronizes the files in the iFolders found locally to the intermediate server, and thereafter replicates them to other computers. With the iFolder Enterprise server, it is also possible to access shared files located on the server via an Internet browser, as well as copy files from the server to other media.
In workgroup sharing, which is more relevant to our pilot, iFolder can share files and synchronize directly without an intermediate server. This is accomplished through add-on modules using Gaim, an open-source instant messaging client, and using Bonjour (see

subsection 2.5.4). The sharing capabilities of workgroup mode are currently under development.
By means of the iFolder client, files are saved into an iFolder and replicated in their entirety, either to other devices or to the iFolder server. As the files are edited and changed, the iFolder client keeps track of the changes and then only synchronizes the changed parts with the other devices.
iFolder (at least for the Linux client and server) is built on top of Simias, a generic data store and logic for the collections of information. Simias uses a local database as storage. It also handles synchronizations of the the collections from machine to machine. Security is handled through access levels which can be either Administrator, Read/Write or Read Only. It is possible to fetch user identities from an external source, for example from Novell's eDirectory service. Simias also includes an embedded web server for browser accessibility. The iFolder project is built on the .Net framework and Mono (software to develop and run .NET client and server applications on Linux, Solaris, Mac OS X, Windows, and Unix).
Like many other open source developments, the documentation for iFolders is not always up to date. Thus, implementation details may have changed.

2.6.5 Google Docs & Spreadsheets

It has been difficult finding any thorough technical information on Google Docs&Spreadsheets, as specification information beyond user descriptions is seemed to be held back. Thus the following information has mainly been fetched from Wikipedia [63]. Google Docs & Spreadsheets is a Web-based word processor and spreadsheet application offered by the company Google and used through a browser interface. Documents and spreadsheets can be created within the application itself, imported through a web interface, or sent via email. Documents can also be saved to the user's computer in a variety of formats, for example Microsofts Word and Excel format. By default, documents are saved to Google's servers. Documents that are opened are automatically saved to prevent data loss. Documents can be tagged and archived for organizational purposes as well as shared and edited by multiple users at the same time.

Google Docs & Spreadsheets does not support certain browsers such as Opera and Apple's Safari. There is also a limit on how much a user can store on his account. Each document must be under 500k plus 2MB for each embedded image.

Although text documents and spreadsheets can be optionally accessed through HTTPS, it is not set by default. There is also a potential security breach as accounts for all Google services have a unified login process. While a unified login simplifies access, it also represents a potential threat to security through cross-site scripting as the access to Google Docs & Spreadsheets then requires no password check. Cross-site scripting allows code injection by malicious web users into the web pages viewed by other users. Code injection is a technique to introduce code into a computer program or system by taking advantage of unenforced and unchecked assumptions the system makes about its inputs.

Originally Google's write program Writely ran on Microsoft ASP.NET, but has since 2006 been developed on a Linux-based platform.

2.6.6 myJXTA

myJXTA is a demo application written for the Java platform which illustrate key concepts of the JXTA platform and P2P. The myJXTA application provides functionality for secure one-to-one chat, group chat, and sharing, searching and downloading documents within a peer group [64]. The myJXTA application uses the JXTA platform core building blocks to discover, join, create groups, create a connection between two peers (chat) and a group of peers (group chat), as well as the resolver and endpoint routing protocol to search and download files. The system is available for Windows (95, 98, ME, 2000, NT, XP), Solaris, Linux, Unix, Mac OS X, or other Java enabled platforms. myJXTA uses a native GUI rather than using a browser.

New peer groups can be created using the JXTA network. The peer groups are public and can be created. It is also possible to join an existing peer group. Every time a user joins a new group, this peer group becomes the default peer group for chatting and/or file sharing.
For chat possibilities, it is possible to create private one-on-one chats. The chats are assigned a secure password and the messages are encrypted as they are sent over the network. For chats, it is possible using the JXTA Peer Group advertisement to create a JXTA invitation. The invitation can be sent to other myJXTA peers and functions like a business card. If the invitation is accepted, the peer will be added to the user list of known peers which allows for secure one-to-one chat.

Search for files and file sharing is always carried out within the current peer group and is based on the JXTA Content Manage Service (CMS). The CMS manages the shared content for a local peer, and allows applications to browse and download content from remote peers. Within CMS, each item of shared content is represented by unique content id and a content advertisement which provides meta-information about the content, such as its name, length, mime type, and description [65]. The CMS also provides a protocol based on JXTA pipes for transferring content between peers.

Each piece of shared content is referenced by a unique content identifier, using a 128-bit MD5 checksum generated from the content data. By using MD5 as unique identifier, it is easy to determine if two files shared by different peers are the same rather than relying on the content name or description.

The CMS manages a persistent store in the cache which includes references to the locally shared file content as well as their associated advertisements. These advertisements are stored as XML files. In the persistent store, only the references to shared files are maintained rather than copying the file contents. This saves disk space when sharing large media files. When content is shared, the MD5 is computed and the reference to the actual content is stored along with the MD5 checksum. When the content is subsequently retrieved by another peer, the content is verified to make sure that it has not changed since last shared.
The CMS service uses JXTA pipes (see subsection 2.6.6) for remote content request and retrieval. Each instance of CMS manages a single input pipe for receiving both content requests and responses. Request and response pipe advertisements are passed in each CMS message so once the initial content request pipe advertisement is discovered for a peer, subsequent pipe advertisements can be obtained from the messages themselves. This allows the CMS to utilize separate pipes for handling different message types, since the initial pipe is only needed to send the first request.

It is possible to allow automatically sharing of all files that have been searched and downloaded from a device by setting the auto share preferences. By default, myJXTA will not allow auto share. myJXTA also has additional options for configuration of rendezvous, router and other network services and protocols (see subsection 2.5.6).

An extended version of myJXTA, called myJXTA2 is under development. Very little information is available, but myJXTA2 aims to include resource search, JXTA import

and export facilities, text-to-speech through integration with Java speech synthesizer software FreeTTS and live graphs through the integration with the Prefuse tool (having features for data modeling, visualization and interaction) [66].

Like many other open source developments, the documentation for myJXTA is not always up to date. Thus, during the time of writing, implementation details in particular for the CMS may have changed.

2.6.7 GRAM

GRAM (Group Revision Assistance Management) is a decentralized P2P based software configuration management tool [45]. The system uses JXTA as a middleware platform.

Every peer holds a shared space synchronized with other peers, and a workspace for a user's ordinary editing. The workspace holds the configuration files and actual source codes for each user. The shared space functions as a repository where a collection of system configuration files and history files are kept, including file control meta data, latest source code, comments and revision records. Using JXTAs group concept, the shared space includes a new group directory to hold group related configuration files and messages for helping peer's administration and coordination. All files except the GRAM source codes are kept in the XML format which makes them interchangeable for adapting to different usages. Group data are synchronized and duplicated among peers in the shared spaces.

In terms of collision detection of source code, GRAM uses preventive approaches rather than file merging after collisions have occurred. GRAM provides the users with context-aware information of the group software development environment so users can be aware of what others are doing and how source codes are revised. This context-aware information includes information on which peers are currently logged on, which files is currently being edited, messages related to files, current version numbers and the possibilities of chatting with other group members using multicast messages. Moreover, GRAM uses agents within the workspaces to watch on the source files being edited, analyze the possibility of collisions by their cooperation, and alert users before possible collisions happen. The agents are connected with a proactive action database which consists of current user's editing processes.

It is the shared space service that checks whether a user is present and offers chatting and other communication services. It also contains file manipulation services based on delta techniques (see subsection 2.4.1). The workspace service has a diff processor (based on the GNU diff command in Unix) which discovers changes to files in different versions by comparing their texts.

The user interface offers a view of commonly shared files and groups, as well as group members, the source code for a selected file, a chatting module and additional information about files.

2.6.8 Gnutella

A number of P2P file sharing systems have been made like Gnutella, FreeNet, Morpheus, eDonkey, Napster and others. Of these, we describe Gnutella as it was the first decentralized P2P file sharing network to come into widespread use. The main motivation for use of a fully decentralized network has proved to be the difficulty to shut the network down or control its content as there is no centralized unit.

Gnutella was developed by Justin Frankel and Tom Pepper in early 2000, and was made available on the Internet only a day before AOL stopped the program distribution over legal concerns. Yet after a few days, the protocol had been reverse engineered, and compatible open-source clones began to appear. In the first version of Gnutella, a client had to know the address of at least one other node to join the network [67, 68]. Once the client had connected to the node, it could broadcast a ping to find the addresses of other nodes. Each node maintained a connection to a number of other nodes, usually about five. To search the network for a resource a peer sent a query message to each of the nodes it was connected to. They then forward the message and when a resource was found the result i.e. resource name and address was propagated back along the path. The number of nodes that get queried would be controlled by using a Time-To-Live (TTL) counter (see subsection 2.5.7).

The first version of Gnutella did not scale well. Flooding the network by messages would lead to network congestion once the number of nodes expanded above a limit. When node saturation occurred, the network became fragmented. Moreover, searching the network was roughly of exponential complexity. To address the problems of bottlenecks, Gnutella developers implemented a tiered system of ultrapeers and leaves. Instead of all

nodes being considered equal, nodes entering into the network were kept at the edge of the network as a leaf, not responsible for any routing, and nodes which were capable of routing messages were promoted to ultrapeers, which would accept leaf connections and route searches and network maintenance messages. Search results were now delivered over UDP directly to the node which initiated the search, respectively a proxying peer, usually an ultrapeer of the node. The queries carried the IP address and port number of either node. It lowered the amount of traffic routed through the Gnutella network significantly, making it more scalable.

If the user decides to download the file, they negotiate the file transfer. If the node which has the requested file is not behind a firewall, the querying node can connect to it directly. However, if the node is behind a firewall, stopping incoming connections, the client wanting to download a file will send it a so called push request to the remote peer to initiate the connection instead (to “push” the file). At first, these push requests were routed along the original chain it used to send the query. However, this was however rather unreliable because routes would often break and routed packets are always subject to flow control. Therefore so called push proxies were introduced. Push proxies are usually the ultrapeers of a leaf node and they are announced in search results. The client connects to one of these push proxies using a HTTP request and the proxy sends a push request to leaf on behalf of the client. Normally, it is also possible to send a push request over UDP to the push proxy which is more efficient than using TCP. Push proxies have two advantages: First, ultrapeer-leaf connections are more stable than routes, which makes push requests much more reliable. Second, it reduces the amount of traffic routed through the Gnutella network.
When a user disconnects, the client software saves the list of nodes that it was actively connected to and those collected from ping response (pong) packets for use the next time it attempts to connect so that it becomes independent from any kind of bootstrap services.

Additionally the Gnutella has adopted a number of other techniques to reduce traffic overhead and make searches more efficient. Most notable are QRP (Query Routing Protocol) and DQ (Dynamic Querying). With QRP a search reaches only those clients which are likely to have the files, so rare files searches grow considerably more efficient, and with DQ the search stops as soon as the program has acquired enough search results, which reduces the amount of traffic caused by popular searches.

The file transfers themselves are handled using HTTP. Gnutella is based on it’s own protocol being further developed and maintained by the Gnutella Developer Forum.

3 Casual resource sharing with shared virtual folders

The chapter presents the concept of shared virtual folders (SVF) and how we will use it for casual resource sharing. First we present some scenarios on where and how an SVF is foreseen to be used in 3.1. In 3.2 we define what we mean by casual resource sharing together with other related expressions. In 3.3 we look into how an SVF functions and in 3.4 we extend the SVF concept to also include simple versioning detection. In order to further clarify the concept of an SVF, in 3.5 we define a set of properties that a SVF will have, and a set of operations that the users can carry out on an SVF. The chapter ends with a comparison of SVF contrasted to other related work described in the previous chapter, section 2.6.

3.1 Scenarios

The thesis work originates from the need of exchanging resources in an ad-hoc manner. We envision the application to be used by people exchanging resources in collaboration or project groups for a longer or shorter period of time, but lack server access. An example could be children playing in a local football team, where some of the parents would be team board members. The team board members get together on a regular basis and some members bring along their computers. The computers are used for minutes, budget estimations, letters and other documents. It functions as the board's electronic archive.

The board meetings take place in a public cafe where a WLAN zone with Internet connection is available. The board has network access but need a common repository for their documents. We propose to let the board members use SVF. The SVF could consist of parts of all board members harddisks. Files stored on the shared parts of the harddisks become available to all board members visible as a common repository.

The SVF itself will keep track of where the different files are located, hiding location details from the users. Moreover, it would provide security mechanisms so the repository can only be accessed by the SVF members. The SVF members need not worry about a third party administrator getting access to their documents since there are no servers involved. They can log on to the SVF or off as each device owner wish. It is also possible to update a document within the repository of an SVF. Since there is a risk of a board

member updating a document without telling the others, versioning detection within the SVF is supported by the application. Should the board members wish to divide themselves into smaller groups working independently on separate networks, the SVF will split itself as well. The members in each group will only see the shared resources from devices they are currently networked with. When they join again the two instances of the SVF will melt back into one large group. Updates carried out on any of the documents stored in the repository will be visible to the others.

There is no requirement for Internet connectivity. As new members join the board, they can get SVF membership through invitation from current members who pass on the credentials for group joining. If any members withdraw from the board, they may also withdraw from the SVF membership. It is not possible for the SVF members to evict any other members. The SVF cease to exist when the last device withdraws its own SVF membership.

Another scenario is university students from activity groups storing images and films from holidays and week-end trips they have been to. While the students have common servers available for university courses, they are not allowed to use disk space for other purposes than assignments. Thus they establish a SVF amongst them consisting of parts of harddisks from their personal devices. After a while the harddisks fill up, leaving the students with the option of either deleting older images or not adding new ones. A third option is to find additional space. If for example one of the members has an Xbox at home with available disk capacity, the Xbox could be included in the SVF to function as backup storage. The SVF will provide a simple means for exchange of files between the Xbox and another device, for example a portable PC. The Xbox owner could transfer some older, less popular images to the Xbox. If anyone requests the content, she will be able to get the content for them later by moving music from the Xbox at home back to her portable PC. Afterwards, taking her portable PC with her to the university campus, the SVF members can access her shared harddisk from the campus network.

3.2 Casual resource sharing

By *casual resource sharing* we mean ad-hoc exchange of resources between computing devices connected in a network. By *resources* we include any exchange of data using any kind of protocols, for example web pages download, text documents and images

exchange, chat-functionality, audio-streaming, video-conferencing, game playing interaction etc. For our implementation we have narrowed resources down to file sharing only to avoid the pilot becoming too large. Thus in the succeeding text, the words “file” and “resource” will be used interchangeably.

A *device* could be any computer that can be configured and networked. By *casual* we mean resource exchange not usually planned in advance, although it could be. It happens ad-hoc as people interact during work or through entertainment. This type of interaction cannot rely on stationary servers to support resource exchange, although servers, if occasionally accessible, could be taken advantage of.

3.3 Shared virtual folders

Shared virtual folders (SVF) can be looked upon as a concept with similarities to tuple spaces [29]. A SVF is a repository shared between a number of devices connected to the same network. The *shared folder* is *virtual* because all users see it as one folder or repository, while in reality it consists of a several different disks where the resources are located.

During network connection, the devices discover each other. In order to start exchanging resources, one of the devices must *initiate establishment* of a SVF. A SVF is initiated by giving the SVF a name and issuing a group *advertisement* to ensure all potential participants receive information about the group. In addition, a password or another credential to the group must be provided to potential new members.

If the others *logs on* to the group, they will be *registered as members* of the group. If they choose to join, they will become *member devices* of the SVF, otherwise the SVF will be established without them. The SVF will always be established with at least the device that initiated establishment.

Consider Figure 3-1. Suppose A initiates establishment of the SVF by creating a new SVF and joining it herself. Moreover, a group *advertisement* must be issued to B and C before they will be able to join the group. Also some credential for authentication, like a password must be known to all parties. Should both B and C decline to join, a SVF will still be established with A as the only member. If both B and C accept, the SVF will be

established with three participants. In Figure 3-1 both devices have accepted so that devices A, B and C share a SVF and are located on the same network.

Within the network, the devices can share resources with each other. For example in Figure 3-1, A has *added resources* a and b, B has *added resources* c and d, while C has *added resources* e and f. Thus the common *repository* of the SVF contains resources a, b, c, d, e and f which all members has access to as long as they are connected to the same network.

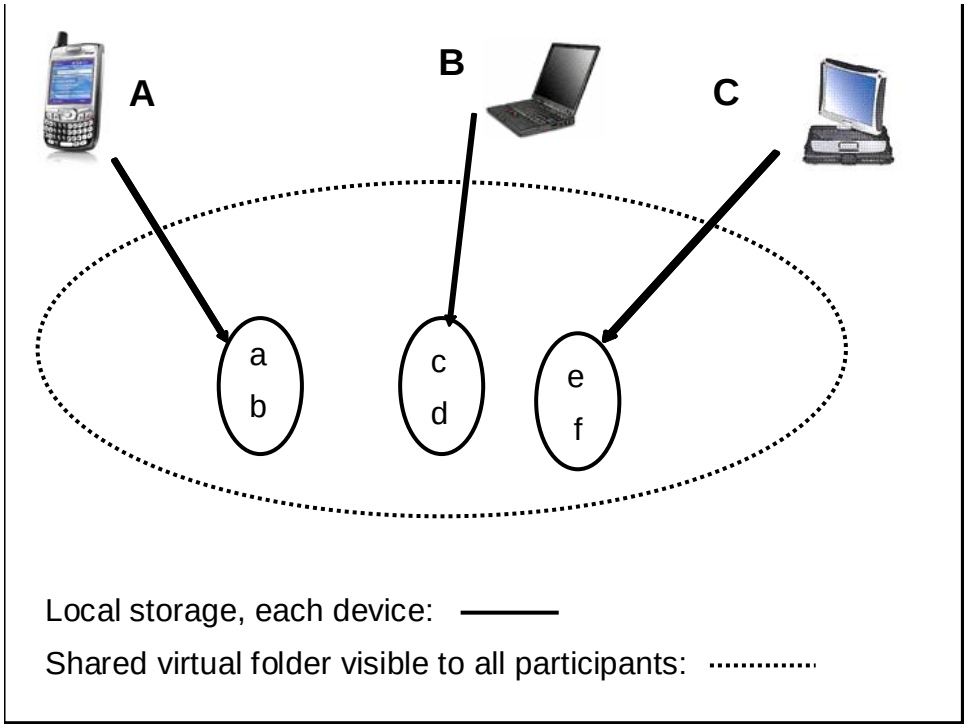

Figure 3-1 Example of a shared virtual folder (SVF).

While all devices have the name of all resources kept visible to them, the resource itself has not been transferred to any of the others, they have only received *resource notifications*. Thus all resources remain on their local disks of the participant which provided the resource. As long as the resources are not *removed* from the repository, a SVF participant can *request a download* of a resource any time. If the request is successful, the resource itself will be *downloaded* to the requester.

For example if A does not wish to download resources c, d, e and f instantly to her harddisk, she can wait until she needs the resources. The downside of waiting is risking

that device B and C has removed their resources from their parts of the *repository*. If B requests a download of resource f and the request is successful, resource f will now be on B's local disk as well and the resource will be listed twice in the SVF repository.

If any of the participants updates their resources, simple versioning detection will be carried out, as explained in section 4.5.

The devices may at any time *log off* the SVF. If they do, their resources will still be visible to the others as long as they stay in their SVF *session* and do not log off. However, should they request any of the resources which only resides on the device that has left, the request for download will not be successful and thus not carried out.

A device could also risk never to be re-connected with the others. If so, the resources that have not been downloaded and stored locally will be lost. Also the notification of resources that are not locally available will be lost.

The SVF permits a device to *log on* and *log off* the SVF ad-hoc. All devices will always have access to all their member SVFs regardless of whether they are connected to a network or not. A device could be a member of as many SVFs as the hardware and software of his device allows. For example, A, B and C can establish a group called "chemistry" and another called "mathematics". They could all share the SVFs. In addition A could share a common SVF called "computer science" with D, where B and C do not have access. B again could share a group called "French" with E and F to which A, C and D do not have access and so forth.

While many groups can be established, it is not allowed to be logged on to more than one group at a time.

While it is possible to establish many different groups, it is also possible to create different *instances* of the same group provided that the group members have split into several networks that are not connected. For example in Figure 3-2, A and B are on one network, while C is alone on another. A and B will have access to each other's resources, while C will be on her own. If they all get together on the same network, there will be only one common SVF. Participants of an SVF sharing the same network would never separate.

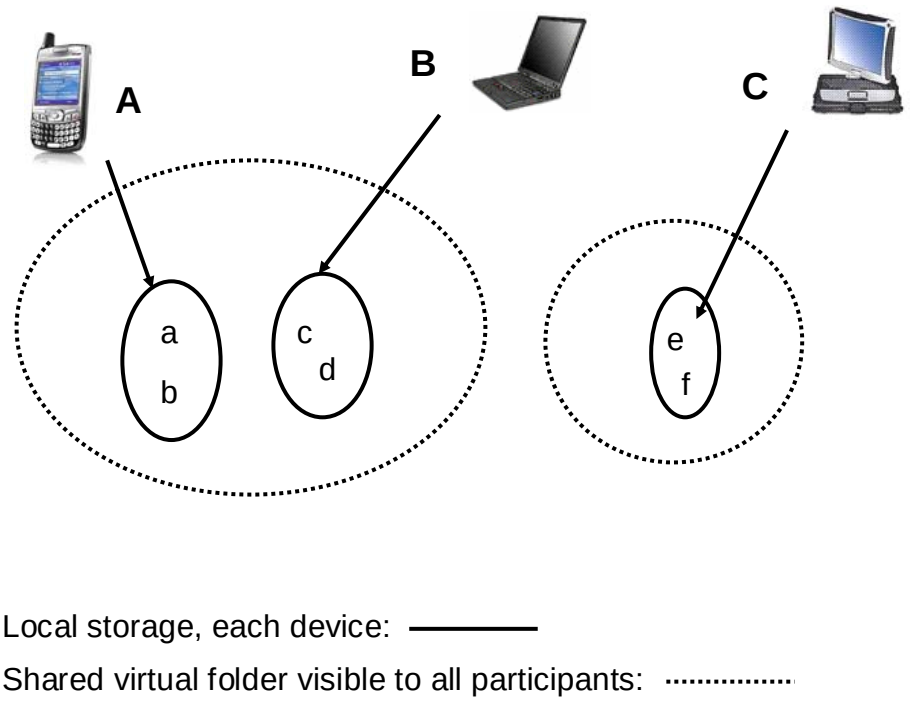

Figure 3-2 Two instances of a shared virtual folder.

All SVFs has their own *id* which is unique. In addition they also have a human-friendly *name*, which may not be unique such as the previously mentioned “mathematics” or “French”. The human-friendly name could be re-established or two groups could be given the same name by coincidence. If a SVF is *removed*, the name can be given to a new group, but not the id. If the SVF is removed, but the advertisement has been kept, the group can be reconstructed. After a while the advertisement itself will also be removed automatically and then the group can no longer be reconstructed.

A particular SVF could be described as the following triplet:

$SVF_{=} <D, R, id>$

where id is a unique identification that identifies the SVF. D is the devices *registered as members* of SVF and R is the *resources* they offer to the SVF repository. Both D and R could vary over time.

As an example of varying devices registered as members, suppose when A, B or C have finished the collaboration, C do not wish to be a part of the SVF anymore. Thus she

withdraws her *SVF membership*. A and B chooses to stay in the SVF as they are not finished with their collaboration.

As an example of varying resources, suppose device A at some point in time wishes to *withdraw resource* a and instead *add resource* g and h. The repository will then be updated to b, c, d, e, f, g and h if all three devices are connected to the same network. Moreover, the same resource can be shared out to several SVFs. For example a resource b can be added to the “mathematics” group, but also to the “French” group if desirable.

If a user has more than one device, it is of course possible to set up a SVF between his own devices as well to transfer resources from one device to another. It is also possible for these devices to become members of many SVFs at the same time as the different memberships will be independent of each other.

It is not possible to evict a member from an SVF. Only the participant themselves can withdraw their own SVF membership. Suppose A and B wanted to evict C, who refused to leave the SVF. A and B cannot evict C, but they can both withdraw their resources and membership from the SVF and re-establish a new SVF without giving C access to the new password or other credentials.

As a part of the SVF properties of a closed group concept, we will assume that for most initiated groups the group members already know each other as they start to collaborate. Thus if the members are not trustworthy, they can be held responsible for their actions as opposed to large P2P networks where participants easily can be anonymous. We will also assume that the chances of re-finding a file and thus persistence (see subsection 2.3.5) within the SVF are good as people can be held accountable. Also injection of unsolicited files will likely be less than for open P2P networks.

We have not seen the need to include any incentive mechanisms (see subsection 2.3.4) within the SVF because we assume the users will get incentives for collaboration through previous knowledge of each other. Similarly, if users are aware of each others identity, they may use other means of sanctions if participants are not behaving well during SVF sessions.

3.4 Versioning detection

When users collaborate, they often not only want to exchange resources, but also collaborate by updating common documents. On a server the operating system will carry out some simple versioning detection like changing modification dates of a file when it is updated. However, this will not be detected when the repository is located on several devices. Thus simple versioning detection might be of interest if several persons simultaneously would work on the same files.

Again consider Figure 3-2, suppose C had already downloaded resource a before separation. Also suppose C *opened the resource, wrote* to *the resource* and *saved the resource* again when located on the other network. Also assume that A did the same thing. Afterwards they all got together again on the same network. Likely the participants would be interested in knowing what had happened to the resource still available from both C's and A's devices.

For simplicity each SVF member is allowed to access all resources in the repository, but must download a resource before *reading* or *writing* to it. Thus a peer is not given write access to remotely stored resources. Each member of the SVF can be sure that the resources they provide for the repository will not be changed as long as the resources are stored on their local disk. Downloaded resources will automatically be a part of the repository as well, but can be *updated*.

In order to carry out versioning detection, the repository must be able to identify two identical resources and two different versions of the same resource. For example two files are *identical* if one of the file copies has been downloaded from another device and not been modified afterwards. Two file copies are not considered identical if for example a file from the repository is copied and issued to another device via e-mail and then imported into the SVF.

If a resource is fetched into the repository by SVF download, modified within the repository and thereafter saved under the same name again, it is called a new version of the resource. In the example of A, B and C where both A and C had updated the same resource, the resource will be marked with version 2 in both cases. This will be enough for the parties to know that the resource has been updated, but they will not know whether A and C both have created their own *successors* of the resource a or whether A

had updated the resource and sent C a copy of the successor. Thus if a resource has copies, the copies will be identified as having the same author, filename and version.

Our SVF will thus be based on trying to keep a file uniquely defined within a group by a combination of filename, author and version. This is analogue to an operating system's demand of never storing a file with the same filename twice within a directory. Thus two files with the same name and version can exist on different devices, as they would either be copies of one another or they would have different authors. The peer that adds a resource to the repository will be set at the author. Similarly, the peer that updates a resource will be the author. If files have different authors, it will be an indication of two different files even with the same filename and version number. A file cannot be downloaded or added to the local repository if these three criteria already match an existing file in the local repository. A file will alo be detected during version update if it has been overwritten to match another file within the repository. If detected, a user must store the file under another name, or the file will be deleted.

If a file is written to without ever being copied, it will not be a new version of the file as long as only one peer has seen the previous version. In these cases, only the modification date will be updated.

Each time a resource has been updated, a *resource notification* will be issued unless the previous version, the *predecessor*, was not submitted to anyone. If so the version number will not be updated but remains the same as long as only the resource owner has seen the intermediate copy.

3.5 SVF operations and properties

In order to understand more clearly the concept of an SVF, the SVF could be described through *operations* and *properties*. SVF properties define abilities designed into the system that the users must relate to. SVF operations are carried out by the users of the SVF as they wish.

SVF properties

id	Each SVF has a unique id.
name	Each SVF has a name which is not unique.
resources	Files that are available and belong to the SVF. The number of files may change dynamically as users add or remove files to the repository.
member devices	The devices with access to the SVF resources (files). A member device can be a member of more than one SVF simultaneously.
repository	Can be looked upon as common storage where all resources are kept. The common storage contains resources that the member devices wish to share.
resource notification	All member devices currently logged on to the SVF issue a list of all resources they wish to share. The resources themselves are not sent, but are physically stored on that member device.
advertisement	A potential new group member must receive a group announcement called advertisement before being able to join the group.
SVF instance	As long as all member devices are connected to the same network there will be only one instance of an SVF running. If the member devices are located on several separate networks, there can be several instances of the same SVF running simultaneously.
initiating device	The device establishing the SVF. This device will always be a member of the SVF at the time of SVF creation.
version	A resource can have several versions available in the repository.
predecessor	The resource (file) version which comes before.
successor	The resource (file) version which comes after.
identical files	Two resources are identical if one of the file copies has been downloaded from another device and not been modified afterwards.
repository	A virtual place where the SVF keeps all resources. In reality it consists of disk space from all participants.

SVF operations

initiate SVF establishment	In order to share resources, one device must create a new SVF. An initiated SVF will always be established. The SVF established will always contain at least one member device which is the initiating device.
register as member	A device has received the password or other credentials in order to join the SVF. It has also received group information or created a group and logged on to the SVF.
log on	A device member logged on to the SVF can see the resources in the repository and may download resources. A device becomes a member of an SVF the first time it logs in to the SVF.
log off	A device that logs off the SVF has temporarily lost access to the resources in the repository and to download resources. Access can be regained next time the device logs on to the SVF.
resource notification	A device adding or removing a resource in the repository will automatically issue a notification message. Sometimes this will happen also if a resource is updated.
request a download	In order to download a resource a request must be issued onto the SVF.
download resources	All logged in member devices can view all resources available in the common repository. These devices may download a resource if they want to look at the actual file contents and modify the file.
withdraw resource	A member device may remove a resource from the repository that it previously has offered other member devices.
add resource	A member device may add a resource to the repository so other device members can download the resource if they wish.
withdraw membership	If a member device wish never to be connected to the SVF again, it can withdraw its membership from the SVF.
remove SVF	When the last member device has withdrawn its membership, the SVF ceases to exist. As long as the advertisement is available, the group can be re-established.

	If the advertisement is lost, the SVF name can be recycled, but not the id.
open resource	Open a resource for reading or writing.
read a resource	Read the content of a resource.
write to/update a resource	Change a resource by writing to it.
saving a resource	Saving it back to the repository after writing to it.

Other expressions used:

Session	The time between log on and log off for a member device.

3.6 Comparison to related works

As mentioned in subsection 2.6, there are quite a few alternatives to SVFs. These can usually be divided into two categories:

1) alternatives based the client-server model
2) alternatives based the P2P model (including SVFs).

Using the first alternative, access to a common server like a web hotel or a file server (through HTTP, FTP, SSH or similar protocols) would solve the problem of storage in a simple way. It would usually also solve issues of getting proper backup, high server uptime and a well-defined security regime.

However, using servers has the disadvantages that the users have to administer the disk space, getting hold of different file versions and carry out versioning control between them, cleaning up and removing old files etc. within the groups. Also, they have to arrange access to the server in advance, typically through a user agreement either online or otherwise. Thus some routines must be planned in advance. There are file servers publicly available online (for example through anonymous ftp), but without security checking there is no concept for closed groups as anybody may remove or change file content.

In addition servers require network access to become available, for example through Internet. Just a common network between the participants is not sufficient. While Internet usually is available, server access also requires the users to trust a third party with their

contents. Being subjected to a third party security routines can be undesirable. For example, physicians in different locations discussing a patient's x-ray may be hesitant to trust a third party with patient information without further guarantees of security routines.

Google Docs&Spreadsheets is a good example of these server solutions. While the administrational process of getting access to using the servers are very simplified and the server uptime is excellent, there is a limit to disk space and document size per user. Also typically for server solutions, one has to have Internet access and trust a third party with user content.

Microsoft Shared Folders and SAMBA described in 2.6.2 are also based on the client-server model, where one of the participating devices now functions as a server. This type of software may be more ad-hoc since software usually comes installed with the operating system and since there is no additional system administrator the collaborators have to relate to. Furthermore, there is no need for network access to a server, only between the participants themselves. These solutions also comes with a security regime and thus group support [58]. Moreover, they do not require network access to a stationary server for collaboration, just access between involved devices.

Still some of the same problems as for other client-server models occur, like the need to administer how users use their common disk space (file modification and update, file removal etc). It also has the undesirable effect of usually not having equally high uptime as an ordinary server.

Solutions like Microsoft Groove Office and at least the enterprise sharing part of iFolder are also based on a client-server model [61, 62]. They also offer a browser interface which simplify use, but require Internet access. Microsoft Groove Office in particular has an advantage as a number of other applications also can be used beside just file sharing. The drawback to vast functionality could be that there is a higher user threshold during initial use of the solution.

The second alternative is using the P2P model. The advantage is no need for Internet access or other network access to reach a server. For example the iFolder workgroup sharing and the myJXTA application described in 2.6.4 and 2.6.6 appear conceptually very similar to the SVF. myJXTA has many similarities with SVF, but it's main aim is to demonstrate the functionality of the JXTA platform whereas for SVF the goal is not to

show middleware functionality but to build solutions for causal collaboration. Thus myJXTA relies on using the JXTA cache, which means that it is not possible to guarantee that a group will still be available the next time a device logs on. While myJXTA too has a casual approach to file exchange (besides chatting), it does not have any versioning detection available in the current version. iFolder workgroup sharing will likely offer re-establishment of a group, similarly to SVF. It is uncertain how iFolder will handle version detection as the implementation of workgroup sharing is not finished.

The project with most similarity to our concept is likely GRAM described in 2.6.7. It is based on the JXTA middleware platform, and has a shared file repository that is central to the application. The architecture uses a database for repository storage. It also offers full software configuration management, a much more extensive version detection and control than for our version. In addition other resources are also shared, like messaging and chatting. The difference between the two is that GRAM is meant for software configuration management, whereas our application is a simple tool meant for users without any technical background that wants to share files and in general collaborate.

The Interactive Workspaces project at the Stanford University (see subsection 2.2.4) also has similarites with our model also because they both are based on the tuplespace model. However, this project was restricted to location boundaries, and were also far more pervasive when interacting between different user interfaces through integrated user views.

P2P file sharing systems like Gnutella described in 2.6.7 does not usually provide a closed group concept, thus everybody would have access to all files in these types of networks [68]. In some of these networks, it may also be difficult to get hold of files that are not particularly popular, as they often are replaced with more popular content.

Both the Bluetooth protocols and OBEX as described in 2.6.1 are relatively easy to use and can be considered building blocks of both a P2P system and a client-server model. These protocols can be used for transferring files, but has no group concept attached to it. Moreover, they are not suitable for WAN communication, nor do they keep any form of versioning detection beside what is available through the operating system.

4 Application design

This chapter describes the overall design of an SVF implementation which will be used to prove our SVF concept. In 4.1 we start by presenting some functionality criteria for our SVF implementation, while in section 4.2 we describe the application's main architecture. For our application we will also need a suitable middleware. 4.3 outlines why we have chosen JXTA as our middleware amongst the four implementations previously described in section 2.5. Section 4.4 deals with more design issues on casual resource sharing and the passing of notifications, while 4.5 describes design issues for the versioning detection system. Section 4.6 explains how organization and repository of the architecture will be, while the last section, 4.7, describes in overall terms the principles behind the application's graphical user interface (GUI).

In this and the following chapters, the words "SVF" and "group" will be used interchangeably since we will base our SVF concept on the JXTA group functionality.

4.1 Functionality criteria

Based on the functionality described in the previous chapter, we want the application to have the following capabilities:

- Casual resource sharing should be carried out through a closed group concept described as shared virtual folders (SVF). The groups should be closed in order to avoid outsiders to read or write to files, to remove them or to add unwanted files.
- Persistent storage using disks as well as the cache. The cache itself will be overwritten eventually, thus it is desirable to employ a more permanent storage.
- Traffic between devices should be kept to a minimum to avoid network congestion. The middleware will issue coordination messages also, thus application messages should be kept at a minimum.
- The application should function both on WANs as well as LANs. Ideally we would like our application to function everywhere regardless of distance or location.
- Establishment of new SVFs should not be limited by the application. Ideally it should be possible to create as many groups as desirable because we assume it will make resource sharing more casual.

- Resources in an SVF can be added or removed at any time. Again we wish to allow resource sharing to be as casual as possible.
- All resources made available in a SVF will be visible to the others within the group. All members can also download all resources should they wish so. In order to simplify and include inexperienced users, we allow all resources to be shared without further restrictions.
- A member can only withdraw his own membership from a SVF. While more complicated designs could be implemented, our model will emphasis simplicity.
- In order to remove a SVF, all participants must withdraw their memberships. As long as a member has not withdrawn his membership, it should be possible for him to get hold of at least his own files.
- The SVF should contain versioning detection when users modify a document. Since we believe that users will be updating files within the repository during collaboration, there should be a structure to handle versioning detection within the SVF.
- The application should focus on simplicity at the expense of user freedom. Only basic functionality should be offered and the GUI should provide little freedom through configurations to also attract inexperienced users.
- The application should be agnostic to various operating systems and devices in the largest degree possible. It should be possible to run the application regardless of device type or operating system.
- The device and service discovery should be highly dynamic. Devices and files should be allowed to arise and vanish without notice to accommodate ad-hoc behaviour.

4.2 Application architecture

In this section we outline the main application architecture. We have built our architecture around the tuplespace model described in section 2.2.4. We could also have used other models, for example both one-way RMI/RPC (see subsection 2.2.1) and messages in MOMs (subsection 2.2.2) are basic building blocks in distributed systems and thus could be used. But it would require more time to start with the very basic building blocks rather than to make further use of middleware abstractions. Programming everything from scratch could give a fast application, but development would be time consuming. Another advantage of using middleware is separations of concerns as

described in subsection 2.5.1, which is important in order to not clutter up the application and to make error search and correction easier.

All devices participating in a SVF holds their local part of the SVF repository. Together the parts form the SVF repository quite analogues to the tuplespace. All member devices may add (analogue to the *out* command) or remove files (the *in* command), read the files (*rd* command) or write to the files (*in* in combination with the *out* command).

When considering a resource sharing concept, we could also have used the publish-subscribe model (see subsection 2.2.3) for our application architecture. While the publish-subscribe model is well established, it does not provide any temporal decoupling so that devices removed during subscription can get access to past resources again after reconnection. This must be accounted for somehow, for example by buffering in the sender application, but would be connected with uncertainty of when to buffer and maintaining the accurate buffer size. Although we did not choose the publish-subscribe model, our architecture will also have to deal with it because of the middleware chosen, as seen in the next section.

Since one of our functionality criteria is simplicity, we will avoid adding a profile as described in 2.3.3. But we will offer a basic setup file because different devices may have different editors and choose to install software in different places.

Figure 4-1 shows the architecture and division between the application and the middleware. All traffic between devices goes through the middleware's network interface. The middleware is responsible for device discovery, dynamic routing between peers and a group concept used for SVFs. The latter provide a framework for closed groups forming the shared virtual folders (SVFs). It is also the middleware that handles the transmission and routing of group and peer information (advertisements) that is stored in the local cache.

The application connects to the middleware through a middleware interface. The application contains some logic which acts upon the different user decisions provided via the GUI (see section 4.7). The application also interacts with the SVF repository and gives feedback to the GUI. The repository consists of an embedded database and a directory where all the files will be located (see subsection 4.6). In addition it contains the cache which holds information about groups and peers.

Altogether, the SVF repository will be responsible for:

- Keeping track of a device's current SVF memberships through communication with the middleware's group concept.
- Keeping track of all peers offering files to member SVFs.
- Administer local disk space where repository files are stored.
- Keeping track of which device holds the files not stored locally.
- Keeping track of the current session's notifications, i.e. what files are currently offered by the system.
- File versioning detection.
- Keeping track of groups and peers discovered in the network.

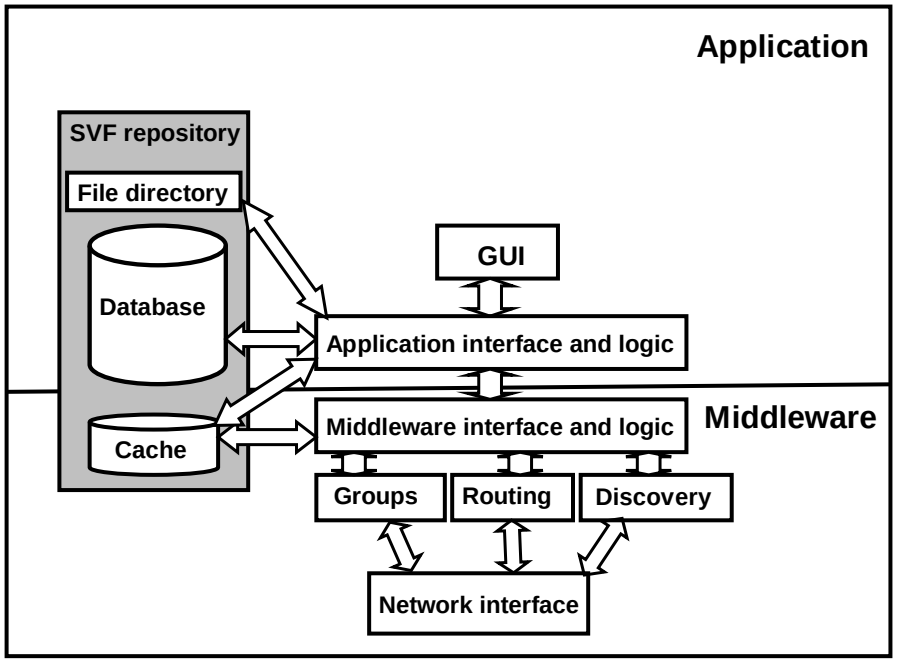

Figure 4-1. Architecture overview.

4.3 Choice of middleware

In 2.5.1 we explained briefly why middleware is used during software development. The most important concerns is that it allows for abstractions to be made and thus to save time during development, but also that it separates concerns well if the middleware is fitted for the purpose.

In 2.5 we also presented four different types of middleware implementations. In order to evaluate these, we have put up some desirable criteria based on our functionality criteria from 4.1:

- Highly dynamic device and service discovery. Devices and services could arise and vanish in a short notice. We assume that ad-hoc behaviour will be the norm for our devices.
- A group concept, preferably with a co-existing security implementation. In order to avoid unwanted files being distributed within the groups or requested files being removed, a closed group concept is considered of importance.
- Dynamic routing. This is a demand as we expect devices to move around in different networks a lot.
- A minimum of traffic between devices. As little traffic as possible is of interest to avoid network congestion, especially since the application also will issue messages itself.
- Platform ubiquity. The operating system or the device resources should not matter.
- Based on Internet protocols and well-functioning on both WANs as well as LANs. Preferably the pilot should function at any location and with and distance between the devices.
- Vendor supported. A middleware with broad vendor support it is more likely to be maintained and installed on more devices.
- The purpose of the middleware should fit with our SVF concept to the largest degree possible. While one usually has to test the middleware or at least study the protocols closely before knowing how well the middleware fits, middleware that appears to lack mechanisms needed with our concept should be excluded.

We have chosen JXTA (see subsection 2.5.6) as our middleware and "building block" for our SVF model. The JXTA platform's group concept is attractive to ensure file protection in the repository to outsiders, although JXTA does not offer security for multicast messages, only for unicast. The JXTA platform provides a highly dynamic discovery service through the use of rendezvous peers using time-to-live (TTL) counters. JXTA functions both on LAN as well as WANs due to the relay peers that do the routing and contains information about routes to other peers. These relay peers can forward messages on behalf of peers that cannot directly address another peers due to NAT,

firewalls or routers. This is performed by using a protocol that can traverse the firewall, like HTTP, for example (see subsection 5.3.6). Also, the JXTA is implemented in the Java language which provides implementations of a wide variety of platforms and OS. Thus there will be no need for the application to interact directly with different OS, as the Java Virtual Machine (JVM) will handle the integration.

As mentioned in section 4.2, JXTA supports the publish-subscribe model as devices has to subscribe to groups. This is not desirable since peers will loose messages while not logged on, but we will account for it by additional notification messages.

As for the other middleware platforms investigated, The Socialized.Net described in 2.5.7 has an advantage since the support and development environment is directly available. On the other hand, TSN lacks a group concept as it is more loosely based on the current interests of a user. Furthermore, a platform choice with broad support from international environments would be preferable in order to ensure maintenance. Broad support platforms software is also usually subjected to more thoroughly testing and has more resources for further development.

Bonjour described in subsection 2.5.4 is a popular middleware in particular for exchange of music files, but was not chosen due to its inability to function across firewalls and routers which excludes use in WAN settings (at the time implementation decision was made, a newer version is now available which promises to also handle WANs).

Universal Plug and Play (UPnP) described in subsection 2.5.5 is also popular standard with many implementations, but was not chosen because the goal and some of the vital procedures seemed not to support our project. According to the UPnP form the goal is “to allow devices to connect seamlessly and to simplify network implementation in the home and corporate environments” [69] which is a very overall goal, but may fit our SVF model. Going into details however, UPnP defines a control point (see section 2.5.5) to retrieve a device's description which is vendor-specific, manufacturer information like the model name and number, serial number, manufacturer name, URLs to vendor-specific web sites, etc. The description also includes a list of any embedded devices or services, as well as URLs for control, eventing, and presentation of the device. Issues connected with how to remotely control a device seemed without relevance to our project, although we do not know whether it could perhaps have been possible to adjust them also to fit

with the SVF. However, it was easier for the project to use JXTA as it seemed viable for our purposes.

We did not consider any other middleware but these four described in section 2.5.

4.4 Casual resource sharing

This section explains design issues and how messages are passed in our pilot application. Subsection 4.4.1 motivates our choice to base the application on notification messages and the downloading itself on a pull model. The subsections thereafter describe the different types of message passing in different situations which are of importance to the application. Subsection 4.4.2 describes the notification messages, while 4.4.3 describes how the application handles log on and log off of a device. 4.4.4 describes how files are downloaded.

4.4.1 Resource sharing issues

As devices come and leave ad-hoc, we need to make sure that the network itself is stable. As mentioned in subsection 2.5.3, it is of great importance that device activity does not block or take down other devices or even worse, the network itself. Thus also avoiding or handling network congestion well is of importance especially since both the middleware and the application will issue messages.

We have chosen to base our communication on multicast notifications (see subsection 2.3.1) for the SVF application itself. The multicast notifications are push messages issued to all members of a SVF. Notifications inform the others when a file is being added or removed from the repository, during file requests and when a new device joins a session. However, by use of JXTA the participants will only receive push messages for as long as they stay logged on to a SVF. As the JXTA group concept is based on subscription, it is necessary to account for the loss of messages issued when the devices are not logged on. The application will do this by issuing special notification messages as a peer log on to the network again, see subsection 4.4.3.

If a notification message is lost, we will not try to account for it, as that may lead to network congestion. The user may gain it later on if a new device logs on to the group, or

it may have to log off the group and log on again to trigger another issues of update notification messages. In most cases we foresee that messages will reach the receiver.

For the file downloading, we have chosen to use pull requests rather than push. The choice was made as it is difficult to foresee when users need a file.

We want to avoid loading down devices with little resources and to minimize the possibilities of network congestion (see subsection 2.3.2). We also foresee that not all users participating in an SVF will want to download all resources, thus downloading upon request seems most viable.

Implementing a swarming protocol as suggested in subsection 2.3.1 could have been beneficial for download in a setting where a larger number of devices transfer files casually. For example one could envision a tutorial where the lecturer would like to distribute presentation material. Often this type of material is large (typically Microsoft Powerpoint presentations with many images) and a swarming protocol could be employed to off-load the lecturer's PC so other devices reach the files at an earlier stage. A multicast would be even more efficient, but requires all participants to arrive on time. Moreover, the functionality criteria of including also low resource devices restrict use of large multicast messages. Thus, for our pilot implementation, we have not implemented any of this functionality.

Notifications combined with file transfer overall demands more bandwidth than just file transfer alone, but will ensure that the views each device has of the SVF repository will be kept reasonably updated. In cases when devices prefer not to download the files immediately it will be particularly useful.

4.4.2 Notification messages

When a new file is added, updated or deleted in the repository the other SVF member devices need to be informed. In order to update them, multicast notification messages are sent to other member devices.

The types of multicast notifications used are:

- *Update* : When a device has added a file to the repository or updated a file already

in the repository. The file can be added either by downloading from one of the other member devices or by copying the file into the SVF repository locally. Update messages are also used in response to “new” messages to inform other member devices for files available from the local repository.

- *Delete*: When a file has been removed from the repository.
- *New*: Similar to an update message, but is only issued when a new device logs on to a SVF.
- *Request:* If a device wants to download an entire file, not just the notification.

All notifications are issued as multicasts. Figure 4-2 shows an example where the smartphone has added or updated a file locally and issues an update multicast message to the others. Similarly, Figure 4-3 Remove notification using multicast. Here the smartphone has removed a file from the repository and sends the others an update.

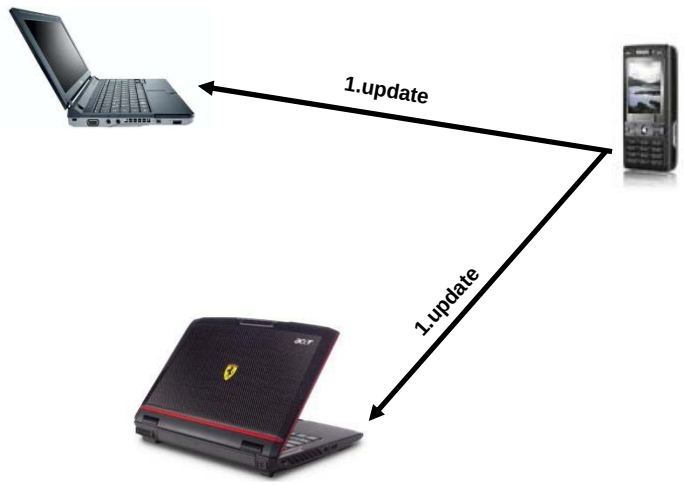

Figure 4-2. Update notification using multicast.

Should a device loose any messages they are simply lost, and if the notification is of great importance, the user can restart the SVF application and thus initiate an update over again.

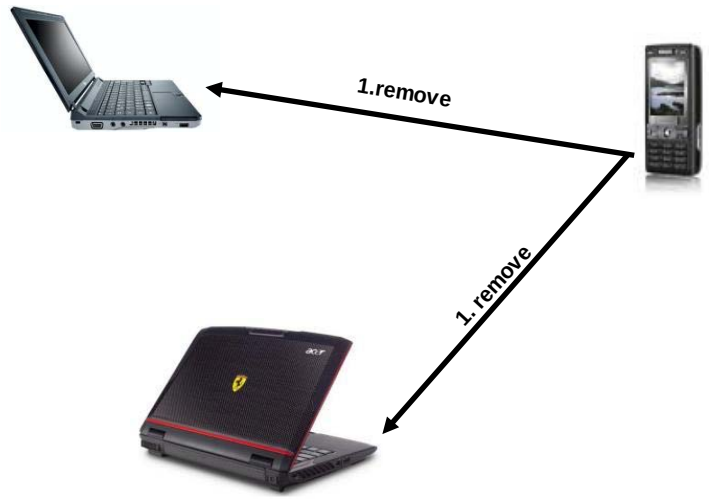

Figure 4-3 Remove notification using multicast.

4.4.3 SVF log on and log off

When a peer logs off a group, they will loose the messages issued during their absence as the group is based on member subscription. The application must account for this the next time a peer logs on to the group again. Thus, when a device logs on to the SVF application, the peers need to get the newcomer's latest changes to his local SVF, and the newcomer needs to know what the others currently offer in their local SVF repositories.

Thus the newcomer issues a "new" message including all files it currently has in its local repository. As the others receive the message, they will respond to the new message by issuing a multicast update message containing all files they currently have in their local repositories.

Figure 4-4 shows an example of a smartphone logging on to an already established SVF. As a newcomer, the smartphone issue a multicast "new" message to inform the others that it has now arrived and to notify about it's local files. As the smartphone's message are received by the others, they get all the current updates from the smartphone's local repository, and they respond by issuing an update message containing information on the current content of their local repository. The messages labelled "new" are printed in dashed lines, while the update messages are drawn in solid lines. Response messages to a

"new" message must be labelled differently as a "new" message automatically will trigger a response from the other member devices.

If any messages are lost, restarting the SVF application may help as it will repeat the process.

Since a device can leave the network any time, there will be no time for additional notifications before leaving or closing the application. Devices that request any of the files from the device that has left, will be unsuccessful in it's search if a file copy is not kept amongst the remaining peers.

At the application level there is no need to issue particular messages to establish groups or remove them as JXTA will take care of messaging at this level (see subsection 5.3.5).

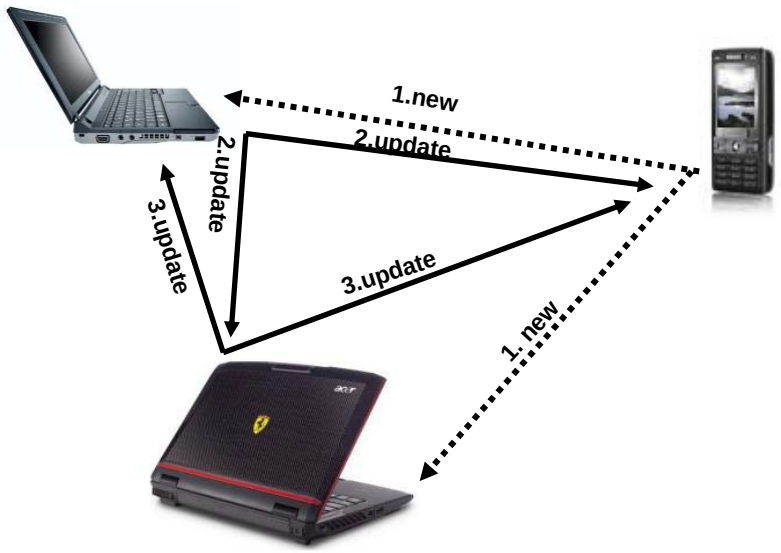

Figure 4-4 Multicast messaging as a device connects to a SVF.

4.4.4 File download

A file download starts with a multicast request to the others from the device that requests a file. If a device issues a request notification, the other devices will either do nothing or respond by starting to transfer the requested file. If the issuing device has received no response to the request, it may repeat the request later on. If nothing is received, it will assume that the file at least currently is not available within the SVF.

Figure 4-5 shows an example of request for a file download. The smartphone requests a file from the others in the SVF by issuing a multicast request for a file, shown in dashed lines. The lowermost PC has the file and thus responds to the request. If the request message is lost, the request also has to be repeated.

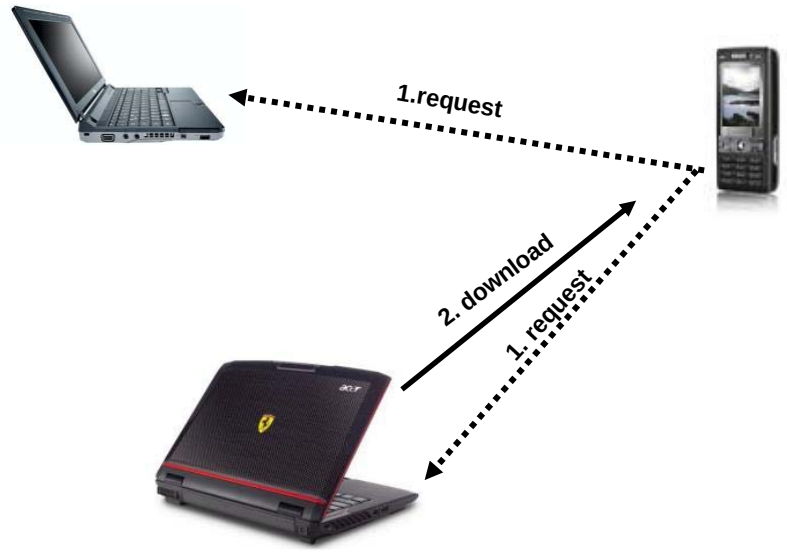

Figure 4-5. Request for file download.

4.5 Versioning detection

Versioning detection in the SVF is foremost to detect if several persons simultaneously has been updating a file. Thus, the distributed software versioning detections outlined in 2.4.3 is fitting for software development, but too elaborate for this pilot. Instead we have chosen a very simple scheme for document versioning.

File versioning for an SVF depends on whether a file has been distributed to others or not. If the last version of the file has not been downloaded by anybody, the file will not receive a new version number if updated. As long as the file version number has not changed, a notification message will not be issued either. Figure 4-6 shows the smartphone issuing a multicast notification message to the others after fetching a file into the local repository. This file will be marked with version number 1. Thereafter the smartphone writes to its own local file. Because the file itself has not been distributed, there is no need to update the version number, thus it is still version number 1. Since the others have not read nor written to the file, they will not know what version number 1 look like, and thus there is no need for version update nor notification issuing.

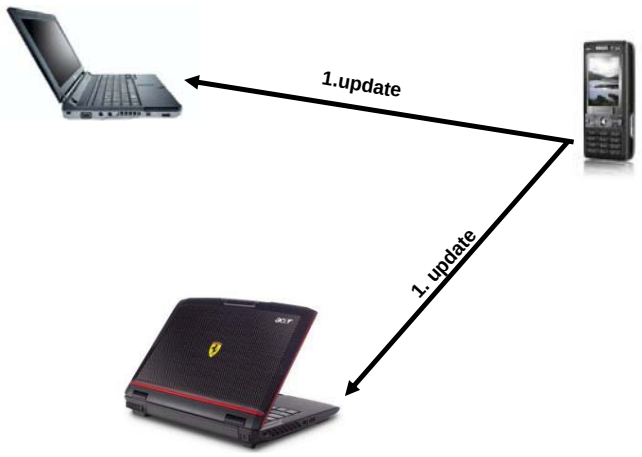

Figure 4-6. Version control if the previous version has not been distributed.

If the file has been downloaded by others, there will be file copies with the same filename, author and version. The messaging scheme will become more complicated as Figure 4-7 shows. Here the smartphone has issued a notification on a file that it has added to the local repository, similar to the previous example. This file will also be marked with version number 1. However, the PC to the left now requests the file by issuing a multicast request, shown in dashed lines (2). The smartphone responds by sending the file, as shown with a longer, dashed line (3). The file transfer itself is carried out as a unicast. As a response to the received copy, the leftmost PC issues an update (4) to inform the others. Finally when the smartphone updates the file thereafter, the file must now be given a new version, number 2. When the file is given a new version

number, this must be announced by two multicasts; one to remove the old file version followed by an update multicast to inform the others of the new file version (5).

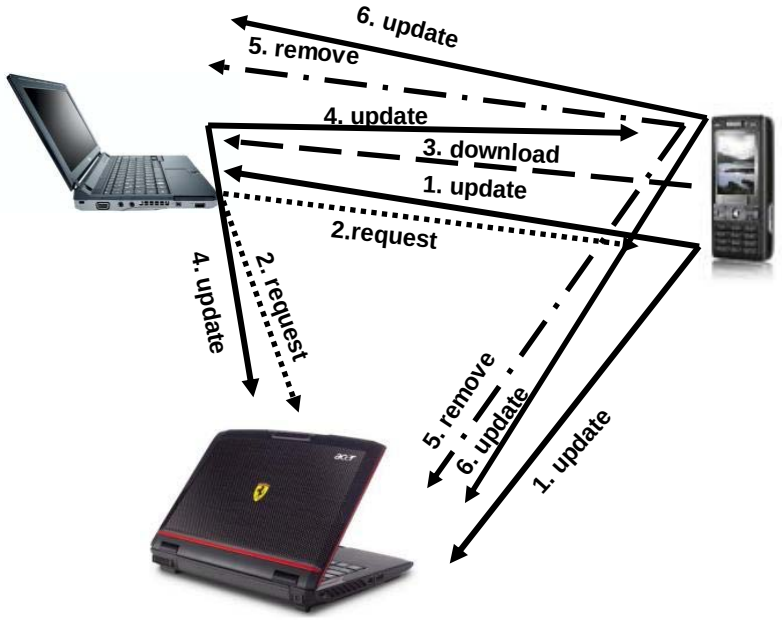

Figure 4-7. Version control if the previous version has been distributed.

In Figure 4-7, if also the PC to the left updates the same file, there will be two different versions of the same file named with version 2. The group members could incorrectly perceive it as one file being a copy of the other. In order to know the difference between a file copy and two versions of the same file, the files that are copies will have the same authors. Thus there will be a difference between a file that is a copy and a file that has two separate versions.

As mentioned previously in subsection 3.4, a file cannot have the same filename, version and author if they are located on the same device and belongs to the same group. Thus downloading and adding a file is not allowed if the device already contains a file with the same combination of filename, author and version. Similarly, during version update a conflict will be detected and give the user the option of renaming or deleting the last file.

If a file is saved under another name in the repository, the file will not be a part of the local repository at all unless it is fetched into the repository again by adding the file to the repository.

4.6 The repository

In order to store information about files and updates, a repository is needed. This section describes how the repository would be implemented. Without modification, the JXTA middleware will save all announcements both for groups and other services in the caches of each peer (see subsection 5.3.4). For our application it is not desirable to use the cache alone as storage, as the cache will eventually replace old content with newer once it has been filled up. Thus groups seldom gathering may risk that information about their SVF has disappeared from the cache. However we will employ the cache for discovery of new groups and peers in the network.

An important design choice is how information would be stored locally besides caching. The two most obvious choices are either using a directory structure or a database. We have chosen a database structure for our application.

A file system only allows folder names and file names to be present. In a directory structure, typically naming conventions (like the use of the filename extension to indicate a content type) or location conventions are used for mapping data. In a database structure, mapping to available types can be carried out through adding fields and tables to the database. It is also possible to add more meta-data than for a directory structure. Moreover a database can often organize and structure the content better, thus simplifying search and allow for more metadata to be gathered. Using a database will also simplify implementation of versioning detection.

A database usually supports access and updates by well-defined APIs, like the Structured Query Language, SQL. One of the benefits of using a fairly abstract query language is to make the database engine carry out basic tasks like adding temporary variables, loop structures etc and thus simplify application programming. Also, the database management system (DBMS) offers additional attractive functionality, like indexing in order to improve performance.

Another DBMS advantage is the ability to carry out transactions. For simplicity, features

involving transactions have not been implemented in the pilot. All SVF changes could be envisioned as transactions. If demanded, modifications associated with a partial transaction could be undone without compromising the integrity of the SVF. For example, one could demand notifications to be visible either for all device views of the SVF repository or none.

For storage of the files themselves, we have chosen to create a local directory (see Figure 4-1) where all files are stored identified through unique identifiers. The files are linked to their SVFs and real filenames through entries in the database. Alternative options to the approach could have been to use binary large object (blobs) or character large objects (clobs) in relational databases, alternatively to use object-oriented, object-relational or multimedia databases. We have not explored these alternatives and will thus not employ them in the pilot project.

While separate file storage will likely take up more space if the users want to store the same files also outside of the repository, it has the advantage that the users will easily recognize files belonging to the SVF repository in contrast to other files. When the users update a file, the file will be fetched from the directory and saved under the same name, which will indicate a new version of a file. If the file is saved by the user under a name not recognized in the database, the file will not be included in the SVF repository, although it resides in the same directory as the local SVF repository files. However, the user may add the file to the SVF if desirable.

4.7 The graphical user interface (GUIs)

Today many applications use web browsers as GUIs rather than making native ones. Web browsers are available on all platforms, and have a relatively well-defined API which makes this type of GUI attractive. Moreover, through Internet browsing the users have familiarized themselves with the GUI.

But the browser client is also strongly connected with the client-server concept where a server's address (URL) must be given in order to locate the content. The well-defined client-server model is contradictory to our vision of the SVF as a resource pool. In our model the URL would change often depending on accessible devices. Re-routing could perhaps have been possible, but would increase the time frame for making the pilot. Moreover, for the peers to be located through browsers, we would need to install

additional software on each peer anyway which would minimize the benefit of using a browser client as opposed to employing a native GUI. Another approach which maybe could have worked was to construct a dynamic webpage stored locally which were refreshed from time to time as messages arrived. While this may have been possible to carry out we have not chosen this solution.

Since a decentralized solution would anyway require local software installation, we have chosen a native GUI. Loosing the user's familiarity with the browser, we on the other hand have the benefit of stripping the application GUI for any additional functionality. The users will be limited in application configuration, since most functionality will be hidden by decisions taken during the design process. Experienced users may dislike the lack of configuration, but for inexperienced users the threshold will be lowered.

For design development and to further clarify application functionality, we outline some GUI functionality here. One rule of thumb towards simplicity is to avoid several windows that give the user alternative choices in ways to proceed. If only one window is opened, the users are limited to work with that window [70]. Furthermore, the number of buttons and choices in the window should be kept to a minimum.

The GUI should offer the following choices:

- Fetching a file into the SVF repository.
- Deleting a file from the SVF repository.
- Open a file by connecting it to another application.
- Fetching a remote file.
- Create and join a SVF.
- Withdraw a SVF membership.

5 Implementation

This chapter presents the implementation choices made as well as how the programming was carried out. Section 5.1 describes the implementation environment, while 5.2 explains the choices of software with respect to programming language and database choice. Section 5.3 goes into detail about the implementation itself.

5.1 Implementation environment

In order to make the application versatile, we preferred the application to run in as many environments as possible. However, due to time limitations, the application has been tested on Microsoft XP (ver 5.1 with Service Pack 2). The version of Java used has been Java 2 Platform (J2SE), version 1.4.2 with JXTA library version 2.4. The database chosen was One$DB 4.1 Beta embedded version (see 5.2.2 for database choice).

The application was developed and tested on a portable Dell Latitude D620, with CPU Intel Core Duo, 2048 RAM and a harddisk of 100 GB.

5.2 Software choice

This section elaborates on the choice of software used. In 5.2.1 we explain why we choose Java as the programming language and in 5.2.2 we explain our choice of database.

5.2.1 Programming language

The choice of a programming language was more or less bound to be Sun’s Java since a functionality criteria was to be platform agnostic. Java is available on both open source Unix-like platforms and vendor specific platforms like operating systems from Microsoft and Macintosh. Java is also available for a large number of hand-held devices which is attractive. C, Perl, Python, C# could have been reasonable alternatives, but does as easily offer the same range of platform availability.

5.2.2 Database

There exist a number of commercial databases specifically targeted as "embedded". "Embedded" refers to a database not running as a separate process, but instead being

directly linked into the application requiring access to the stored data [71]. This is in contrast to more conventional database management systems (for example MS SQL server, Oracle, or postgreSQL), which run as a separate process, and where the application connects using some form of inter process communication (for example TCP/IP sockets).

The main advantage of embedded database systems lies in their application availability and ease of administration [71]. As the data is kept in ordinary files in the user's space, there is no need to obtain special permissions to connect to the database process or to obtain a database account. Furthermore, since embedded databases require only an available library, they can be useful in constrained environments where resource devices are limited. Embedded database can also be linked to an application and shipped along with the rest of the software.

We chose our database due to the following criteria:

- It should be an embedded database as we want a tight integration with the application. Moreover, only one application instance should use the database.
- The database should preferably be implemented in Java to avoid additional language installations.
- Available on as many operating systems platforms as possible. We would like our application to function in as many different settings as possible.
- A well-defined API, for example a version of SQL. A well-defined API makes it easier to integrate the database with the application.
- Easy to install and well documented. This is necessary to avoid spending additional time in finding out how to set up and use the database.
- Available without costs as there is no implementation budget.

Also, some other criteria were considered, but not equally emphasised as our implementation is a demonstrator:

- Software maintenance available. This is useful if the application should be released and distributed to a larger audience.
- Licence allowing re-distribution. Correct licensing is also important during a software release.
- Preferably open source, so that optimizations/changes could be carried out if needed.
- Small footprint, considering our functionality criteria of availability on a large number of devices also including those with limited resources.

Due to financial limitation of the pilot project only freely available databases could be considered. Of these, One$DB was chosen (see appendix A for a list of possible databases). The choice of using One$DB did not come as a result of thorough evaluation of all possibilities, but rather as an acceptance of the first database found that actually suited our needs.

One$DB fulfils most of the criteria except it had a larger footprint than some other databases. For example Hypersonic SQL had a very small footprint of less than 100 Kb. On the other hand, One$DB comes with a number of features of interest like sequences and triggers. One$DB is written in the Java language which is desirable since it will not require support for additional languages. Otherwise, one of the most appreciated features about One$DB is the well-written and up-to-date documentation which was lacking for a many of the other available databases. One$DB also comes with LGPL (Lesser General Public License) which could be a reasonable starting point for further development later on and is thus also open source.

5.3 Application implementation

An overview of the application classes are shown in Figure 5-1, which may be compared with the architecture overview in Figure 4-1. The most important class is the `LookupManager.java` responsible for the interfaces and logic, together with the `SVFGUI.java` implementing the GUI.

We have chosen to start with the GUI description as it gives an overview of the application features. Thus `SVFGUI.java` is described in 5.3.1 together with the java class responsible for message feedbacks, `messageGUI.java`. In order to create a loose coupling between the main logic and the GUI, we use the interface class `LookupListener.java` and a class holding a structure for events, `LookupEvent.java`.

In 5.3.2 we describe the features of the repository and a description of the repository database itself. The repository is connected with the classes `LookupManager.java` and `SVFConnectToDatabase.java`. The first class is responsible for the interaction between the different modules and the SQL calls to the database. `SVFConnectToDatabase.java` handles routines for the SQL statements as well as implementing additional logic to initiate the database and open and close database connections.

The implementation logic and connection to JXTA in `LookupManager.java` is explained throughout the rest of the section. As an aid to get an overview, for each subsection we have added the methods from LookupManager.java connected with the text.

In order to utilize the JXTA platform, a configuration must be set up as described in 5.3.3. Central to the implementation is the JXTA discovery and rendezvous service described in 5.3.4, which provides connection to other peers as well as routing. The group concept is elaborated in 5.3.5, where we have implemented additional security to demonstrate how a closed group concept could function. Subsection 5.3.6 describes communication using ports, pipes and queues. The messaging itself is described in 5.3.7 while versioning detection is handled in 5.3.8.

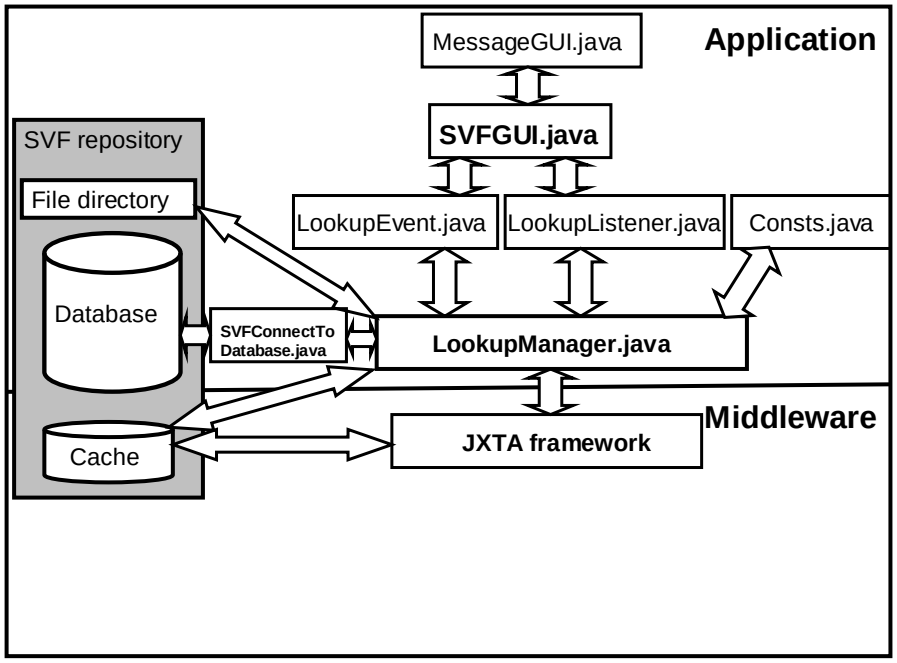

Figure 5-1 Overview of the software modules.

The SVF is a fully decentralized solution and where all peers have the same software installed. Thus the architecture of Figure 5-1 is implemented by all peers connecting to the SVF network.

In this section the term “this device” refers to a random device interacting with other devices through the SVF application.

5.3.1 The graphical user interface (GUI)

As mentioned in our functionality criteria in 4.1, the user interface design should be focused on simplicity. Thus all requested functionality should be available but nothing else, to avoid confusing the users by giving them many alternatives to choose from [70]. The panel in Figure 5-2 thus consist of the following elements:

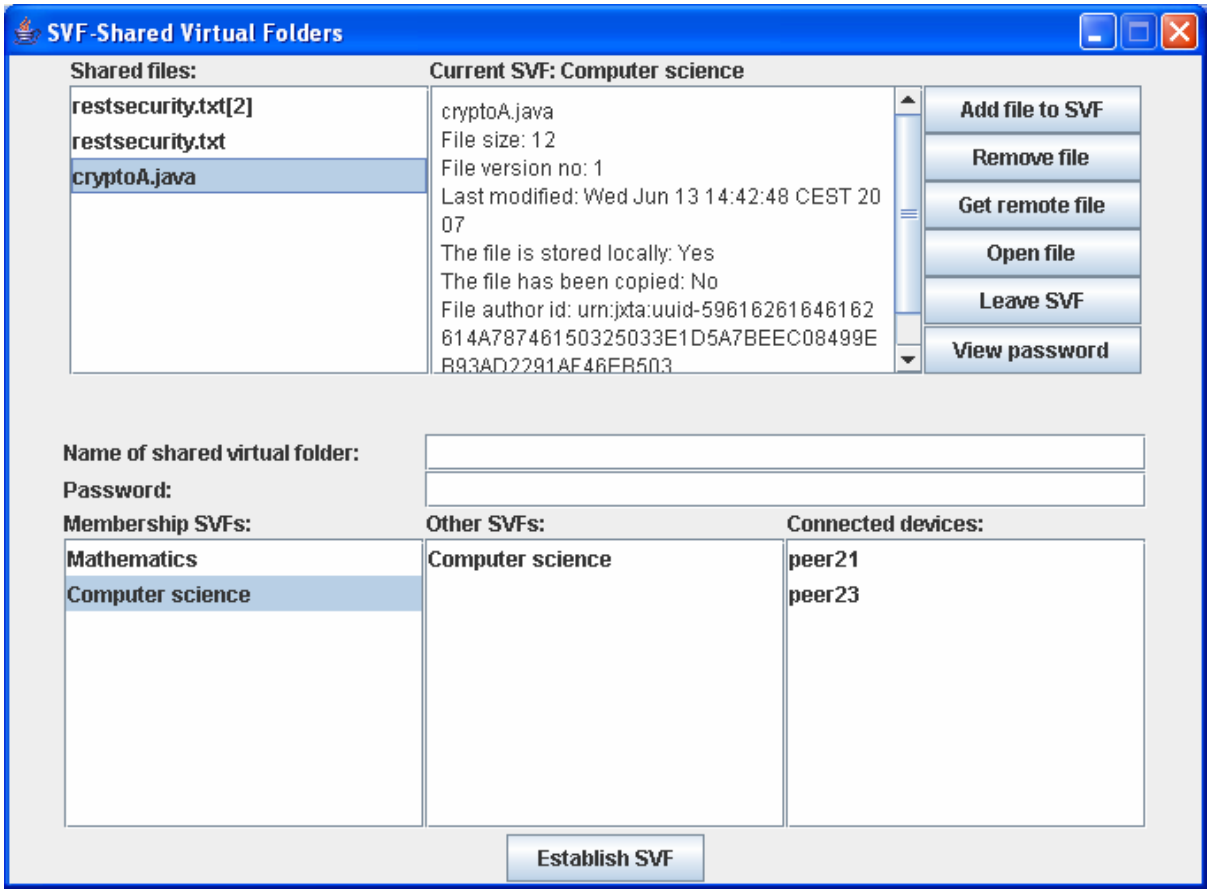

Figure 5-2 The SVF graphical user interface.

- A heading that displays the name of the currently active folder or alternatively "No Active Shared Virtual Folder" if no folder has been selected.
- A list-box marked "Shared files:" where one file can be selected at the time. If a file is a later version than number one, the version of the file will also be displayed.
- Between the "Shared files:" list-box and the buttons there is an area displaying

information on the file currently selected. Here the file name, size, version and author ID are displayed as well as last modification date, information on whether the file is remotely or locally available, and whether the file has been copied to other devices. Additional information could also have been displayed, but we consider this information to be sufficient for demonstration purposes.

- An “Add file to SVF”-button is used to copy a file into the repository.
- The “Remove file”-button is used to remove a file from the repository and copy it to outside the repository.
- The “Get remote file”-button is used to request a remote file from the repository for download from another SVF member.
- The “Open file”-button is used to open the file for modification. Today only an editor is connected for demonstration purposes.
- The “Leave SVF”- button is used to withdraw a SVF membership.
- The “View password”-button is a somewhat unusual security handling of the group login password. Each SVF folder has its own password, to ensure the group is closed to outsiders. Since a peer could be member of many different groups, there will be a lot of passwords to handle for each peer. Thus it would be unreasonable to demand the users to remember all passwords. At the same time, the users will need to know the group password in order to invite new members into the group. Section 6.4.2 describes other alternatives to this approach, but for the pilot we have chosen to allow a user currently logged in to a group to view the password.
- A text-field marked “Name of shared virtual folder” where the name of the new folder could be written or copied from the list-boxes below.
- A text-field marked “Password” where the password of the group is demanded.
- For groups that one is already a member of (displayed in the lower, left-most list-box) it is not necessary to apply a password for accessing them. This is done because there would be too many passwords to handle otherwise.
- A list-box marked “Membership SVFs:” where all SVFs the device currently is a member of are shown. By selecting one of the SVFs, it will be copied to the above text-field.
- A list-box marked “Other SVFs:” where all SVFs currently available from the device’s cache are shown. The groups shown here are not filtered in any way by the application in the demonstration version, thus new groups, membership groups and other JXTA groups are shown.
- A list-box marked “Connected devices:” where devices currently using JXTA are

shown. This box does not have any purpose except that users may wish to check whether collaborating devices are currently logged on the same network and thus could connect to the same group.

- The button marked “Establish SVF” at the bottom of the application can be used to establish a new SVF or to log on to an already existing group.

The graphical user interface is programmed in the java class `SVFGUI.java` (see Figure 2-1). The most important method is `layoutComponents()` which does all component layering and display. Most of the buttons, text-fields and list-boxes initiated here have mechanisms for handling user interaction attached to them as `SVFGUI.java` implements a listener interface for receiving action events:

```
btnFetch.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            btnFetch_actionPerformed();}});
```

Here `SVFGUI.java` and the object created with the class are registered with the button `btnFetch` component, which is using the button’s `addActionListener` method. When a user presses the button, an action event occurs that invokes the `btnFetch_actionPerformed()` method. Calls to other methods connected to components are carried out similarly.

As the users operate the GUI, they will expect a fast application response to interaction. In order to ensure this, we will need to provide the GUI with a number of threads that can carry out tasks simultaneously:

```
SwingUtilities.invokeLater(
     new Runnable() {
         public void run() {
```

The GUI components are created from the Swing package in Java. We use the `SwingUtilities.invokeLater()`to update the Swing components from a different thread than the thread that dispatched the event. For instance, when an item from a selected list are populated with data, there may be a perceptible delay from the time a button is activated and until the list is updated. If user interaction would be implemented

within one of the `actionPerformed` methods, the button would remain painted in its pressed state until the call to the `actionPerformed` method had returned. It would take a while for the button to return, and thus lengthy operations could not be performed in event handler methods as other events would not be dispatched until the event handler method had returned.

A main design issue has been to create a loose coupling between the GUI and the application logic. Changes added to the `SVFGUI.java` should not lead to greater changes to the application logic code in `LookupManager.java` and vice versa in order to save time during programming and to separate different concerns. Moreover, as the GUI require a number of threads, it is also desirable to co-ordinate these.

In order to separate the GUI from the implementation logic, and to control interaction between the many threads, the GUI is created over an asynchronous model as outlined by Simon [72]. An overview is shown in Figure 5-3, enlarging four classes from Figure 5-1 with snippets of code. `LookupEvent.java` holds a structure of events. The data is a string array holding the results (`result`) and the name of the receiver (`fromWho).` The `LookupEvent` is immutable, so it should not be changed regardless of which method that is processing the events. Thus many methods may utilize the same class.

The interface `LookupListener.java` is offered to classes that would like to receive the structure of events. The `SVFGUI.java` class has implemented the `LookupListener` interface, and thus receives information on events put on the array. The events are handled in the method `lookupCompleted().`

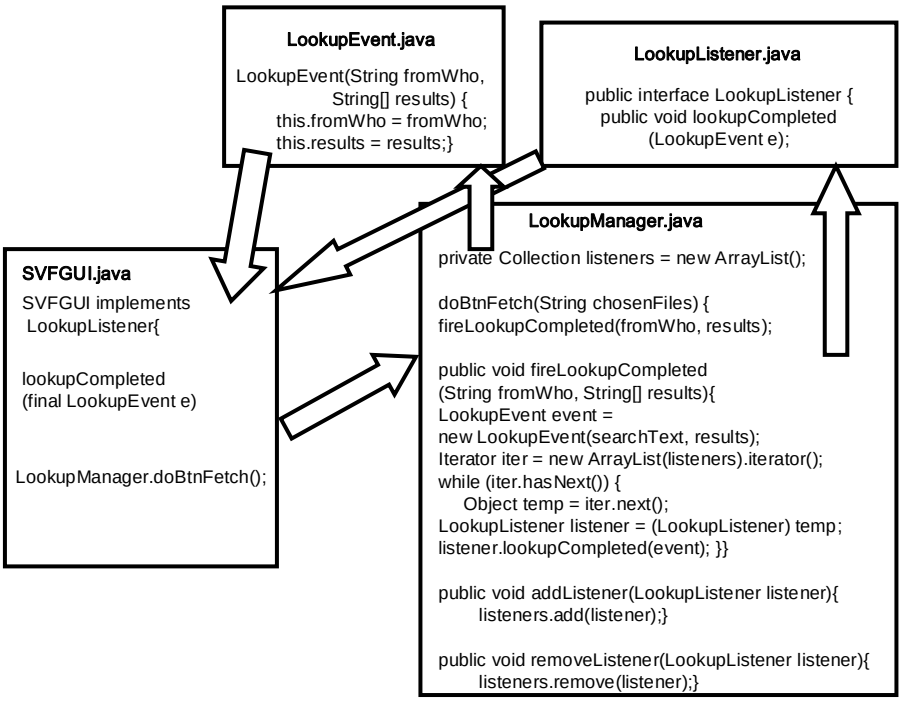

Figure 5-3. Separations of concerns between the GUI and the application logic.

At the other end, `LookupManager.java` has added a collection of `LookupListeners` in the declaration list so more than one event can be passed at the same time through the event array. Also, two methods named `addListener()` and `removeListener()`in `LookupManager.java` add and remove elements in the `listeners` array.

`SVFGUI.java` calls the application logic directly (shown with `doBtnFetch()` as an example in Figure 5-3). In `LookupManager.java` the methods carrying out the work never returns any results back to `SVFGUI.java` directly. Instead they call a method `fireLookupCompleted()`to put the results on to the array of events. The `fireLookupCompleted()` constructs an event, iterate through the array of listeners and calls the appropriate methods of the listeners. The listener registers when elements are put on the array to be processed. Once the listener has been set up, the peer can continue with other tasks because the listener will be triggered asynchronously when an element is received. In this way the GUI and the implementation logic can be more loosely coupled and more easily allow for code removal and additions.

Error messages and messages of information are written to the display by the class `messageGUI.java` called from `SVFGUI.java` (see Figure 5-1).

5.3.2 The file repository

Since the SVF application is decentralized, all peers must run a copy of a database repository and keep a directory available for file storage. The database takes care of all information related to peers, groups and files in the repository. The peers are identified in the database through their IDs, whose are generated by JXTA and assumed to be globally unique. This also holds for the ID of each of the groups (SVFs). JXTA builds an ID from a Universal Unique Identifier (UUID), which is a 128-bit hexadecimal number that functions as a unique identifier for each object [73]. The UUID itself is generated according to the "ISO/IEC 11578:1996" specification [74]. The last two hex characters of the ID define the type of ID; for example whether it is a peer ID or a group ID.

All local files will be stored in a directory whose location is specified in the class `Consts.java` (see Figure 5-1). The file contains all parameters that is needed to be set before startup. The files are not stored under their original names but are renamed after their identifier (FID) from the local database, a number given in ascending order (using sequences in SQL). Thus the FID is a local id, and not an id uniquely defined across the whole SVF network.

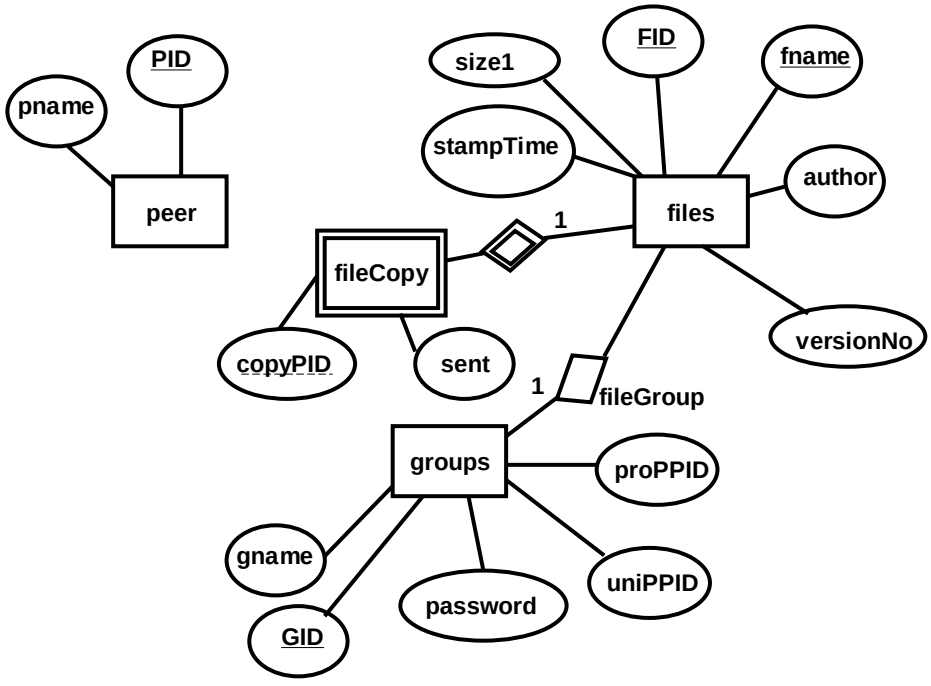

Figure 5-4. ER schema diagram for the SVF database.

The files are added or removed from the repository upon the user's request. It is possible and necessary to allow for files to have the same original name within a SVF as there could be different versions of the same file, but it requires that the files are kept in different locations with different devices.

groups:	The groups that this device is a member of.
GID:	The unique group id given by JXTA.
gname:	The name of the group used by JXTA which is not unique.
password:	The password of this group.
uniPPID*:*	The unicast pipe ID belonging to this group.
proPPID:	The propagate pipe ID belonging to this group.

files:	All files within groups that this device is a member of.
FID:	A local unique file id given by a sequential number generated by the database.
fname:	The name of the file which may not be unique.
stampTime:	The timestamp of the file.
author:	The author (creator) of the file.
versionNo:	The file version.
size1:	The file size.

fileGroup:	The connection between files and groups.

fileCopy:	Which file belongs to which peer in the repository.
copyPID:	The ID of the peer that issued the file notification, aka who holds a file or who has removed a file.
sent:	Whether the file has been distributed to other members or not. This is of importance in order to update the file version.

After normalization of the tables created from the ER schema diagram, we get the tables shown in Table 5-1. The first column represents the field name, the second the data type.

peer

PID	String
pname	String

groups

GID	String
gname	String
password	String
uniPPID	String
proPPID	String

files

FID	String
fname	String
author	String
stampTime	String
versionNo	String
size1	String

fileGroup

GID	String
FID	String

fileCopy

FID	String
fname	String
sent	Boolean
copyPID	String

Table 5-1. The normalized database tables.

The table `peer` is used only to register the peer name and JXTA ID of this device. It is necessary especially during initialization of the JXTA platform, but is also used by the many threads of `LookupManager.java`.

The table `groups` identify all groups that this device is currently a member of and keep track of their group ids and passwords. A new row will be inserted as this device joins a new SVF and a row will be deleted when this device leaves the SVF. The `groups` table is often searched in order to find a file or notification within a group.

The `files` table keeps track of all files stored in the SVF repositories this device is a member of. It also holds information about each file. The table is frequently updated as notifications are received from other peers or files are added or removed locally.

The `fileGroup` table connects each file with a group. A copy of a file can be added to several groups, but will be given a unique `FID` each time it happens. Thus the `FID` identifies exactly one copy of a file. Also, each new copy of the file will be put out into the file directory on disk. This is necessary as a file can be overwritten independently within each group. Thus two copies of the same file can become two different files if one of them is updated.

The `fileCopy` table contains information about files that are currently available in the repositories. This table is updated every time a file is copied, added or removed from the SVF. `sent` are there for versioning reasons; `sent` is set to true if the file has been copied between devices. `CopyPID` holds the peer ID of the device which sent the file update notification, or alternatively the peer ID of this device for locally stored files.

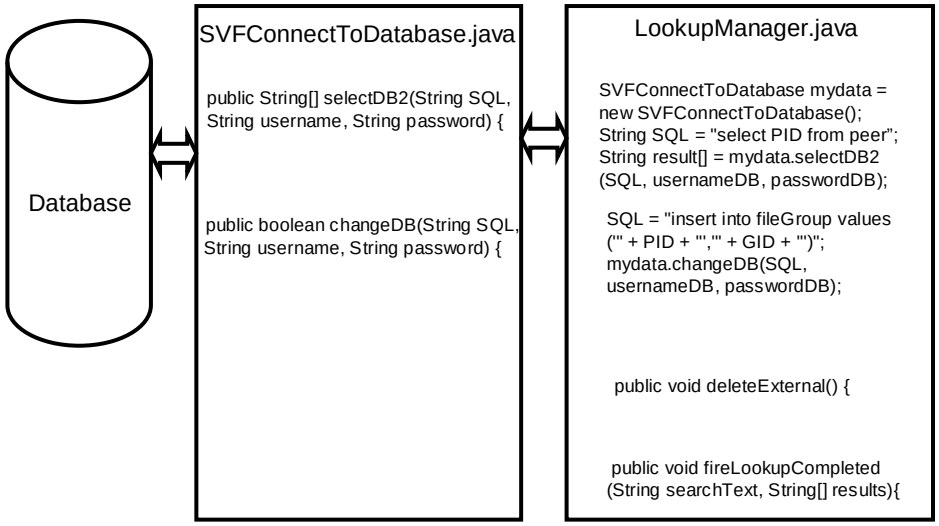

Figure 5-5. Separations of concerns during database connection.

The basic interaction with the embedded database are carried out by the class `SVFConnectToDatabase` (see Figure 5-1 for overview and Figure 5-5 for details) which

has methods `changeDB()` and `selectDB2()` which handles SQL insert, delete and select commands issued to them. `changeDB()` is synchronized as we do not want several threads to delete or update the same row simultaneously which would lead to database error. The interaction with the database goes through the `LookupManager.java` class which uses `fireLookupCompleted()` to pass on database results as events (also see Figure 5-3).

During a session, rows can be added or deleted, but the major cleaning up of tables will be carried out every time a device starts the SVF application using method `deleteExternal()` in `LookupManager.java`. Rows from files not stored locally will be identified by their `copyPID` in table `files`, and deleted. The tables `files`, `fileGroup` and `fileCopy` are affected by these updates. We do the cleaning up during log on because the device can log off from the SVF in an uncontrollable manner (for example through lack of battery capacity).

5.3.3 JXTA platform configuration and application setup

Method	Description
`start()`	Initiates the JXTA platform, the discovery and rendezvous service. Also join the base JXTA peer group, netPeerGroup.

Table 5-2. Main methods in LookupManager.java for JXTA configuration.

In order to start the JXTA peer, a platform configuration must be set up. The setup is carried out in the method `start()` in `LookupManager.java`. The NetworkConfigurator provides a simple programmatic interface for JXTA configuration.

The NetworkConfigurator takes care of JXTA initialization such as generating a new peer id if not already set:

```
p = IDFactory.newPeerID(PeerGroupID.defaultNetPeerGroupID);
```

Moreover the configuration itself must have a home directory, as well as a device nickname, a user login (using the `setPrincipal()` method), a password and a

description of the peer which must be set. The peer itself should also be set to a startup mode, either as an edge node, a relay node, a rendezvous server or a proxy server as explained in subsection 2.5.6. Our peers start the application as edge nodes:

```
config.setMode(NetworkConfigurator.EDGE_NODE);
```

In order to be sure we get a stable rendezvous peer, it is possible to initially connect to an already localized peer. We use one of Sun's servers:

```
config.addRdvSeedingURI(new URI("http://rdv.jxtahosts.net/cgi-
bin/rendezvous.cgi?2"));
```

Saving the new setup for the NetworkConfigurator object completes the configurator initialization. Once set, JXTA will install a subdirectory under the defined JXTA home directory called `cm` where the configuration is stored. Deleting the subdirectories makes it possible to reconfigure the JXTA peer.

In the pilot application, the JXTA home path can be set in the `consts` file (described in subsection 5.3.2, see also Figure 5-1). The `Consts.java` file also contains the home path of the embedded database as well as the directory where the local repository files are kept.

5.3.4 JXTA advertisement, discovery, service and rendezvous

Method	Description
`start()`	Initiates the JXTA platform, the discovery and rendezvous service. Also join the base JXTA peer group, netPeerGroup.
`startGroupSearching()`	Searches for peer group advertisements.
`startPeerSearching()`	Searches for peer advertisements.
`searchForSubPeerGroup()`	Searches for a specific peer group advertisement in the local cache (for group establishment).
`getPeerCache()`	Searches for peers in the local cache and displays the result in the GUI.
`getGroupCache()`	Searches for groups in the local cache and displays the result in the GUI.
`discoveryEvent()`	Handles incoming discovery responses from other peers.
`waitForRendezvousConnection()`	Blocks if not connected to a rendezvous, or until a connection to rendezvous node occurs.
`rendezvousEvent()`	Receives an incoming rendezvous event
`stop()`	Removes a peer from subscribing to rendezvous events.

Table 5-3. Main methods in LookupManager.java connected with discovery and rendezvous.

All network resources in JXTA such as peers, peergroups, pipes, and services are represented by *advertisements* [73, 75]. Advertisements are meta-data structures represented as XML documents. The JXTA protocols use advertisements to describe and publish the existing network resources. Peers discover resources by searching for their corresponding advertisements, and may cache any discovered advertisements locally.

Each advertisement is published with a lifetime that specifies for how long the advertisement will remain valid in the publisher's cache. It renders possible the deletion of obsolete resources without requiring any centralized control. An advertisement can be republished before the original advertisement expires to extend the lifetime of a resource.

In addition one can specify the expiration time of an advertisement which determines when the advertisements will expire in receiver's caches. If the lifetime is long enough, the advertisement will stay in the receiver's caches for the full expiration time. For our implementation, we use the default lifetime which is one year for locally created advertisements and an expiration time of two hours for remotely published advertisements.

A *service* is uniquely defined by its advertisement which provides all necessary information. Both the Peer Discovery Protocol and the Rendezvous Protocol described below are examples of services offered which may be implemented by a peer. It is also possible to define one's own services in a JXTA network and not just to use already defined services as shown in subsection 5.3.5.

To search for advertisements, we use the *Peer Discovery Protocol.* First a pointer to the Peer Discovery Service must be obtained which is carried out at the end of the `start()` method in `LookupManager.java`:

```
discovery = netPeerGroup.getDiscoveryService();
```

Next a Peer Discovery Service event listener is registered to process discovery responses after we sent discovery queries:

```
discovery.addDiscoveryListener(this);
```

The event listener functions similarly to the event listener for the GUI described in subsection 5.3.1. When the listener has been initialized, the peer can continue with other tasks because the listener will be triggered asynchronously when an event is received. For our implementation the group and the peer search are run as two different threads thus we have added listeners both to `startGroupSearching()` and `startPeerSearching()`. These methods search for groups and peers remote and fetches them to the local cache.

To search for an advertisement in the cache (if it is not in the database which we search first), we use either the `getRemoteAdvertisements()` or the `getLocalAdvertisements()` methods. These methods are used in `searchForSubPeerGroup(), getPeerCache()` and `getGroupCache()`.

In `searchForSubPeerGroup()` we are searching for a peergroup advertisement which contains a tag named `Name` with the variable `groupName` as value. We are requesting a single response match (threshold value of 1). The first input is null which indicate use of the rendezvous service for searching (see below). The last null value indicate the use of callback or not, which we do not use.

```
discovery.getRemoteAdvertisements(null, DiscoveryService.GROUP,
"Name", groupName, 1, null);
```

After sending the discovery request, we wait for response. Discovery responses are processed by the discovery event listener `discoveryEvent()` method. Here we collect the peer and the peergroup advertisements.

It is possible to limit the number of incoming advertisement as there can be many. In `getGroupCache()`the number of advertisements are limited to groups established by the SVF application only using the description tag in the group advertisement:

```
discovery.getLocalAdvertisements(discovery.GROUP, "Desc", "SVF*");
```

A peer uses the Peer Discovery Service to find advertisements in the JXTA network which again uses a Peer Resolver Service based the Peer Resolver Protocol for issuing queries and receiving back answers. The Peer Resolver Protocol wraps a query in its own message, and sends the new message to other peers. The Peer Resolver Protocol running on the remote peers will receive the wrapper message and then forward the underlying message to the appropriate handler. Hopefully there will be a response to the request. The remote peer will put the response into a Peer Discovery Service response message, and the Peer Discovery Service will pass the message to the Peer Resolver Service to deliver to the requesting peer. The Peer Resolver Service will then wrap the message into its own message again and forward it to the first Peer Resolver Service for unwrapping and forwarding.

The Peer Resolver Service may use either the Peer Endpoint Protocol or the *Rendezvous Protocol* or both for transportation. The Peer Endpoint Protocol is the protocol which is responsible for the routing within the network. It discovers a route (sequence of hops) to

send a message to another peer potentially traversing firewalls and NATs (see subsection 5.3.6). The Rendezvous Protocol is used for propagating a message within a peergroup.

In order to find other peers on the network and connect to them in groups, we use the Rendezvous Protocol. The Rendezvous Protocol is designed to propagate messages between peers within a group using a rendezvous peer. A rendezvous peer is a device that has the ability to propagate received messages to other rendezvous peers. Thus it has a list of several rendezvous peers it may forward to, instead of just one.

It is possible to designate a peer as a rendezvous peer at startup (for our implementation it must be set in `start()` in `LookupManager.java)`or to connect to other rendezvous peers that could be servers in the network (For example Sun has set up some servers, see subsection 5.3.3). It is also possible that a device will become a rendezvous peer by dynamically assignment.

If a peer is not a rendezvous peer, it may be an edge peer which we have designed our peers to become. An edge peer will only connect to one rendezvous. If that rendezvous fails, the peer will failover transparently to another available rendezvous. Once a peer is connected to a Rendezvous, the peer can start to search and create pipes as described in subsection 5.3.6.

For our peers, at `start()` we connect to the Rendezvous Service through a call to the base peer group `netPeerGroup` object:

```
rendezvous = netPeerGroup.getRendezVousService();
rendezvous.addListener(this);
```

The `netPeerGroup` object also gives access to other services within a peer group (discovery, pipe, etc). We get a pointer to the Rendezvous service and register an event listener for that service. Afterwards we wait for a Rendezvous event connect before proceeding further through the call as in method `waitForRendezvousConnection()`:

```
if (!rendezvous.isConnectedToRendezVous()) {
```

The `rendezvousEvent()` listener method is called whenever a new Rendezvous event occurs. Here as soon as we get a Rendezvous connect or reconnect event (in case the peer was already connected), we notify the main thread to proceed.

In order to remove the peer from the rendezvous listener service, in the `stop()`method we use:

```
rendezvous.removeListener(this);
```

5.3.5 JXTA secure group concept

Method	Description
`start()`	Initiates the JXTA platform, the discovery and rendezvous service. Also join the base JXTA peer group, netPeerGroup.
`doBtnEstbSVF()`	Responsible for creation of a peergroup and for a peer to join the group.
`searchForSubPeerGroupDB()`	Search the database for a peer group ID matching a peer group name.
`searchForSubPeerGroup()`	Searches for a particular peer group advertisement in the local cache (for group establishment)
`createPeerGroupAdvertisement()`	Creates a peer group advertisement.
`createPeerGroup()`	Responsible for creation of a secure peergroup
`createPasswdMembershipPeerGroupModuleImplAdv()`	Creates the module implementation advertisement connected with the group advertisement.
`createPasswdMembershipServiceModuleImplAdv()`	Creates the password membership service implementation advertisement to be put into the module implementation advertisement.
`joinPeerGroup()`	Responsible for logging on to a secure peer group.
`completeAuth()`	Authentication towards a secure peer group.
`createInputPipe()`	Responsible for pipe administration and setting up the propagate input pipe. Administers the message queue and coordinates incoming and outgoing messages.
`createGroup()`	Responsible for creation of a non-secure peer group with advertisement.
`joinSubPeerGroup()`	Joins a non-secure peer group.
`doBtnLeaveSVF()`	Logs off the current peer group and deletes all related peer group information from the database

Table 5-4. Main methods in LookupManager.java connected with the group concept.

A closed group concept is a part of the functionality criteria for SVF (see 4.1) to avoid others to remove files or to fill up the SVF with unwanted resources. While a fully secure group concept is outside of the scope of the thesis, we have implemented a secure peer group to get an impression of how JXTA handles security within groups.

As we start the SVF application, we must connect to the base JXTA peergroup called `netPeerGroup` (with the group name "World Peergroup") which all peers must be members of. We join this group in `start()` in `LookupManager.java` by executing:

```
NetPeerGroupFactory factory  = new NetPeerGroupFactory();
netPeerGroup = factory.getInterface();
```

Here we instantiate the `netPeerGroup`. We also add the `netPeerGroup` to the database because we need it for comparison with other groups later on. This base peergroup connection is important in order to be connected to the discovery, rendezvous and other services (as shown in subsection 5.3.4). As soon as we have connected to the `netPeerGroup`, users may access another group (a SVF) in three ways:

1) They may write a new group name in the "Name of shared virtual folder" text-field, add a password and create a new group which others may join through publishing of the group announcement.
2) They may select a group from the list box "Membership SVFs" whose ID are stored in the database together with the password. Thus choosing this alternative a user need not apply any password.
3) They may select a group from the list box "Other SVFs" whose advertisement resides in the cache but where password needs to be supplied in order to access the group the first time. Groups named in the "Other SVFs" may also reside in the database, in which case the password is obtained already.

All three alternatives are followed by pressing the "Establish SVF" button. The button calls the `doBtnEstbSVF()` method in `LookupManager.java`, which carries out the main work during establishment or re-establishment of a group. The password from the database are never shown in the text-field, but if ever needed there is the button called

“View password” that can be used if a user need to give away the password to another peer as he logs on.

All groups beside `netPeerGroup` must be joined in two operations. First the group advertisement must be found or created, thereafter the peer must authenticate itself against the membership service in order to join the group. For the first part, establishing the group advertisement could be done in three ways carried out in the following priority:

1) If the group ID is found in the database, a group advertisement could be recreated.
2) If the group advertisement is found in the cache, we have the advertisement and do not need to reconstruct it.
3) Construct a new group ID and group advertisement.

The `doBtnEstbSVF()` is the method in `LookupManager.java` which is responsible for group creation and joining. `doBtnEstbSVF()`first checks if the group ID can be found in the database by calling the method `searchForSubPeerGroupDB()`. If the group ID is found, it is fetched and used for re-creation of the advertisement. If there are no fitting entries in the database, we start searching the local cache for the group advertisement using the discovery service:

```
discovery.getLocalAdvertisements(DiscoveryService.GROUP, "Name",
groupName);
```

The local cache is searched by use of the groupname. Quite similar to the local advertisement we also search for the group remotely by using getRemoteAdvertisement as explained in subsection 5.3.4 above.

If the group ID or advertisement still cannot be found, we will have to generate a new ID as done in the method `createPeerGroupAdvertisement()`:

```
subPeerGroupAdvertisement.setPeerGroupID
            (IDFactory.newPeerGroupID());
```

Regardless of whether we have the peergroup advertisement itself, or only an ID we must call the method `createPeerGroup()` from `doBtnEstbSVF()` which establishes the peer group itself:

```
subPeerGroup=netPeerGroup.newGroup(subPeerGroupAdvertisement);
```

When creating a secure peer group, JXTA is neutral to cryptographic schemes or security algorithms [76]. It does not mandate any specific security solution. Instead JXTA provides a framework that allows different security solutions to be plugged in. For example, all messages have a designated credential field that can be used to store security-related information. JXTA does not specify how the information is interpreted as it is beyond the scope of the specification and is left to services and applications themselves to decide.

For our application we have decided to use a simple password encryption which has been cracked for more than 20 years ago [55]. Still it demonstrates the principles of a closed group concept which is our aim.

If we do not have the advertisement, we need to create it according to our security demands. The peer group that is being built does not have the same characteristics as a standard peer group has, since we demand password authentication before joining the group. The additional authentication must be conveyed to other peers so they know how to log on to the group. In order to convey this, we need to modify the module implementation advertisement (see Figure 5-6) that is issued together with the group advertisement. The module implementation advertisement needs to call the net.jxta.impl.membership.PasswdMembershipService (B in Figure 5-6) instead of net.jxta.impl.membership.NullMembershipService (A in Figure 5-6) which is used for peer groups that does not demand authentication.

A new membership service will be a part of a module implementation advertisement which contains many services and is published together with the peergroup advertisement in order to give potential group members access to it. (See Appendix B for a listing of a group advertisement example and connected module implementation advertisement.) We carry out the publishing in the `createPeerGroup()` method, where the fields `DEFAULT_LIFETIME` and `DEFAULT_EXPIRATION` functions as explained in subsection 5.3.4.

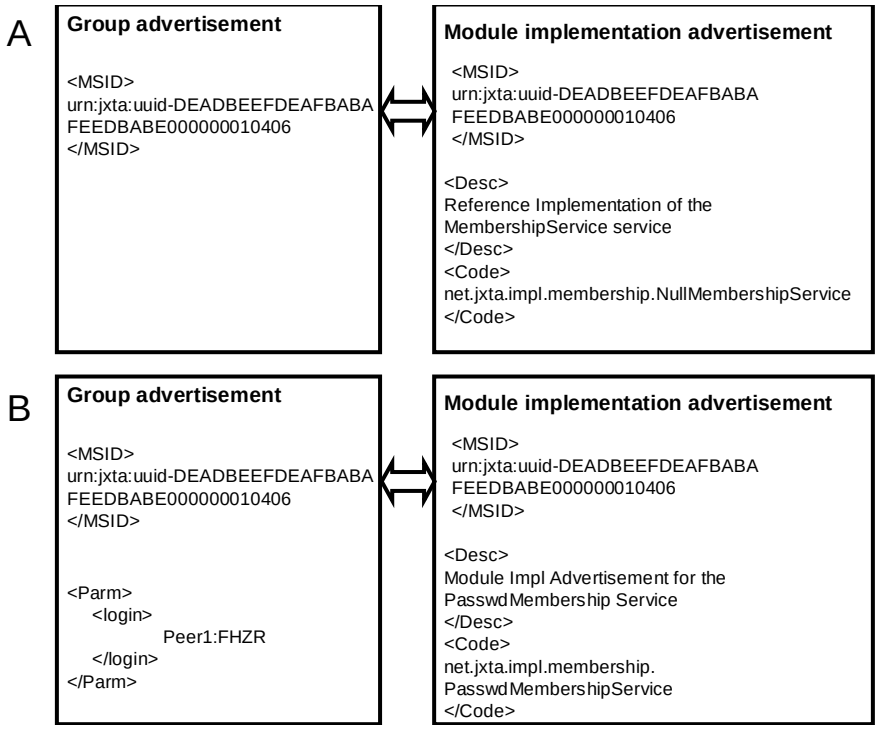

Figure 5-6. Module implementation advertisements with different levels of authentication.

`createPeerGroup()` calls the `createPeerGroupAdvertisement()`which is responsible for the actual creation of the new peergroup advertisement itself. The peergroup advertisement is constructed similarly to the above description; either by means of getting the group ID or a previously created peergroup advertisement as described above. In addition it is necessary to connect the peergroup advertisement with the new module implementation advertisement, done by adding a new Module Spec ID (MSID) which connects the two advertisements (see Figure 5-6):

```
subPeerGroupAdvertisement.setModuleSpecID(passwdMembershipModuleIm
plAdv.getModuleSpecID());
```

Moreover, the login name and the encrypted password must be included into the peergroup advertisement. Thus a tag called `<login>` is added to the XML document which is the peergroup advertisement (see the group advertisement in Figure 5-6, B):

```
subPeerGroupAdvertisement.putServiceParam(PeerGroup.membershipClas
sID,loginAndPasswd);
```

`createPeerGroup()` also calls the `createPasswdMembershipPeerGroupModule-ImplAdv()` which is responsible for the creation of the module implementation advertisement. As seen in Appendix B, the module implementation advertisement is quite extensive, thus instead of creating all the different service advertisements over again, we have chosen to use a standard module implementation advertisement and only modify the membership advertisement. The standard module implementation advertisement `allPurposePeerGroupImplAdv` is first fetched:

```
allPurposePeerGroupImplAdv=netPeerGroup.getAllPurposePeerGroupImpl
Advertisement();
```

The `allPurposePeerGroupImplAdv` has the structure of a hashtable from which all the different service advertisements can be extracted. In order to modify the module implementation advertisement, we must first get hold of the hashtable itself:

```
Hashtable allPurposePeerGroupServicesHashtable = new
Hashtable(passwdMembershipPeerGroupParamAdv.getServices());
```

As we wish to update the membership implementation advertisement only, we need to remove the all purpose membership service implementation advertisement and replace it by the password membership service implementation advertisement:

```
allPurposePeerGroupServicesHashtable.remove(allPurposePeerGroupSer
viceID);
allPurposePeerGroupServicesHashtable.put(PeerGroup.membershipClass
ID,passwdMembershipServiceModuleImplAdv);
```

Then we insert the new element into the hashtable:

```
passwdMembershipPeerGroupModuleImplAdv.setParam((Element)
passwdMembershipPeerGroupParamAdv.getDocument(new
MimeMediaType("text/xml")));
```

Finally we update the module implementation advertisement with the new MSID to connect it with the group advertisement:

```
passwdMembershipPeerGroupModuleImplAdv.setModuleSpecID(IDFactory.n
ewModuleSpecID(passwdMembershipPeerGroupModuleImplAdv.getModuleSpe
cID().getBaseClass()));
```

The `createPasswdMembershipServiceModuleImplAdv()` is called from the `createPasswdMembershipPeerGroupModuleImplAdv()` and creates the actual password membership service implementation advertisement which is put into the hashtable structure. The creation of the password membership service implementation advertisement is straight forward: the MSID is created and inserted, the implementation description is inserted and the `<Code>` element, which contains a reference to the package needed in order to load and execute the code of this implementation (see Figure 5-6 B). The other tags of the password membership service implementation are not changed and can thus be adopted from a generic template.

After the group is created, the `doBtnEstbSVF()` calls `joinPeerGroup()` which does the peer authentication towards the group. `joinPeerGroup()` gets hold of the membership service that we created which allows the peer to establish an identity within a peer group.

The peergroup membership service first establishes a default temporary identity for the peer within the peergroup. This identity only allows the peer to authenticate itself to establish the true identity. Thus, first the peer must apply for a temporary identity:

```
Authenticator auth = membershipService.apply( authCred );
```

The peer provides the membership service with an initial credential which may be used by the service to determine which form of authentication should be used to establish the peer's true identity. An authenticator object is returned to find the authentication form. The actual authentication completion is done in `completeAuth()` called by `joinPeerGroup()` where the authentication methods needs to be extracted from the authenticator object. The login and password will be tested if the `"setAuth1Identity"` and the `"setAuth2_Password"` are present:

```
if (doingMethod.getName().equals("setAuth1Identity")) {
doingMethod.invoke( auth, AuthId);
```

```
if (doingMethod.getName().equals("setAuth2_Password")) {
doingMethod.invoke( auth, AuthPasswd );
```

Afterwards, the completed authenticator object is returned to the membership service and the identity of the peer is finally tested against the new credential available from the authenticator. The identity of the peer remains as it was until the join operation completes:

```
membershipService.join(auth);
```

If accepted, the group is successfully joined by the peer.

The method `doBtnEstbSVF()`which is responsible for setting up and joining the group also takes down and cleans up after the previous group (if it exists) by setting a global flag called `pipeStopped` which takes down the pipes and resign from the membership service by the command:

```
membership.resign()
```

issued in method `createInputPipe()`. Here the existing peer identity that was established is discarded and the current identity is set to the “nobody” identity.

In order to demonstrate the group concept even without security, we have enclosed the methods for logging on to a group also without any access restrictions. The methods `createGroup()` and `joinSubPeerGroup()` carries out the group creation process and joining but without any demand for authentication.

When a SVF no longer is needed, it may be abandoned and deleted from the database. The `doBtnLeaveSVF()` handles this situation, when a user presses the “Leave SVF” button. In order to withdraw a SVF membership, the group must be logged on to before withdrawal, as the `doBtnLeaveSVF()` always deletes the current group. The `doBtnLeaveSVF()`deletes the group from the database, deletes all files connected with the group, and issues notification messages on the delete to the other devices currently logged on to the group. It will also take down any pipes that are currently up, but only after the last notification messages has been sent. Because it may take some time to issue

the final notification messages, there is a delay before the pipes are taken down which also the users will experience.

For a while the group advertisement may stay in the cache. Thus, in case of regrets, it should be possible to reconstruct the group even after deletion. After some time, it will also be deleted from the cache, and the ID of the group will be lost.

5.3.6 Communication: Ports, pipes and queues

Method	Description
`createPropagatePipeAdv()`	Create a propagate pipe advertisement
`createUniPipeAdv()`	Create a unicast pipe advertisement
`createInputPipe()`	Responsible for pipe administration and creating a propagate input pipe. Administers the message queue and coordinates incoming and outgoing messages.
`createUniInputPipe()`	Creates a input unicast pipe.
`createPropagateOutputPipe()`	Creates an propagate output pipe.
`sendUniPipe()`	Creates a unicast output pipe.
`pipeMsgEvent()`	Receives messages from incoming pipe events.
`handleMsg2()`	Opens incoming messages and initiate a message response if needed.
`sendOutputPipe()`	Creates output messages and sends them.

Table 5-5. Main methods in LookupManager.java connected with pipe use.

One of the primary reasons behind the JXTA is to facilitate the transfer of information between peers. The services are made known to the peers through advertisements, and messages and file downloads are usually transferred through *pipes*. The pipe concept is similar to that found in the Unix system, in which a pipe connects two commands.

In the application, all message communication is carried out by means of pipes. The pipes themselves are abstractions on top of the lower *Peer Endpoint Service* layer. The Peer Endpoint Service is the mechanism used for building communication channels between peers. The Peer Endpoint Protocol is responsible for determining a route

between peers in the JXTA network and is the protocol that actually carries out the message transmission. The service may use transport protocols TCP/IP, HTTP, Transport Layer Security (TLS), Beep and ServletHTTP and also be built around others. All transport protocols must implement a number some functionality to be used with JXTA. The two most important implementations are a method to send a message from one peer directly to another without any type of routing needed, and a method for issuing a message to all local peers reachable from the current peer.

The Peer Endpoint Service is responsible for only sending messages between peers. The peers can be directly connected, or relay peers can act as intermediates. When endpoint routing is needed, the routing peer will try to minimize the amount of work by use of a route cache. A route cache is used when the routing peer checks its internal cache to find the route. If a route is found, a route query message is sent to the JXTA network.

All messages from the endpoint router use both the discovery and router services. These services allow the query messages to be published and routed throughout the JXTA network. Finding the address is based on getting a JXTA ID and returning an endpoint address that can be used to communicate with the peer. First it is checked whether the peer is directly connected, secondly previously used routes in the cache are checked, and finally a query is issued if the route cannot be found. If a response is received for the query, this route will be added to the cache. If returning unsuccessful, an error message will be written back to the user.

During connection, the receiving peer must offer some endpoint to which the sender peer can connect to. The endpoints could be either an IP-address and a port number as specified in the TCP/IP protocol, but could also be a secure endpoint using the TLS or using an endpoint defined by a JXTA ID. The receiving code registers a listener which determine whether any messages that arrive at the endpoints should be processed. The listener is designed to monitor messages based on an endpoint service and parameter values.

When the listener is triggered, the listener method will receive both the source and destination addresses of the communication, as well as a message object. The string associated with the message can be extracted and processed (se subsection 5.3.7).

The *Pipe Binding Protocol* is built on top of the Peer Endpoint Protocol, and is responsible for allowing messages to be passed from peer to peer. The Pipe Binding Protocol may run on top of the Rendezvous Protocol (see subsection 5.3.4) or the Peer Endpoint Protocol. The Pipe Binding Protocol outlines three different types of pipes; a unicast pipe, a secure unicast pipe and a propagate pipe. The latter is used in one-to-many connections.

In order to connect two peers using a pipe, a peer will create an input pipe as well as a pipe advertisement. A remote peer must get the pipe advertisement through a query or through information provided directly by the host peer. The Pipe Service is obtained through the peer group of which the peer is currently a member of:

```
PipeService pipe = subPeerGroup.getPipeService();
```

In our application we use two pipes; a propagate pipe for issuing notification messages and a unicast pipe for carrying out peer download. The methods `createPropagatePipeAdv()` and `createUniPipeAdv()` creates the pipe services for the two pipes. The service provides the ability to create input and output pipes and create a message object to be sent through the pipe. In `LookupManager.java` the `createInputPipe()` method sets up the propagate input pipe:

```
InputPipe pipeIn = pipe.createInputPipe(pipeAdv, this);
```

Similarly, the unicast input pipe is generated in the `createUniInputPipe()` method. For the output pipe there is a similar approach:

```
OutputPipe pipeOut = pipe.createOutputPipe(pipeAdv, timeout);
```

The `createPropagateOutputPipe()` method and the `sendUniPipe()` method are responsible for setting up the output pipes. Our `createInputPipe()` method controls the setting up and taking down of the pipes. Most of the time a propagate input pipe will be up listening for messages. If a message is received, it is taken care of through a `PipeMsgListener` service. Through the `PipeMsgListener` service the `pipeMsgEvent()` is executed, and the incoming messages extracted are dealt with in `handleMsg2()` (see subsection 5.3.7 for message handling).

Within intervals, the output pipe is set up and messages transferred if there are any coming in from other methods. Methods in `LookupManager.java` may throw messages they want to submit as elements on to the vector `outputQueue`. Before setting up the output pipe, the `outputQueue` is checked for messages. If the message is a file download, the pipe set up will be the unicast pipe instead of the propagate pipe. Similarly, the device issuing a request for a file will automatically set up a unicast input pipe listening as soon as the request message is issued. `sendOutputPipe()` and `sendUniPipe()` are responsible for the actual message sending.

Similarly to the peers and groups in JXTA, all the pipes have a pipe ID. The application uses two fixed pipe IDs, in order avoid pipe advertisement distribution. Similar to the groups and peers, pipe advertisements can be distributed through publication. However, this is not desirable, as we do not know exactly which peer is sending out an advertisement at what time [76]. Often with JXTA, two peers wanting to connect both issue a pipe advertisement with different IDs. Some peers may pick up the advertisement with one of the IDs while other peers pick up the advertisement with another ID. Thus communication will be very uncertain after a while, since the peers also cache advertisements. Although we could delete all pipe advertisements from local caches at an early stage, we would then have the problem of finding a pipe service at all.

Since using one pipe corresponds to using one port (port number 9700), we can only have one pipe up and running at a time. Thus, during the time that the propagate input pipe is not up because we have to send messages or receive a file, the device could loose messages. While we have used queues to avoid downtime as much as possible, especially during a long file download a device could loose a lot of messages. Because of this issue, we have also not allowed the users to be connected to more than one group at the time. As a side effect, this restriction also helps preventing network congestion.

The propagate pipe is also an obstacle to security within the group as JXTA does not currently offer built-in security here. Thus for a secure SVF, security would have to be carried out by the application itself. Since the propagate pipe could not be securely implemented and security had to be put aside in the thesis due to time constraints, we have not implemented secure unicast channels either. A secure unicast channel is similar to an ordinary unicast, but requires all files to be divided into chunks of maximum 64 kb before transmission.

5.3.7 Messaging

Method	Description
`sendOutputPipe()`	Creates output messages and sends them.
`sendOutputPipeUpd()`	Sends a update or "new" message with several message elements.
`pipeMsgEvent()`	Receives messages from incoming pipe events.
`handleMsg2()`	Opens incoming messages and initiate response if demanded.

Table 5-6. Main methods in LookupManager.java connected with messaging.

Notification messages are an important part of the SVF as described in 4.4 and 4.5. All application messages (notifications well as file downloads) are issued as messages using pipes. The messages have different formats depending on their type. The messages themselves are separated by different element types:

new: A new device has logged on to the group and sends the latest updates.
upd: When a new message is received, the other group members should respond by sending their latest updates. An update message should also be issued when a new file has been added to the repository.
rmv: A remove notification should be issued when a file has been removed from the repository.
req: Issue a request for a file from any of the other members in the group.
dwn: A download message contains the actual file to be downloaded.

Each message element type is accompanied by a number of other elements as well:

new:
no: The number of recurring elements issued in this message
FID+ <no>: The local file ID.
fname+<no>: The file name.

author+<no>: The file author ID.
version+<no>: The file version.
sizes+<no>: The file size.
sender+<no>: The peer ID of the peer who sent the request.

The <no> indicates that each element has a number added to it. Thus all elements can be repeated as many times as there are files in the local repository.

upd:
no: The number of recurring elements issued in this message
FID+ <no>: The local file ID.
fname+<no>: The file name.
author+<no>: The file author ID.
version+<no>: The file version.
sizes+<no>: The file size.
sender+<no>: The peer ID of the peer who sent the request.

The <no> indicates that each element has a number added to it. Thus all elements can be repeated as many times as there are files in the local repository.

rmv:
fname: The file name.
author: The file author ID.
version: The file version.
sender: The peer ID of the peer who sent the request.

req:
fname: The file name.
author: The file author ID.
version: The file version.
sender: The peer ID of the peer who sent the request.

dwn:
fname: The file name.
author: The file author ID.
version: The file version.

file: The file itself.

Two of the notification messages require a response from the receivers. First, issuing a "new" message requires that the others respond with an update also issuing all files that is currently contained within the group of their local repository. Second, a file request notification must be responded to by a file download if any of the peers have the requested file.

The messages themselves are created using a message object. Elements are added to the message object in this way:

```
StringMessageElement version1 = new StringMessageElement
("version", version , null);
msg.addMessageElement(null, version1);
```

Here an element is created called "version" where the variable version is inserted. The last null argument is used for digital signatures. If no signature is specified, null is passed. The null value passed in the addMessageElement indicates that the default namespace is used (which is the set of name tags in this MessageElement, which again is just "version").

The message itself is sent by the command:

```
pipeOut.send(msg);
```

All message creation and issuing are carried out in the methods sendOutputPipe() and the sendOutputPipeUpd(). The latter handles update and new messages when this device need to inform the other group members of all files in the local repository, not just the change of one file.

The pipeMsgEvent()method picks up the messages as incoming events from the OutputPipeListener. The handleMsg2() extracts the messages by the following commands:

```
version = getMsgElement(msg,"version");
```

where `msg` is the message object and `version` is the element type.

5.3.8 Versioning

Method	Description
`doBtnAddSVF()`	Adds a new file to the SVF repository.
`doBtnDelFile()`	Removes a file from the repository.
`doBtnOpenFile()`	Opens a file by means of a third party application.
`checkRepositoryUpdates()`	Checks if any of the files located in the SVF directory has received a new modification date. If they have, the database is updated and notifications created.

Table 5-7. Main methods in LookupManager.java connected with versioning.

File versioning is carried out in order to ensure other participants are informed as file repositories are updated as described in 4.5. When a new file is added to the repository through the method `doBtnAddSVF()`, the file itself will be copied into the common directory, and the database will be updated, setting the filename, author and other information. The version will always be set to one. But before the update, a check must be carried out to ensure the file is uniquely defined within the group and local repository.

A file in the local repository can be opened for read and write access by a third party application. In order to do so, the path of the execution file for the third party application must be set in the `Consts.java` file, the setup file which has been mentioned several times in this chapter (see also Figure 5-1).

The `doBtnOpenFile()` will open the file if a user presses the “open file” button. The method chooses a fitting application to open the file with, based on the file extension. In our pilot version of the SVF, the Notepad application and a browser have been connected. Of course more applications could be connected or alternatively one could connect the file opener of the operating system which we have not done as the pilot is a demonstrator only.

The file itself has to be saved back to the same location under the same name in order to be registered as a new file version. If it is stored under another name, the application will not register the file as a member of the local repository at all. In order to include such files in the repository, the file must be added to the repository again as a new file and will be marked as a first version.

If a file is written to and saved back again under the same name, within a time interval that can be set, the repository directory will be swept looking for file updates. The repository sweep is initiated at the `startSVF()` in the `SVFGUI.java` class which calls the `checkRepositoryUpdates()` method that does the actual work.

The sweep starts with comparing the database `files` table with the file's modified date in the directory. If the files modified date is newer than the registered date, the `stampTime` in the database `files` table will be updated. The file's version number will then be incremented. But if the increment causes a collision with another uniquely defined file, the user will be given the option of saving the file under another name or alternatively the file will be deleted. As a confirmation of the version change, an update message is issued to the other group members. If updates are only found for groups that this device currently is not logged in to, no update messages will be issued since the changes to the files will be registered as we log on to the group by the issue of a `"new"` message.

When a file is downloaded from one peer to another, the file will be registered with the same file version and author as it had during download time. As mentioned in section 4.5, a file with an incremented version number could either be a copy or it could be a previously downloaded version that has been updated independently. In order to see the difference the field `sent` in the database table `fileCopy` are set as a file is copied or received. Thus the files will be marked as a copy, while a file without the `sent` field set will be an unique file. The application users will see the differences because the author will be changed as a file is updated locally. For files that are not the first version, the file list in the GUI will show the version number in square brackets (see Figure 5-2). Similarly to above, it is not allowed to download a file if the file version, name and author is the same.

For remote files, the sent field will not be accessible to others. This is to avoid additional messages to be issued as the `sent` field changes after a download.

Removing a file is carried out by pushing the “Remove file” button and running the corresponding `doBtnDelFile()`. Here the file and related information are deleted from the database and a notification message on the removal is issued.

6 Testing and discussion

In this chapter we will discuss the results from the project. In 6.1 we describe the results from testing the system and in 6.2 we describe the demonstrator's limitations. In 6.3 our research contribution is summarized. In 6.4 we comment on the thesis results both for the model and the implementation, An important issue is how well the system scales as discussed in 6.5. Finally in 6.6 we suggest improvements and extensions to the system.

6.1 Testing

Most of the development has been carried out between two instances of SVFs on the same computer. Creating groups and joining them function well, also to withdraw a SVF membership although it may take some time as remove messages are issued before the pipe is taken down.

We have not taken any measures to separate groups with the same name, as it was considered irrelevant to demonstate our model. However, should the software ever be released, this must of course be accounted for.

Importing a file and removing a file is also working, with the exception of some filenames with unusual characters like ' which we did not have time to sort out. (We experienced troubles with a file called "Beethoven's symphony No. 9 (Scherzo)") .

The other device sees the file through notifications well, but we may get trouble in downloading files if the network connection to the rendezvous is lost for some reason, also temporarily. The rendezvous we tested against was external, which means that the Internet network connection from time to time could be temporarily unstable or slow to establish. When this happens we usually get trouble during file transferal. We have found that retrying later on and ensuring that the rendezvous connection usually is there, is the best way of coping with file transferal failures.

During file versioning, the author is shown as a JXTA ID. We could also have converted the ID to a peer's nickname, which we think would have worked better. Thus, we have added a field `me` in the peer table to differenciate between this device and others. However, we have enough time to finish this change.

Regarding versioning of the files, this is working well with the exception of repository updates that today allow one file to update to another with the same name, version and author. This should not happen, and is of importance to our model, but be still consider the model to be a sufficient proof of concept.

During testing, we also discovered that our GUI is not optimal, as we think the file list-box should have been extended to also include the different authors and locations. Today we must select a file in order to see these details, as files with the same filename and versions appear equal to the users otherwise.

During the testing of the secure login we experienced that this feature is not functioning since it is possible to log on to a group also with another password than the one requested. We have also tested Sun's tutorial code in this matter receiving the same result. There seem to be errors with the authentication toward the rendezvous as the rendezvous never asks for a password confirmation. The group concept itself with unicast and multicast is however functioning according to specifications.

The application is stopped by executing the close button in the upper right corner. Sometimes the closing can be delayed, as the rendezvous need to be contacted for the peer to be removed from all groups and services.

6.2 Limitations of the implementation

Besides the already mentioned weaknesses, our implementation is limited by the lack of large scale tests and the lack of tests using a number of different operating systems and a variety of devices with different resources available. Moreover, while we have demonstrated how to use secure access to get into a group, security for the whole group concept has not been considered.

Finally, but still crucial to the application, we have not received any user feedback on our system nor carried out usability testing for the GUI.

6.3 Research contribution

As described in related work in section 2.6, quite a few has previously offered solutions for file exchange, but to our knowledge combining the following elements together is new:

- Casual collaboration (through a simplified GUI to avoid a steep learning curve and through an application that aims at being accessible in a wide range of settings).
- Access to a secured repository containing files over a user-defined time period without need of server access.
- A collection of files located differently appearing to the user as one unity.
- Document versioning detection as group users collaborate.

The most closely related applications usually focus on one or two of the elements that the SVF contains, but not the combination of all three.

The GRAM pilot project is probably the project that is closest to ours and thus worth mentioning in particular. GRAM is similar, but also far more elaborate in offering resources amongst the participants. A difference is that the GRAM pilot is created for software developers, and not for people without particular technical background who just wants to share resources. Thus we would not call the GRAM pilot "casual" in its approach, although otherwise the solution are much more elaborate than our model.

6.4 Discussion of results

We have divided the discussion of the results further into a discussion about the SVF model itself in 6.4.1 and a discussion of the implementation in 6.4.2. At the end we have also included a small discussion of the shortcomings of the JXTA as middleware in 6.4.3.

6.4.1 The SVF model

We have used our pilot as a proof of concept for our SVF model. The pilot testing has so far been successful, although some testing remains, see section 6.2.

The tests we have carried out so far raises a few questions about the model we have used. The tuplespace model was used as a template for our SVF model. Our main reason for choosing the model was that resources were put into a "pool" which is shared by all

participants. This approach is particular attractive also to our approach with many distributed devices. However, the tuplespace model is also not concerned with who put out a tuple into the tuplespace. As long as only the notifications are issued, this also works well for our application. But we think that for file download, the SVF users would like to know which device currently holds the different files, since some peers would be more attractive than others to download from.

Had the network connection between the devices been optimal and all devices resourceful it would probably not have been important from where the file was offered. But as the networks functions today with varying bandwidth and device capacity, some devices are more easily accessible than others when it comes to downloading files. This could to some extent be accounted for if several devices hold the same file copies (see 6.6). But if not, the user may prefer to download a file later on if network connection is poor or the device has little resources.

As a device leaves the network, we may also like to know which files are not available any longer. We may carry out an unsuccessful search or ask around who had the file, but we foresee that it would be easier to see which device(s) the file is located on and wait until this device reconnects before starting a search.

Moreover, in spite of our choice of the tuplespace model for our application, the JXTA middleware uses the publish-subscribe model for their group concept where a peer subscribe to propagate messages as long as the peer is connected to the group. Thus our SVF implementation is also under influence of the publish-subscribe model as long as we employ JXTA as our middleware. In this way our SVF application and not the middleware becomes responsible for finding a solution to message loss for devices that are removed ad-hoc. The mechanism we employed is the passing of new and update messages as a peer joins a group. Thus, we may claim that the SVF implementation also uses the publish-subscribe model. Had we employed another middleware than JXTA, the publish-subscribe model would maybe not have been used.

Another issue is the file update and collaboration around it. As the model is today, all files have to be downloaded before they are read or written to. By this organization updating in this way, we get a very simple but also functional model with few messages going around in the network. But the model, especially combined with a common repository where all files are kept and where different groups have their own copy of a

file, is not optimal with respect to disk space. However, for most modern computers disk space is not critical, while network capacity and the overhead working time to organize other more elaborate models could be bottlenecks. This conclusion may not hold for small portable devices where disk space can be critical.

6.4.2 The implementation

As mentioned in the previous subsection, the pilot implementation has worked well during the tests that we have carried out. However, we have not strained the implementation by running it over narrow banded networks or tested it large scale. Thus, so far the notification messages seem to work well, although we suspect that messages could be dropped especially during file download. To some extent it could be accounted for by logging off and on to the group, but we would maybe need additional mechanisms in order to account for lost messages especially if a group grows large both in files and members.

A critical issue in the use of the SVF pilot today is the JXTA Discovery Service harvesting too many advertisements. The peer advertisements in the pilot could be excluded, as it has no function beside the nice ability to know whether your collaboration partners are online. But the number of groups on the JXTA network may grow too large for the system to handle properly, especially if a device is connected to the Internet over longer periods of time and thus has well established routes in the network. To some extent we have accounted for this by filtering the group advertisements based on the XML tag <Desc> (description) which is set specifically for all group instances generated in the SVF application. This approach works as long as the SVF application has not been put into large scale use.

To also filter away SVF groups that we are not interested in, we could carry out publishing of group advertisements only issued specifically to devices we would like to invite into the group. Thus other devices could avoid the burden of additional incoming group advertisements for which they are not invited anyway. For peers joining later on, we could send them the advertisement in the same way.

This approach raises issues about peer discovery in a similar way as do group discovery for our pilot. As the approach would help filtering the number of incoming group advertisements, we now have an issue with filtering peers instead. A solution could be

not to make a peer visible to all users in the network, but only issue peer advertisements to the peers that you would like to collaborate with. However, at some stage new peers would have to be added to a device's peer list in order to know where to send group advertisements. Thus it will be hard to avoid the Peer Discovery Service to be collecting peers we are not interested in collaborating with as well. The solutions would maybe be to keep an address book, and issue our peer ID by another media like e-mail to collaborating partners. Then we could perhaps limit the Peer Discovery Service sufficiently. We could also use a catalogue service analogue to the Domain Name Service, but then we are back to relying on servers or routing peer messages in the network.

The group security is a shortcoming of the application. Although we in section 1.3 defined it outside of the scope of our first pilot, in some situations clearly a secure group concept would be vital to application success. Although unsuccessful, we have implemented a basic password routine to demonstrate secure group logon, but we have not secured the passwords available on each peer in any way. Moreover, there are also some shortcomings of the JXTA security concept which lacks a security regime for propagate messages and advertisements (see subsection 6.4.3). In addition, there is the issue of securing the many groups with different means of authentication. In the pilot a group is authenticated by sharing common passwords, one for each group which would be too much for one user to remember as the numbers of groups grow large. An alternative could be to use asymmetric cryptography with digital signatures where all group advertisements could be joined by invitations which were signed by the person inviting to ensure that the message had not been tampered with. In order to login to the group, each participant could use their own private key. Thus the number of tokens to log on to a group could be reduced to a private key only. JXTA offers the possibilities to add asymmetric cryptography including digital signatures to messages and unicast pipes.

Port access is a potential bottleneck to the SVF application. While a large file is downloaded, the SVF today will not be able to catch notifications at the same time. Using sockets to split the traffic between two ports could provide us with more access capacity, but still the receiving port would be busy with the incoming file instead of listening to notifications.

In our application we use the same pipe ID for all unicast pipes. One could foresee that if several pairs of peers in a large group initiated download simultaneously, there could be a

cross-connection between the peer couples as they all connected to the common rendezvous peer. However, so far this has not been confirmed during testing. To avoid this, we could send a newly generated pipe ID together with the notification as we request a file. Using a separate pipe advertisement should be avoided as it could lead to communication problems as described in 5.3.6.

Since JXTA offers no secure propagate pipes, we have not implemented secure unicast channels. A secure unicast channel is similar to an ordinary unicast, but requires all files to be divided into chunks of maximum 64 kb before transmission.

Today there are no application mechanisms that handle the situation where two different groups have the same name. Since this is a matter of getting hold of the right advertisement ID, the ID or another type of identification must be added to the list box in the GUI to make the group appear unique to the user and to the application. As this is not considered important for demonstrator purposes, we have omitted the feature in the pilot implementation.

Finally, the GUI, the many listeners provided by JXTA and the juggling of the pipes altogether requires many threads to be used. All these threads make the application quite resource intensive which makes it difficult to run several instances on the same PC, let alone to run it as it is today on a smartphone or another device with little resources. To encounter for the situation, JXTA promise the use of minimal edge peers which can send and receive messages, but does not cache advertisements or route messages.

6.4.3 Shortcomings of the JXTA

Routing and discovery issues as peers move around are still a large obstacle in P2P networks. Even searching for a particular peer or group using the JXTA ID will not necessarily provide results especially if the devices are separated wide apart. The rendezvous peers connected to the devices may not have a direct route to each other, and as queries needs to be issued and responded to, the search expires before reaching the other party. While routing in JXTA seems to be well considered and carefully planned, we believe routing and discovery for P2P networks for a particular peer moving around on the Internet deliver far poorer quality than for stationary devices today.

Also, in 6.4.2 we mentioned the need for reducing the amount of advertisements to an acceptable level, which is only partly supported today by the JXTA. Today, it may take some time for Internet connected JXTA devices to find a route to a new device that is being connected. But after a few days or weeks, the route is found, and there is a large response of unknown groups and peers fetched by the Peer Discovery Service if not filtered. Even limiting the groups to only SVF applications as we have done could be a problem if the SVF were put into large scale use.

Moreover, as mentioned in 6.4.2, JXTA provides a security regime, but the regime itself comes with some drawbacks with respect to our SVF model as we need propagate messages and advertisements to be encrypted and connected to a security model as well. Without providing security for these types of communications, we will not have a fully secure group concept. As of today, we would have to implement this security ourselves. Regarding our problems in making our security solution work, we have not had time to investigate the errors thoroughly.

Searching for pipes through pipe advertisement does not function well in JXTA as mentioned in 5.3.6 and 6.4.2. Too many different pipe IDs could be generated by different devices which will be an obstacle in choosing a common pipe ID for communication. Thus pipe IDs must be distributed along with the group advertisement or by passing messages directly by means of pipes whose ID already has been distributed.

6.5 Scalability

Many of the issues we have mentioned 6.4 also affect scalability. These are the difficulties of P2P routing over the Internet, the need for additional filtering of incoming advertisements, the number of file copies generated taking up space as they are copied one time per group and the lack of access to more than one pipe at the time. In addition we do not know how traffic will grow in the network as the number of SVF peers grow large. But we do know that JXTA functions fairly well at least where distances are not too large.

If the groups become large, there is a risk that there could be a lot of messages in the network if files are updated intensively. Also, messages could be dropped, which is a serious obstacle.

Also, we have chosen Java as our platform with scalability in mind. Java offers wide platform accessibility from smaller and portable devices to servers and resourceful computers. It is also available on a wide range of operating systems which is beneficial. The SVF application itself today runs on a very limited number of these platforms, thus the application would need adjustments in order to function well. Of particular interest would be to modify the code to fit some of the smartphone devices running operating systems like Symbian. As JXTA offers a special minimal edge peer connection for these devices without routing or caching, likely the pilot would have to be modified quite a lot to fit, as databases could then not be used either. Thus storage need on such devices would have to be considered carefully.

6.6 Future work

Foremost, a reliable security concept and an advertisement limitation should be carried out for our SVF implementation as described in subsection 6.4.2. Along with the limitations, we could also carry out peer group invitations through use of advertisements issued only to invited peers.

Depending on how the SVF behaves in a large scale environment, we would maybe need to implement a more failure proof system for notification messages if these are lost for some reason.

Regarding file versioning detection, as mentioned in the testing in 6.1 we would like to change the GUI file list-box not only to show file names and versions, but also to show other features like on which peer the file resides and the file author as well. This would be helpful in finding the specific file, without having to select many different files for viewing.

We would also like to extend the versioning detection carried out in our pilot by a graphical representation showing the branching of the file versioning tree. Moreover, we could have marked the graphical representation by file location and version number. Also, allowing the users themselves to mark two files in the repository as versions of one another could have been useful. This should be allowed even though the files had different names.

There are times when creating a group could be too cumbersome. For example, if only one file needs to be transferred, and thereafter we wish to withdraw the SVF membership. For these instances it would have been possible only to issue files on the network without any need to establish a SVF. Technically our application allows for it, as one could transfer files without being connected to a JXTA subgroup as well. All devices log on to the base netPeerGroup, which could also be used to transfer files directly. We would not have any security here, but for a casual transferral of one file only, it could represent an attractive feature.

If more than one device has a file copy which is requested for download, we would also like to add mechanisms for choosing the most resourceful device with the best connection. This could have been carried out by letting the less resourceful devices be waiting a little bit before they responded to the file request. Resourceful devices could respond immediately. Alternatively, a swarming protocol as described in 2.3.1 could be useful especially in class learning environments to avoid the problem of “hearding”, where a number of peers request a popular file all at the same time. Relying on multicast would be the most effective, but a swarming protocol could also be used to improve the spreading of large files quickly if all requests do not come exactly at the same time.

In our pilot we did not implement search functionality neither for files, peers nor groups although one could foresee it as useful with the growth of a larger SVF network. Search of peers and groups would be straight forward using the Peer Discovery Service, while file search could respond by finding the file in the database and selecting the file entry in the GUI. A device could search within the currently available SVF repository for a file or a particular version of a file. An extension of the search could be to store the search request in the database if it could not be answered by the member devices currently logged on to the SVF. Later on, when other SVF members logged in, the search could be repeated to see if any of these devices had the file. For some files request, it would be beneficial to extend the search period in this manner, while for others, the file would loose interest if not downloaded immediately.

During the opening of a file by a third party application, we could have connected our application to the application chooser available from the operating system instead of just adding a few applications as we have done for our pilot model. Also, during download we could have given the user more feedback on how long time the download would take.

Moreover, today the system only handles files as resource exchange. Offering other resources for exchange as well could be of interest. Of course, this and the other suggestions must be considered also from a simplicity point of view. If the application becomes extensive, the threshold for use will be higher as well. By implementing all functionality described in this section, the abilities of the system could change considerably, which also should be taken into account.

7 Conclusion

In this thesis we have considered different models for casual collaboration based on the P2P technology. We have employed the tuplespace model to build a tool for casual collaboration of resources called the Shared Virtual Folders (SVF). Thus a number of peers share resources from a common virtual folder. The folder is a repository where each peer offers a part of their local harddisk for sharing. The repository is virtual since it is made up of many harddisks, while the users perceive it as one unity. In order to ensure resource handling through collaboration, versioning detection within the folder is part of the model.

Thereafter, we implemented a pilot application based on the SVF model where resources were implemented as file exchange only due to time constraints. The application appears to be well functioning, while we have not yet tested it in a larger scale. While improvements could be made to our pilot, our model for SVF seems viable.

We think that our approach to resource sharing through SVFs contains some valuable contributions. While many solutions for file exchange exist, our SVF combines the elements of 1) casual collaboration through a simplified GUI to avoid a steep learning curve and through an application that aims at being accessible in a wide range of settings, 2) access to a secured repository containing files over a user-defined time period without need of server access, 3) a collection of files located differently appearing to the user as one unity and 4) document versioning detection as group users collaborate. Futhermore, despite limitations, we have gained some insight in casual collaboration using the P2P approach through our implementation.

8 References

1. Borch, N. and L.K. Vognlid. *Searching in variably connected P2P networks*. in *International MultiConference in Computer Science & Computer Engineering* 2004. Las Vegas, Nevada.
2. Androutsellis-Theotokis, S. and D. Spinellis, *A survey of peer-to-peer content distribution technologies.* ACM Computing Surveys, 2004. **36**(4): p. 335-371.
3. Denning, P., et al., *Computing as a discipline.* Communications of the ACM, 1989. **32**(1): p. 9-23.
4. NRC, *Academic careers for experimental computer scientisks and engineers*. 1994, National Research Council, National Academy Press. p. 17-20.
5. NationalResearchCouncil, *Academic careers for experimental computer scientists and engineers.*, in *National Research Council*. 1994, National Academy Press: Washington, DC. p. 17-20.
6. Schoder, D. and K. Fischbach, *Peer-to-peer prospects.* Communications of the ACM, 2003. **46**(2): p. 27-29.
7. Barkai, D., *Peer-to-peer computing*. 2001, Hillsboro, OR: Intel Press, Intel Coorporation
8. Schoder, D., K. Fischbach, and C. Schmitt, *Core concepts in peer-to-peer networking*, in *Peer-to-peer computing : the evolution of a disruptive technology* R. Subramanian and B.D. Goodman, Editors. 2005, Idea Group Pub: Hershey, PA
9. Groove. *Groove Virtual Office*. 2006 [cited 2006 01.03]; Available from: http://groove.net.
10. Milojicic, D.S., Kalogeraki, V.,Lukose, R.,Nagaraja, K.,Pruyne, J.,Richard, B.,Rollins, S.,Xu, Z. *Peer-to-peer computing.* 2002 [cited; Available from: http://citeseer.ist.psu.edu/cache/papers/cs/25966/http:zSzzSzwww.hpl.hp.comzSztechreportszSz2002zSzHPL-2002-57.pdf/milojicic02peertopeer.pdf.
11. Napster. *Napster*. 2007 [cited 10.04.2007]; Available from: http://www.napster.com/.
12. Clarke, I., Sandberg, O., Wiley, B., Hong, T. W. *Freenet: A Distributed Anonymous Information Storage and Retrieval System* in *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability* 2000. Berkeley, CA.
13. Cohen, B. *Incentives Build Robustness in BitTorrent*. in *1st Workshop on Economics of Peer-to-Peer Systems*. 2003. Berkeley, CA.
14. Kubiatowicz, J., Bindel, D.,Chen, Y.,Czerwinski, S.,Eaton, P.,Geels, D.,Gummadi, R.,Rhea, S.,Weatherspoon, H.,Weimer, W.,Wells, C.,Zhao, B. *OceanStore: An Architecture for Global-Scale Persistent Storage.* in *Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000).* 2000. Cambridge, MA.
15. Anderson, D.P., Cobb, J.,Korpela, E.,Lebofsky, M.,Werthimer, D, *SETI@home: An experiment in public-resource computing.* Communications of the ACM, 2002. **45**(11): p. 56-61.

16. fightAIDS@home. 2006 [cited 01.03.2006]; Available from: http://fightaidsathome.scripps.edu/.
17. Cohen, E., Shenker, S. *Replication Strategies in Unstructured Peer-to-Peer Networks*. in *In Proceedings of ACM SIGCOMM*. 2002. Pittsburgh, PA.
18. Adar, E. and B.A. Hubermann. *Free riding on Gnutella*. First Monday 2000 [cited; Available from: http://www.firstmonday.dk/issues/issue5_10/adar/index.html.
19. Lui, S.M., Lang K R, and S.H. Kwok. *Participation incentive mechanisms in peer-to-peer subscription systems.* in *35th Hawaii international Conference on System Sciences*. 2002. Hawaii.
20. Johanson, B. and A. Fox, *Extending tuplespaces for coordination in interactive workspaces.* The Journal of Systems and Software, 2004. **69**: p. 243-266.
21. Marshall, D. *Remote Procedure Call*. 1999 [cited 23.03.07]; Available from: http://www.cs.cf.ac.uk/Dave/C/node33.html.
22. Alonso, G., et al., *Web services. Concepts, architectures and applications.*, ed. M. Carey and S. Ceri. 2004, Berlin Heidelberg, Germany: Springer.
23. Tanenbaum, A.S., Steen, M. v., *Distributed systems: principles and paradigms*. 2002, Upper Saddle River, N.J.: Prentice Hall. XXII, 803 s.
24. Wikipedia. *Message -oriented middleware*. 2005 [cited 27.03.07]; Available from: http://en.wikipedia.org/wiki/Message_Oriented_Middleware.
25. Terpstra, W.W., Behnel, S., Fiege, L., Zeidler, A., Buchmann, A.P. *A Peer-to_peer Approach to Content-Based Publish/Subscribe*. in *Second Int workshop on Distributed Event-Based Systems*. 2003. San Diego, CA, USA.
26. Oki, B., Pfluegl, M.,Siegel, A.,Skeen, D. *The Information Bus – an Architecture for Extensible Distributed Systems.* in *14th ACM Symposium on Operating System Principles*. 1993. Asheville, NC, USA.
27. Hansen, K. and C. Damm. *Building flexible, distributed collaboration tools using type-based publish/subscribe — the Distributed Knight case.* in *IASTED International Conference on Software Engineering*. 2004. Innsbruck, Austria.
28. Muthusamy, V., Jacobsen, H-A. *Small-Scale Peer-to-Peer Publish/Subscribe*. in *Second Workshop on Peer-to-Peer Knowledge Management*. 2005. La Jolla, San Diego, California, USA
29. Carriero, N., Gelernter, D., *Linda in context.* Communications of the ACM, 1989. **32**(4): p. 444-458.
30. Johanson, B., A. Fox, and T. Winograd, *The interactive workspaces project: Experiences with ubiquitous computing rooms.* Pervasive computing, 2002(April-June): p. 71-78.
31. Perich, F.J., A., Finin, T., Yesha, Y. , *On Data Management in Pervasive Computing Environments.* IEEE Transactions on Knowledge and Data Engineering, 2004. **16**(5): p. 621-634.
32. Xu, B. and O. Wolfson. *Data management in mobile peer-to-peer networks*. in *Databases, information systems, and peer-to-peer computing (DBISP2P)*. 2004. Toronto, Canada.
33. Stutzbach, D., D. Zappala, and R. Rejaie. *The Scalability of Swarming Peer-to-Peer Content Delivery* in *Networking 2005*. 2005. Waterloo Ontario Canada.
34. Wilson, B.J., *JXTA*. 2002, USA: New Riders Publishing.

35. Friday, A., Roman, M.,Becker, C.,Al-Muhtadi, J., *Guidelines and open issues in systems support for Ubicomp: reflections on UbiSys 2003 and 2004.* Personal Ubiquitous Computing, 2006. **10**(1): p. 1-3.
36. Sousa, J.P. and D. Garlan. *Aura: An Architectual Framework for User Mobility in Ubiquitous computing Environments*. in *Software Architecture: System Design, Development and Maintenance (Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture)*. 2002. Kalsruhe, Germany: Kluwer Academic Publishers.
37. Verma, D.C., *Using peer-to-peer systems for data management*, in *Peer-to-peer computing : the evolution of a disruptive technology* R. Subramanian and B.D. Goodman, Editors. 2005, Idea Group Pub: Hershey, PA
38. Borch, N.T. *Improving semantic routing efficiency*. in *Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*. 2005. San Diego, California.
39. Elmasri, R.A. and S. Navathe, *Fundamentals of database systems*. 2000, Reading, MA, USA: Addison-Wesley.
40. Kubiatowicz, J., *Extracting guarantees from chaos.* Communications of the ACM, 2003. **46**(2): p. 33-38.
41. Stoica, I., Morris, R.,Liben-Nowell, D.,Karger, D.R.,Kaashoek, M.F.,Dabek, F.,Balakrishnan, H., *Chord: A scalable peer-to-peer lookup service for internet applications.* . IEEE/ACM Transactions on networking, 2003. **11**(1): p. 17-32.
42. Ferreira, R.A., et al. *Search with Probabilistic Guarantees in Unstructured Peer-to-Peer Networks.* in *Proceedings of IEEE P2P'05*. 2005. Konstanz, Germany.
43. Conradi, R. and B. Westfechtel, *Version Models for Software Configuration Management.* ACM Computing Surveys, 1998. **30**(2): p. 232 - 282
44. Collins-Sussman, B., B.W. Fitzpatrick, and C.M. Pilato, *Version control with subversion*. 2004, O'Reilly Media: Stanford, California
45. Takata, K. and J. Ma. *GRAM - A P2P System of Group Revision Assistance Management.* in *18th International Conference on Advanced Information Networking and Applications (AINA).* . 2004. Fukuoka, Japan: IEEE.
46. Wikipedia. *Revision control*. 2006 [cited 26.03.07]; Available from: http://en.wikipedia.org/wiki/Revision_control.
47. Emmerich, W., *Enigneering Distributed Objects*. 2000, Chichester, West Sussex, England: John Wiley & Sons Ltd.
48. Ding, C.H., Nutanong, S., Buyya, R., *Peer-to-Peer Networks for Content Sharing*, in *Peer-to-Peer Computing: The Evolution of a Disruptive Technology*, R. Subramanian, Editor. 2005, Idea Group Publishing: Hershey, PA, USA.
49. Joseph, S. *Semantically routing queries in peer-to-peer networks.* in *Proceedings of the International Workshop on Peer-to-Peer Computing* 2002. Pisa, Italiy.
50. Wikipedia. *Bonjour*. 2006 [cited 09.02.2006]; Available from: http://en.wikipedia.org/wiki/Bonjour_%28protocol%29.
51. Apple. *Bonjour*. 2006 [cited 09.03.2006]; Available from: http://images.apple.com/macosx/pdf/MacOSX_Bonjour_TB.pdf
52. Asterisk. 2006 [cited 09.02.2006]; Available from: www.asterisk.org.

53. Microsoft. *Universal Plug and Play in Windows XP*. 2001 [cited 09.02.2006]; Available from: http://www.microsoft.com/technet/prodtechnol/winxppro/evaluate/upnpxp.mspx.
54. UPnPForum. *Software Development Kits (SDKs)*. 2007 [cited 27.03.2007]; Available from: http://www.upnp.org/resources/sdks.asp.
55. JXTA. *JXTA v 2.3. x:Java Programmer's guide*. 2006 [cited; Available from: http://www.jxta.org/docs/JxtaProgGuide_v2.3.pdf.
56. Borch, N. *Social peer-to-peer for social people*. in *International conference on Internet technologies & applications*. 2005. Wrexham, United Kingdom.
57. Borch, N. *The Socialized.Net*. 2005 [cited 27.03.2007]; Available from: http://www.socialized.net/files.html.
58. Wikipedia. *Shared resource*. 2007 [cited 27.03.07]; Available from: http://en.wikipedia.org/wiki/Shared_file_access.
59. Wikipedia. *Samba (software)*. 2007 [cited 27.03.07]; Available from: http://en.wikipedia.org/wiki/Samba_%28software%29.
60. J_K9@Linux. *Proposing An Open Source Groove Alternative*. 2007 [cited 28.03.07]; Available from: http://wolphination.com/linux/2007/02/13/proposing-an-open-source-groove-alternative/.
61. Wikipedia. *Microsoft Office Groove*. 2007 [cited 28.03.07]; Available from: http://en.wikipedia.org/wiki/Microsoft_Office_Groove.
62. iFolder. *iFolder*. 2006 [cited 27.03.07]; Available from: http://www.ifolder.com.
63. Wikipedia. *Google Docs&Spreadsheets*. 2007 [cited 02.06.07]; Available from: http://en.wikipedia.org/wiki/Google_Docs_&_Spreadsheets.
64. JXTA. *myJXTA User Guide*. 2006 [cited 27.03.07]; Available from: http://instantp2p.jxta.org/Userguide.html.
65. JXTA. *The JXTA Content Manager Service*. 2006 [cited 27.03.07]; Available from: http://cms.jxta.org/cmswhitepaper.html.
66. JXTA. *myJXTA2 project home*. 2006 [cited 27.03.07]; Available from: http://myjxta2.jxta.org/.
67. Stone, E., Czerniak, T., Ryan, C., McAdoo, R. *Peer to Peer Routing*. [cited 28.03.07]; Available from: http://ntrg.cs.tcd.ie/undergrad/4ba2.05/group6/index.html.
68. Wikipedia. *Gnutella*. 2007 [cited 28.03.07; Available from: http://en.wikipedia.org/wiki/Gnutella.
69. UPnPForum. *UPnP*. 2007 [cited 27.03.2007]; Available from: http://www.upnp.org/about/default.asp.
70. Arlov, L., *GUI-guiden*. 1996, Oslo, Norway. 42-53.
71. Janert, P. *Embedded databases*. 2004 [cited 31.03.07]; Available from: http://www.perl.com/pub/a/2004/09/12/embedded.html.
72. Simon, J. *Rethinking Swing Threading*. 2003 [cited 25.05.07]; Available from: http://today.java.net/pub/a/today/2003/10/24/swing.html?page=1.
73. Gradecki, J.D., *Mastering JXTA. Building Peer-toPeer Applications*. 2002, Indianapolis, Indiana, USA: Wiley Publishing, Inc.
74. ISO/IEC, *Information Technology - Open Systems Interconnection - Remote Procedure Call (RPC)*. 1996, 11578:1996. ISO (International Organization for Standardization). .

75. JXTA. *MiniJxta Sample Application*. 2005 [cited 260507]; Available from: http://people.jxta.org/tra/minijxta/MiniJxta.html.
76. Oaks, S., B. Traversat, and L. Gong, *JXTA in a nutshell*. 2002, Sebastopol, CA, USA: O'Reilly & Associates, Inc.
77. Freebyte. *Freebyte's Guide to Free Databases* 2007 [cited 310307]; Available from: http://www.freebyte.com/programming/database/#opensourcedatabases.

9 Appendix A: Embedded databases

Evaluation of embedded databases. Some databases have been left out because of lack of information about them or because they clearly was not freely available. This guide has been carried out based on the guide of Freebyte [77] in addition to database homspages. The footprint is not included in the table.

*: Embedded? X = yes **: Open source? X = yes ***: Embedded routines for ping, FTP, telnet, SMTP, POP3, HTTP

Database	*	**	Language	Platform	SQL type	URL	Maintenance	Cost /licensing	Access
Embedded MySQL	X	X	ANSI C	Win95/Win98/NT, Linux, Solaris, FreeBSD, AIX, SunOS, etc. drivers available.	SQL	www.mysql.com/products/embedded/	MySQL AB	GPL	Native C API, JDBC, ODBC, Python,Perl, PHP, .NET, Ruby, VB
PostgresSQL	X	X	C, C++, or Java	Runs on various flavours of unix, like Linux, FreeBSD. Clients available for OS/2 and Win32.	SQL	www.postgresql.org/	Yes, many	BSD	ODBC and JDBC drivers available.
Firebird	X	X	C and C++	32-bit Windows, Linux (i586 and higher), Solaris (Sparc and Intel), HP-UX (i386), FreeBSD and MacOS X. Some Firebird 1.0 builds are also available for WinCE and AIX.	Modified ANSI SQL:99	www.firebirdsql.org/	Firebird foundation	Initial Developer's PUBLIC LICENSE	native/API, dbExpress drivers, ODBC, OLEDB, .Net provider, JDBC native type 4 driver, Python module, PHP, Perl, etc.
SQLite	X	X	C	Linux-x86, Windows. Binaries: Linux-x86, Windows, and Mac OS-X ppc and x86.	SQL-92	www.sqlite.org/	Maintained by SQLite. Active contributors.	Public domain	C/C++, Python applications and more
Gadfly	X	X	Python	All Pyhon supported OS.	ODBC 2.0 SQL	gadfly.sourceforge.net/sql.html	No ongoing support	Freely use and copy it as long as you don't change or remove the copyright	Python
GNU SQL	X	X	C	Unix	Supports SQL89 and some extensions from	directory.fsf.org/gnusql.html	Freeware	GNU General Public License, Version 2	C/Unix

					SQL-92.				
CQL++	X	X	C++	Based on KDE. Windows NT, Windows 95, Windows 3.1 or Windows for Workgroups(client only), OS/2, Sun O/S, Sun Solaris, HP-UX, SCO UNIX, Linux, or any UNIX or other environment with a compatible C++ compiler.	SQL	www.cql.com/	Machine Independent Software Corporation	Free, GPL license.	ODBC
SolidDB Embedded-Engine	X	?	Built on mySQL	Linux, Windows, Solaris, Unix, Sun Netra HA Suite, and VxWorks.	SQL	www.solidtech.com	Solid	Commercially available	ODBC, JDBC
Empress RDBMS	X	?	C API, rest uncertain	Solaris, SUN O/S, HP-UX, AIX, Tru64 UNIX (Compaq), IRIX, SCO, Linux, Red Hat, SUSE, FreeBSD, etc, WIN NT, WIN 2000, WIN XP, QNX 4 & 6, Lynx O/S, Bluecat, RTLinux, Linux PPC, Lynx O/S	SQL	www.empress.com/	Empress Software Inc	Empress licence	C, JAVA, Microsoft Excel, Visual Basic and HTML.
RDM Embedded	X	?	C and C++	OS running C/C++, perhaps more	XML, SQL	www.birdstep.com/start/	Birdstep	Birdstep's licence	XML interface. Interfaces for Java, C/C++, and SQL. ODBC
c-treeSQL Server	X	?	C	QNX,LynxOS,Windows, Mac OS, HP-UX, Solaris.	SQL-92, ISAM	www.faircom.com	Faircom	Faircom licence Commercially available	Embedded SQL, Interactive SQL, JDBC, and ODBC). C and C++ interfaces and VCL/CLX components
Integra4 RDBMS	X	?	C/C++	?	SQL-92	www.co-soft.com/	Cosoft India Ltd	Cosoft India licence	ODBC, C, C++ or Java
H2 database engine	X	X	Java. H2 is built from scratch to overcome some limitations of Hypersonic SQL / HSQLDB.	Java enabled platforms. Can also be compiled to native code using GCJ.	SQL	www.h2database.com/html/frame.html	H2 group	modified version of the MPL 1.1 available at www.mozilla.org/MPL	JDBC and (partial) ODBC API; Web Client application
HSQLDB/ Hypersonic SQL	X	X	Java	Java enabled platforms.	SQL	www.hsqldb.org/	Hypersonic SQL Group.	Hypersonic SQL Group licence	embedded (into Java applications)
One$DB	X	X	Java	One$DB can run on any operating system for which the JVM (Java Virtual Machine) is available.	SQL:99	www.daffodildb.com/	Daffodil DB	LGPL license	ODBC, JDBC 3.0
MS SQL server 2005 Express Edition	X	N	Windows	Windows	SQL	www.microsoft.com/sql/editions/express/default.	Microsoft	Free to download, free to	ODBC, JDBC

						mspx		redistribute,	
IBM Cloudscape	X	X	Java	Java (all platforms), C (ODBC, X/Open CLI) on Windows, PHP (cross-platform).	SQL-92 and partial SQL:1999 and SQL:2003	www-306.ibm.com/software/data/cloudscape/	IBM	Zero-cost licensing, for redistribution or departmental use.	JDBC 3.0 compliant ODBC, X/Open CLI
FlashFiler	X	X	Delphi/Kylix(Pascal)	Different versions of Windows	SQL	sourceforge.net/projects/tpflashfiler/	Maintained by a team of individuals.	Mozilla Public License 1.1 (MPL 1.1)	Borland Delphi and C++ builder
DiamondBase	?	?	C++	?	?	www.csse.monash.edu.au/~darrenp/diamondbase.html	Maintained by individuals. Very little info available	Free non commercial use, and is negotiable for commercial use.	?
DataReel	***	X	C++ development kit	HPUX, MSDOS, RedHat Linux, Solaris, Windows	C++ routines	www.datareel.com/	DataReel Software Development	GNU Lesser General Public License (LGPL)	C++ routines
Oracle Berkeley DB	X	X	Java, XML. Bindings C, [C++]], Java, Perl, Python, Tcl, Smalltalk and more	Oracle Berkeley DB is a library that links directly into the application.	Linking routines	www.oracle.com/database/berkeley-db/index.html	Oracle	Dual license	Linking routines, Java, XML

10 Appendix B: Group and service advertisement examples

This appendix consists of samples of a group and a service advertisement. The group advertisement is listed first. The <GID> tag represents the group ID and the <MCID> is the Module Class ID which is unique and connects the peergroup advertisement with the module implementation advertisement. From the peergroup advertisement we see that the name is “TestGroup” and under the <login> tag are the login name and the encrypted password stored.

Both the group advertisement and the connected module implementation advertisement are published at the same time. The <Parm> elements which is found in both advertisements contains arbitrary parameters that are interpreted by each implementation. The <Svc> tags, also found in both advertisements, are elements that describe the association between a group service denoted by its MCID, and arbitrary parameters encapsulated in a <Parm> element.

In the module implementation advertisement, under the first <Svc> tag, the membership implementation is announced.

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PGA>
<jxta:PGA xmlns:jxta="http://jxta.org">
        <GID>
                urn:jxta:uuid-4D6172676572696E204272756E6F202002
        </GID>
        <MSID>
                urn:jxta:uuid- DEADBEEFDEAFBABAFEEDBABE000000010406
        </MSID>
        <Name>
                TestGroup
        </Name>
        <Desc>
                Peer group using Password Authentication
        </Desc>
        <SVC>
                <MCID>
                        urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000505
                </MSID>
                <Parm>
                        <login>
                                Peer1:FHZR
                        </login>
                </Parm>
        </SVC>
</jxta:PGA>
```

Figure 10-1. The group advertisement

```
<?xml version="1.0"?>
<!DOCTYPE jxta:MIA>
<jxta:MIA xmlns:jxta="http://jxta.org">
        <MSID>
                urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000010406
        </MSID>
        <Desc>
                General Purpose Peer Group Implementation
        </Desc>
        <Comp>
                <Efmt>
                        JDK1.4.1
                </Efmt>
                <Bind>
                        V2.0 Ref Impl
                </Bind>
        </Comp>
        <Code>
                net.jxta.impl.peergroup.StdPeerGroup
        </Code>
        <PURI>
                http://www.jxta.org/download/jxta.jar
        </PURI>
        <Prov>
                sun.com
        </Prov>
        <Parm>
                <Svc>
                        <jxta:MIA xmlns:jxta="http://jxta.org">
                                <MSID>
                                        urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000050206
                                </MSID>
                                <Desc>
                                        Module Impl Advertisement for the PasswdMembership
Service
                                </Desc>
                                <Comp>
                                        <Efmt>
                                                JDK1.4.1
                                        </Efmt>
                                        <Bind>
                                                V2.0 Ref Impl
                                        </Bind>
                                </Comp>
                                <Code>
                                        net.jxta.impl.membership.PasswdMembershipService
                                </Code>
                                <PURI>
                                        http://www.jxta.org/download/jxta.jar
                                </PURI>
                                <Prov>
                                        sun.com
                                </Prov>
                        </jxta:MIA>
                </Svc>
                <Svc>
                        <jxta:MIA xmlns:jxta="http://jxta.org">
                                <MSID>
                                        urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000020106
                                </MSID>
                                <Desc>
                                        Reference Implementation of the Resolver service
                                </Desc>
                                <Comp>
                                        <Efmt>
                                                JDK1.4.1
                                        </Efmt>
                                        <Bind>
                                                V2.0 Ref Impl
                                        </Bind>
                                </Comp>
```

```
                <Code>
                        net.jxta.impl.resolver.ResolverServiceImpl
                </Code>
                <PURI>
                        http://www.jxta.org/download/jxta.jar
                </PURI>
                <Prov>
                        sun.com
                </Prov>
        </jxta:MIA>
</Svc>
<Svc>
        <jxta:MIA xmlns:jxta="http://jxta.org">
                <MSID>
                        urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000060106
                </MSID>
                <Desc>
                        Reference Implementation of the Rendezvous service
                </Desc>
                <Comp>
                        <Efmt>
                                JDK1.4.1
                        </Efmt>
                        <Bind>
                                V2.0 Ref Impl
                        </Bind>
                </Comp>
                <Code>
                        net.jxta.impl.rendezvous.RendezVousServiceImpl
                </Code>
                <PURI>
                        http://www.jxta.org/download/jxta.jar
                </PURI>
                <Prov>
                        sun.com
                </Prov>
        </jxta:MIA>
</Svc>
<Svc>
        <jxta:MIA xmlns:jxta="http://jxta.org">
                <MSID>
                        urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000030106
                </MSID>
                <Desc>
                        Reference Implementation of the Discovery service
                </Desc>
                <Comp>
                        <Efmt>
                                JDK1.4.1
                        </Efmt>
                        <Bind>
                                V2.0 Ref Impl
                        </Bind>
                </Comp>
                <Code>
                        net.jxta.impl.discovery.DiscoveryServiceImpl
                </Code>
                <PURI>
                        http://www.jxta.org/download/jxta.jar
                </PURI>
                <Prov>
                        sun.com
                </Prov>
        </jxta:MIA>
</Svc>
<Svc>
        <jxta:MIA xmlns:jxta="http://jxta.org">
                <MSID>
                        urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000040106
                </MSID>
                <Desc>
                        Reference Implementation of the Pipe service
                </Desc>
                <Comp>
                        <Efmt>
```

```
                        JDK1.4.1
                    </Efmt>
                    <Bind>
                        V2.0 Ref Impl
                    </Bind>
                </Comp>
                <Code>
                    net.jxta.impl.pipe.PipeServiceImpl
                </Code>
                <PURI>
                    http://www.jxta.org/download/jxta.jar
                </PURI>
                <Prov>
                    sun.com
                </Prov>
        </jxta:MIA>
</Svc>
<Svc>
        <jxta:MIA xmlns:jxta="http://jxta.org">
                <MSID>
                    urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000100106
                </MSID>
                <Desc>
                    Reference Implementation of the Always Access service
                </Desc>
                <Comp>
                    <Efmt>
                        JDK1.4.1
                    </Efmt>
                    <Bind>
                        V2.0 Ref Impl
                    </Bind>
                </Comp>
                <Code>
                    net.jxta.impl.access.always.AlwaysAccessService
                </Code>
                <PURI>
                    http://www.jxta.org/download/jxta.jar
                </PURI>
                <Prov>
                    sun.com
                </Prov>
        </jxta:MIA>
</Svc>
<Svc>
        <jxta:MIA xmlns:jxta="http://jxta.org">
                <MSID>
                    urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000080106
                </MSID>
                <Desc>
                    Reference Implementation of the Endpoint service
                </Desc>
                <Comp>
                    <Efmt>
                        JDK1.4.1
                    </Efmt>
                    <Bind>
                        V2.0 Ref Impl
                    </Bind>
                </Comp>
                <Code>
                    net.jxta.impl.endpoint.EndpointServiceImpl
                </Code>
                <PURI>
                    http://www.jxta.org/download/jxta.jar
                </PURI>
                <Prov>
                    sun.com
                </Prov>
        </jxta:MIA>
</Svc>
<Svc>
        <jxta:MIA xmlns:jxta="http://jxta.org">
                <MSID>
                    urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000070106
```

```
                        </MSID>
                        <Desc>
                                Reference Implementation of the Peerinfo service
                        </Desc>
                        <Comp>
                                <Efmt>
                                        JDK1.4.1
                                </Efmt>
                                <Bind>
                                        V2.0 Ref Impl
                                </Bind>
                        </Comp>
                        <Code>
                                net.jxta.impl.peer.PeerInfoServiceImpl
                        </Code>
                        <PURI>
                                http://www.jxta.org/download/jxta.jar
                        </PURI>
                        <Prov>
                                sun.com
                        </Prov>
                </jxta:MIA>
        </Svc>
        <App>
                <jxta:MIA xmlns:jxta="http://jxta.org">
                        <MSID>
                                urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000C0206
                        </MSID>
                        <Desc>
                                JXTA Shell Reference Implementation
                        </Desc>
                        <Comp>
                                <Efmt>
                                        JDK1.4.1
                                </Efmt>
                                <Bind>
                                        V2.0 Ref Impl
                                </Bind>
                        </Comp>
                        <Code>
                                net.jxta.impl.shell.bin.Shell.Shell
                        </Code>
                        <PURI>
                                http://www.jxta.org/download/jxta.jar
                        </PURI>
                        <Prov>
                                sun.com
                        </Prov>
                </jxta:MIA>
        </App>
    </Parm>
</jxta:MIA>
```

Figure 10-2. The module implementation advertisement.