PROJECT “BABEL”

UNIVERSITY OF BATH
COMPUTER SCIENCE

FINAL YEAR PROJECT

DALE LANE

Project “Babel”

http://students.bath.ac.uk/ma8cdl/project

“Develop a peer-to-peer network of wireless devices, and demonstrate using a real-time system to allow collaboration and communication”

DALE LANE
ma8cdl@bath.ac.uk

PROJECT DETAILS

ABSTRACT

Current networking approaches for mobile devices often follow a client / server approach – the traditional and most common way of networking computers. This approach was first designed many years ago, and reflected the use of networking at the time. It was the age of the mainframe, when the high cost of memory and processors meant that computing resources were stored in a single super-computer.

However, times have changed. The mainframe is no longer the main information store - we are. Each of us carries a wealth of electronic information – from calendars, and contact details to business files, carried on a variety of mobile devices. Sharing this information with colleagues and friends is vital if we are to be productive and make best use of this mobile information. As a result, the client-server approach breaks down.

The solution is **mobile peer-to-peer networking**: a technique that allows mobile device users to exchange information, communicate and collaborate on their work; a technique that better reflects our new use of technology.

This project is an attempt to demonstrate this argument by building a peer-to-peer network of wireless devices, and a suite of applications that shows this exchanging of information between peers.

ADMINISTRATIVE DETAILS

Student Name: Dale Lane (ma8cdl@bath.ac.uk)

Supervisor Name: Eamonn O'Neill (eamonn@cs.bath.ac.uk)

Project Timescale: September 2001 – April 2002

Completion Date: 6 May 2002

Related URL: http://students.bath.ac.uk/ma8cdl/project

http://www.dalelane.co.uk/ *(mirror)*

CONTENTS

PROJECT DETAILS 1

CONTENTS.. 2

1. INTRODUCTION 7

2. BACKGROUND 12

3. SPECIFICATION AND DESIGN.......................... 20

4. DEVELOPMENT AND IMPLEMENTATION...... 47

5. RESULTS AND EVALUATION 70

6. FURTHER WORK.. 90

7. CONCLUSIONS 107

8. REFERENCES 110

APPENDICES 114

TABLE OF CONTENTS

PROJECT DETAILS 1

ABSTRACT 1
ADMINISTRATIVE DETAILS 1

CONTENTS 2

TABLE OF CONTENTS 3
TABLE OF FIGURES 5
ABBREVIATIONS USED 6

1. INTRODUCTION 7

1.1 OVERVIEW 7
1.2 GOALS 7
1.3 REQUIREMENTS 8
1.4 MOTIVATIONS 9
1.5 DOCUMENTATION 11

2. BACKGROUND 12

2.1 OVERVIEW 12
2.2 CONTEXT 12
2.3 LANGUAGES AND TECHNIQUES 13
2.4 METHODOLOGY 16
2.5 STANDARDS 19

3. SPECIFICATION AND DESIGN 20

3.1 OVERVIEW 20
3.2 APPLICATION LAYER 20
3.3 APPLICATION LAYER – THE CHATROOM 21
3.4 SERVICES LAYER 27
3.5 APPLICATION LAYER – THE WHITEBOARD 35
3.6 APPLICATION LAYER – FILE TRANSFER 37
3.7 CORE LAYER 40
3.8 USER INTERFACES 40

4. DEVELOPMENT AND IMPLEMENTATION.. 47

4.1 OVERVIEW 47
4.2 CORE LAYER 47
4.3 SERVICES LAYER 48
4.4 APPLICATION LAYER 58
4.5 USE OF TOOLS 64
4.6 OTHER PROBLEMS 66
4.7 CONCLUSION 67

5. RESULTS AND EVALUATION 70

5.1 OVERVIEW 70
5.2 TESTING 70
5.3 EVALUTATION 72
5.4 RESULTS 89

6. FURTHER WORK 90

6.1 IMPROVING JXTA 90
6.2 FURTHER WORK 103

7. CONCLUSIONS 107

7.1 OVERVIEW 107
7.2 ANALYSIS 107
7.3 MAIN RESULTS 108
7.4 SUMMARY 109

8. REFERENCES 110

8.1 OVERVIEW 110
8.2 BIBLIOGRAPHY 110

APPENDICES 114

TABLE OF FIGURES

Figure 1 - Project Layer Architecture 17

Figure 2 - Project Development Approach 17

Figure 3 - Overall chat room application design - showing main operations 26

Figure 3 - Initial system architecture design for the services layer 35

Figure 4 - Initial File Transfer Application design 38

Figure 4 - Section of interface design - used in all applications 42

Figure 5 - Code Excerpt: attempt1/babel/src/babelservices.java 48

Figure 6 - XML files produced for debugging - attempt4/chatapp/useradvert.xml 49

Figure 7 - Prototype design – an initial sketch 50

Figure 8 - Screenshot of the prototype running under MS Windows 51

Figure 9 - Chat application interface running on PJEE under MS Windows 59

Figure 10 - Implementation of IGNORE function in the chat-room 60

Figure 11 - Final Test Case table 71

Figure 12 - Data Transfer Accuracy Evaluation Results (low network load) 73

Figure 13 - Message round-trip timed test diagram 75

Figure 14 - Message round trip times (low network load) 76

Figure 15 – Data Transfer Time between two peers (low network load) 78

Figure 16 - Effect of message packet size on data transfer rate 78

Figure 17 - File Transfer Application transfer times 79

Figure 18 - File Transfer application interface screenshot 83

Figure 19 - Existing use of pipes in JXTA 90

Figure 20 - Existing connection protocol 94

ABBREVIATIONS USED

API	Application Programming Interface – describes the interface between the application and the lower services layer.
AWT	Abstract Windowing Toolkit – a Java interface library for producing graphical user interfaces
CLDC	Connected Limited Device Configuration – Java API and virtual machine meant for resource-limited mobile devices
FTP	File Transfer Protocol – common Internet application protocol for transferring files
GUI	Graphical User Interface – describes the interface between the application and the user.
J2ME	Java 2 Platform Micro Edition – configuration of the Java programming language aimed for limited-resource devices such as smart mobile phones.
J2SE	Java 2 Standard Edition – standard configuration of the Java programming language.
JXTA	(Stands for Juxtapose) Started as a research project at Sun. Provides basic building blocks and services for construction of peer-to-peer networks.
LAN	Local Area Network – computers that share a common communications line
MIDP	Mobile Information Device Profile – J2ME profile for mobile devices
PDA	Personal Digital Assistant – portable hand-held devices that provide computing and information storage and retrieval.
PJAE	PersonalJava Application Environment – Java application environment for devices such as high-end phones and PDAs rather than desktop computers.
PJEE	PersonalJava Emulation Environment – a software test tool that allows testing of applications against a desktop-based implementation of the PJAE.
P2P	Peer-to-Peer – a network where devices do not rely on a server for communicating, and all nodes are equal[1].
TTL	Time To Live – used in networking to specify when something should expire.
UML	Unified Modelling Language – a standard notation for modelling as a first step in developing an object-oriented design methodology
WLAN	Wireless LAN – allows a mobile device to connect to a LAN via a radio connection

[1] A discussion of “peer-to-peer” is contained in the **Background** section.

1. INTRODUCTION

1.1 OVERVIEW

This section provides an introduction for the project, describing the motivation behind it and outlining the overall project goals. It also provides an analysis of the overall project requirements, and details the documentation structure for this report and the attached appendices.

1.2 GOALS

> *This project aims to demonstrate that collaborative network applications are a practical and viable way for users to work together and add value to their use of mobile computing devices such as Personal Digital Assistants (PDAs).*

The most common approach to networking today, even networking of the latest developments in computing – mobile computers – is to network machines using a client-server topology.

As I will discuss in detail in **Background** (2.2.1), this was a result of tradition arising from the fact that it was the most appropriate approach when it was first developed. It is appropriate where the server is the primary information store and processing centre for the network or for the coordination of large-scale networks, but this does not cover all scenarios where networking is used.

I believe that **peer-to-peer**[2] is a more appropriate approach to networking where data is carried around on mobile devices. On mobile devices, data tends to be accessed and updated frequently, and copying data a central server encourages the use of out-of-date and incorrect data.

> *This project aims to demonstrate that peer-to-peer networking is a viable solution for the networking of mobile computing devices such as Personal Digital Assistants (PDAs).*

I believe that **wireless networking** is a practical and useful approach to networking small devices. The mobile nature of PDAs are another reason why **peer-to-peer** is the ideal approach. Using a client-server approach to wirelessly network mobile computers means that the users can only communicate or collaborate when they are within range of a server. This is acceptable while users are in their office or other such place of work, but would be very restrictive when the users took their mobile devices with them to a customer site or other work in the field.

[2] A detailed explanation of the term "peer-to-peer" can be found in the **Background** section.

Relying on a central server to coordinate and control communication and collaboration means that users are forced to work independently while away from their central place of work – which is when it would be most useful for users to work together. **Peer-to-peer** networking does not impose such restrictions, and allows users to continue working together wherever they are.

> *This project aims to demonstrate that true peer-to-peer networking can reliably support chat rooms and other collaborative work applications, without any need for a central server.*

Famous "peer-to-peer" applications such as the ill-fated Napster file-sharing program often have to rely on servers for some aspect of control or coordination. Applications such as instant messagers have traditionally relied on servers. Even those applications which boast of providing direct connections between chat users tend to rely on servers to maintain a list of who is online at any given time. Users login to a server to inform the server of their presence. The server can then inform them which of their friends are currently online, and provide them with the connection information necessary to initiate communication with them.

The use of a server in such cases can be a way of avoiding the challenges involved in creating a peer-to-peer network, and sometimes a necessity brought about by the size or scale of the network or nature of the data involved. However, in implementing a network to support collaboration between users of mobile, small-scale computing devices, I believe that the use of a server would introduce restrictions and limitations that cannot be justified by convenience of implementation.

The project aimed to demonstrates the benefits of peer-to-peer networking, and highlights the differences between it and client/server techniques. I explored the implications of mobile networking and how mobile device users can effectively exchange information, communicate and collaborate.

1.3 REQUIREMENTS

The principal requirement was that the system must support the flexible creation of groups of mobile devices, allowing for communication and file-sharing between them.

My vision was of a flexible network of PDAs or other mobile devices – for example a group of students in a large lecture room communicating without a direct physical connection or cumbersome link, or a group of people in an office working together and collaborating freely and easily without resorting to using PCs or laptops. This would enable effective group work by allowing information to be accessed and messages sent to this unit as a whole. It would also allow individuals to communicate, allowing for collaboration that is more private.

To achieve this vision, the project had the following overall requirements – to create a peer-to-peer network of mobile devices, and create network applications that show this exchanging of information between devices. The applications must demonstrate the network in operation, allowing wireless device users to exchange information, communicate and collaborate on their work.

The project required me to create the services necessary for the generation of the network, and to build a real network. It included the construction of a small suite of applications to demonstrate the network. Deciding on which applications to create is discussed in more detail in **Specification**

and Design (3.2.1), however it included both considerations of what applications would best demonstrate the network and what applications would best meet the needs for group work and communication. The applications decided upon were as follows:

- Chat program – instant messaging between groups or individuals or mobile devices
- File swapping – exchanging of documents and files between wireless users
- Collaborative whiteboard – allowing collaborative work on a shared free sketch-pad

To support the goals identified above, the project requirements also included creating a network capable of supporting a system that can understand availability and be able to communicate to each individual. It needed to enable users to place communications on other individual devices and be able to share and retrieve information. It had to allow instant messaging and file sharing. The project had to support unequal devices, and manage the different processing speeds and search capabilities.

It was a requirement that the system supports the formation of ad hoc groups of individuals and allows information to be accessed and messages sent to this unit as a whole. The project tackles the problem of connecting individuals and devices in an effective manner, and provides a way for mobile users to communicate and exchange information.

1.4 MOTIVATIONS

1.4.1 INTRODUCTION

In this section, I discuss my motivations for choosing this as my Project. I identify each of the main areas of study, explaining why it warranted investigation and what I hoped to gain from it.

1.4.2 WHY NETWORKING?

Networking has become the underpinning for much of our economy and is a critical foundation of computer science. It enhances everyone's ability to communicate, giving people throughout the world unprecedented access to information. In software development these days, networks are all-important. Sun Microsystems uses the slogan, "the network is the computer" and IBM promotes a network-based business model called "e-business." Nowadays, it is often assumed that programs communicate across networks. As a result, networking is an ideal area of further academic study.

This project helped me to learn about the benefits of different approaches than the traditional client/server technique and give me an insight into development of network applications and protocols. It provided the opportunity to learn a lot and gain an insight into networking and in particular mobile networking and peer-to-peer networking techniques.

1.4.3 WHY MOBILE DEVICES LIKE PDAS?

One area of networking that seems to be of increasing interest in recent years is mobile networking, with the introduction of devices like the Palm Pilot and other such Personal Digital Assistants (PDAs). I decided to explore this for my project, and look at the some of the issues and implications associated with developing mobile applications.

As well as introducing technological limitations and considerations such as speed, processing power and bandwidth, the nature of the hardware also introduces specific user interface requirements

that are quite different to developing for PCs and other desktop machines. Differences such as the absence of a traditional keyboard, the use of stylus devices instead of a mouse, and the limited screen size all have big implications for the design and implementation of applications.

1.4.4 WHY WIRELESS NETWORKING?

One of the first things that I did while considering what I could do for this project was look at the current mainstream use of networking in PDAs.

Many PDAs currently available provide an infrared port allowing a direct link between two similar devices[3]. This supports such activities as file swapping – allowing address book entries and other such small files to be exchanged between users quickly and easily.

My research of usage of PDAs amongst my peer-group suggested that this was the dominant use of networking. Restricting users to exchanging contact details or small files does not really encourage collaborative activities. The limited bandwidth of an infrared connection means that to exchange larger files users will either use e-mail (requiring a PDA modem) or using PCs instead. In this way, the potential for collaboration and cooperation between mobile device users is largely lost – the requirement of a direct line of sight between devices and need for close proximity is cumbersome, and the restriction of only connecting two users at a time is limiting.

My research into devices and technologies currently available did reveal alternatives. Wireless LAN (WLAN) allows laptop users to access a LAN without wires. This brought the benefits of PC networking to the laptop, including communications, cooperative work and file sharing.

The IEEE standard[4] adapts well to PDAs and other wireless devices, and enables the creation of wireless networks of PDAs. With a WLAN module, a Palm Pilot can use the same standard, enabling interaction with not only other PDAs, but also other devices such as a laptop, PC or server.

This introduces potential for huge increases in productivity. Networking mobile and non-mobile devices makes collaboration a real possibility, and this project demonstrates this potential.

1.4.5 WHY COLLABORATIVE IDEASTORMING APPLICATIONS?

Currently, the main options for remote design or idea-storming collaboration are videoconferencing (which is very expensive and awkward) or applications such as Microsoft's NetMeeting (which is not very open or extensible). Neither of these is easily mobile, and both are quite cumbersome. I believe that there is a gap for applications such as mobile collaborative whiteboards and other communications opportunities, and I wanted to explore this.

1.4.6 WHY PEER-TO-PEER NETWORKING?

The functional requirements of the project were the first to be identified – a project that would tackle communication problems and facilitate collaborative work. From my vision described above, one of the first decisions that I had to make was the basic network architecture that I would use.

---

[3] For example, Palm include it as standard on all their handhelds – cf. http://www.palm.com/products/what.html

[4] Current IEEE wireless LAN standard is 802.11b. More details can be found at http://grouper.ieee.org/groups/802/11/
The IEEE standard is not the only WLAN available. Another example is Bluetooth - http://www.bluetooth.com/

After researching the different possibilities available to me, I decided to use a peer-to-peer (P2P) paradigm. Exploring the differences between this and more traditional topologies became one of the key project goals as described above, but I chose this over the more traditional client-server approach because it was the best solution to the identified problem.

It also provided an opportunity to learn and explore a different type of networking to anything that I had learnt about or worked with before – and to explore the differences and their implications.

1.5 DOCUMENTATION

1.5.1 DOCUMENTATION OVERVIEW

This report describes work carried out for my project. It contains the following main sections:

- **Introduction** – introduces the project with an overview of the aims and objectives
- **Background** – puts the project into context, and introduces some key ideas
- **Specification and Design** – explains what the software is required to do, and gives a high-level design of how it does it
- **Implementation** – explains in detail how the software was created
- **Results and Evaluation** – discusses to what extent the project goals were achieved
- **Further Work –** how the project was extended, and what future work is possible
- **Conclusions –** summarises the main achievements of the project

As discussed in **Background** (2.4.3), I followed a formal software development lifecycle. As a result, I produced a number of requirements analysis and design documents. These are attached to in the **Appendix** and referenced in the report body where relevant.

These documents and all code listings are also attached on a CD-ROM to allow for searching. The CD also contains videos showing examples of testing and GUI operation.

1.5.2 SCOPE

This project is an educational exercise, intended to increase my knowledge and understanding of the areas identified in the goals (1.2), and improve my software engineering. There is no intention to produce this project as a commercial end-user product, and as such, there are no security requirements, or other such commercial concerns.

The focus of this report itself is therefore focused on these goals – more on the method of the development and the learning process, rather than defining the produced software. As such, it is written in quite a narrative rather than technical or scientific style – intended to document and justify the actions carried out, some of the problems encountered and how they were approached.

2. BACKGROUND

2.1 OVERVIEW

This section intends to provide some of the background information about the project area, introducing some of the key ideas and areas that the reader may not be familiar with.

It describes the tools, standards, languages, and techniques that are used in the project, together with a rationale for their use where appropriate. It also describes the methodology used, and identifies the key developmental stages.

2.2 CONTEXT

2.2.1 PEER-TO-PEER NETWORKING

As discussed previously in the **Introduction** (1.4.6), very early in the project I identified that a peer-to-peer networking approach would be the ideal way to implement the sort of system and network that the project required. In this subsection, I will discuss why I believe it is better suited, and why I did not go with the more traditional client-server approach.

Traditional client-server approaches reflect the way we accessed information when computers were introduced – the age of the mainframe, when the cost of memory and processors meant that resources were stored in super-computers, and people had "dumb" workstations on their desk.

Linking all devices through a server grew from the traditional use of computers where useful information was stored in a central location that everyone could access. In the days when all useful information was stored on a mainframe, and a central server provided the bulk of the processing and file-storage, this was an effective solution.

However, times have changed. The growth of the PC meant that processing power came to people's desks. PCs no longer need to rely on a central server for processing or information. This growth continued and took a step further – entering people's pockets and briefcases with the advent of laptops, PDAs and mobile phones. Mobile technology means that people can take processing power with them – no longer restricted by requiring access to a central server or mainframe.

The mainframe is no longer the main information store – we are. Each of us carries a wealth of information – from calendars, diaries, addresses and phone numbers to business documents and files, carried on a variety of mobile devices. Sharing this information with colleagues, customers and friends is vital if we are to be productive and make best use of this mobile information.

Changes in technology and its use are driving the development of networking. The fact that useful information is now carried with people on mobile devices means that there are potentially huge benefits from being able to network and share information between mobile devices.

Relying on networking between PCs and workstations when the useful information is on mobile devices is slow and ineffective, introducing delays and reducing productivity. While tried and tested,

client-server approaches are not necessarily the ideal solution for every problem. As information and processing power is more widely distributed, to be shared efficiently a different approach is required.

The P2P paradigm is a more communal approach. To consider an analogy, imagine a group of students with common ideas about living together and sharing certain belongings. We will have shared rooms (such as the kitchen) where we put everything that we are willing to share, but we will also have our own individual rooms where we store our private belongings.

This more flexible approach is much like the P2P ideal. Each user can specify which information they want to make public, and can access what others have made public. So if a file or piece of data is available on any of the hard drives of the connected devices, you can find it without needing them to have placed a copy on the central server.

As well as offering greater convenience and access to a wider selection of files, copies of files on peer's devices are likely to be more up-to-date than copies left on servers. In this way, users can work with and share the most current information.

The analogy extends to services and other resources, too. In a student house, we may have resources that we make available to others – for example, the microwave I put in the kitchen for all to use. However, some resources I will want to keep private – such as the hi-fi in my bedroom.

In the same way, the P2P paradigm allows distributed use of a variety of services and resources. As new devices enter the network, the capabilities of the network are increased – without being restricted to the capabilities of a central server.

I decided that this was an ideal solution for this project – in this way, users could communicate to other individuals without having to pass through a central location. Without relying solely on the centralised control, it tackles the problem of connecting individuals and devices in a cheap and effective manner, providing an extensible and flexible network.

Users can enter and leave networks as easily as walking into a room, instantly having access to a wide variety of information from people that they work with, without being tied to requiring proximity to a suitable server to work together.

2.3 LANGUAGES AND TECHNIQUES

2.3.1 NETWORK PROTOCOLS

During my research into peer-to-peer networking, I found numerous references to **Project JXTA**[5], a project being developed by Sun.

[5] Full details about Project JXTA can be found at http://www.jxta.org/

> "Different protocols, different architectures, different implementations. That accurately describes current P2P solutions. Currently, developers use diverse methodologies and approaches to create P2P applications. Standards, abundant in the client/server world, are noticeably absent in the P2P world. To tackle this deficit, Sun developed JXTA.
>
> From the Jxta vision statement:
>
> Project Jxta is building core network computing technology to provide a set of simple, small, and flexible mechanisms that can support P2P computing on any platform, anywhere, and at any time. The project is first generalizing P2P functionality and then building core technology that addresses today's limitations on P2P computing. The focus is on creating basic mechanisms and leaving policy choices to application developers.
>
> Jxta strives to provide a base P2P infrastructure over which other P2P applications can be built. This base consists of a set of protocols that are language independent, platform independent, and network agnostic (that is, they do not assume anything about the underlying network). These protocols address the bare necessities for building generic P2P applications. Designed to be simple with low overhead, the protocols target, to quote the Jxta vision statement, 'every device with a digital heartbeat.' "

http://www.javaworld.com/javaworld/jw-10-2001/jw-1019-jxta_p.html

JXTA seemed to be an ideal foundation upon which to build my project. From a design point of view, it is more efficient to build on an existing set of protocols, rather than creating my own. My aim was not to use the protocols as is, but to edit them as required, using them as a core set.

From an educational point of view, I thought that I would learn more by adapting existing well-developed protocols, rather than trying to create new ones. Given the timescale of the project, there was not the time to design the system sufficiently to know enough of what would be required from all of the protocols to be able to write them all myself. Using an existing set allowed me to learn a lot about protocols whilst still giving me time to learn about other network layers.

In addition, some argue that Sun is making JXTA a standard for P2P development, making it a very useful choice of technology to gain experience with.

2.3.2 PROGRAMMING LANGUAGE

2.3.2.1 JAVA

The aim of the project was to implement the application to run on a high-end PDA. During my initial research, I identified **Java** as an ideal implementation language for many reasons, including the fact that it is well suited to development with Project JXTA. It is well suited to the object-oriented approach to requirements analysis and design that I decided upon for the project, as discussed below.

Another reason for choosing Java initially was Java 2 Platform, Micro Edition (**J2ME**) version, which allows Java applications to run on small-scale, reduced memory footprint platforms.

J2ME is based on configurations and profiles. A configuration defines the minimum set of class libraries available for a range of devices. A profile defines the set of APIs available for a particular family of devices. By developing applications to a specific profile, you can aim the code to specific hardware capabilities and know what API set to expect.

One profile is the Mobile Information Device Profile (MIDP) based on the Connected Limited Device Configuration (CLDC). In my initial design and requirements work, I identified this as being the profile to which I would work. I stated that a requirement of the final code was that it would be MIDP/CLDC-compliant – thereby limiting the Java functions, methods and data-types that I could use to those specified in the Basic J2ME and MIDP APIs.

However, after further research and after attempting initial development using J2ME, I reconsidered this decision. MIDP is meant for smart pagers, mobile phones and small devices with small LCD screens and limited input entry methods (such as a mobile phones with simple keypads). Also, CLDC has very limited Java technology, with no support for floating point, no on-board bytecode verification, no Java Native Interface, no Abstract Windowing Toolkit (AWT), and no advanced Java 2 technology features (such as reflection and security).

This is not appropriate for developing an application for a high-end PDA, and my attempts to use it showed it to be too restrictive to satisfy the requirements detailed in the **Introduction** (1.3).

2.3.2.2 PERSONALJAVA / THE PERSONAL PROFILE

In contrast, the J2ME Personal Profile, based on the Connected Device Configuration (CDC), has all those missing features. J2ME CDC is fully compliant with the Java 2 VM specification, which makes it easier to port code written for standard Java (J2SE). It is a more appropriate choice – meant for higher end devices such as high-end PDAs.

However, the Personal Profile is relatively new, and hardware support is still quite limited. The predecessor of the Personal Profile was called PersonalJava. PersonalJava was announced before J2ME, and is therefore not a profile in the traditional sense. However, it shares many of the same aims of the Personal Profile, defining a subset of the core Java APIs for use within reduced memory footprint devices. It includes most of the AWT component set and support for network connectivity.

The increased availability of PersonalJava led me to identify this as the best choice.

2.3.3 TOOLS

The following were some of the tools were used during the development and testing of this project:

- Rational Rose[6] – used during the requirements analysis and design stages to produce UML documentation of requirements and design decisions.
- PersonalJava emulator[7] – used during implementation to aid development. The rationale for using this tool is discussed in **Implementation** (4.5.1.2).

[6] More details about Rational Rose can be found at http://www.rational.com/products/rose/

[7] Sun's PersonalJava Emulation Environment is available for download from http://java.sun.com/products/personaljava/

- JavaCheck[8] – a development tool for testing classes for compatibility with particular Java API's. Logs produced by the JavaCheck tool allowed me to identify the areas of code where I used classes or methods not supported by PersonalJava.

- log4j[9] – a tool used to control log output generated by my code. The rationale for using this tool is discussed in **Implementation** (4.5.1.3).

- Java Development Kit – all development and initial testing was carried out on Linux and MS Windows desktop machines, using Java 2 SDK v.1.3.1

- JXTA Shell[10] – provides command-line UNIX shell-type interface to JXTA platform, used during early development as a test aid. The rationale for using this tool is discussed in **Implementation** (4.5.1.4).

- tcpdump – used during optimisation to examine network behaviour when running Services layer under heavy load. The rationale for using this tool is discussed in **Implementation** (4.5.1.5).

- Other – a variety of other tools and utilities including make and Windows Performance Viewer (part of MS Management Console)

2.4 METHODOLOGY

2.4.1 DESIGN APPROACH

This project could clearly have become very large if I had attempted to implement it all from scratch. In the timescale available to me, it would not have been practical to design and write a full set of network protocols and design and build a suite of applications to use them.

One option I considered was to focus on a single area – either creating an application based on existing protocols or to focus on creating a full set of network protocols.

My decision regarding this was largely driven by academic and educational reasons: I wanted to learn about developing all the layers of a network – not just the protocols or just an application. While this reduces the scope for work in each area, I learnt more from such an overview – giving me a chance to work in several areas. This served better to help me accomplish one of the objectives of the project – to learn more about the implications of mobile networking. A more detailed discussion in what I gained from this project is included in the **Conclusion** (7.3), later in this report.

My approach to this project was to divide the implementation of the network into three layers:

---

[8] Sun's JavaCheck tool is available for download from http://java.sun.com/products/personaljava/javacheck.html

[9] log4j is available for download from http://jakarta.apache.org/log4j/

[10] JXTA Shell can be obtained from http://shell.jxta.org/

The **core layer** contains the network protocols. The P2P protocols are defined and created here.

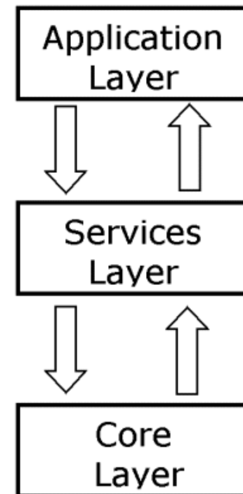


Figure 1 - Project Layer Architecture

The **services layer** contains common P2P functionality. These are generic functions that can be used by any application. Effectively, these are a higher-level API.

The **applications layer** – applications access services located in the services layer, which provide interfaces for the network protocols in the core layer.

Dividing the overall system into layers like this had a number of benefits, including allowing the possibility of reusing functions and protocols between applications. This modularization also aided the development, by breaking down a large task into smaller, more manageable sections.

I designed the system one layer at a time, starting with the Application Layer. From this design, I was able to identify what services I needed in order to implement it, and designed the necessary services accordingly. Designing the services in turn told me what protocols I needed to implement them.

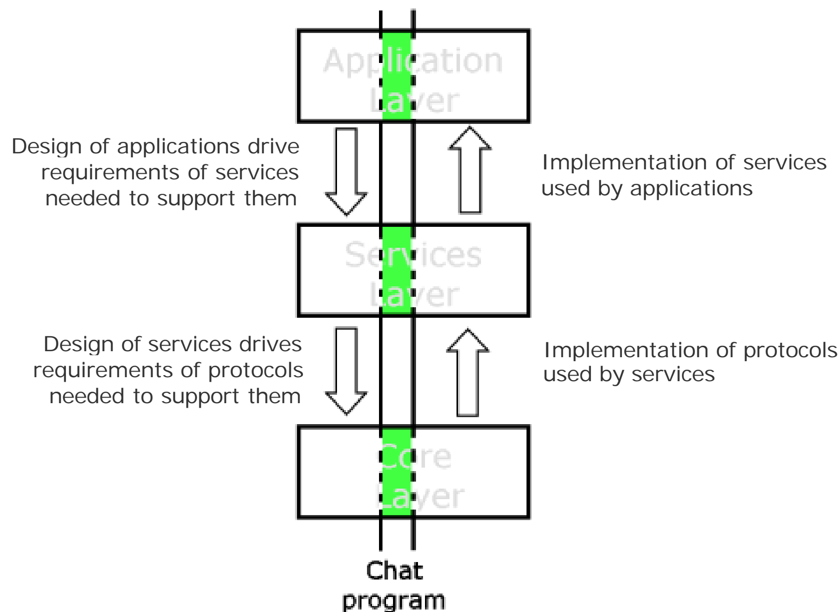


Figure 2 - Project Development Approach

In this way, I was not attempting to create a complete set of P2P network protocols and services. Instead, I designed and implemented as many protocols as were needed to implement the services. Likewise, I only created enough services to be able to implement my application. This allowed me to carry out design and development work in all three layers. Visually, each application can be described as taking a thin slice through each of the three layers as shown in the diagram above.

Once the first application was complete, I then began designing the second. From there, I decided what additions were needed to make to the services to implement it. In turn, I then knew what additions I needed to make to the protocol set. To return to my visual metaphor - I took another 'slice'.[11]

2.4.2 PROJECT DELIVERABLES

This approach therefore left me with the following deliverables, which were used as milestones for the project. The process was repeated for each application.

- **Application Requirements** – Specifies the requirements of a Chat Application
- **Application Design** – High Level Design for Chat Application, including GUI design specification
- **Services Requirements** – Specifies the requirements of the services required to implement the Application Design
- **Services Design** – High Level Design for Services required for Chat program, including interface specifications for all services
- **Protocols Requirements** – Specifies the requirements of the protocols required to implement the Services Design
- **Protocols Design** – Matches up Protocols Requirements with protocols available in the JXTA set, and details required extensions / modifications
- **Protocols Implementation** – Low Level Design and Coding of the Core Layer
- **Services Implementation** – Low Level Design and Coding of the Services Layer
- **Application Implementation** – Low Level Design and Coding of the Application

2.4.3 DEVELOPMENT APPROACH

As suggested by the deliverables and milestones detailed above, I decided to follow a formal waterfall development lifecycle for this project. As this is the first large-scale project I have undertaken, I thought that following a formal development process would help to direct my work and manage my time efficiently, as well as making it easier to demonstrate work carried out.

I also kept an online logbook to document my progress. This can be found on the attached CD-ROM and contains a chronological description of work carried out and decisions made, with the rationales behind them. Again, this served to help organise my efforts, and provide a reference source from which this report was written.

I decided to design using an object-oriented approach. This is well suited to implementing in Java – as classes identified can be mapped to Java classes. It also fits in well with a communications-based system, allowing analysis of messaging flow. I had decided to create a services layer that could

[11] Examples of service-layer functions added with later "slices" include: FileHandler required by the file transfer application.

be used by different applications and an object-oriented approach supported a modular design to produce classes that could be reused by different applications.

For most of requirements analysis and design work, I used UML[12] notation to describe decisions. This was because it is a well known, accepted standard and maps well to implementation in Java.

2.4.4 TARGET PLATFORM

The overall project requirements detailed above specify that the project be aimed at a mobile computing device such as a PDA. I did not want to restrict development by naming a specific device. Naming **Palm Pilot** as my intended platform, for example, would bring a number of implications.

Initially, I stated that the target platform would be any mobile computer that was MIDP/CLDC compliant (as discussed in the above section on languages). However as stated above I decided that this would be too restrictive. The final requirement was that it would be any PersonalJava-compatible PDA, which includes file I/O support (optional for PersonalJava but required for my project).

Development was carried out initially in Java 2 Standard Edition (J2SE) on MS Windows and Linux development environments. After initial prototyping, the code was ported to PersonalJava, and development continued using the PersonalJava emulator mentioned above (2.3.3).

Although I did not specify a particular target platform during requirements, during implementation the development was loosely targeted towards the Compaq iPAQ. This was largely due to the practicalities of it's availability, however also because it is a flexible and powerful high-end PDA with a good example of a typical PDA interface.

I did consider developing this project for the Palm computing platform PalmOS, as it is a very successful and widespread mobile computing platform. There would have been some benefits to this approach, because of the wide availability of support, tools and advice for developers.

However, I decided against it because of the specialist development knowledge required to develop with it, when compared to products such as PersonalJava. In addition, Palm devices currently use the Motorola 68328 (Dragonball) processor, which is limited in terms of speed and computation capabilities thus limiting the level of handheld application processing.

2.5 STANDARDS

2.5.1 CODING STANDARD

I defined a coding standard for the project to aid code readability and clarity. The coding standard is an adaptation of the Java standard presented in the *Java Language Specification*, from Sun. A copy of the coding standard used for the project is attached to this report[13].

[12] More information about UML can be found at http://www.rational.com/uml/

[13] Appendix H – “Coding Standards Document”

3. SPECIFICATION AND DESIGN

3.1 OVERVIEW

This section provides a description of the requirements analysis phase, and the design of the software. It includes a rationale for the decisions made, together with some of the alternatives considered and the reasons for not using them.

3.2 APPLICATION LAYER

3.2.1 CHOOSING THE APPLICATIONS

For this project, I identified three applications that I would develop. There are:

- Chat-room / Instant messaging application
- Collaborative whiteboard
- File transfer application

I chose applications that would provide a good cross-section of possible types of data that can be sent and received. The chat-room sends text data, the file transfer application transfers file data, and the whiteboard sends graphical data.

I also wanted three applications that would impose differing strains on the network – with different bandwidth properties. I identified the chat application as being the one that would impose the least strain on the network. The nature of text messages sent through the instant messager applications I encountered during my research showed that they were relatively small. The time taken to write each message means that messages are also sent infrequently.

I identified the whiteboard application as being the application that would impose the greatest strain on the network. The way that messages can be sent so quickly, with a single stroke of the stylus, means that it is possible (and indeed quite likely) that messages will be sent very frequently.

The file transfer application does have the potential for the greatest amount of data being transferred, however this obviously depends on the nature of files chosen by users. However, unlike the other applications, the messages do not need to be sent or received in real-time. For collaborative work and communication, the other two applications need to communicate with as small a delay as possible. This is not the case with a file swap program. Fast transfer is desirable, but not as vital.

The collaborative whiteboard is one of the most interesting applications. By providing a free sketchpad for joint design and idea storming, the potential of group work using networking can be greatly exploited. As mentioned previously, there are currently few other options for such work and developing the whiteboard for this project demonstrates the promise for such applications.

By selecting applications that reflect different amounts and types of data, and use different patterns and frequencies of data transmission, I explored a variety of uses of mobile networks.

As described in the Project Approach explained in **Background** (2.4.1), the first application that I began to develop was the Chat Program. This was chosen because it posed the least strain on the network, which I felt would be helpful during development.

3.3 APPLICATION LAYER – THE CHATROOM

3.3.1 SCOPE OF REQUIREMENTS ANALYSIS

The division of the project into three layers defined in the **Project Approach** (2.4.1) means that at this stage, I only needed to define the application at quite a high level.

I needed to identify key functionality required, but not how it works, as this would be provided in lower layers. The scope of this analysis was therefore limited to a high-level functional view.

3.3.2 CHAT APPLICATION FUNCTIONAL REQUIREMENTS

The Requirements Document deliverable that I created is attached to this report[14]. It contains a detailed definition of all of the requirements that I identified.

A discussion of the rationale behind requirements identified is contained in the following pages.

The first thing that I did was to identify the primary functions of a chat application. I did this in a number of ways, including:

- Researching features of existing chat applications, identifying the positive and negative features of applications already available
- Research of my peers to find what they would want from a chat program, using questionnaires to find their requirements – as they are the end users as identified in my initial **Vision** (1.3)
- Personal experience – what my requirements for a chat application are, based on the initial project vision as described in the **Introduction** (1.3)

A full detail of this research is attached to this report[15]. This was done to ensure that I did not just implement a chat program the way I might like it, or simply duplicate an instant messaging program that I had used before.

Because of this research, I identified the following basic functions as being functional requirements of the chat-room application:

- Join room / Log in
- See list of available / logged-in users

---

[14] Appendix A – "Chat Application: Application Layer Requirements"

[15] Appendix E – "Chat Application: Requirements Investigation"

- Send and receive chat-room messages – messages to all
- Send and receive private messages – messages to individual named user
- Ignore messages from named users
- Transferability
- Graphical user interface
- Performance requirements

These requirements are discussed in detail in the Requirements Document[16], but I will include some of my reasoning for them here.

3.3.2.1 JOIN ROOM

My reasoning for giving "joining the chat room" as a function was not user authentication, (as logging-in is often associated with), because security is not an issue that I considered for this project. Instead, it was intended to be a mechanism to inform other users on the network of the user's arrival.

Much of the functionality for joining is the responsibility of the services layer – this is something that all applications will need to do, so the functionality is better suited to the services layer. The User Interface is responsible for dealing with joining a room – to ensure that the user receives feedback when they try to join the room, however this will be discussed later.

I specified a requirement that the chat application must be able to join specified rooms, rather than a single global room. This is a result of the global requirement of the project where I specified that the system should support the ad hoc formation of groups of individuals. Such an approach better supports the vision discussed in the Project Concept where a group of people can easily start working together – the inflexibility of a single global group would hinder this.

3.3.2.2 SEE LIST OF AVAILABLE USERS

How the application identifies who is in the chat-room (the "discovery" method) is beyond the scope of the application layer. This functionality must be common to almost any network application, and is therefore better suited to a lower layer. However, using this network functionality to maintain a list of current users, and displaying this clearly to the user was a relevant requirement.

3.3.2.3 MESSAGES TO ALL

I wanted to create a chat-room where all users in a room receive messages. This supports the Project Concept of supporting collaboration between groups, rather than pairs, of people.

3.3.2.4 PRIVATE MESSAGES

To maintain flexibility, I also wanted to have the ability to send messages to selected user(s). This would allow the formation of sub-groups within rooms – allowing private communication between some people in the room, while allowing them to see what is being said in the room. It means that

[16] Appendix A – "Chat Application: Application Layer Requirements"

they do not need to create a new room (peer group) for themselves if they want to chat privately, therefore excluding them from the others in the room / group.

3.3.2.5 IGNORE

I did initially consider where ignored messages should be removed. One option considered was to ensure that the Services layer never passes them to the application. An alternative is for applications to check when a received message is passed to it.

My final decision was to include it as an application requirement. I think that it makes more sense to leave *ignore* as an application function, as it allows greater flexibility – allowing the user to ignore a certain individual in one application but not in others, without having to enforce a global ban. For example, the user might want to prevent a certain user being able to send them a file in the file transfer application, while still wanting to receive messages from them in the chat room.

The disadvantage of this approach is that the services layer is still receiving and processing messages from ignored peers. If a user decides to block incoming messages from a particular peer, this is probably because their messages are a nuisance. If they are repeatedly sending messages to the user to attack their performance in a sort of denial-of-service attack, then putting the ignore filter in the application layer will do nothing to combat this. However, security is not a primary concern of this project and the increased usability from putting the ignore functionality in the application layer outweighs the irrelevant risk of planning for malicious users.

3.3.2.6 TRANSFERABILITY

My original thought was that I wanted the application to be able to work on any mobile device – based on the original global requirement of a system that is able to deal with the variety of performance characteristics of different mobile devices. However, I was very aware that this may not be practical, given the timescale – it could have been difficult to implement, and would almost certainly have been difficult to verify.

A compromise that I considered was to select a single platform, and define a requirement that the system should be designed to be able to be ported easily to any other mobile device.

As discussed previously, I made it a requirement for the implementation to be compliant with the PersonalJava specification – a Java API for mobile, resource-limited devices. I decided that this was close enough to my aim of making it a global application.

3.3.2.7 GRAPHICAL USER INTERFACE

The nature of the limited control and displays in mobile devices such as PDAs and mobile phones means that there are important User Interface implications. For example, entering large amounts of text on devices such as these is often quite difficult and/or time-consuming, and so a command-line or text-based interface would not be acceptable.

While this is a considerable topic that warrants investigation by itself, I believed that it needed some consideration if I was to satisfy the project's requirement of exploring the implications of networking mobile devices. User Interfaces and the way people use mobile devices is clearly an important part of this. I will return to this later in the **Design** (3.8) part of this section.

3.3.2.8 PERFORMANCE REQUIREMENTS

I did briefly consider specifying formal performance requirements, perhaps in terms of number of users the system can support at any one time, or reliability of message delivery. However, my lack of experience in this field meant that I was unable to suggest realistic or feasible values. I therefore decided to leave this discussion until the creation of a test plan.

3.3.3 CHAT APPLICATION FUNCTIONAL DESIGN METHODOLOGY

As mentioned previously, I followed an object-oriented approach to project development, using UML notation to record it. A full description of this work is too lengthy to describe here, but is attached to this report as an appendix[17].

To summarise the work, after identifying the application requirements I did the following:

1) Identified **use cases** based on requirements in the requirements definition
2) Identified **classes** by highlighting nouns and noun phrases in the requirements
3) Refined **class list** by considering core class requirements
4) Explored the responsibilities of each possible class using **CRC cards**
5) Used the refined classes to produce a basic **class diagram**
6) Analysed the application in action looking at interactions using **sequence diagrams**
7) Use changes discovered to further refine the **CRC cards**
8) Produced **complex class diagram** to explicitly identify class attributes and operations
9) Defined the role of the User Interface using **activity diagrams** which led to further refinement of the **classes**
10) Finalised each **class** by defining each operation and attribute

With each of these stages, I explored different aspects of the application and its use. This enabled me to continually refine and adapt my original design, as well as highlighting problems with the original requirements document – and led to a number of changes.

This design was based on the application functional requirements. While it included designing functional responsibilities of the user interface, it did not include work on the interface aesthetics.

3.3.4 CHAT APPLICATION DESIGN

The design work described above led to the production of the next identified milestone – the High Level Design, which is attached to this report[18]. However, I will summarise the design in the

[17] Appendix F – "Chat Application: High Level Design Work"

[18] Appendix B – "Chat Application: Application Layer High Level Design"

following pages and discuss some of the main decisions made. It is worth noting that the Design Document attached also addresses non-functional design requirements that are not addressed here.

3.3.4.1 MODES

I decided to create two modes – either logged in or out of a chat-room. These modes provide a simple way to enforce requirements such as only being able to send a message while logged in, or only being able to login while not in a chat-room. They also present feedback to the user, showing them what operations are available and making their current state in the network clear to them.

I designed the interface to switch between these modes, altering both the look (so that the user can tell at a glance if they are logged in) and the functions provided.

3.3.4.2 SCENARIOS

From the requirements analysis described earlier, I identified five scenarios – operations that the application would need to be able to support.

I then designed each of the scenarios in more detail, breaking down these high-level operations into their responsibilities. There is not space in this report to detail all of these, however the application functionality was relatively straightforward – as the services layer provides the network functionality. The design of these operations is described in more detail in the Design document.

As Figure 3 shows, while in the logged in mode, it has three main responsibilities, which are:

- **Send Message**

  The application grabs the message and the intended recipients from the interface, packs it into a network message, and gives it to the services layer for sending.

- **Receive Message**

  The application implements the interface specified by the services layer, able to respond to incoming messages. Most message processing is carried out by the services layer, which provides the application with sender details and the message body. The application is responsible for displaying the message in the interface.

- **Update List**

  The application is responsible for maintaining a list of the users in the same group. This should be displayed to the user and updated regularly. The functionality to find the users in the group is provided by the services layer, but it is the responsibility of the application to use this often enough to create and maintain an up-to-date list.

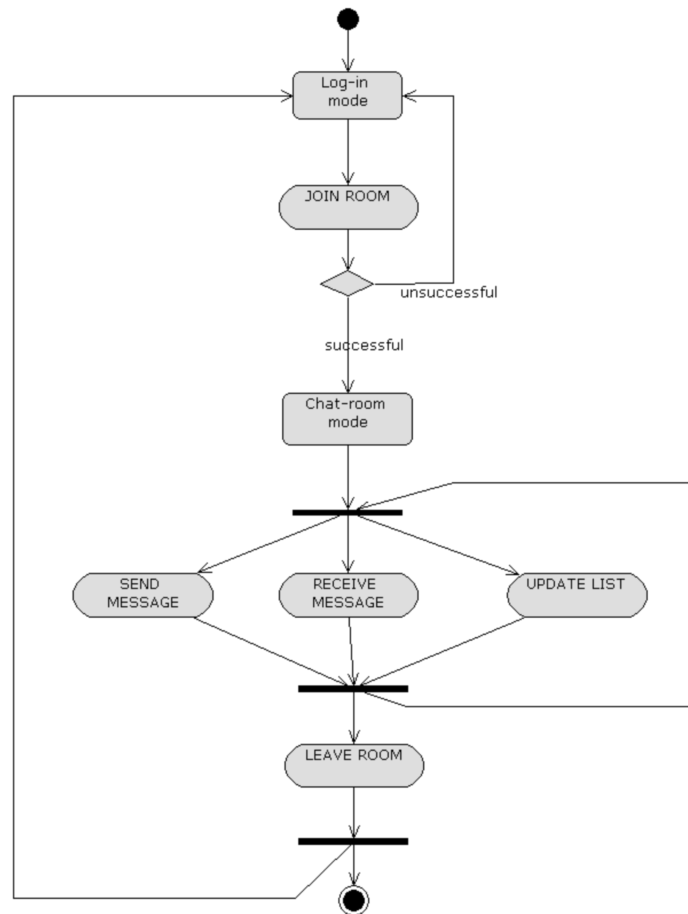


Figure 3 - Overall chat room application design - showing main operations

As the above diagram shows, these three operations all need to be carried out and repeated while the user is in the chat-room mode. My original design suggested a priority-based approach to managing these three different tasks, assigning each of the three responsibilities a priority level.

The priorities would work like this:

- System will send a message. If there are no messages to send…
- System will receive a message. If there are no messages to receive…
- System will update user list

In this way, the application would use its "idle" cycles to keep the current user-list up to date, maintaining the user-list every loop cycle where the system is not busy sending or receiving a message. As a result, the application would pause updating the list while sending or receiving messages, but such a pause would be relatively brief.

However, during implementation I decided that this was an inefficient approach – leading to the user list being updated far more often than is really necessary and causing excessive use of system

resources. Instead, I implemented the maintaining of the user list using a thread that updates the user list periodically. This is discussed in detail in the **Implementation** (4.4.2.3) section.

3.4 SERVICES LAYER

Returning to my original **Project Approach** (2.4.1), the Services layer holds common functionality – functions that are useful to more than one application. My approach meant that at the time of identifying the requirements for the services layer, I had only one application designed. As a result, it was difficult to specify with great certainty what functionality would be common.

However, my general approach was that the Services layer would provide the interface to the protocols. My aim was to make them as generic as possible, to allow for re-use for later applications.

3.4.1 SERVICES FUNCTIONAL REQUIREMENTS

The first thing that I did was to identify the primary functions required of the Services layer. I did this by going through the application functions in the Application Design Document. For each requirement, I identified where the application needed to make use of network protocols, and specified a more generic function that could implement this. This left me with the following list:

3.4.1.1 JOINGROUP

This allows the application to join a specified peer group. As I had already identified that I was using JXTA to implement the network protocols, it made sense to use the existing JXTA PeerGroups. However, this was a design decision, and will be discussed later.

As mentioned previously, security is not a consideration for this project, so the joinGroup service does not include any user authentication. Instead, it is a mechanism to allow communication with other users in the same group and to inform other users on the network of the user's arrival. It was also a way of ensuring that there are no duplications in the chat-room – by making a set time when user names could be checked.

3.4.1.2 LEAVEGROUP

This allows the application to leave the current peer group – closing all connections and freeing all network resources associated with the group.

Without a server to keep the peers up-to-date with when users leave, it is up to the peers to find this for themselves. To aid peers with this "discovery" – being able to see what other users are in the peer group, it was helpful for users to inform all members of a group before they leave. As a result, the leaveGroup design included an EXIT messages to be sent to all users on leaving.

During development, it was found that relying on an EXIT message was not sufficient to manage the different ways that people might leave, and additional safeguards were implemented. These are discussed in detail in the **Implementation** section (4.3.6).

3.4.1.3 SENDMESSAGE

This allows the user to send a message to one or more users. The application can specify a list of intended recipients and the message, without having to call a "send" function multiple times. The Services layer then implements the sending to multiple addresses.

3.4.1.4 SENDMESSAGETOALL

I thought that it might be helpful to have a service where a message can be sent to all members of the current group, without requiring the application to specify each individual member.

I did consider that it makes little difference whether the application or the services layer identifies the message recipients, and that this service might not be very useful for a chat application. However, it may be useful for other applications that may not store a list of local users.

This requirement was not derived from the application design, but I believed that it would be a useful function to provide.

3.4.1.5 RECEIVEMESSAGE

This should be self-explanatory. The Services layer is responsible for identifying the sender of the received message, and extracting the message contents.

The Services layer is then responsible for delivering the extracted message contents to the correct application. I decided that the application should not need to know how the messages are sent or encoded – that it is the responsibility of the services layer to turn the data the application wants to send into a network message and then to deliver received message data to the application.

These two responsibilities – listening for incoming messages, and then processing and delivering them – will be discussed in greater depth in the **Design** section (3.4.3).

3.4.1.6 DISCOVERY

This shall allow the application to see what other users are members of the current peer group. It shall return the application a list of users in an agreed format. This is discussed in detail below.

3.4.1.7 CONCLUSION

This initial list should enable all of the functionality required by the chat application. My next step was to go through these requirements and consider them in more detail.

For each requirement, I considered what would be required of a Services layer in order for the application to be able to function, trying to specify it in as generic a way as possible so as not to tie the service to implementing chat-room functionality.

The final requirements analysis is detailed in the Services Requirement milestone document attached to this report[19].

3.4.2 SERVICES LAYER DESIGN CONSIDERATIONS

I continued my development of the Services Layer by beginning a high level design of the service functions. I began by further breaking down the services' functions identified in the requirements.

[19] Appendix C – "Chat Application: Services Layer Requirements"

3.4.2.1 DISCOVERY CONSIDERATIONS

One thing that I considered was how I would manage the "discovery problem". The P2P nature of the project, without the provision of a server to maintain a list of current logged-in users, means that I have to address how the services layer identifies other users currently in the group. This can be implemented in a number of ways. Some of the methods that I considered include:

3.4.2.1.1 Explicit peer configuration

Each peer could be given the details of other known nodes in the network. This could take the form of a configuration file listing neighbour details. This is not really a discovery mechanism – it is more a way to avoid having to implement one. Each peer would come into existence already "knowing" the other peers in the network.

Benefits of this approach include the simplicity of implementation, and reliability. However, the drawback is that it is inflexible – requiring all nodes to be reconfigured to introduce a new peer. This is not in line with the Project Vision of a dynamic, fluid network. As a result, this approach would not be suitable for this project.

3.4.2.1.2 Dynamic Peer Discovery – using directory services model

I could use a server (either stand-alone or a specialised peer) to implement a directory service to track current users, effectively requiring nodes to login to the directory when joining a room.

While this has potential benefits of being straightforward to implement, it also is not really in line with the original vision of a peer-to-peer network where PDAs could communicate freely. The addition of requiring a server breaks the true P2P model, and is something that I would rather avoid. As a result, this approach would not be suitable for this project.

3.4.2.1.3 Dynamic Peer Discovery – using network model

In a network model, no single peer knows the structure of the entire network or the identity of every peer participating in the network. Instead, peers know only of the peers with which they are in direct communication, participating in the larger network vicariously. As a result, peers must cooperate to carry out tasks.

While useful for many P2P cases, it would not easily satisfy the requirements of this project such as the ability to see what other users are in a chat-room. As a result, this would not be suitable.

3.4.2.1.4 Dynamic Peer Discovery – using Multicast

Multicast is like the network model except that the nodes do not necessarily assist with discovery. Instead, it takes advantage of features offered by the network itself to locate and identify peers.

Multicast enables a peer to send a message to multiple receivers simultaneously. The sender does not need to know how many receivers exist or whether any exist at all. It simply packs up a message and releases it into the network. All local nodes would receive a copy of the message.

This could be used for discovery by having peers announce their existence using multicast. Other chat-room members detect this message, extract the user-name, using it to maintain the user-list of current chat-room members. This announcement would propagate through the network to all users.

This seemed to be the most suitable approach for the project, and I decided to use this. Another benefit of this approach is that there is already an implementation of IP Multicast in the JXTA protocol set. Development based on these building blocks was therefore quite straightforward.

3.4.2.2 DISCOVERY OPTIMISATIONS

I decided on a two-stage implementation for the discovery process. Discovering peers needs to be done often. As a result, it makes sense to cache some information to save finding it repeatedly.

If the service layer maintains a local cache of peer information, then this cache could be used in searches. This is quicker than searching the network for peers.

As a result, this led me to needing two types of discovery operation – the type that uses the local cache, and the type that actually makes a remote search for new peers. I called the search of local cache "localDiscovery" and the process of refreshing the local cache "remoteDiscovery".

One implication was that a peer might not see new users unless it has a reason to run the remoteDiscovery service. As a result, I needed to implement a safeguard to prevent new users joining the chat group being ignored. One approach I tried was to start the findUser service when a message is received from a peer - to ensure that the user is added to the local cache. If the user is already in the cache, then this will only result in a quick search of the local cache. If the user is not already in the cache, then the findUser service will try to find them remotely (using remoteDiscovery) and therefore ensure that they are added to the cache. I thought that this would be an effective complement to the discovery protocols, as most of the time this would involve only a quick search of the local cache.

Unfortunately, this added an unacceptable overhead to the receiving of every message. The fact that these services are intended to be run on devices with limited processing power (PDAs etc.) meant that this is something that I needed to avoid. As a result, I implemented an approach where peers periodically announce their presence. This is discussed in **Implementation** (4.3.7).

There is a risk that the application will continue to use cached information about peers after they have left the chat-room. As a result, I implemented two safeguards to prevent this:

- Add functionality to leaveGroup so a node will inform all members in the chat-group that it is leaving the group, so that they may update their cache accordingly

This provides a use for the sendMessageToAll function that I decided to include earlier – as it allows a quick way for the exit message to be sent to all users in the room.

This also required the introduction of a new service to provide the functionality of removing users from the local cache, to be used when an exit message is received. This was called updateCache.

This will obviously not help if the application exits in an unexpected way or if a connection to a peer is lost, but should catch the majority of cases. For those users who exit without running leaveGroup (by killing the application, leaving the network while the application is running or otherwise), there will also be the following safeguard:

- Add functionality to the cache such that cached information expires after a certain amount of time. In this way, peers have to periodically check if users are still there.

The expiry period should be definable by the user – too short a period, and the cache will be continually refreshed thereby negating the benefits of having a cache, too long a period and the application risks having an out-of-date user-list giving the user the mistaken impression that users are still in the room long after they have left.

3.4.2.3 MESSAGE CONTENTS

In order to satisfy the requirements of the various applications, I designed a message structure that would contain the necessary information. As well as the message data body, messages include a number of headers including sender name and the type of application the message is meant for.

3.4.2.4 CONCLUSION

The results of these and other decisions led to the creation of the Services Layer High Level Design Document attached to this report[20]. This details the final design for all services.

3.4.3 SERVICES LAYER DESIGN

The table on the following pages detail the initial design of the services layer. The tables describe briefly the nine main original functions of the services design – including the definition of their inputs, outputs and pseudo-code functionality.

The method descriptions provide an overview of the main service layer functions and how they were approached. Notice the division of discovery into two sections – local and remote. This is discussed above (3.4.2.2).

[20] Appendix D – "Chat Application: Services Layer Design"

JOINGROUP

Overview

This service will allow the application to join a specified group.

Each instance of the services layer shall enable an application to join one group only. If already a member of a group, further join operations shall fail.

Inputs

- Name of chat group / room : String
- Requested chat-room user name : String

Outputs

Outcome of join operation, either:

- Successful
- Unsuccessful – user already in a chat-room
- Unsuccessful – requested user name already in use
- Unsuccessful – other error

Functionality

1. Check if the user is already logged in to a room (if so, return output and exit)
2. Check if requested username already in use (if yes, return output and exit)
3. Publish user's presence
4. Check that user's entry into chat-room has been published – use discovery to try and find self
5. Store peer group information needed to send messages
6. Start listening thread (receiveMessage)
7. Set loggedIn state to TRUE
8. Run remoteDiscovery to populate list of chat-group members

LEAVEGROUP

Overview

This service will allow the application to leave a specified group.

To aid network integrity, the service will inform all peers that user is leaving.

The service shall then be responsible for closing all relevant connections to the group.

Inputs

- Name of chat group / room : String

Outputs

None

Functionality

1. Check if user is logged in to a room (if not, exit)
2. Use sendMsgToAll to send message of type "user exiting" to all peers
3. Close all network connections

SENDMESSAGE

Overview

This service shall allow the application to send a message to one or more users.

It shall require the application to specify the message type – to enable this service to distinguish between messages from different applications.

Inputs

- Intended message recipients : List of Strings
- Message
- Message-type

Outputs

None

Functionality

1. Build message – including sender name and unique identifying data
2. For each named recipient:
   a. Run findUser to get information necessary to send message – peer data
   b. Open connection to peer
   c. Send message
   d. Close connection

In the event of any errors occurring during stage 2, the service will retry stage 2 for that recipient once.

SENDMSGTOALL

Overview

This service shall allow the application to send messages to members of a named group without requiring the application specify individual recipients.

There is no need to specify the group name, as this will be implied to be name of the group that the application is currently a member of.

Inputs

- Message
- Message-type

Outputs

Outcome of send attempt, either:

- Send will be attempted
- Unsuccessful – user not in a chat-room

Functionality

1. Check if user is logged in to a room (if no, return "unsuccessful" and exit)
2. Build message – including sender name, message type and unique identifying data
3. Use discovery (defined below) to identify list of users in current room
4. For each user found:
   a. Run findUser to get information necessary to send message – peer data
   b. Open connection to peer
   c. Send message
   d. Close connection

In the event of any errors occurring during stage 4, the service will retry stage 4 for that recipient once.

RECEIVEMESSAGE

Overview

This service shall respond to incoming messages.

It will identify the type of incoming msg to decide what to do with it.

It will be run continuously as a listening thread.

There are no inputs to the thread for the service or application that starts it.

The output will be the reason for the continuous thread ending.

The inputs and outputs referred to below refer to the input when a message is received, and the output that it produces from it.

Inputs

- Message

Outputs

- Parsed Message
- Message sender

Functionality

1. Create incoming network connection to enable messages to be received
2. Run endless listening loop, waiting for incoming message
3. If a message is received:
   a. Extract contents from the message, removing any transport or other header information
   b. Extract message-type
   c. If message type is "user exiting", send message to updateCache
   d. If message type is "app X message"
      i. Send parsed message to correct application

LOCALDISCOVERY

Overview

The localCacheDiscovery service shall identify the users that are currently members of the group that the application is a part of.

It shall look for users in the local cache of user adverts only.

Inputs

- Name of group : String

Outputs

- Local users found: List of Strings

Functionality

1. Search locally cached information about local peers for user information which has passed the user-defined expiry period
2. Delete all user-information from the cache which is found to have expired
3. Return list of users contained in cache

FINDUSER	UPDATECACHE	REMOTEDISCOVERY
Overview	**Overview**	**Overview**
This service shall search for connection information about a named user – returning sufficient information to open a connection to that user. If the user cannot be found, the service shall return null.	This service shall allow explicit manipulation of the locally stored cache of peer information. *Note: This currently only provides facility to remove users from the cache, but I may implement further functionality in future.*	This service shall update the cache of locally stored peer information, by searching for all users in the current peer group. All cached information that is updated will have its expiry period reset.
Inputs	**Inputs**	**Inputs**
- User name : String	- updateType - User name : String	None
Outputs	**Outputs**	**Outputs**
- Connection information – peer data - Or null, if not found	None	None
Functionality	**Functionality**	**Functionality**
1. Search locally cached information about local peers for username (if found, return info and exit) 2. Use remoteDiscovery to update locally stored peer info 3. Search updated locally cached peer information for username (if found, return info and exit) 4. If nothing found, return null	1. If updateType is "remove": a. Search locally cached information about local peers for username b. If found, delete information from cache	1. Begin search for peers in the specified peer group 2. Wait for replies for a user-definable period of time a. If a peer is found, add connection information to the locally stored cache of peers

Figure 3 shows the system architecture design – outlining how and where the nine main operations detailed previously will be implemented. The previous table outlines the dynamic behaviour for each of these operations.

These nine methods shown in the tables were grouped to provide the following classes – shown in the class diagram to show the designed services layer structure.

- **BabelServices** – main interface to services layer functionality
- **BabelListener** – responsible for listening for and responding to incoming messages
- **BabelGroup** – implements a peer group, responsible for the creation, publishing and joining

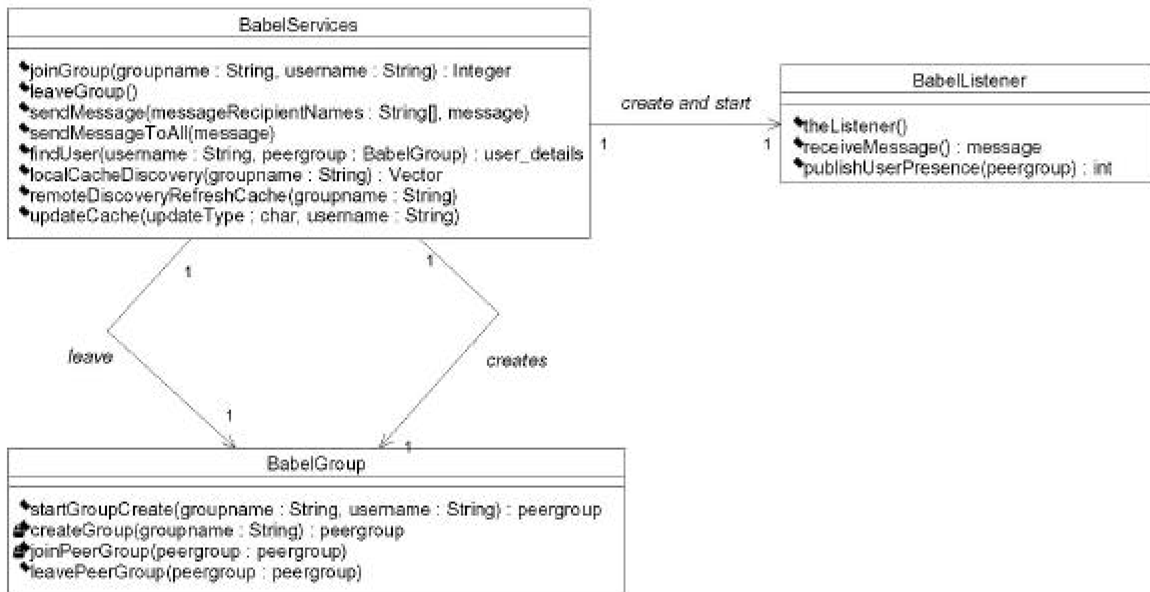


Figure 3 - Initial system architecture design for the services layer

3.5 APPLICATION LAYER – THE WHITEBOARD

3.5.1 REQUIREMENTS

The whiteboard application aims to develop a "collaborative virtual whiteboard". The idea is that members of a peer group see the drawing actions of each other in real-time, enabling them to collaborate on a design or idea-storming session.

The functional requirements were largely the same as those of the chat-room application, with the main difference being how the interaction is displayed – graphically instead of textually.

The application has the same basic functions:

- Join room / Log in – Same as chat-room (3.3.2.1)

- See list of available / logged-in users

  While I thought it was important that the user be able to find out what other users are involved in a collaborative design, I did not think that this warranted screen space that could otherwise be used by the paint canvas if displayed permanently. Instead, I decided to show this as a pop-up dialog box, which could be accessed by clicking on a button on the canvas. This was an interface decision, and is justified in detail in the **User Interface** section (3.8.4.2).

- Send and receive – messages to all

  I decided that all messages would be sent to all users in the group. Group work on a diagram means that all users should be involved. Adding private messages would

only confuse matters by introducing different peers seeing different images on their canvas. This was a usability decision, as I thought that a public forum approach better suited the concept of a collaborative sketchpad.

3.5.2 DESIGN

3.5.2.1 GENERAL

As with the chat-room application, I designed the whiteboard application to operate in one of two “modes” – either logged in, or logged out. The interface design supported this by making it clear which mode the application is in at any time – see **Interface** discussion (3.8.4.3).

An explanation of the use of modes and the implications on group management can be found in the earlier discussion about the chat-room application. The unique design requirements for the whiteboard application came in being able to implement an efficient paint engine, capable of displaying user tracings in real-time.

While in a peer group, the user can send and receive messages – with received messages being interpreted and represented as painting on canvas.

3.5.2.2 USE OF COLOUR

I wanted the interface to include some indication of which user was responsible for different parts of the sketch. I considered a variety of approaches, including annotating or labelling lines, but finally decided on using a different colour for every user. In this way, I could use a key or legend that matches up colours to names.

Every user that joins the peer group can be assigned a colour, thereby identifying any lines or shapes that they draw as being theirs. I identified twelve different colours that I believed were distinctive enough to be distinguishable. In the event that more than twelve users join a peer group, the available colours are cycled (so that the thirteenth user has the same colour as the first).

While this does mean that users cannot use different colours in what they do, I felt that this was an acceptable sacrifice. The whiteboard is not meant for artistic drawings or real-life representations – it is meant to be a free sketchpad for collaborative diagrams. The addition of knowing which user is doing what adds value that justifies the loss of any other semantic meaning that colour could have.

As this is a peer-to-peer application, and there is no server or coordinating peer to allocate colours, I thought that the best solution would be for each peer to allocate the colours that appear on it’s screen itself. Each peer would allocate colours in the order that it encounters other users. This does mean that the allocating of colours to users could be different between peers, but as the colour holds no meaning for the sketch other than to identify the user, this does not matter – provided that each peer maintains a key of colours to users correctly.

The key of colours to users was implemented using a Dialog window, opened from the “i” button shown in the designs below. The interface for this dialog was kept as simple as possible – just a list of usernames with a colour for each. When a user leaves a sketch group, I decided that the best interface would be for their contributions to remain on the screen. Therefore, it would be helpful for them to be included in the user/colour key even after they leave.

3.5.2.3 OTHER FEATURES

I also included in the design of the paint canvas engine the ability to alter the effect of drawing on the canvas screen – effectively altering the size and shape of the virtual pen "nib". This was intended to reflect the flexibility of the application, and demonstrate the increased usability possible by giving users greater choice over how they draw their sketch.

3.6 APPLICATION LAYER – FILE TRANSFER

3.6.1 REQUIREMENTS

The file transfer application aims to provide a forum where users in a peer group can freely exchange files. The idea is that members of a peer group can easily swap documents and work together on files, without needing a common server to store them.

As explained before, I formally specified the requirements in a Requirements Analysis document. However, in this section I will briefly summarise the main functional requirements.

The interface shall allow users to see who is available in the peer group, and select users to send files to. The application shall provide a seamless efficient link with the platform's native file storage system, allowing users to select files to send and specify where to store incoming files.

The application shall enforce an informed choice as to whether to receive an incoming file. All incoming file transfers must first have asked for and received permission before commencing. When being asked for permission, the receiving user shall be presented with enough information to make the decision including at least the file type, file size, and name of user trying to send the file.

3.6.2 DESIGN

The functionality of the file transfer application design was mostly divided into two classes. The first, **BabelFileApp** provides the interface. It is responsible for allowing the user to select a file to send and a recipient from a list of users logged into the peer group. It is also responsible for receiving incoming file transfer requests and inform the user.

The second, **BabelFileHandler**, provides the engine responsible for breaking a file down into segments small enough for sending or reconstructing incoming received file segments into the original file.

Figure 4 shows the system architecture design – outlining how the functionality is divided. The functionality design is explained in detail in the pseudo-code below (3.6.2.1.1 - 3.6.2.1.2).

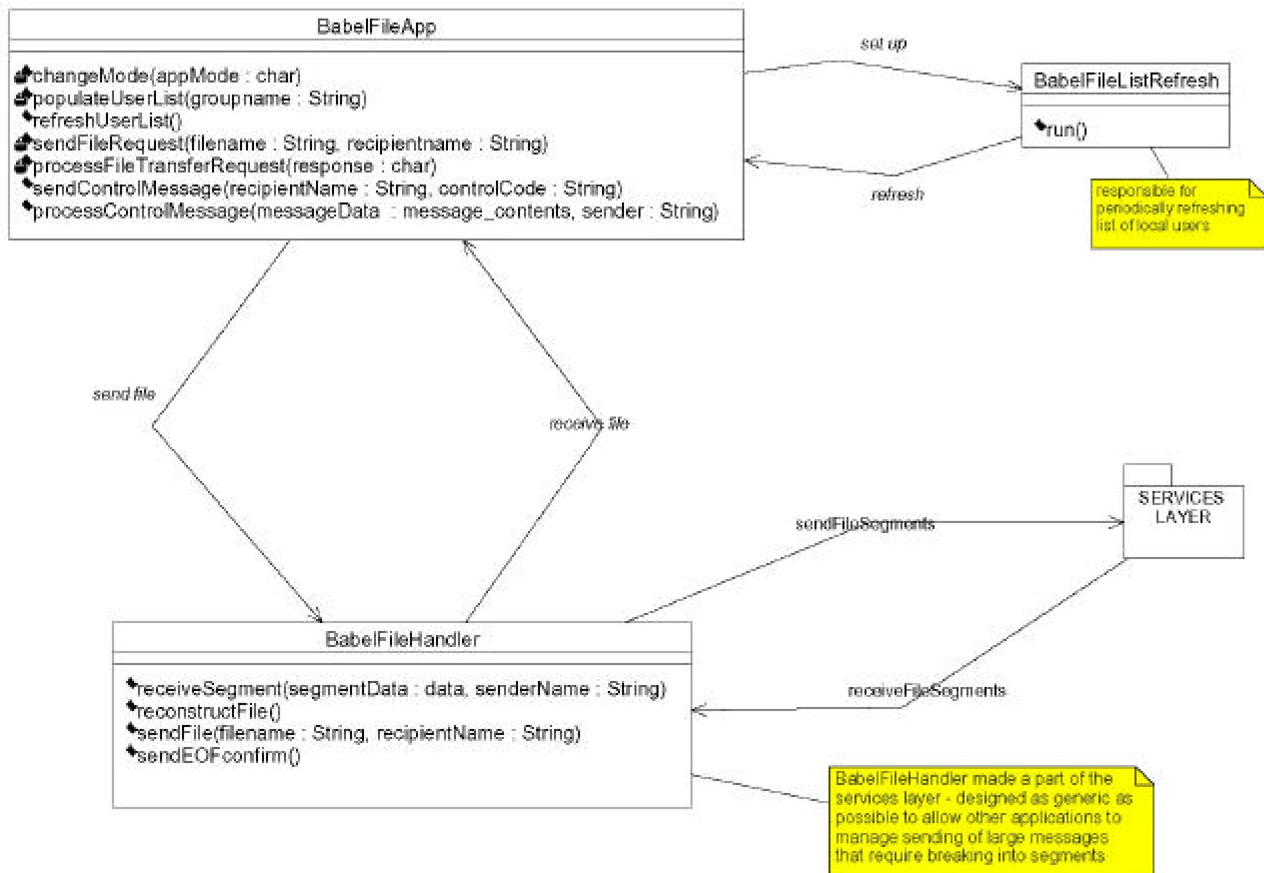


Figure 4 - Initial File Transfer Application design

3.6.2.1.1 Sending a message – pseudo code

1) User selects a file to send – with the interface providing a way for the user to choose a file from the local file system

2) User selects a remote user to send the file to, from a list of logged-in users

3) User confirms their choice and **BabelFileApp** creates a control message – a file-transfer-request message, and sends it to the recipient. The message includes the filename, file type and size. – `sendFileRequest` (*This meets the requirement of providing the user with an informed choice about whether to receive a file.*)

4) User waits for recipient to reply – with appropriate feedback displayed while waiting

5) User receives a control message – the file-transfer-request-reply message from the recipient – `processControlMessage`

6) If received control message is a REJECT (for any reason), send is cancelled, and appropriate feedback is displayed – including the reason if known.

7) If received control message is an ACCEPT, send file and recipient details to the **BabelFileHandler** for processing -- `sendFile`

8) **BabelFileHandler** breaks the file into segments small enough for sending, packs each segment into a project network message and sends it to recipient

9) When all file segments have been sent, the **BabelFileHandler** sends a control message to the recipient – a confirm-end-of-file message containing details required to reconstruct, such as number of packets sent – `sendEOFconfirm`

10) **BabelFileHandler** returns message to **BabelFileApp** to inform that file has been sent – allowing the interface to display feedback to the user

3.6.2.1.2 Receiving a file – pseudo code

1) **BabelFileApp** receives a control message – containing a file-transfer-request message -- `processControlMessage`

2) Control message contains file details (name, type, and size) and the sender's name – which are displayed by the **BabelFileApp** interface

3) **BabelFileApp** provides user with way of responding to request – to either ACCEPT or REJECT the request. Reply is sent to send-requester – `sendControlMessage`

4) If reply was an REJECT, nothing more is heard – transfer attempt over.

5) If reply was an ACCEPT, **BabelFileApp** will ask the user where the received file should be saved and what name the received file should be given. -- `processFileTransferRequest`

6) The Services Layer will receive a file segment. Finding that there is no **BabelFileHandler** running, it will start one, and pass the segment to it. – `receiveSegment`

7) All future segments are received and processed by active **BabelFileHandler –** received from the main services layer -- `receiveSegment`

8) When **BabelFileHandler** receives control message indicating end-of-file, all packets have been received, and the received file segments are reconstructed and saved as a file using information obtained from **BabelFileApp** – `reconstructFile`

9) **BabelFileHandler** informs **BabelFileApp** that file has been successfully received, so appropriate feedback can be displayed.

This implements a simple handshake-protocol between the two peers, which does not attempt to provide a secure, safe platform. In effect, the protocol is more like etiquette to be conformed to rather than a secure, abuse-proof system. The receiver relies upon the sender to effectively ask for permission before sending a file. If a sender were to start sending file segments without sending the request message first, the services layer would still create a file handler and start receiving the file.

The ability to reject incoming files is intended to improve usability, not provide secure protection against unwanted files sent maliciously – as specified in project requirements, security is not an issue. As the application is not being designed to inter-operate with other applications, and this application is designed not to allow the user to send a file without asking for permission first, this is secure enough for these purposes.

Note that my design placed the file handler class in the services layer, allowing it to be reused by any other applications wanting to send messages large enough to require being broken down into smaller segments for sending.

3.7 CORE LAYER

Originally, my intention was to carry out requirements analysis and design for all three layers of the project before beginning implementation. The **Project Approach** (2.4.1) contains a detailed rationale for this approach.

This approach was based upon the idea that Project JXTA provided a core set of network protocols upon which I could build. However, as I learnt more about JXTA, I found that JXTA is a much more extensive set of network protocols and services than I had originally realised.

As a result, I redefined my approach. My further research into JXTA showed that it was possible to develop an implementation of the Services layer using the protocols provided in the JXTA set. From there, I went on to implement the chat application as designed using these developed services.

As I still wanted to include work on protocols in this project, I decided that I would further investigate the implementation of the JXTA protocols that I use. My aim was to try to identify weaknesses or shortcomings in the protocols, and identify protocols that I could improve upon.

As this work relies upon a discussion of what I learnt from the Implementation, I have detailed this in the **Further Work** section below (6.1) to avoid confusion.

3.8 USER INTERFACES

3.8.1 REQUIREMENTS ANALYSIS

As well as the functional requirements discussed previously, I also analysed the other interface requirements. The first thing that I did was to research the requirements of the final target platform. Although my project requirements did not specify any particular device, my research into high-end PDAs such as PocketPCs and iPAQs seemed to suggest that a colour screen resolution of 240 x 320 pixels was standard. A wide variety of PDAs stated this in their specifications. This was the first requirement that I identified of the application interface.

Other requirements that I identified were derived from the nature of the PDA interface. For example, my initial prototype chat application[21] provided a text box where users could prepare a message and send it when the user clicked on "Enter". However, while this made a quick and easy

[21] The use of a prototype chat room application for the development of the services layer is discussed in the **Development and Implementation** section (4.3.1.4).

interface for the testing and development of the services layer, this would not have been a suitable implementation for a device without a keyboard (where "Enter" relies on a series of stylus-input symbols). Providing a click button that the user can use to send a prepared message is a more efficient and appropriate way to get user input.

Other such problems include the lack of being able to distinguish between types of "click". With a mouse, the application can provide different functionality in response to a click on the left or right mouse buttons. However, the stylus provides a single type of input – thereby limiting the types of input I could use. (I did consider simulating the variety of click types using a modifier – such as a click while holding down a certain button, or prefixed with a certain command or symbol, but I felt this would be overly complex for the requirements of the application).

Another example is the use of such GUI elements as Tool Tips, or "Bubble Help" – providing information when the user hovers the mouse pointer above a part of the GUI. Such interface conventions are impossible for PDA applications. Other examples include such visual cues as tab order for data entry controls – very useful in guiding users in windows applications, but not relevant here. Designing an application for such platforms is not simply a matter of designing a standard Windows application and using a stylus in place of the mouse. The absence of a keyboard and mouse introduce a number of requirements – both restrictions and advantages.

An example of one of the advantages that I identified was that the use of a stylus allows for smaller buttons and controls than would be usable with a mouse. Small controls are difficult to use with a mouse, because of the time and increased effort required to line up the pointer. However, the way that a user directly uses a stylus to click on a control with a PDA means that smaller controls that would be fiddly and cumbersome with a mouse become acceptable with a pen-based pointer.

3.8.2 INITIAL DESIGN

With all the interfaces, I began by sketching out initial designs on paper and comparing them for their usability. Finding the optimum layout of controls took a lot of consideration because of the limited screen resolution available. Each design approach was mocked-out on paper – drawing the interface to scale and using it with a pen to simulate the use of a PDA stylus.

I took a number of steps to make the most of the space available, such as deciding to combine the "Login" and "Logout" buttons of the initial prototype. As only one of them was ever enabled at any one time, it made sense to use a single button in which the button text alters to reflect the currently supported operation. After deciding on a design, and choosing the optimum layout for the screen controls, I began to implement it. This implementation work is described in the following **Development and Implementation** section (4.4.1).

3.8.3 CONSISTENT STYLE FOR THE APPLICATION SUITE

The initial Project Proposal described the development of a "suite of applications". With that in mind, the design that I finally settled upon was influenced in part by its suitability for implementing all of the applications.

My intention was to create a consistent style that all Project Babel applications could share. For example, I decided to use the upper part of the interface where group and usernames are controlled in all Babel applications.

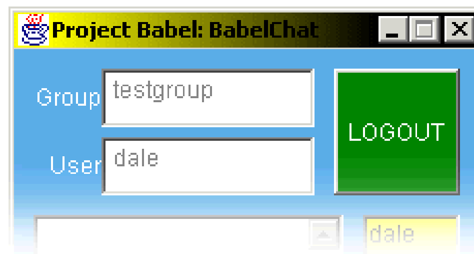


Figure 4 - Section of interface design - used in all applications

As well as providing a consistent interface and style aiding both the aesthetics and learnability of the application suite, it also allowed for reuse of code thereby reducing development work required.

For the same reason, I decided to maintain the colour and general look between all of the applications I would develop.

3.8.4 FINAL DESIGN

Much of the design was based on things learnt from creating and using the prototype chat application used in the services layer development[22]. For example, the method of selecting which users to send messages to in the chat application (by selecting or de-selecting them from a list – see the design diagrams following this section – 3.8.4.3) was kept. The prototype demonstrated that it was a flexible way to switch between private one-to-one messaging, private conferences between small groups, and global public messages.

As well as learning from the prototype which approaches worked well and which did not, the interface design was also influenced by the research into PDA applications carried out in the initial project investigation, and the functional and aesthetic requirements analysis carried out previously.

3.8.4.1 DISPLAYING ERROR MESSAGES

The initial prototype chat application displayed all logging debug information and error output to the standard output – the terminal where the application was started. While this was acceptable for debugging purposes, it was not a very user-friendly approach for running on a device where one application is run at a time, with the application filling the screen.

Much of the information output would not need to be seen by the user, and was useful only during development and debugging. An alternative way of outputting this was used – logging the output to a file (this is discussed in greater detail in the **Implementation** section – 4.5.1.3).

Error messages to provide the user with feedback were necessary, however. An example of this is where a user tries to login using a name that is already in use in the peer group. I decided to display error messages using pop-up error dialog windows. Examples of the design of these dialog windows are shown on the following pages. My approach was to have a standard box for all errors – with a description of the problem, and a confirmation button to show that the user has read the error.

[22] The use of a prototype chat room application for the development of the services layer is discussed in the **Development and Implementation** section (4.3.1.4).

3.8.4.2 SHOWING ADDITIONAL INFORMATION

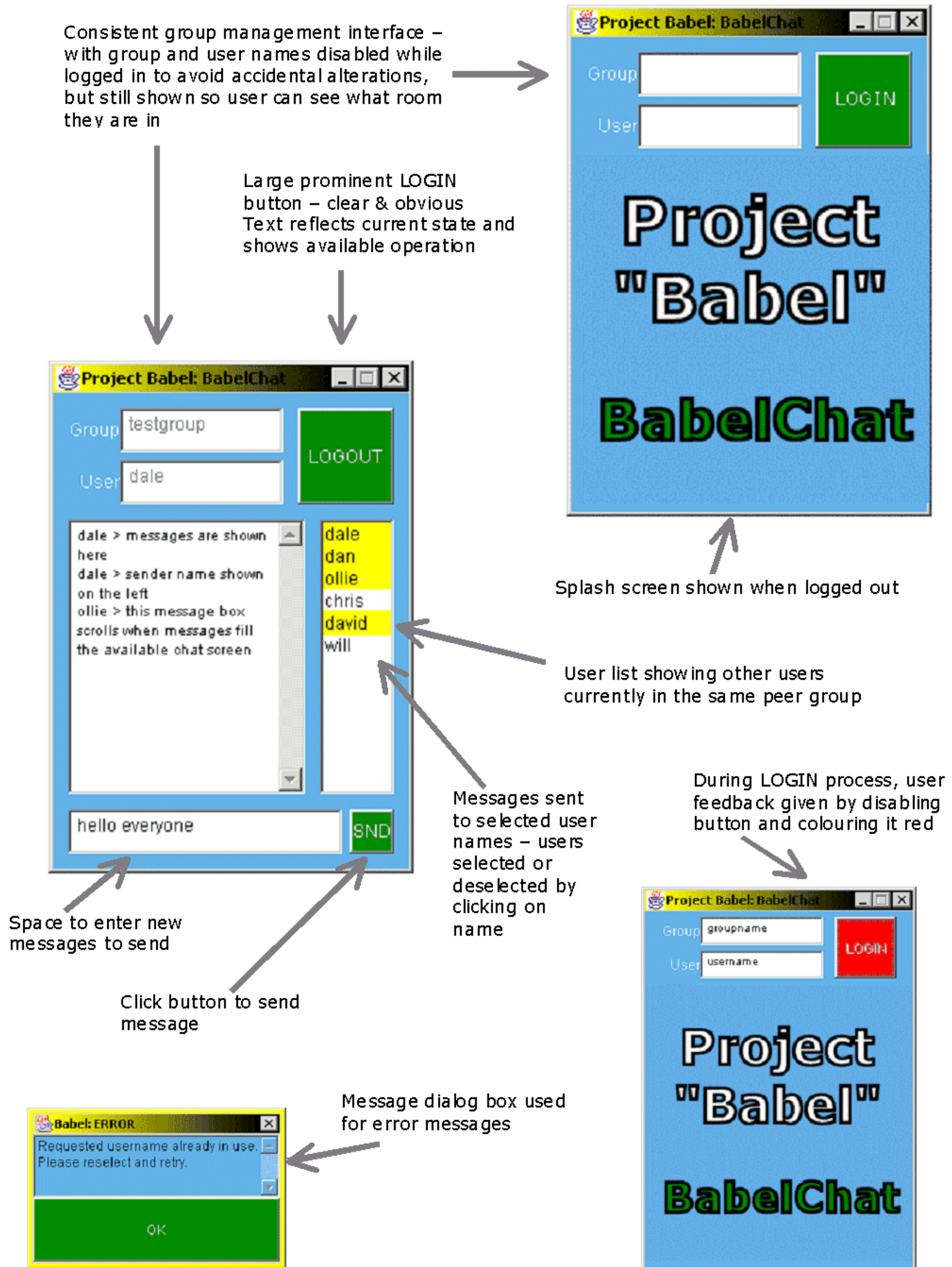
The limited screen size meant that I had to be very efficient with the layout and selection of GUI elements. This was especially important with the interface design for the collaborative whiteboard application. For the application to be at all useful as a design tool, the canvas needed to be as large as possible. This meant keeping the interface as clear and uncluttered as possible. While I did want to display information such as the names of other users in the group, I felt that there simply was not the screen space to include such information permanently.

Instead, I decided to use pop-up dialog boxes for all such information, leaving the screen as free as possible for use as a whiteboard canvas. As discussed previously, my investigation into PDA interfaces showed that buttons could be made quite small without sacrificing usability. I tried to avoid always putting the information and controls on the screen, instead providing small, discreet buttons that could be used to open dialog windows where settings could be viewed and changed. Such an approach does sacrifice some ease of use by making it harder for the user to access controls, but I believe that this is justified by the increase in usable screen space available to the user.

The design diagrams on the following pages show this use of buttons in the whiteboard application. The original design diagrams shown there (3.8.4.4) show the buttons labelled with simple letters to identify them ("i" for Information and "o" for Options), however these are meant as placeholders to be replaced with icons during implementation.

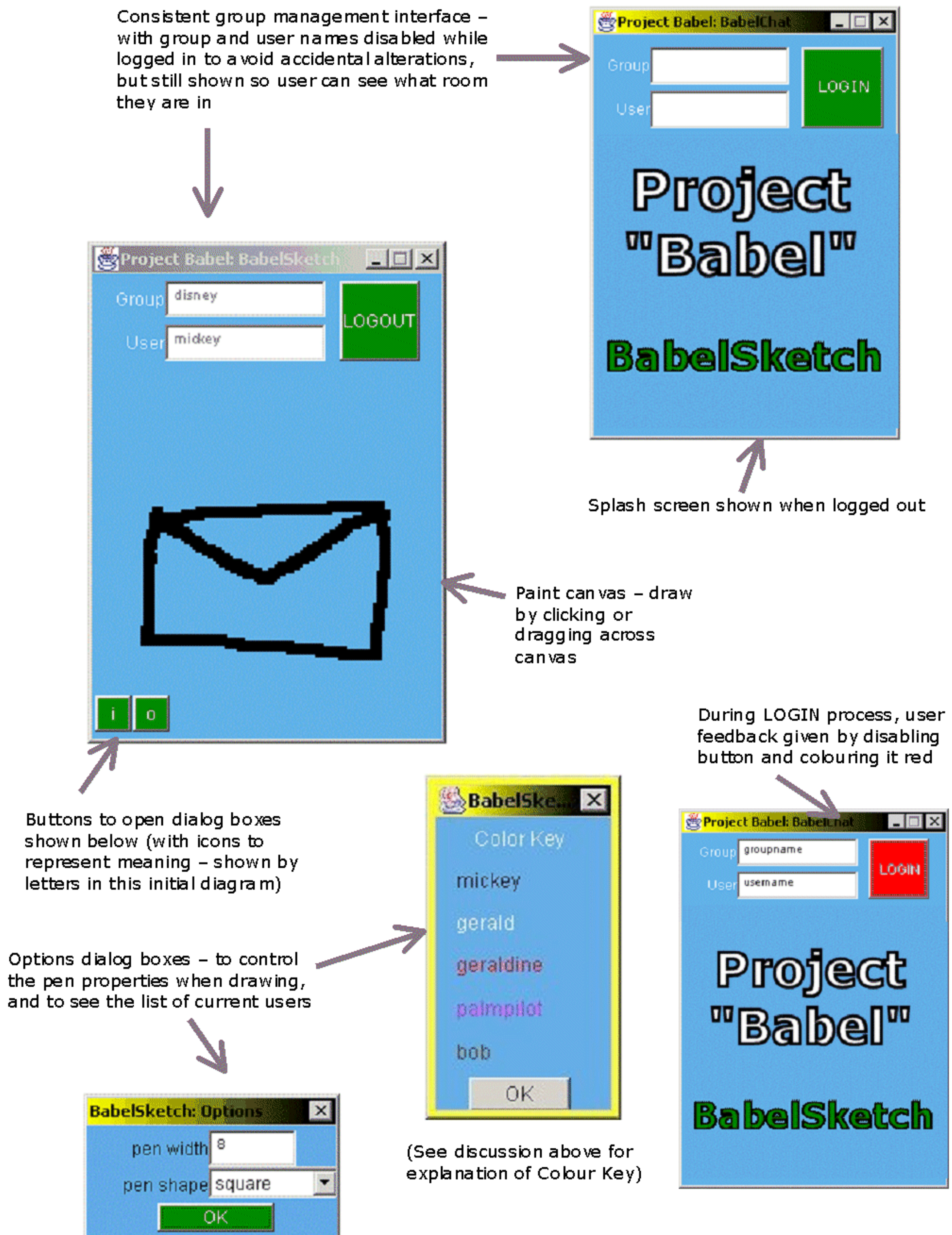
3.8.4.3 CHAT ROOM INTERFACE

The following diagrams briefly describe the interface design of the chat application.



3.8.4.4 WHITEBOARD INTERFACE

The following diagrams briefly describe the interface design of the chat application.

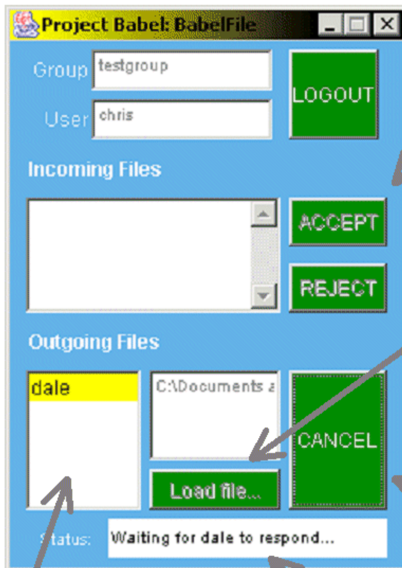


3.8.4.5 FILE TRANSFER INTERFACE

The following diagrams briefly describe the interface design of the file transfer application.

Standard interface with same group control and splash screen while logged out

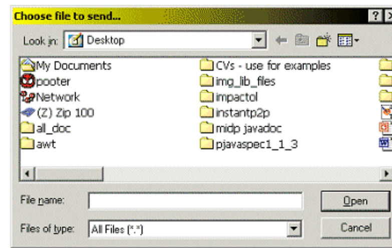
SENDING a file...



Main screen divided into 2 sections – for sending (bottom) and receiving (top) files

Click here to open file selection dialog (shown on the right) to select file to send

Filename displayed above button



List of currently logged-in users – click on user to send file to

Shows current status of send operation

Click SEND to send file. Button label changes to CANCEL while it is possible to cancel file transfer in progress

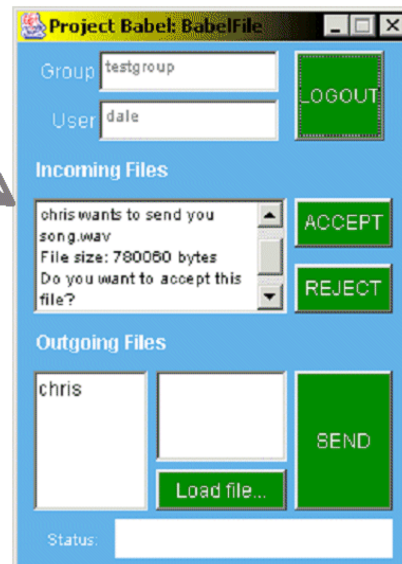
---

RECEIVING a file...

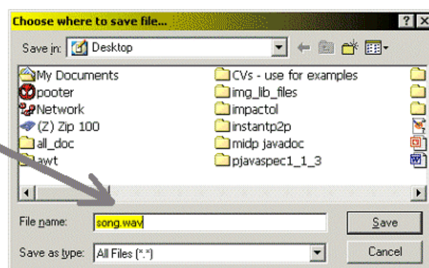
Incoming file transfer requests shown here – giving file name and size

Incoming file transfer request enables ACCEPT/REJECT buttons on the right.

Clicking ACCEPT opens the file save dialog below to specify where to save file



Filename pre-entered



4. DEVELOPMENT AND IMPLEMENTATION

4.1 OVERVIEW

This section provides a description of how the system was implemented to meet the requirements detailed previously. It includes an overview of some of the main problems faced, and the solutions adopted.

I also kept a chronological journal of the coding and implementation, describing all of the obstacles faced. A copy of this journal is attached to this report as an appendix[23].

4.2 CORE LAYER

As discussed previously, I decided to build my services layer on the JXTA network library. To do this, I downloaded a stable build of the JXTA jar packages from the JXTA website[24]. The JXTA website offered stable builds of the Java binding, written in J2SE – standard Java.

The easy availability of the pre-compiled jar packages, together with the availability of a full Javadoc[25] API reference for the J2SE implementation meant that this was a good choice for initial implementation and experimentation.

After I completed an initial implementation of the services layer, capable of supporting a prototype chat-room application (discussed below – 4.3.1.4), I ported the services layer code to use PersonalJava – the target language decided on for the implementation as discussed previously.

As well as making my code compliant with the PersonalJava API, I also needed a PersonalJava-compatible version of the JXTA platform. To obtain this, I found a copy of the JXTA source code written in PersonalJava from the JXTA website. I then compiled this using make files, to produce my own new set of PersonalJava compatible jar files.

Together with these, I was able to run my code on the PersonalJava Emulation Environment, and development continued in PersonalJava – using the JavaCheck tool listed above (2.3.3) to periodically check the code for compatibility.

[23] Appendix G – “Implementation Journal”

[24] The JXTA packages were obtained from http://download.jxta.org/

[25] A description of what Javadoc is and how it works can be found at http://java.sun.com/j2se/javadoc/

4.3 SERVICES LAYER

4.3.1 GENERAL APPROACH

4.3.1.1 INITIAL FRAMEWORK

As discussed previously in **Methodology** (2.4.1), my approach to the implementation was to work "upwards" – starting with the services layer and then creating the applications, adding to the services layer as additional functionality was required by each application.

My approach was to first create the class files as defined in my design documents. I wrote these simply as empty definitions, declaring the class names and their methods – identifying input parameters and return values. An example of this is shown in the code excerpt below.

```
public int joinGroup(String reqGroupName, String reqUserName) {

    int joinOutcome = 1; // assume successful for now

    System.out.println ("SER: Running joinGroup service ");
    System.out.println ("SER: You requested chat room: " +
                         reqGroupName + " and username " + reqUserName );

    return (joinOutcome);
} // end of joinGroup()
```

Figure 5 - Code Excerpt: attempt1/babel/src/babelservices.java

This gave me a stable, compile-able framework in which I could then develop the services' functionality.

I continued with this, fleshing-out the services layer classes. When enough functionality was complete to support a sizable system activity – such as creating and joining a peer group, I wrote small test programs to run the services. In this way, I began to test each of the Service methods in turn. The first method tested was the joinGroup method.

4.3.1.2 EARLY PROBLEMS

Running the Service layer methods revealed a number of problems in the code that would require too much space to describe fully here. However, the following example is a good illustration of how I investigated problems that I encountered using a variety of tools and techniques. It involves the creation of JXTA pipes – virtual connections created between peers to send messages through.

My approach to creating Pipes was to create a "pipe advertisement", setting the values of the various elements manually. Advertisements in JXTA are created as XML files, so setting elements involved adding a new tag-pair to the file. (For example, adding `<Name>testname</Name>` sets the Name element of the advertisement.)

I then tried to create pipes to send and receive messages through, based on my advertisement. This led to run-time errors, and finding the source of the problem required a lot of investigation.

I began by converting the advertisement to a string and printing it to the standard output. However, I could see no problems with the advert. To help debug further, I wrote a small function that would write the advert to an XML file. This produced a document similar to the following:

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
   <Id>
       urn:jxta:uuid-
       C2C85041752E4AC39202B0D9AACFE8B91F11D0544BA146AA9464A653B59D748204
   </Id>
   <Type>
       JxtaUnicast
   </Type>
   <Name>
       BabelUserName:babeltestuser
   </Name>
</jxta:PipeAdvertisement>
```

Figure 6 - XML files produced for debugging - attempt4/chatapp/useradvert.xml

Again, I could see nothing wrong with the advert format. I then used this file with the JXTA Shell to check the advert. The JXTA Shell provides a UNIX shell-type command-line interface to the JXTA platform. The shell commands provide a lot of built-in functionality, such as creating and joining PeerGroups, and made a useful test aid during the early development. By importing the file into the shell and using it to create a pipe advertisement, I could confirm whether there was a flaw in the advertisement, or in my approach to use it to create a pipe.

When attempting to create a pipe based on this advert, the JXTA Shell also threw exceptions, confirming that there was a fault with the advertisement itself.

I then used the Shell to generate a new, valid pipe advert, and compared it with my generated pipe advert. At first, I could see little difference between the two – they both contained the same elements, and seemed virtually identical. The only difference was the ID number, but as the shell and my implementation were based on different peer groups, I thought that this was to be expected.

After creating several more adverts from the Shell, and comparing them to advertisements generated by my code, I noticed that the format of the UUID[26] ID in my advertisements was slightly different to those created by the Shell. Noticing this led to me finding the problem in how I was generating Pipe IDs – a problem which, once found, was relatively straightforward to correct, by referring to the JXTA specification for PipeIDs and writing an ID generator which conformed to it.

4.3.1.3 EARLY TESTING

The JXTA Shell was a very useful test aid during early development – allowing me to confirm that the Services layer could successfully create, publish and join a peer group. I ran my joinGroup method in the Services layer to create and join a group. On another computer on the same Local Area Network (LAN), I ran the JXTA Shell and searched for local groups. The Shell was able to find and successfully join the peer group created by my services layer.

I was also able to confirm that my early services layer could successfully receive messages, by sending messages to the peer created by my code, from the Shell. My code printed to the standard output that a message was received whenever I sent a message from the Shell.

---

[26] UUID number – JXTA term, standing for Universally Unique Identification Number
Full details can be found at http//spec.jxta.org/v1.0/docbook/JXTAProtocols.html

4.3.1.4 PROTOTYPING

The “fleshing-out” of the services layer continued until I believed that it was stable enough to support an application. The testing that can be carried out on a network services library without an application is limited, even using the JXTA Shell to locate and correct the most obvious bugs. As a result, my next step was to create a prototype application. I wanted to test the services layer more thoroughly before beginning work on the final application layer.

At this stage, I wanted to produce an application purely for the testing and development of the Services layer – my aim was not to produce a fully functional or refined application. I wanted the application interface to be able to do enough to allow me to run and test all of the Services’ functions and methods – but not to really develop any application functionality.

As explained previously in **Methodology** (2.4.2), the initial implementation of the Services layer came after the requirements analysis and design of the chat application. As a result, the prototype used to test and develop the services layer implemented a basic chat-room.

The use of the prototype served many purposes. These include:

- An opportunity to learn about using AWT and the different GUI elements supported before I produced the final applications.

- Helped to confirm design sanity by demonstrating the division of responsibilities between application and service. (e.g., the services layer providing the functionality to find details about local users, but the application turning this into a useful list)

- Helped to identify a number of bugs and problems with the services layer, which would have been distracting if encountered while developing an application.

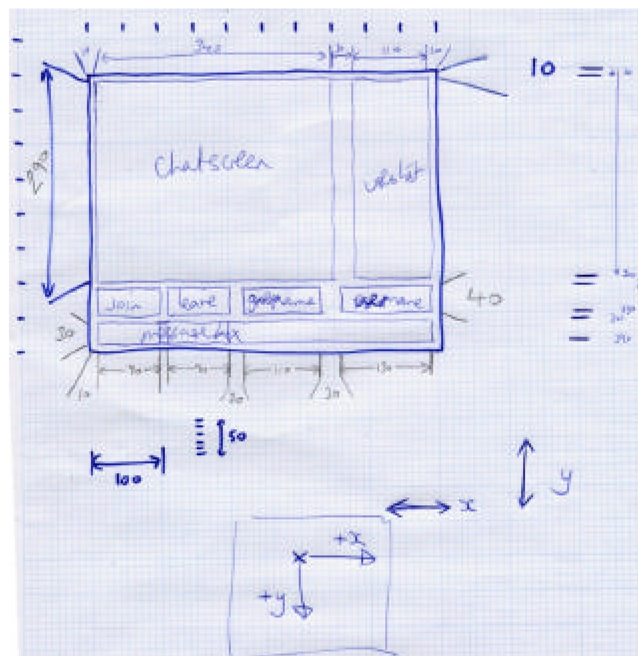


Figure 7 - Prototype design – an initial sketch

My initial prototype used an AWT interface – providing text boxes where chat messages, usernames and group names could be entered, and buttons which the user could click on to login and logout from peer groups.

My approach to designing interfaces is discussed in detail in the section on **User Interfaces** (3.8). After the design, I produced a scale drawing of the interface on paper, and use the drawing as a guide to mark-out the GUI elements in the code. Figure 7 is an example of the sort of scale drawing that I used – my initial design sketch for the chat-room prototype. It shows the large "chatscreen" where received messages could be displayed and a list where local users could be shown.

I also added basic error-handling to the interface – such as disabling the "login" button while a user was already logged in and enabling the "logout" button, and vice versa when a user logged out.

My approach to implementing interfaces is described in detail below (4.4.1).

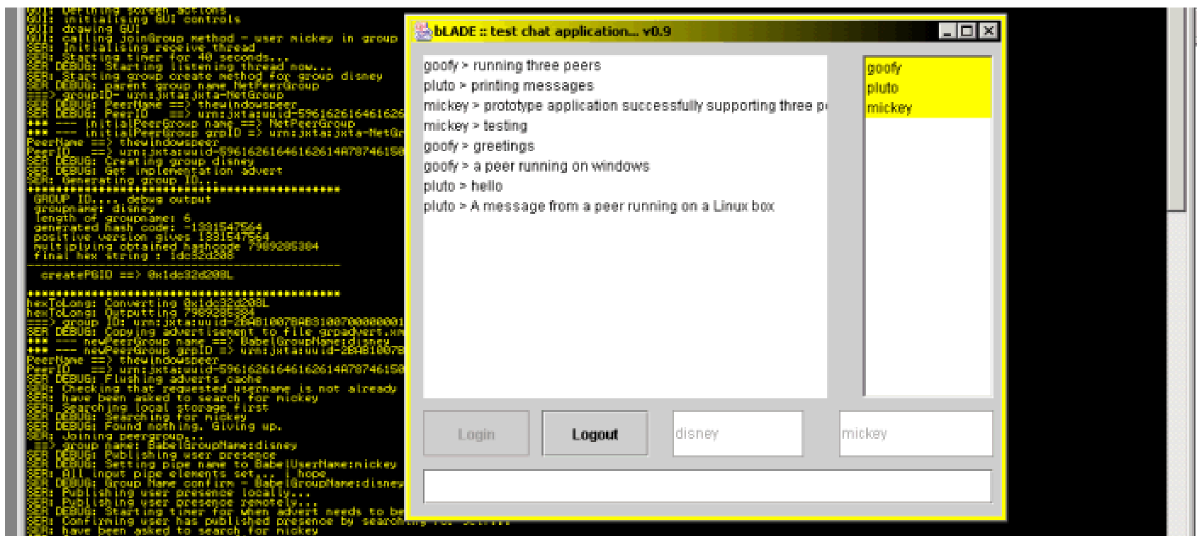


Figure 8 - Screenshot of the prototype running under MS Windows

4.3.2 DATA TYPES

The selection of data types was influenced by my use of PersonalJava[27]. Both the available data types and the methods available are limited in PersonalJava. I also tried to use data types and methods that are more efficient, such as avoiding String manipulations and println statements.

4.3.2.1 MESSAGES

Messages sent between peers contain a variety of information, including the message itself, and headers including the username of the message sender and the type of application the message is meant for. I decided that the best way to implement this would be to use a class to implement each message. I then provided "get" and "set" methods to access and set elements in the message.

[27] PersonalJava full specification can be found at http://java.sun.com/products/personaljava/spec-1-1-2/index.html

The body data contained inside the message class was originally implemented using a string, because of the convenience provided by string manipulation methods. While this worked well with the prototype chat application, I decided to change the message class to use an array of bytes. This better suited the variety of applications – able to be converted to storing the graphical data of the whiteboard, or the file data of the file transfer application.

4.3.2.2 DEFINITIONS CLASS

I implemented a form of header file to store constant variables – used to define message tags, error codes and return codes. To implement this I created a single class to store all of the constants, and used inheritance to let all other classes use them. By extending the definitions file, I helped to make my code more readable, and be consistent with my use of parameters and message values. It also made the code more maintainable, by putting all of the values in one place where they can be changed without having to edit multiple values throughout the code.

4.3.3 USING OUTPUT PIPES

To send messages to discovered peers in the same peer group, I used JXTA *OutputPipes* – uni-directional virtual connections between nodes. OutputPipes point to a single peer – to a specific InputPipe, and are created based upon discovered advertisements for remote user's InputPipes.

My initial approach to using OutputPipes was to create a new pipe for every message. Each peer would cache the pipe advertisements that it discovered. To send a message to that peer, it got an advert out of the cache and used it to create a one-use pipe to that user. After sending the message, the pipe would be closed and destroyed. Much of the documentation that I was able to find which discussed OutputPipes referred to them as one-use connections, and while they did not justify this, I could see several advantages to such an approach.

Using each pipe once was a secure, stable way to maintain the connections. If an advert was out-of-date, or the remote user was experiencing some problem that prohibited receiving messages, then the attempt to create an OutputPipe would throw an exception, which could be caught and dealt with. This was a reliable method, which worked well for both the prototype chat application, and the final chat application when it was integrated with the services layer.

However, when running the whiteboard application, the need for a more real-time, streaming-type network meant that I had to look for ways that I could optimise the services layer. The services layer was simply too slow to support the whiteboard application in a useful way. These optimisations are discussed in greater depth below, but one of my approaches was to optimise my use of pipes.

Instead of creating a new pipe for every connection, I created a cache of output pipes, so that pipes could be reused. This did mean that I lost the implied check that the remote user was still there that happened when the pipe was created, but I dealt with this in other ways as discussed below. However, avoiding the need to create a new pipe, negotiate a new connection, as well as closing it every time and using the garbage collector to destroy it, produced significant performance benefits.

The cache of OutputPipes was indexed by the user details, and stored in a dynamic array maintained by the services layer.

I did also introduce finite blocking to the usage of OutputPipes, but this is discussed below.

4.3.4 USING THREADS

Early versions of the services layer used to support the prototype chat application made limited use of the thread functionality in Java – allowing different processes to be executed concurrently. As a result, trying to do one task at a time, and waiting for it to complete before proceeding meant that many of the network operations (particularly when error-handling or dealing with "unexpected" situations) would lead to the interface seeming to stall while the services layer was run.

A particularly bad example was when the user clicked on LOGIN. Creating, publishing and joining a peer group, checking that the requested name was not already in use, and then joining the group and publishing the user's presence in the group, all took time. While this was being done, the interface stalled – seemingly crashed. Although logging to the standard output terminal showed that the application was active behind the scenes, the interface stalled, unresponsive to user commands and failing to redraw screen GUI elements if covered or obscured by other windows.

While this could have been solved using some time-sharing workaround, the most efficient solution was to use threads to allow processes such as the login process to run without causing everything else to stop and wait. The thread would then inform a listener when it was complete.

Threads are used for a variety of functions – examples include providing a "listening thread" (waiting to receive and process incoming messages), providing a "timer thread" (republishing a user's presence when adverts expire) and a "message queue handler" (dealing with a queue of outgoing messages, sending messages at the front of the queue and working through them).

Ensuring that the threads did not require excessive use of system resources required careful implementation and use of logging to verify. Simply using a `while(true)` loop in a run() method of a thread to get required functionality to repeat can cause CPU usage approaching 100% if not carefully handled. No matter how fast the machine that the code runs on, it will always cause the application (and computer in general) to crawl as system resources are eaten up by the runaway threads – the faster the machine, the more times it runs each loop per second, not helping overall performance. As a result, the usage of threads required careful consideration and testing to ensure that all threads remained in control.

Stopping the threads also required careful implementation to ensure that all threads stopped when required. My initial approaches of throwing interrupts to stop threads were not always reliable, with interrupt signals sometimes going unnoticed.

My research into Java threads showed that using the stop() method to halt threads is fundamentally unsafe, and prone to deadlock and other errors[28]. Instead, I decided to override the stop() method of many of the threads, so that they threw an exception. I then added a catch for that exception in the thread run() body. This seemed to be an effective solution.

4.3.5 CREATING PEER GROUPS

My first step to creating the peer group management service was to separate the process of starting the JXTA platform and joining the default initial NetPeerGroup, and the process of creating a user peer group. Once separated, I could then alter the way that they are called, so that they are

[28] The Java API documentation written by Sun contains a good explanation of why using stop, halt, and kill to stop threads is fundamentally unsafe. (http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html)

both called the first time, but then only the latter after that. In this way, I could avoid the overhead of starting the JXTA platform every time a user logs out and then into a new peer group.

As with the creation of pipes, groups are created by first creating an advertisement – an XML file containing the group parameters. I first created a blank peer group advert document, and then set each of the elements individually. These were created by adding XML tags, with the name of the parameter in the XML tag, and the value inside.

These elements included a group name, a text-based description, and a unique group ID. The challenge in creating a group ID was creating something that would be unique enough so that it would not match any group ID created by another peer not part of this project. The lack of a server to co-ordinate group management meant that each peer that wants to join a certain group needs to be able to generate the same group ID, but ensure that no peer that does not want to join that group coincidentally generates the same ID.

To do this, I wrote a hash function that used the group name requested by the user to generate a 128-bit group ID. The group ID had to satisfy the requirements described above – always generating the same group ID when given the same group name, but do so in such a way that different group names will not lead to the same group ID. The hash function also had to satisfy the format of group IDs – which contain encoded information about the group. I found the details of this format in the **JXTA Protocol Specification**[29] document, available from the JXTA website, and ensured that group IDs created by my services layer conformed to the format specified there.

4.3.6 WHEN USERS LEAVE

Without a server to inform the peers when users leave the chat-room, the peers have to find this out for themselves. The nature of JXTA pipes being used for sending and receiving messages means that if a peer tries to send a message to a user that is no longer there, this is not handled very well.

My first approach was for all of the peers to send an EXIT message to all users in their group before leaving. Creating and sending an "I'm leaving now" message was added to leaveGroup, so that a user always informs all peers before they leave. These users can then close their pipes to them.

This was an effective approach that caught most of the cases. During the development I enhanced and added to the functionality when EXIT messages were received, including removing adverts from the cache, closing output pipes to them, updating the user list sent to the application layer for displaying, and deleting any cached output pipes from the store.

However, this was not a foolproof approach. If a peer exits in any way other than expected (such as network failure, or any other sudden or unexpected crash or event which ends the application) then the EXIT message will not be sent. Therefore, an alternative method was needed to handle users leaving. This had to be considered, particularly when designing for a wireless LAN where users might move out of range of other peers without intentionally leaving a peer group.

I emulated this by killing one of the project applications (without closing down the application properly so that it would send the EXIT message) and seeing how the other peers reacted to this. I also tried unplugging network cables to see how the peers dealt with a sudden lack of network.

[29] The JXTA Protocol Specification can be obtained from http://spec.jxta.org/v1.0/docbook/JXTAProtocols.html

My initial approach to discovery consisted of each peer creating an advertisement containing all information necessary to connect to them, and then "publishing" it – broadcasting it to all remote peers. Receiving peers then cached this advert so that they could refer to a locally saved copy in the future (the benefits of this are discussed in the **Design** section previously). This meant that in the event that a user leaves without sending the EXIT message, the user could continually use the local cache to "discover" the remote user after they have left. I approached this in two ways. Firstly, I introduced a time-to-live (TTL) value for the adverts – telling the remote peers which discover the advert that after a specified amount of time, the adverts should be deleted. (This did require the user to periodically re-publish their adverts, but this is discussed in the following section on the **Using a Timer** – 4.3.7). Secondly, I introduced code that detected the absence of remote users by identifying problems while trying to send messages to absent peers.

With the initial approach to using OutputPipes as described above (where I created a new OutputPipe for every message), this was relatively straightforward. Trying to create an output pipe to a user that is no longer there causes an IOException exception to be thrown. By catching this exception, I could assume that the user had left, and added a call to the functionality used when an EXIT message is received. In the event that the exception was a one-off glitch, this would lead to the user removing a user who is still present. Their name would disappear off the application user list temporarily until the next time a remote discovery is done, however this is not a huge problem.

This became more complicated with the introduction of caching the OutputPipes, as described above. Using an old OutputPipe (with the connection information that this implies) when the remote user is no longer there led to send attempts stalling for a considerable time as the JXTA platform tried to get the message to the destination. The need for caching the output pipes has already been discussed, however it made the managing of users who leave without informing less stable.

I attempted to solve this by introducing a timeout value into the OutputPipe – specifying how long the pipe should block for when trying to send a message. This was a difficult choice when dealing with a mobile network with potentially variable properties. Too short a time and the pipe could give up before creating a connection, but too long leads to the system seemingly stalling when trying to send a message to a user which is no longer there. It was difficult to find a suitable value by experimenting on the development platform of three desktop machines with fixed cable connections – where send times were relatively consistent. However, I specified a time that I thought was the maximum length of time that a user would find acceptable to pause while trying to send a message.

(This does leave a problem if a user leaves without sending an EXIT message. Remote users will not realise that they have left if they just sit in the chat-room to listen and do not try to send a message to them. However, this is not a major problem, and is handled by the fact that their cached adverts will expire eventually even if they never try to send a message).

4.3.7 USING A TIMER

As mentioned previously, the fact that I introduced an expiry time into all published adverts helped to make the network more tolerant of users leaving without being able to send an EXIT message. However, it also meant that publishing the peer's own presence was not something that it could do while logging-in and then forget about. It was necessary for peers to re-broadcast their presence often enough that remote users would receive a new advert when the current one expires.

I tried a number of approaches to this, including looking for system behaviour that happens infrequently enough that I could include a call to publish adverts. However, I decided that the only reliable way to do this would be to add a separate timer thread, responsible for republishing adverts.

I implemented this by storing the time when an advert would expire. A timer thread then periodically checks that time against the current time to see if the advert has expired – if the current time is greater than the advert is republished.

Using the system time was a precise, reliable method to monitor the expiration. It behaves in the same way on different platforms, unlike using an empty loop or iterator to wait. While not checking the time, I put the thread to sleep to conserve system resources, and reduce the load on the system.

By making the timer thread keep a local copy of the advert, so that the *same* advert is published and not a new one created each time, I ensured that remote users would not end up with multiple copies of the same advert if the expired advert has not yet been removed.

4.3.8 OPTIMISATIONS

The first complete services layer supported the chat application adequately. However, when used with the whiteboard application, the services layer was shown to be too slow. The nature of a real-time whiteboard, able to share a free sketch canvas across a network introduced a need for a more real-time, streaming-type network. Although the messages being sent were smaller than the chat room, the nature of the stylus interface meant several messages could be created and sent in a short space of time by dragging the pointer. The chat messages took time to write, which had allowed the services layer time to send before the next one was ready.

This meant that I had to look for ways that I could optimise the services layer. The services layer was simply too slow to support the whiteboard application in a useful way. I used three approaches to optimising the services layer code:

- Optimised the search algorithm, and the way it was used – I improved the way that a peer searches for a user while preparing a message to be sent

- Peephole optimisation throughout the services layer – I re-examined all of the 5000-odd lines of code to find places where small improvements could be made. These included use of more efficient data types or methods, avoiding unnecessary printing, optimising threads (as described above), and improving efficiency of loops in the code.

- Introduced a cache of OutputPipes to avoid creating a new pipe for every new message. This has been discussed at length above.

While this work led to a noticeable improvement in the performance of the services layer, the whiteboard showed that messages would go missing when the network was placed under heavy load. This was evident in the whiteboard as segments of the line drawn on one peer not appearing on other peers. This was quite noticeable and affected the usability of the application.

Investigating the problem with network monitoring tools (see the discussion on **Use of Tools** – 4.5.1.5) showed that the messages were being prepared at a rate faster than they could be sent. The services layer uses a buffer to store messages requested to be sent during processing of an outgoing message, however this buffer was being overrun and messages were being lost.

There were three possible approaches to this problem:

- Alter the applications so that they sends fewer messages

- Alter the services layer so that it does not lose messages if asked to send more messages than it can handle

- Increase the send message buffer size

I decided to try the second. I did this by implementing a message queue in the services layer, so that when an application tries to send a message it is placed in a queue. The services layer can then work through the queue, sending one at a time at a rate that would not overrun the buffer. This sacrificed some of the real-time nature of messages but ensured that all messages were sent.

I implemented the message queue as a separate class to the existing send functionality – an optional method that applications can use. Applications remained free to access the send functionality themselves, as a queue is not always necessary or appropriate. The chat room, for example, where the time taken to prepare a message means that the services layer can easily cope, and the overhead of starting and stopping the queue for every message would in fact reduce performance.

The queue is managed by a queue-handler thread, responsible for sending all messages added to the queue. Instead of calling the "send" functionality, an application can simply add its message to the queue. The queue checks to see if the send-thread is running (if the queue is currently being emptied). If it is not, it starts the thread. If it is, it just adds message onto queue and does nothing, knowing that the send-thread will get to it eventually.

This did require careful implementation to avoid the risk of deadlock with starting and stopping the send thread (with the queue being stopped as it reaches what it thinks is the last message in the queue, just as a message is added without starting the queue thinking that it is running). Java synchronisation is costly, and I tried to avoid this affecting the performance of the services layer by avoiding using it in every iteration of the send-thread.

4.3.9 CONFIGURATION

The JXTA platform contains it's own security functionality, including the need to log-in through the built-in JXTA window-based authenticator every time I start the JXTA platform. As security is not part of this project, this username/password check was inappropriate and quite annoying. I researched the JXTA platform to find out where usernames and passwords are stored, and passed them in as Java System Properties when starting the application. As JXTA found the required username and password it no longer opened the Login window that was such a nuisance.

This does require that a username and password is already stored in the pse directory that JXTA creates, but as long as the project is delivered with this directory, then this does not pose a problem.

4.3.10 USE OF FAGAN INSPECTIONS

Another development technique that I used was to periodically evaluate the coding produced using the Fagan inspection method[30]. Fagan Inspections are a peer inspection technique developed by Michael Fagan – involving a four-man inspection team going through all code produced line at a time, looking for defects. Each inspector fulfilled a specific role – reader, author, moderator and tester. I have been on a comprehensive training course in the Fagan inspection process, and each inspector was a fellow Computer Science student.

[30] Full details of Fagan Inspections and the training I went on can be found at http://www.mfagan.com/

This process was very helpful, both in terms of finding defects in the code and as a chance to see what to do next. The process of explaining my work to others not familiar with the project made it easier to identify solutions to problems that I had been having.

4.4 APPLICATION LAYER

4.4.1 INTERFACE IMPLEMENTATION

Deciding to use PersonalJava meant that I had to implement the interface using AWT – the basic Java interface library, as there is no support for the more extensive Swing GUI libraries in PersonalJava. This was somewhat restrictive, but I was able to work around the limitations and implement the interface that I originally designed.

As described in the **Design** section previously (3.8), when designing the user interfaces I produced careful scale-drawings of the interface – showing the size and position of all elements.

As a result, I decided not to use the Java layout managers to arrange GUI elements in the frame. The layout managers allow developers to describe how screen elements should be arranged, which is suitable where you want to support different window sizes. However, as I had identified the pixel size of the target platform and knew exactly where I wanted GUI elements to go, I decided that layout managers would not be suitable.

Instead, I explicitly defined the pixel dimensions and coordinates for every screen element. For code readability, this was done using constants defining the height, width, and x,y coordinates of every group of elements and using the constants when placing or sizing. This made the code easier to read and maintain than a hundred lines of code with hard-coded meaningless numbers. It also made it quicker to make changes to the interfaces when I wanted to try something slightly different.

Creating a carefully laid out and designed interface did require me to learn how to use AWT properly, but most problems encountered during this work were syntactic in nature. One of the biggest problems I encountered was because I did not realise that AWT window coordinates refer to the window as a whole, including the title bar and border. My initial solution of measuring and then adding the required offsets in the code led to problems. These occurred when I tested the GUI on a different platform as the window properties such as title bar and border dimensions are determined by the system window manager and vary from platform to platform. My final solution determined the border and title bar and border dimensions dynamically and used them to create the offset.

Figure 9 shows a screenshot of the first version of the chat-room interface. It was obtained after running the interface class in the PersonalJava emulator discussed previously.

As well as running the application in the emulator, I also checked the code regularly using JavaCheck to ensure that I was writing code that conformed to the limitations of the target platform.

With implementing all of the applications, my initial aim was to create an application capable of running the interface, interpreting user input and packing them as network messages ready for sending using the services layer, and receiving, processing and interpreting incoming network messages – but without actually using any services layer functions. Instead of sending messages using the services layer, I passed the message directly to the receive message functionality.

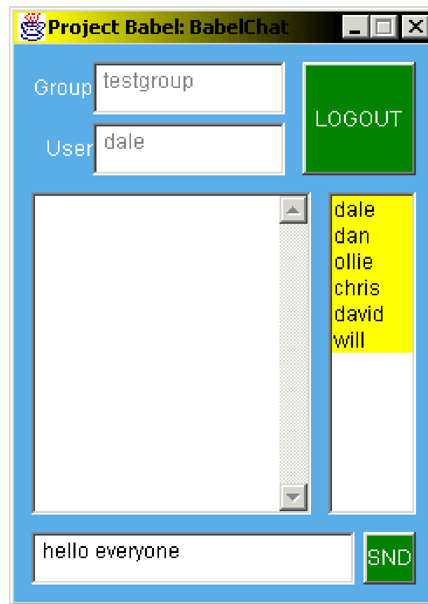


Figure 9 - Chat application interface running on PJEE under MS Windows

By effectively sending messages to myself without the use of the services, I developed and worked out problems with the application as a stand-alone program before using it as a network application. I then added calls to the services layer to enable sending and receiving of messages.

Particularly with applications such as the whiteboard application, this was very helpful to verify the paint canvas engine before considering the network implications. By avoiding the mouse action directly calling the graphical method to draw to the canvas, I ensured a more network message-friendly design – without hard coding in the draw operations to the mouse actions. Instead, when the application determined that something needed to be drawn (when a user mouse drag or click was detected) it packaged the information into a message. This was then unpackaged and used to draw. While inefficient for a stand-alone whiteboard application, this approach also meant that integrating the application with the services layer would require minimum effort.

4.4.2 CHAT ROOM IMPLEMENTATION

4.4.2.1 APPROACH

The implementation of the chat room was relatively straightforward. This was partly because the prototype provided a learning experience, making the development of the final application more of an exercise in creating a usable interface, taking lessons from using the prototype. The functionality was the simplest of the applications to implement as most of the work is done by the services layer.

4.4.2.2 IGNORE

One function added to the chat application was the ignore feature discussed in the requirements analysis. As mentioned previously, I had decided to implement this as an application function – acting as a filter to incoming messages. By allowing the user to add names to a list they want to ignore, the application can check the sender of incoming messages against the list before displaying.

I implemented the ignore function interface as a separate dialog box – as I felt that this wasn't something that a user would want to do often enough to justify using any of the limited screen space, and combining it with the user list (which is already used to select which users should receive a message) would have been confusing.

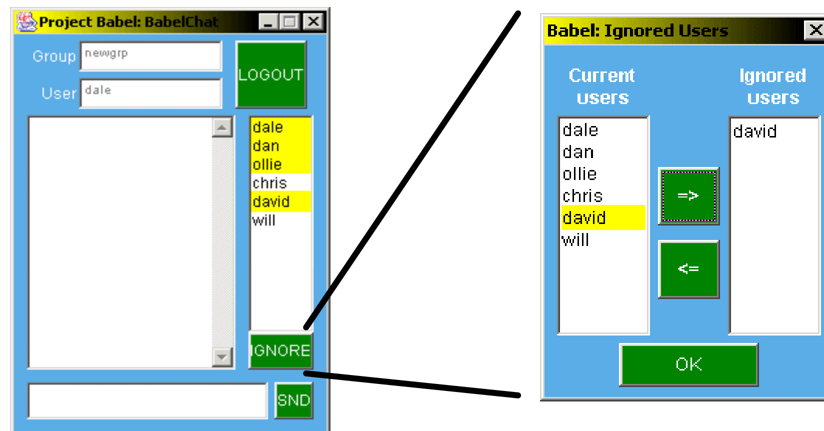


Figure 10 - Implementation of IGNORE function in the chat-room

The interface provides lists of current users to choose from, and users being ignored.

Although I implemented this as an application function, as it was a feature that could be used by any application, I implemented it as a separate class – BabelAppIgnoreList, and provided a standard interface, allowing it to be used by any application which wanted to implement an ignore feature.

4.4.2.3 MAINTAINING USER LIST

As discussed previously in **Prototyping** (4.3.1.4), the Services layer is responsible for providing the discovery methodology capable of producing a list of local users. However, it is the responsibility of the application to ensure that this is used frequently enough to maintain and display a user list.

This was implemented using a timing thread which repeatedly refreshed the user list displayed in the GUI with the latest information from the services layer. I was unable to find an efficient method of merging these two user lists (the list of users currently displayed, and the list of currently logged-in users returned from the services layer). I needed to ensure that any users which were being displayed but were not included in the services layer list were removed from the GUI, and that any users mentioned in the latest services layer update but not currently being displayed were added to the GUI. As this needed to happen quite frequently, I decided that the optimum approach would be to discard the user list and create a new one every time a new list was obtained from the services layer.

4.4.3 WHITEBOARD APP IMPLEMENTATION

4.4.3.1 APPROACH

The application engine for the whiteboard was more extensive than the chat room because of the need for handling of interactive graphics. This required me to learn how AWT handles graphics.

My approach to using the existing services layer functionality with a whiteboard application was that I would define a canvas, within which user mouse clicks would be detected. On detecting a

mouse click or drag, the event listener would grab the (x, y) coordinates of the mouse pointer. These could then be packaged into a message ready for transport by the service layer.

4.4.3.2 PAINTING

I implemented the paint canvas used by the whiteboard in a separate class (BabelPaintEngine). After considering alternatives, my approach was to use Java listeners to detect mouse clicks and drags, and using mouse coordinates to draw a shape with `fillRect` or `fillOval`.

The interface allows the user to choose what style and size of "pen nib" they want, and these form the widths and heights given to the fill method and choice of fill method. I implemented four styles of pen tool – a circle (fillOval with size specifying both height and width), a square (fillRect with size specifying both height and width), a vertical line (fillRect with width of 1, and size specifying height) and a horizontal line (fillRect with height of 1, and size specifying width).

I then refreshed the canvas using the repaint() method, revealing the user's marks.

This did lead to a problem – the mark where the user clicked or dragged was drawn on screen, but this did not remain when the user drew on the canvas somewhere else. This gave the effect of a pen nib following the mouse pointer around the canvas that leaves no trail.

This was a result of being limited to using AWT. In Swing, any drawing actions are stored to memory, and then displayed to the window. However, there is no buffering in AWT, and so when the screen is redisplayed after the repaint any earlier changes are not remembered – only the most recent. Without being able to use Swing, I could think of two possible options – firstly, implementing a buffer myself, where as well as drawing to the screen I also saved to an image buffer, or secondly, finding a way to avoid refreshing the whole screen.

I decided to use the second approach, as it seemed the most efficient way. My solution was to override the implied repaint command with a selective refresh of a particular section of the canvas – only the part of the canvas that the user most recently changed. By using the same coordinates used in the fill method, I could ensure that only the changed area of canvas would be refreshed when the canvas was "repainted", and that the rest of the image was left untouched.

The lack of buffering does mean that if another window obscures the canvas, or if the whiteboard application window is hidden or minimised, then the diagram is lost. While this seemed like a problem on the multi-tasking, multi-window desktop machines used during development, it is not a problem when running the application on the target platform. When running on a PDA, it will be the only application running, with no other windows to wipe the screen – so I did not try to implement any additional buffering to correct this.

4.4.3.3 PERFORMANCE

As discussed in the earlier section on **Optimisations** (4.3.8), the initial performance of the whiteboard was quite poor, with large breaks in lines drawn that never appeared on other peer's screens. This led to the implementation of the message queue mentioned earlier. While this meant that the whiteboard was no longer as close to real-time as before, however it did mean that it created a more accurate representation of what a user sketched.

4.4.4 FILE TRANSFER APPLICATION

4.4.4.1 APPROACH

The interface for the file transfer application was quite complicated because of the amount of information and number of controls that needed to be displayed to the user on a very small screen. The interface design is shown in the **Design** section previously (3.8.4.5), and was implemented in the same way as the other applications.

My general approach to implementing the file transfer application can be found in the **Design** section (3.6.2), which includes class diagrams and pseudo-code. This section contains descriptions of changes and additions made to that design.

Testing during development started with sending small text files at first, to confirm messaging and interface worked correctly. Then I started sending larger files – mainly images, which provided a nice quick visual way to see if a file had been received correctly. Corrupted files were evident as images with random use of colours, or with scan lines shown in an incorrect order.

4.4.4.2 SENDING FILE SEGMENTS

To avoid the problems encountered with the whiteboard, where messages went missing before being sent, I decided to make use of the message queue functionality that I added to the services layer while developing that application. The message queue was better suited to the file transfer application where accuracy is even more vital, and real-time streaming is not important.

I divided the file into segments by reading the segment size's number of bytes from a FileInputStream containing the file to be sent. Each chunk read from the File stream was packed into a Babel message and added to the send queue ready for sending. Then the file pointer was incremented, allowing the next read from the file stream to grab the next segment along. This was repeated until the end of the file was reached.

The size of the segments that the file was divided into seemed to have a noticeable impact on the performance of the application. In particular, the size of the file segments affected the time taken to transfer files and the proportion of messages to go missing (a problem encountered which is discussed at greater length below – 4.4.4.3). My initial hypothesis was that I was creating messages that were larger than the TCP frame size used for transport. As a result, my messages were being further broken down into segments by the lower layers – increasing overhead and workload, and resulting in poorer performance. I verified this by experimentation – trying out a variety of file segment sizes a number of times to gauge which size resulted in the optimum performance.

I eventually settled on a segment size (which was indeed below the TCP frame size) that resulted in the best performance.

Originally, the file segments were being manipulated as strings, because of the easy availability of string manipulation methods. However, while this worked with text documents, when used on binary data files such as executables, audio files and images, it resulted in the file being corrupted. The effect was similar to sending files over FTP using ASCII transfer mode instead of binary.

This was corrected by breaking the files into byte arrays instead of substrings, and using array offsetting and traversing instead of string concatenations to manipulate them. The resulting code was more complicated, but avoided converting the data to a string that was causing the problem.

4.4.4.3 RECONSTRUCTING FILES

I tried a number of approaches to this. The first, and simplest, was creating a stream to store incoming file segments and appending each new segment to the end of the data stream when it arrived. When the last segment arrived, I wrote the data stream to a file.

While this worked with small files and was a useful quick test of the interface and messaging, it was not acceptable as a solution. Large files, where the file handler sent many messages, led to the receiving peer receiving file segments out of sequence as message transport time varied for each message. This meant the file was being reconstructed with segments in the wrong order.

As a result, I needed a way of sorting the segments into a correct order before writing to a file. To do this, I first added a header to every file segment, giving it's packet number and the start and stop byte addresses of the segments place in the file. I then implemented a buffer in the file handler class where all incoming files segments were stored until they were all present, when they could be sorted into the correct order, using the message headers as a guide.

4.4.4.4 RETRANSMISSION REQUESTS

Although this did lead to small to medium files being reconstructed in the correct order, there were still problems. The larger the file being sent and the more messages required to be sent, the more messages went missing.

When writing the requirements for the chat-room and whiteboard applications, I stated that reliability was not a major concern. In a chat room, if a user does not receive a message, the sender will send their message again when they get no reply. This is not a major cause for concern.

However, for a file transfer application, reliability is vital – messages going missing result in corrupted files. As a result, I implemented additional functionality to the file handler – so that it could deal with missing messages.

My approach to missing messages was to use the confirmEnd-of-File message included in my design. The message contained details about what packets the receiver should be expecting. I added functionality so that the receiver checked the contents of the file segment buffer (containing all received segments) to make sure that there are no missing packets, making a list of any segments found to be missing. This list is then sent to the sender as a retransmit-request requesting that the sender send just those missing file segments again.

The file sender then re-opened the file, and offset the required number of bytes into the file to create the file segments required. Once all retransmits are complete, it resends the confirmEnd-of-File control message, and the process begins again, (allowing the receiver to check if there are still any packets missing, and send a new retransmit-request list again if necessary). This is repeated until no packets are found missing, when the sorting and reconstruction can begin.

4.4.4.5 RECEIVING FILE TRANSFER REQUESTS

Another issue raised during development is that the use of a single FileHandler for sending and a single FileHandler for receiving means that each peer can only send one file at once, and receive one file at once. As a result, the control messaging needed enhancing so that if a user tries to send a file-transfer-request to a user while they have a file receive in progress, they receive an automated reply informing them that the user is busy and to try again later.

I also added safeguards to the file handler so that if file segments of a different file are received from a different user while in the middle of receiving a file, then these segments would be discarded, so as not to corrupt the file being received.

The nature of the file handler – whether being used to break a file into multiple segments and send, or being used to collect segments and group into a file, was handled by having two constructors for the **BabelFileHandler** class – one for when being used as a receiver and one for use as a sender.

4.5 USE OF TOOLS

4.5.1.1 INTRODUCTION

I used a large number of tools during the development and implementation of the project, some of which have been discussed previously. This section gives examples of some of these, describing the rationale for using them and explaining how they were used.

4.5.1.2 EMULATOR

The PersonalJava Emulation Environment[31] (PJEE) provided an opportunity to develop in PersonalJava without needing repeated access to the target environment. It helped me to verify that the code I developed would run on an implementation of the PersonalJava application environment.

Developing on desktop machines, while repeatedly loading my compiled code onto a PDA and running it on the target platform would have been impractical and time-consuming. The emulator allowed me to test-run my code on the desktop machines used for development – allowing for a much faster turn-around of testing, identifying problems, correcting them and retesting.

4.5.1.3 LOGGING

My initial approach to debugging was to use `println` commands to display all logging debug information and error output to the standard output – the terminal where the application was started. However, this approach was not ideal for a number of reasons:

- `System.out.println` is a costly method – particularly when trying to optimise the whiteboard, I noticed that print statements were responsible for some of the lag.
- When running on the target platform, the standard output is not as accessible – without a multi-window environment such as the desktop machines that I developed on, it is not possible to see both the application interface and the terminal where the application is run. Therefore, printing to standard output was not a suitable way of displaying errors and debug information for the final product.
- Applications like the file transfer application have a lot going on at any one time with the file handler breaking up files into segments, sending, checking, rebuilding files, checking for and requesting missing segments and much more. During development, with so much happening it was vital to know what was happening, so extensive use of debug output was required. However, during usual operation these

[31] More information about the emulator can be found at http://java.sun.com/products/personaljava/pj-emulation.html

debug statements needed to be easily and quickly removed. Without a pre-processor to specify whether or not DEBUG lines should be printed, this would mean repetitive editing of code to comment-out or delete unwanted print statements.

As a result, I decided to use Log4j[32], an open source project, to manage logging. Log4j offered the solution to all of these problems, and is fully configurable at runtime using external config files.

The biggest advantage of log4j over plain System.out.println was its ability to disable certain log statements while allowing others to print unhindered. I divided my messages into groups of importance (debug, info, warn, error, and fatal) and could specify whether all should be printed, just errors, just warnings and errors and so on. I could also set logging priorities for individual classes, or groups of classes. As a result, simply by altering a configuration file I could reduce the large amount of debug information printed by the Services Layer after testing showed it to be stable enough to support application, while increasing the debugging output from the application classes.

Another major advantage was that log4j allowed me to specify where debug information should be sent – to a terminal for displaying in real-time, and/or to a log file for examining after testing. This was very helpful during testing, and is well suited to the target platform where terminal printing is unsuitable. In this way, once complete I can alter the config file to specify that all logging be sent to a log file in the event of any problem. This log file can then be studied after any problems to determine the source of the error.

The output produced by the logging was also very helpful, displaying the class name and line number of the logging statement as a prefix to every line of logging output – which would have required extra work to accomplish using traditional println statements.

Finally, log4j is a very fast efficient system, allowing me to avoid many of the overheads of the traditional println statements[33].

4.5.1.4 JXTA SHELL

The JXTA Shell[34] was very useful during my initial investigation into the JXTA Platform. The Shell provides an application that illustrates the use of JXTA Technology, and was a useful tool to understand the use of peer-to-peer computing and the nature of the JXTA platform. The Shell permits interactive access to the platform building blocks through a simple, command-line interface.

By providing a peer with which I could create, publish and join peer groups, and send and receive messages, it also proved a useful tool during initial development to try out approaches to these tasks, and check if they worked by running them alongside the Shell. For example, to confirm that my services layer was creating and publishing groups correctly, I could run the Shell search command to see if the Shell peer would find the group created by my code.

---

[32] More information can be found about log4j from the Project website: http://jakarta.apache.org/log4j/

[33] A discussion of Performance of the log4j package can be found at http://jakarta.apache.org/log4j/docs/manual.html

[34] A technical overview of the JXTA Shell can be found from the Project JXTA website at http://www.jxta.org/project/www/docs/TechShellOverview.pdf

In the event of problems with my peers created by my services layer not being able to "see" each other, I would be able to identify if the problem was in publishing a peer's presence or detecting it – by using the Shell as a control node for comparison.

Once the services layer was stable, and I started to use the applications instead, my usage of the Shell decreased – however it was an invaluable tool during the early development.

4.5.1.5 TCPDUMP

As discussed previously (4.3.8), the development of the whiteboard and the file transfer led to concerns over the efficiency of my services layer. Before deciding how to alleviate these performance issues, I wanted to understand the source of them. To do this, I used tcpdump and other network monitoring tools to examine the quantity and size of packets being sent and received across the network. By writing such output to a log file, and studying it in depth, I was able to understand better why so much information was being sent to affect performance, and how this could be corrected.

It was also useful when determining the optimum size for file segments in the file transfer application. The file handler class, responsible for chopping up files into segments for transfer, determined the size of each segment – affecting the load placed on the network. Too small, and the size and speed overheads of transporting each message outweighed the information carried by each message, leading to the receiver being flooded with too many messages – each one requiring processing by the receiver, with the time that this entailed. Too large, and each message would need to be further broken down by the lower transport layers, with the overheads of reassembling each message that this entailed. I found the optimum size through experimentation, and network-monitoring tools such as tcpdump were essential in this work.

4.6 OTHER PROBLEMS

4.6.1.1 JAVA

As discussed previously (2.3.2.1), Java was very well suited to this project for a number of reasons. However, when starting the project I had not used Java before. As a first attempt at using a language, the project was therefore hindered by my lack of knowing how things could be implemented and what was available to me. In retrospect, my progress would have been quicker had I selected a language with which I had some experience.

4.6.1.2 JXTA

One of the biggest initial problems faced in the implementation of this project was learning about the JXTA platform. The JXTA project, and the related work produced by the JXTA community, is growing at an amazing rate. At the time of writing this document, the JXTA was only a year old, and in the seven months from initial research to completing this project, I saw the JXTA website and documentation grow and expand.

When I began using JXTA, there was a distinct shortage of supporting materials or documentation. There were few articles, examples or tutorials on how to use JXTA or it's capabilities – only press releases and discussions that were relatively free of any technical content. Another problem was that as JXTA is advancing so quickly, much the documentation that I was able to find was out-of-date and somewhat misleading.

This meant that I had to teach myself a lot of how JXTA works, by looking through source code, reading the Javadoc API documentation, and experimenting with it. Initially a large amount of development time was spent doing this – learning how JXTA works, what it's capabilities are and how to use it. Many of the aspects of JXTA that I investigated in my initial work are now only recently described in a variety of documents and manuals, while others are still difficult to find documentation for.

One information source that I did not notice until very near the end of the project was the Project JXTA mailing lists[35]. The mailing lists provide a chance for developers to ask questions and compare ideas. I found that the answers to many of the problems that I spent considerable time working through and finding the solution to myself could be found in the mailing list archives. By encountering problems that other developers had encountered before, and ignoring how they had solved them, I was in some ways re-inventing the wheel each time. While this was not intentional, and purely a result of the fact that I was not aware of the presence of these mailing lists, I do believe that from an educational point of view, I learnt more from solving these problems myself than I would have done from reading and implementing other people's solutions.

In addition, after discovering and subscribing to the mailing lists, I have found that I have been able to answer a number of JXTA queries from other developers myself, and have become an active contributor to the *user* mailing list.

Another problem was the over-emphasis on the JXTA Shell that I found in virtually all JXTA tutorials that I was able to find. The JXTA Shell mentioned earlier provides a command-line interface to the JXTA platform. The Shell is easily extensible, and additional user-defined commands can easily be integrated to the core functionality. While it was useful as a test aid, and as a way of experimenting and learning more about JXTA, I did not want to incorporate it into my project. Most tutorials for writing JXTA applications rely on being incorporated into the Shell, and I could find no examples of stand-alone applications written to use the JXTA platform without using the Shell. Finding out how this could be done therefore took some time to investigate.

I wanted to avoid writing my project as an extension to the Shell because I felt that, while useful as a demonstration tool, the Shell was too broad for my requirements. It was more efficient to create my own network engine to match my own application requirements, and not have to include the un-required, unnecessary functionality included in the very powerful Shell application.

4.7 CONCLUSION

4.7.1 OVERVIEW

To summarise, the project produced the following:

- Relatively stable network services layer
- Effective file transfer application – reliable and able to deal with missing packets
- Reliable multi-user chat room and instant messager

[35] The Project JXTA mailing lists can be accessed from the project website at http://www.jxta.org/project/www/maillist.html

- Multi-user collaboration idea storm network whiteboard

4.7.2 IMPLEMENTATION BREAK-DOWN

4.7.2.1 INTRODUCTION

The implementations of these have been described in detail in this section, however the following list summarises this implementation.

4.7.2.2 SERVICES LAYER

The services layer consists of the following classes:

- **BabelServices** – main services class providing the application API and access to services layer network functionality.
- **BabelMessage** – implements instances of messages for the Babel network.
- **BabelGroup** – implements instances of peer groups, with functionality to create, join and publish peer groups (including generating groupID codes).
- **BabelListener** – implements the listening thread, constantly listening for new incoming messages and processing them on arrival.
- **BabelTimer** – implements timer thread responsible for publishing user's presence (based on a timer, which republishes advert when the expiry time passes).
- **BabelFileHandler** – provides functionality to deal with sending large items of data (used by the File Transfer application) – either splitting a file into segments and sending individual segments, or receiving file segments and reconstructing them into a file. Supports requesting retransmission of any missing segments.
- **BabelMsgQueue** – implements a queue of outgoing messages waiting to be sent. Wakes up send thread if not already active.
- **BabelSendQueue** – implements a thread responsible for sending messages waiting in BabelMsgQueue. Puts itself to sleep when all messages in queue are sent.
- **BabelDefinitions** – inherited "header"-file containing definitions and constants used throughout services layer for consistency and code-readability.

4.7.2.3 APPLICATION LAYER

The application layer consists of the following classes:

- **BabelChatApp** – implements the chat room application, providing the user interface, and access to services layer
- **BabelChatListRefresh** – implements a timer thread, responsible for periodically refreshing the chat-room's list of local users

- **BabelSketchApp** – implements the whiteboard application, providing the user interface, and access to the services layer

- **BabelPaintEngine** – extends the Java `canvas` class to provide a whiteboard graphical engine – intercepting and interpreting user mouse commands, displaying user sketches, and providing different "paint tools"

- **BabelWhitListRefresh** – implements a timer thread, responsible for periodically refreshing the whiteboard's list of local users

- **BabelFileApp** – implements the file transfer application, providing the user interface, and access to the services layer

- **BabelFileListRefresh** – implements a timer thread, responsible for periodically refreshing the file transfer application's list of local users

- **BabelAppGenericErrorDialog** – extends the Java dialog class to provide a project-custom dialog message box for displaying error and information messages

- **BabelAppIgnoreList** – implements filter to ignore incoming messages from unwanted users, providing standard interface allowing it to be used by any application

4.7.3 EVALUATION

Notice that the listing of implemented classes above is longer than that given in the **Design** section previously (3.4.3). All additions made during implementation have been detailed in this section, however they are mainly a result of requiring separate classes to manage threads, and the addition of the file handler, timer and message queue functions in the services layer.

Overall, however, the implementation went ahead without any major problems or needing major changes to the original design.

5. RESULTS AND EVALUATION

5.1 OVERVIEW

This section provides an overview of the testing carried out as a part of this project, and an evaluation of the system that the project has produced.

The testing subsection (5.2) explores whether or not the system produced meets the functional requirements for which it was developed – checking whether the system is capable of carrying out the tasks specified.

The evaluation subsection (5.3) explores how well the system performs at these requirements, and discusses whether the system meets the non-functional requirements.

5.2 TESTING

5.2.1 DURING DEVELOPMENT

Testing during development was carried out on a LAN of desktop machines, running on both Linux and MS Windows computers of varying processor speeds and memory capacities. The project code was run on the PJEE emulator on all machines. Although the emulator provided the ability to limit memory use (with configurable heap and stack sizes) to mimic the performance of the target platform, I wanted to confirm that the project functioned well on machines of varying capabilities and that I was not developing to a machine of a certain speed and abilities.

Testing during development was informal – simply running the applications, trying to test each area of functionality. Problems encountered were recorded in a list of issues to be investigated.

5.2.2 FINAL TESTING

5.2.2.1 FUNCTIONAL REQUIREMENTS

Testing after completion was more formal, with a detailed test plan covering all functions specified in the requirements documents.

The testing was carried out with a variety of network environments – first on the desktop machines used in development across a LAN. The collection of test cases to test all of the application and service layer functions formed a TestSet. The TestSet was run six times to measure the performance under different network environments.

The distribution of TestSets is shown in Figure 11. The aim of the testing was to confirm that the applications correctly carried out all of the operations included in the application requirements – with no reference to the performance or quality of this operation.

All tests passed successfully – confirming that all applications satisfied all specified functions listed in the requirements documents. (There were some performance concerns over the whiteboard application, however this is discussed in the following section.)

	App run with **2 peers**	App run with **4 peers**	App run with **6 peers**
Network load – low (LAN not being used)	TestSet 2A	TestSet 4A	TestSet 6A
Network load – high (LAN being used to transfer multiple files between machines while running test)	TestSet 2B	TestSet 4B	TestSet 6B

Figure 11 - Final Test Case table

5.2.2.2 TARGET TESTING

A variety of testing was planned to confirm that the system satisfied the requirements for compatibility with the PersonalJava API.

Unfortunately, complications with incompatible Java virtual machines meant that I was unable to complete the target testing on the Compaq iPAQ target platform discussed previously (2.4.4). Instead, wearable computers running Windows 2000 were used. While the specifications of these machines was closer to desktop PCs than the PocketPC environment originally envisaged, they still provided a similar interface to that intended, with a portable screen and stylus. Another interface difference was that the different screen proportions meant that my applications ran in a window instead of filling the screen. Again, while this meant that the testing was a different experience to that originally intended, it was still similar enough to make testing and evaluation using them valid.

Most importantly, they provided an opportunity to test the performance of the system when running on a wireless network – using Bluetooth network cards.

The target testing carried out included:

- Checking all classes produced using the JavaCheck tool

  Result: All classes were confirmed, with only minor warnings due to file I/O support being optional in the PersonalJava specification

- All testing in the previous section (5.2.2.1) carried out using PersonalJava Emulation Environment

  Result: All tests carried out without errors or warnings due to compatibility

- Running all TestSets on the target platform.

  Result: All tests carried out without any errors or warning encountered due to compatibility. All operations could be completed correctly, (there were performance issues, however these are described in **Evaluation** – 5.3)

5.2.3 USE OF LOGGING

The **Implementation** section (4.5.1.3) discusses my decision to use log4j as a logging tool. This was a very effective aid during testing, helping me to understand and verify what the system was doing and trying to do at any time.

In addition, in the event of errors, bugs or other failures, logging provided detailed context for the failure, aiding the correct and speedy identification of the problem source. The use of logging was an essential complementary tool in my testing.

5.3 EVALUTATION

5.3.1 PERFORMANCE

5.3.1.1 OVERVIEW

In this section, I will provide a brief evaluation of the performance of the applications produced, and the underlying services layer. This evaluation is divided into the various non-functional requirements identified during the initial research and requirements analysis stages.

5.3.1.2 STATIC

Each implementation of the application supports as a single user, as required, without causing an excessive load on the platform.

The network created by the system should not enforce a limit on the number of users that can be in a peer group simultaneously – the only limit on this should be limits imposed by the hardware and network services and/or protocols. This has not been tested for practicality reasons – the closest to stress testing carried out in this respect was to run a chat-room with ten users (although due to hardware shortages, four were run using different port numbers on the same physical machine).

The size of the application and required configuration files exceeds 2Mb. While this seems quite large, a satisfactory maximum size limit was never specified.

5.3.1.3 DYNAMIC

The up-time or reliability requirement of the system was not specified precisely – only that it is required to be as high as possible.

The requirement for response time of the application responding to user commands was specified to be almost instantaneous – as close to appearing real-time as possible. This was satisfied during evaluation, with no noticeable delay to operations. (The notable exception to this being the lengthy login operation, which is reflected in the interface to show a busy status so that a response of some sort is given almost instantaneously).

5.3.1.3.1 Accuracy

5.3.1.3.1.1 SERVICES LAYER: Data transfer accuracy

My first evaluation of the efficiency of the system was measuring the accuracy of data transfer between peers, and how this is affected by the amount of data being sent.

This was tested using a custom-written test application, which is described below.

1. ***Master*** creates a test file of the required size (as described below).
2. A copy of the test file is placed on both ***Master*** and ***Slave*** peers

3. ***Master*** peer breaks the file into messages and sends the file segments to the ***Slave*** peer

4. ***Slave*** peer records:

   a. Number of file segments that do not arrive

   b. Number of file segments that are corrupted (file segments that differ when compared with the file segment obtained from the ***Slave***'s own copy of the file)

No efforts were made to retransmit missing packets, as this was an evaluation of the services layer message efficiency only. The test application also allowed the user to choose whether the sender uses the message queue in the services layer (discussed previously – 4.3.8), as reflected in the results.

The file was created using a small custom-written Java utility. I wrote this as a small command-line utility capable of creating files of a specified file size, and filling it with a pseudo-random sequence of bytes.

This was carried out both on the development platform (using fixed-cable connections) and on the target platform (using a Bluetooth connection). The development platform evaluation was carried out with the system supporting six chat-room users, while the target platform evaluation was carried out with only two peers (due to hardware availability issues).

Figure 12 shows an example of the sort of statistics collected, showing the repetition of the test using a range of file sizes, and size of segments that the file is broken up into. The results shown are the result of repeating each test three times, and recording an average.

File size	Packets	**Network load – low** (LAN not being used)							
		Without message queue				**Using message queue**			
		128byte packets	1024byte packets	2048byte packets	4096byte packets	128byte packets	1024byte packets	2048byte packets	4096byte packets
5K	lost	0%	0%	0%	0%	0%	0%	0%	0%
	corrupted	0%	0%	0%	0%	0%	0%	0%	0%
100K	lost	2%	1%	0%	0%	6%	5%	4%	5%
	corrupted	0%	0%	0%	0%	0%	0%	0%	0%
500K	lost	23%	18%	12%	8%	2%	4%	5%	7%
	corrupted	0%	0%	0%	0%	0%	0%	0%	0%
1M	lost	26%	24%	24%	17%	1%	4%	4%	8%
	corrupted	0%	0%	0%	0%	0%	0%	0%	0%
5M	lost	28%	26%	22%	16%	0.4%	2%	5%	6%
	corrupted	0%	0%	0%	0%	0%	0%	0%	0%

Figure 12 - Data Transfer Accuracy Evaluation Results (low network load)

The results show that the services layer alone is not very reliable – with the unreliability increasing as more data is sent.

The results also seemed to confirm what I discovered during development – that packing too much data into each JXTA message leads to an increase in the number of messages lost. The less the messages need to be broken up by the lower layers, the more effective they seem to be.

The testing also verified the benefit of the message queue in reducing the proportion of messages that are lost. It was interesting to see that the results seem to suggest that the effectiveness of the message queue acting as a buffer actually *decreases* as the packet size increases. With a queue, the correlation between proportion of messages lost and packet size is reversed.

5.3.1.3.1.2 APPLICATION LAYER: File transfer accuracy

My next evaluation of the efficiency of the system was to measure the accuracy of the file transfer application. This differs from the previous test method (5.3.1.3.2.2), which aimed to measure the efficiency of only the core data transfer service. This test aimed to evaluate the efficiency of the file transfer application when the retransmit of missing packets is used.

Again, the test file was created by the file creation utility described earlier (5.3.1.3.1.1), allowing testing using a variety of file sizes.

This was carried out both on the development platform (using fixed-cable connections) and on the target platform (using a Bluetooth connection). The development platform evaluation was carried out with the system supporting six chat-room users, while the target platform evaluation was carried out with only two peers (due to hardware availability issues).

The transferred files were compared using UNIX diff commands to verify accuracy. All files tested were transferred accurately – with the application retransmission method compensating for the unreliability of the services layer.

5.3.1.3.1.3 APPLICATION LAYER: Whiteboard accuracy

My next evaluation of the accuracy was to measure the performance of the whiteboard application. This was carried out informally, using the application to determine if the performance was accurate enough to be acceptable to end-users.

This was carried out both on the development platform (using fixed-cable connections) and on the target platform (using a Bluetooth connection). The development platform evaluation was carried out with the system supporting six chat-room users, while the target platform evaluation was carried out with only two peers (due to hardware availability issues).

The performance on single points or shapes drawn was acceptable in all cases. When drawing longer lines or shapes, the performance deteriorated. The evaluation on the development platform in these cases was acceptable. While some parts of lines would fail to arrive, the general shape or design being drawn was still recognisable. The evaluation on the target platform was noticeably slower, with unacceptable gaps and missing sections of drawings as the lines grew longer.

5.3.1.3.2 Speed

5.3.1.3.2.1 SERVICES LAYER: Message Round-trip time

My first testing of the speed of the system was measuring round-trip time for a single message.

The aim was to see how quickly messages created by an application can be passed to the services layer, processed and sent, and then how quickly incoming messages come up through the layers to reach the application. Any application, no matter how fast, will face this as an operational overhead and so it made sense to measure this.

It would have been impossible to accurately measure the travelling time for a single message because of the challenge of setting system clocks on two different machines to be synchronous. As a

result, I decided to measure the time taken for a message to be sent, and a reply to be returned – giving me a time that can be approximately divided by two.

This was measured using a custom-written test application, which is described below. The test application was written as efficiently as possible, to avoid the test application itself influencing the scale of the results collected. Examples of this include that messages are pre-prepared before timing begins, and that no printing to the standard output is carried out during timing.

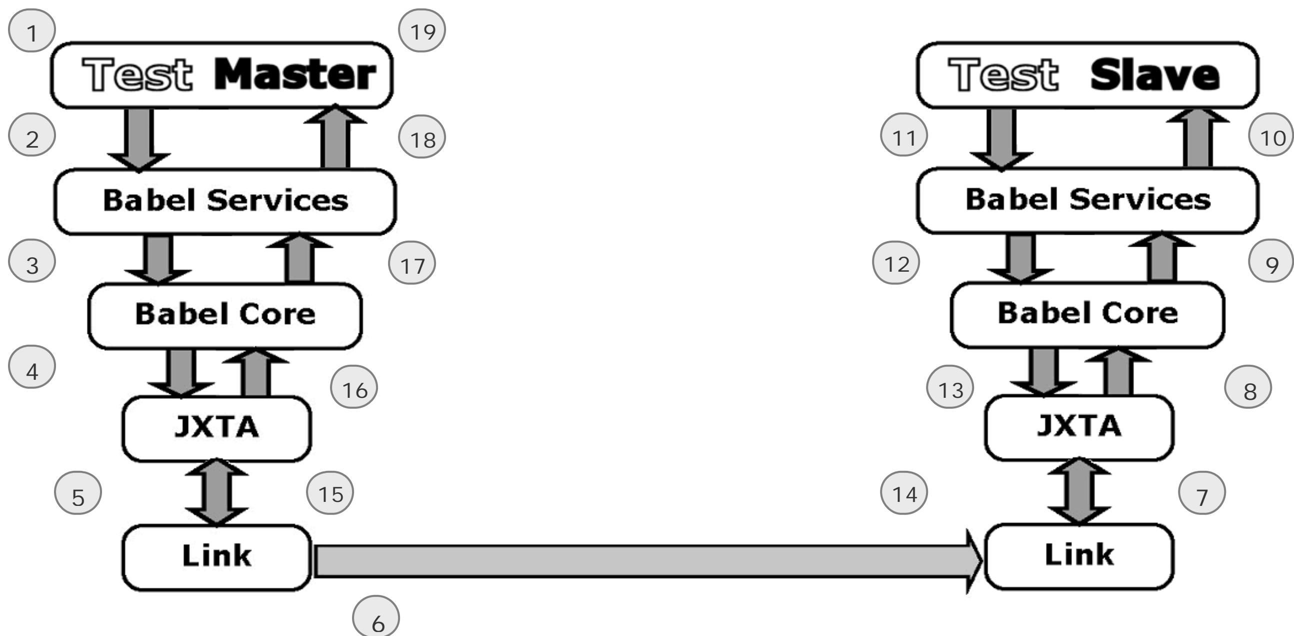


Figure 13 - Message round-trip timed test diagram

The above diagram depicts the test design. The following table describes the numbered stages.

1	The test application `MsgRoundTripTimer` on the ***Master*** peer creates and joins a peer-group, using functionality from the services layer. It creates a new empty test message and prepares it for sending. It prepares a connection to the ***Slave*** peer, creating both an input and output pipe and initialises them by sending a message.
2	The ***Master*** peer starts a timer – **the test starts here**. The empty test message is passed to the services layer for sending.
3	The Services layer in the ***Master*** peer sends the message.
4	The message is sent using a standard JXTA uni-directional, unicast pipe.
9	The message is passed to the listening thread blocking on the input pipe on the ***Slave*** peer, and is parsed.
10	The message type is identified by the services layer to be MSGTYPE_TEST, and is passed to the test application `MsgRoundTripSlave`
11	The ***Slave*** application immediately sends a **pre-prepared** reply message to the ***Master***
17	The reply message is passed to the listening thread blocking on the input pipe on the ***Master*** peer, and is parsed.

18	The message type is identified by the services layer to be MSGTYPE_TEST, and is passed to the test application `MsgRoundTripTimer`
19	The ***Master*** peer stops the timer – **the test stops here**, and the elapsed time is displayed.
	The test re-starts from **2** until interrupted by the user.

The table below contains the results obtained from running this test.

	Network load – low (LAN not being used)	**Network load – high** (LAN being used to transfer multiple files between machines while running test)
Average time for a round-trip (in ms)	144	155
Number of samples taken (number of tests run)	2870	2870

The average time for two empty messages to be sent between peers – was approximately 150ms.

Although the average time was fairly low, the statistics collected showed that round-trip times of approximately 3000ms (3 seconds) were occurring occasionally. I plotted the samples taken on a graph to see if any patterns would emerge.

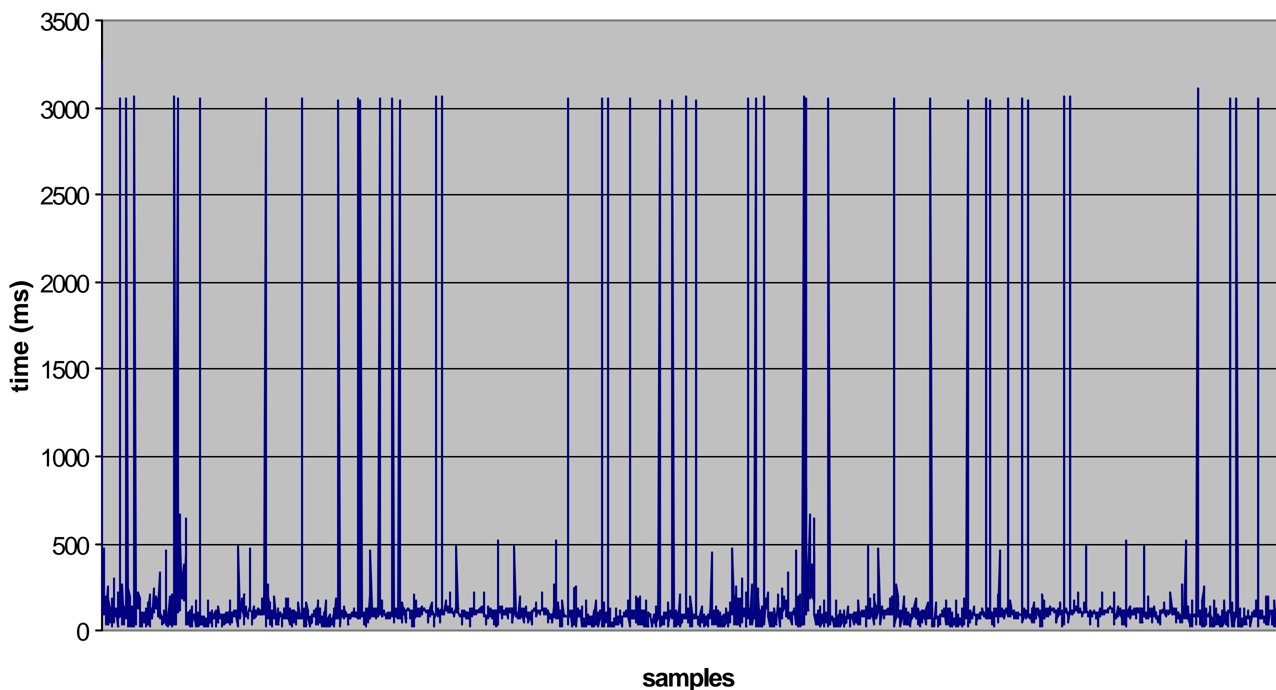


Figure 14 - Message round trip times (low network load)

Figure 14 confirms that the times were fairly consistent, except for the occasional spikes of approximately 3 seconds for a round-trip. This was an alarmingly high value, and was quite unexpected. For an empty message to take so long to travel to a remote peer and return would seem to suggest that there is a problem.

I was unable to identify the source of this problem, or identify any pattern in it's occurance. It seems possible that this problem could be, in part, responsible for the performance problems with the whiteboard application, and if further time allowed, I would have liked to investigate this further.

Repeating the test while putting the network under a heavier use seemed to have a negligible effect on the test result. The graph produced looked very similar to the graph produced under low network load – with similar occasional three-second spikes.

This would seem to support my assumption that the bulk of the round-trip time is spent going through the layers within the peer – through JXTA and my Services layer. The actual transport time is unlikely to be a significant proportion of the overall time, so increasing this by increasing the network load does not have a noticeable effect on the overall round-trip time.

Repeating the test with both Master and Slave peer running on the same computer – using different port numbers, further supported this. Running on the same computer means that messages are sent and received via loopback, rather than sending across a network. As this also produced a very similar set of results, it seemed to confirm that the physical transport time was a relatively small proportion of the overall round-trip.

5.3.1.3.2.2 SERVICES LAYER: Data transfer speed

My next testing of the speed of the system was to measure the speed of data transfer between peers, and how this is affected by the amount of data being sent.

This was tested using a custom-written test application, which is described below.

1. ***Master*** creates a test file of the required size (as described below).
2. ***Master*** peer breaks the file into messages
3. ***Master*** peer records the start time – **the test starts here**
4. ***Master*** peer starts sending the pre-prepared file segments to the ***Slave*** peer
5. ***Slave*** peer records a stop time once all segments received – **the test ends here**

Before starting the test, the two peer's system clocks were synchronised as close as was possible. Because of possible inaccuracies here, the test results were measured to the nearest second – measuring to the nearest millisecond (as with previous tests) would be of little value.

No efforts were made to retransmit missing packets, as this was an evaluation of the speed of the system and not the accuracy (which has been evaluated previously – 5.3.1.3.1). The test application also allowed the user to choose whether the sender uses the message queue in the services layer (discussed previously – 4.3.8), as reflected in the results below.

The received segments were not reconstructed into a file, because this test was not intended to focus on file-handling performance. Although a file was used as a source of data to send, this was for convenience only, and the test aimed to measure the time taken to send and receive an amount of data. What the receiving peer does with that data, (such as reconstructing into a file) is an application concern, and not related to the services layer.

The test files were created using a small custom-written Java utility described earlier (5.3.1.3.1.1).

Some of the statistics are shown in Figure 15, showing the repeating of the test using a range of file and file segment sizes. These figures are an average after repeating each test three times.

File size	**Network load – low** (LAN not being used)							
	Without message queue				**Using message queue**			
	128byte packets	1024byte packets	2048byte packets	4096byte packets	128byte packets	1024byte packets	2048byte packets	4096byte packets
5K	4s	3s	3s	0s	6s	4s	3s	1s
100K	15s	2s	1s	1s	18s	4s	3s	2s
500K	69s	10s	6s	4s	78s	15s	8s	6s
1M	133s	19s	10s	6s	141s	23s	11s	7s
5M	608s	83s	46s	27s	631s	87s	49s	30s

Figure 15 – Data Transfer Time between two peers (low network load)

These results were interesting, because they suggested that there might be some benefits to using larger messages. Earlier results (including those discussed previously - 5.3.1.3.1.1) suggested that using larger messages should be avoided as it led to an increase in lost messages. I believed that this was a result of the messages being broken up into multiple segments for transport – with the increase in number of messages flooding the network and causing an increase in lost packets.

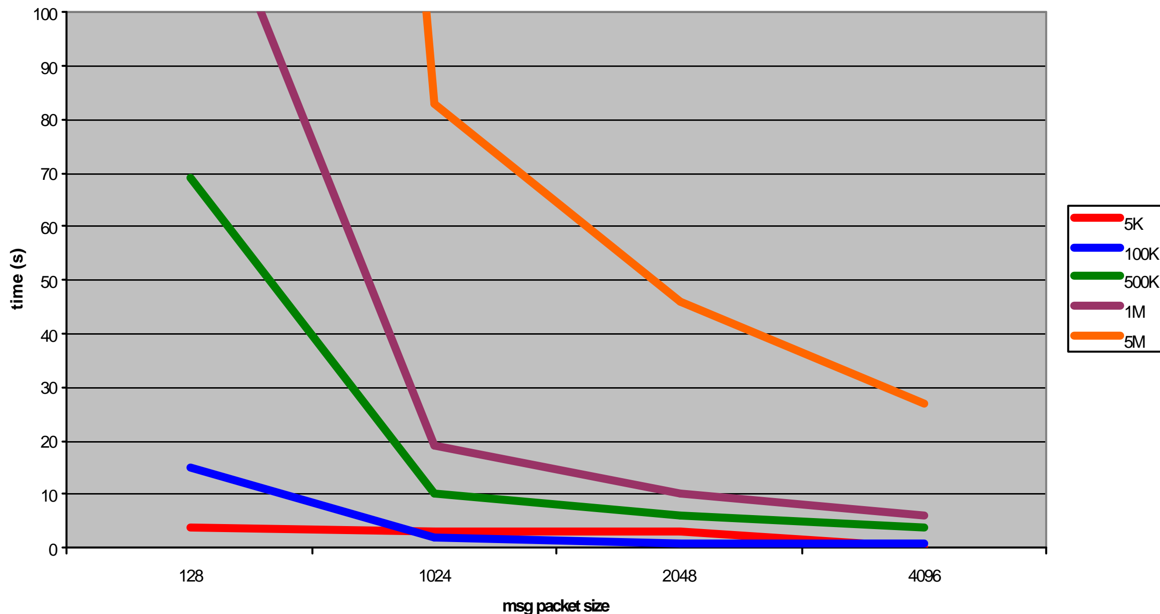


Figure 16 - Effect of message packet size on data transfer rate

However, the results obtained from this test suggested that the use of smaller packets (while more reliable) was much slower. Figure 16 above represents this quite dramatically – with the use of 128 byte messages causing the time necessary to transfer data packets to increase significantly. This could possibly be the true reason for the increase in reliability – that the slower transfer rate means

that less messages are lost. It also suggests that using larger messages might be more efficient – with potentially significant speed increases for the file transfer application.

The benefit of smaller packet sizes is that in the event that a packet is lost, less data needs to be retransmitted. However, the reduction in the amount of work needed to break up and reconstruct the file seems to compensate for this considerably.

The file transfer application currently uses a file segment size of 1024 bytes – these results would seem to suggest that this should be increased.

5.3.1.3.2.3 APPLICATION LAYER: File transfer performance

My next evaluation of the speed of the system was to measure the speed of the file transfer application. This differs from the previous test method (5.3.1.3.2.2), which aimed to measure the speed of data transfer only. This test aimed to evaluate the performance of the final provided system and included a measurement of the time taken to break a file into segments, send the segments, the time for the receiver to evaluate the received segments and request and receive any missing segments, and the time to reconstruct the final file.

Again, the test files were created by the file creation utility described earlier (5.3.1.3.1.1), allowing testing using a variety of file sizes.

All logging was turned off during testing, to ensure that time taken to print out messages or write to the log file did not affect the test results.

The original requirements discussed creating a system that would support a network of different speed devices. To look at the differences in performance that this introduces, the applications were run on two different machines – one 800MHz computer, and an older 300MHz machine.

File size	**Average complete transfer time** (including time required to request any necessary retransmissions and reconstruct file)			
	(slower machine to faster machine)	*(faster machine to slower machine)*	*(fast machine to fast machine)*	*(slow machine to slow machine)*
5K	1s	1s	1s	1s
100K	1s	2s	2s	4s
500K	8s	5s	11s	16s
1M	14s	21s	26s	30s
5M	53s	265s	180s	210s

Figure 17 - File Transfer Application transfer times

Some of the statistics collected are shown in Figure 17, showing the repeating of the test using a range of file sizes.

The results showed that the relative speed of peers had a significant impact on the file transfer speed:

- **Fast to Slow** - this resulted in a very slow transfer, as the faster peer sent the file segment messages at a rate far faster than the slower peer could receive. As a result, the proportion o

- **Slow to Fast** - this resulted in the fastest transfer, as the faster peer was able to cope with the speed of incoming messages the first time they were sent. Although messages were not sent as fast as a transfer between two "Fast" machines, as there was no need for retransmission of missing packets, a lot of time was saved.

- **Fast to Fast** - also led to many segment retransmissions, as the messages were received at the peer's maximum processing speed.

5.3.1.3.2.4 APPLICATION LAYER: Whiteboard performance

My next evaluation of the speed was to measure the performance of the whiteboard application. This was carried out informally, using the application to determine if the performance was fast enough to be acceptable to end-users.

This was carried out both on the development platform (using fixed-cable connections) and on the target platform (using a Bluetooth connection). The development platform evaluation was carried out with the system supporting six chat-room users, while the target platform evaluation was carried out with only two peers (due to hardware availability issues).

The performance on single points or shapes drawn was acceptable in all cases. When drawing longer lines or shapes, the performance deteriorated. The evaluation on the development platform in these cases was acceptable, although slow. The evaluation on the target platform was noticeably slower, with unacceptable delays on the transfer as the lines grew longer.

5.3.1.3.2.5 APPLICATION LAYER: Chat-room performance

My final evaluation of the speed was to measure the performance of the chat application. This was also done in an informal way, using the chat-application to determine if the performance was fast enough to be acceptable to an end-user.

This was carried out both on the development platform (using fixed-cable connections) and on the target platform (using a Bluetooth connection). The development platform evaluation was carried out with the system supporting six chat-room users, while the target platform evaluation was carried out with only two peers (due to hardware availability issues).

In all cases, the performance was determined acceptable – with no noticeable delays encountered in sending and receiving messages.

5.3.1.4 SECURITY

The intention of this project is to demonstrate the benefits of mobile peer-to-peer networking, not to provide a commercial secure chat program. As a result, there was no security testing for the system.

5.3.1.5 TRANSFERABILITY

The application should be compliant with the PersonalJava platform, and be capable of working on any device capable of supporting PersonalJava. This was verified using the JavaCheck class checking tool, and by testing on a PersonalJava emulator (PJEE) and PersonalJava device (see 2.4.4).

5.3.2 USER INTERFACE

5.3.2.1 OVERVIEW AND APPROACH

The aim of this section is to provide a brief evaluation of the user interfaces of the applications produced, identify any potential problems with the interface design, and propose any possible improvements.

Alan Dix[36] divided the concept of evaluating the usability of an interface into three main categories: learnability, flexibility and robustness. I decided to use this approach to evaluate the interfaces and in this section I will cover each of these categories in turn, and discuss how it affects the usability of the applications.

In order to gain a better understanding of the usability of the interfaces, I studied three different users' interactions with the interface. By selecting three different types of user, I hoped to gain a cross-section of the way that different users would interpret the usability of the interfaces.

The first user was a fellow Computer Science student, who is very experienced and confident with using the sort of applications that this project produced – including chat-rooms, instant messagers, IRC, Microsoft NetMeeting's whiteboard and FTP clients.

The second user was also a fellow Computer Science student, who was less experienced with such a variety of communications applications. He had used two chat applications before (MSN Messenger and ICQ) and was relatively confident with the idea and use of chat-rooms and instant messagers, but had limited experience. However, he was very experienced with the use of PDA devices – having both used and owned PalmOS and PocketPC devices before, and had a number of opinions about effective and ineffective PDA user interface approaches.

The final user was a computing novice who has used computers and Windows interfaces before, but does not have a great understanding of them, and minimal competence. She was unfamiliar with any other chat-rooms, messagers, file transfer programs or other such network applications.

I evaluated the interface by watching each user trying to learn how to use the application and asking them what they thought while they did so. Afterwards, I also asked what they thought of the interface in an informal interview – asking about their response to each of the usability principles.

I also recorded videos of the evaluators using the interfaces, for closer examination. This also allowed me to question the evaluators after the testing. Without the concern of disturbing them while they were using the interface, I could get more detailed explanation of their actions. Some examples of the videos I recorded can be found on the attached CD-ROM in AVI format.

[36] "Human-Computer Interaction" – Alan Dix, Janet Finlay, Gregory Abowd, Russell Beale (1997)

The following list details some of the usability aspects that I considered during this process, and used to direct the questioning of the evaluators. It includes rationales of how some of the usability principles relate to the interfaces produced, and how effective they are.

The list is followed with a conclusion (5.3.2.5.1), summarising the evaluators' comments and considerations. This is presented as a list of heuristics, together with a summary of the comments made (5.3.2.5.2).

5.3.2.2 LEARNABILITY

Dix identified learnability as being the properties of an interface which both allows users to be able to use the system initially when they are new to it, and to allow more experienced users to attain a high level of understanding and performance. He identified five principles that support this and I examined the interfaces using each of these principles.

5.3.2.2.1 Familiarity

One way to improve the learnability of an interface is to use the experience that a user will already have – both from the real world and from using other software applications. Also known as guessability it refers to properties that an interface has that enables the user to guess how the system will act, and how they should interact with it from a first impression of the interface.

I applied this principle to the interfaces as explained above (5.3.2.1). The familiarity of the interfaces was most obvious in the users' initial reaction to them – seeing how they guessed the interface would behave before they had a chance to work it out through experimentation.

The group management controls, used in all three interfaces, seemed to be quite clear to all evaluators – even from a first impression, and all users knew how to log-in and log-out easily.

The chat-room and whiteboard applications continued this – with most of the evaluators able to recognise and identify the major GUI elements and their meaning at a glance. However, there were some exceptions to this: while all evaluators realised that the user list displayed in the chat application was showing them the names of other users in their peer group, none of them realised that they could select who they sent messages to by selecting names from it. This functionality was not immediately obvious from the interface, and even the evaluators who were more experienced with chat applications did not realise it's presence. Fortunately, the interface selects all names by default, otherwise they may have taken some time to work out why their messages were not being sent.

One of the evaluators found the "color key" dialog in the whiteboard (see 3.8.4.4 for the design picture) a little difficult to interpret at first. The relatively thin font led to her failing to recognise that the different colours of the different usernames had any meaning. The "Color Key" heading was not prominent enough to bring it to her attention, and she assumed it was a list of usernames only. This could be improved by enhancing the heading, and increasing the font weight of the usernames.

Another of the evaluators found the LOGIN/LOGOUT button turning red during the login procedure (see 3.8.4.3 for design picture and explanation) was unclear – they associated red with indicating an error, and thought that the interface was indicating an error state rather than a busy state. This could be improved by choosing a different colour for the button, or by displaying a more explicitly "Busy…" message in some way.

The file transfer application was less familiar to the evaluators. After logging in, some of them were unsure how to proceed, finding the interface too busy and cluttered. The densely packed GUI controls made it difficult for the evaluators to know how to start, and were not able to guess from a first impression of the screen. This design was largely a result of the small screen space, and it is difficult to see how this could be easily solved.

Also, when first logging in to an empty peer group, the users found it very difficult to tell which box contained which information – the difference between the list of user names to choose from (left) and the box containing the name of the file to be sent (right), was not obvious until some information was entered.

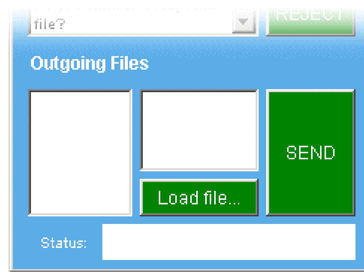


Figure 18 - File Transfer application interface screenshot

This is shown in Figure 18 above. It could be improved by adding labels to the boxes to identify them at a glance.

The proportion of the file name box on the right also was felt to be odd. As it contained a filename (a single line of text), the evaluators found the very tall box to be a little out-of-place. By using a narrow box, it also meant that the filename often continued off the right-hand edge of the text area – cutting off part of the name from view.

This was a result of trying to display two different types of information – a user-list (better suited to a narrow vertical list) and a filename (better suited to a long horizontal box), in a small area of screen space. In an attempt to compromise, the evaluators felt that the result was a little cumbersome.

5.3.2.2.2 Predictability

One of the principles that support learnability is predictability. Predictability supports the user being able to learn how to use an interface by allowing the user to predict the outcome of any possible action that they may make. Any behaviour that helps the user to know in advance what will be the interface's response to what they do aids the learnability. Likewise, behaviour that surprises the user and does not allow them to determine the effects of their actions based on their interaction history hinders the interfaces' learnability.

I applied this principle to the interfaces as explained above (5.3.2.1). I also paused the users occasionally before they did something, and asked them what they thought would happen as a result. Comparing what actually happened with their expectations was a useful indication of the interface's predictability.

Overall, the evaluators agreed that the interface was relatively predictable. The interface makes it obvious what operations can be performed at any time, by enabling and disabling GUI controls as appropriate and in response to user actions. This operation visibility provided a clear indication to the user of the available operations.

5.3.2.2.3 Synthesizability

For the user to be able to predict how the interface will respond to a given action it is necessary for the user to envisage some sort of mental model of the application and how it behaves. Synthesizability supports this by providing enough feedback to allow them to assess the effects of their interactions. Any behaviour that helps the user to clearly see the outcome of an action that they perform aids the learnability. Likewise, behaviour that does not provide clear or sufficient feedback makes it harder for the user to internally visualize the system's state and hinders learnability.

I applied this principle to the interfaces as explained above (5.3.2.1). I also used the application while the user had their eyes closed, and then asked them to open their eyes and tell me what state the application was in without having seen what I had done.

The enabling and disabling of GUI controls, and the switching of screen displays between the logged-in and logged-out modes (3.8.4.3) made it immediately obvious whether the user was currently in a peer group or not, in all interfaces.

The file transfer interface (see 3.8.4.5 for the design picture) was initially a little confusing for one of the evaluators, who found that the changing function of the buttons was not an obvious enough indication of the current position in the operation sequence. An example was changing the "SEND" button text from "SEND" to "CANCEL" (to allow the user to cancel their file transfer request) while waiting for the user to accept or reject the request. This was not an obvious enough form of feedback for her, and she found it a little confusing – unable to confidently tell at a glance if the application was busy sending a file, or busy receiving a file when presented with the application screen after re-opening her eyes.

5.3.2.2.4 Generalizability

Generalizability is another principle that aids the user in being able to predict how an interface will work. An interface can use generalizability by behaving in ways that allow the user to extend specific knowledge of how the system interacts to new similar situations. Generalizability can be applied within an application by making system actions consistent, and between applications where similar situations are possible.

The generalizability of the interfaces was quite good. The chat application provided an interface that was consistent with chat-rooms and instant messagers that the evaluators had encountered previously – with a small space to write messages, a send button, and a large screen of recent messages. The whiteboard application, with the ability to control the pen size and shape, was consistent with basic paint and graphics packages – and the pen stylus made the main control quite intuitive.

The file transfer application was not as generalisable – with the evaluators commenting that it was unlike any file transfer application that they had encountered previously. However, the use of standard file dialogs was appreciated by all evaluators.

The generalizability of managing user and group names between applications also aided the learnability of the interfaces, enabling the evaluators to start to recognise patterns and links between the applications as part of a small application suite, rather than three individual programs.

5.3.2.2.5 Consistency

The final learnability principle is consistency, which refers to the similarity in behaviour arising from similar situations or objectives. Consistency can take a variety of forms, and is strongly related to the other principles. In general though, the more consistent the interface is, the easier it is for the user to generate a mental model of the application and the less the user has to learn – all of which will aid learnability.

The interfaces are very consistent – both internally, and across the application suite. Consistent use of command names, regular system behaviour and interaction requirements when performing similar tasks, and a consistent placement and behaviour of interface controls all aid the learnability.

One inconsistency was noted by all of the evaluators in the file transfer application, who did not like the difference in types of status boxes (see 3.8.4.5 for the design picture). The incoming files box is quite large, allowing the user to see earlier messages and supporting scrolling. The outgoing files box is quite small, showing only a single line of text.

This was done because of the difference in the size of messages required by the two scenarios, however on reflection, a more consistent approach to providing feedback would have been more usable.

5.3.2.3 FLEXIBILITY

Dix identified flexibility as being the properties of an interface that allows a multiplicity of ways for the user to interact with a system. He identified five principles that support this and I examined the interfaces using each of these principles.

5.3.2.3.1 Dialog Initiative

One way to improve the flexibility of an interface is to consider who initiates communication between the user and the interface – whether it is the user themselves or the system. Maximising the user's ability to pre-empt the system and minimising the system's ability to pre-empt the user can allow the user freedom from artificial constraints on the input dialog. . This principle does not really apply to the applications created in this project.

5.3.2.3.2 Multi-threading

Another way to improve the flexibility of an interface is to support the user interaction pertaining to more than one task at a time. Instead of enforcing that the user carry out tasks one at a time, the flexibility of an interface can be improved by supporting several.

I applied this principle to the interfaces as explained above (5.3.2.1). The whiteboard and chat application do not really support more than one type of user task, so this does not apply. However, the evaluators did comment that the ability to both send and receive files at the same time, and being able to interlace the steps required to do this, was quite helpful.

5.3.2.3.3 *Task Migratability*

Another consideration when evaluating the flexibility of an interface concerns the transfer of control for execution of tasks between the system and user. This principle does not really apply to the applications created in this project.

5.3.2.3.4 *Substitutivity*

One way to improve the flexibility of an interface is to allow equivalent values of input and output to be substitutable – and not restrict user input of a specific type where equivalent alternatives are available.

I applied this principle to the interfaces as explained above (5.3.2.1). The nature of the applications means that there is limited scope for substitutivity in the interfaces. Much of the user input is not directed specifically at the system, but is intended for transmission to remote users. One place where an opportunity for substitutivity was recognised was in selecting a file to send in the file transfer application. Files can be selected by entering the file path and name in the box, or by using the "File Load" dialog. These are two equivalent ways of identifying a local file, and the interface interprets them both accordingly.

The evaluators agreed that the interface did not restrict any user input where an alternative might have been used.

5.3.2.3.5 *Customizability*

The final flexibility principle is customisability, which refers to the modifiability of the user interface by the user or the system. Adaptable interfaces, where the user is allowed to customise the interface at will, or adaptive interfaces, where the system alters the interface in response to the user's actions (such as displaying additional help to a novice user) all aid the flexibility of the interface.

None of the three interfaces is customisable in any way, although none of the evaluators felt that this was a problem. It was generally agreed that there was little need for customisability of the interfaces unless a significant number of additional functions were added.

5.3.2.4 ROBUSTNESS

Dix identified robustness as being the level of support that the interface provides the user in determining the achievement of the user's goals. He identified four principles that support this and I examined the interfaces using each of these principles.

5.3.2.4.1 *Observability*

Observability refers to the user's ability to be able to tell the current state of the application from the interface. Overall, the evaluators felt that that this was very possible from the interfaces, however this is largely a result of the applications only having two main states – logged out and logged in. There are other internal states, but these are of no interest to users.

As mentioned previously (5.3.2.2.3), one user found it difficult to tell what stage the file transfer application was in from the interface – whether sending or receiving. This was compounded by a number of problems reported by the evaluators with the message boxes used in the interface (see 3.8.4.5 for a picture of the design):

- Inconsistency in msg box types between incoming and outgoing (noted in 5.3.2.2.5)
- Messages in outgoing-status message box overrunning message window – with no way of scrolling
- Messages in incoming message box not clearly separated – added to continuously maintained large text field, with no indication of when a new message has been added
- Unclear which is the final message (for example, when receiving a file, the message which reports that all file segments have been received implied to some evaluators that the transfer operation was complete, unaware that the application next had to reconstruct the segments into a file) – remaining work is not made clear

5.3.2.4.2 Recoverability

Recoverability refers to the interface providing the user with the ability to recover from errors that they may make. This does not really apply for these applications as there are no recoverable errors – this is not an interface issue.

5.3.2.4.3 Responsiveness

The responsiveness of the interfaces was determined acceptable by all evaluators. The login time was perceived as a little slow, but none of the evaluators felt that this was a large problem. In any case, this was not a fault of the interface, which was responding promptly by displaying a busy status almost instantaneously – the delay was in the services layer.

5.3.2.4.4 Task conformance

Interface supports all tasks detailed in application functional requirements documents – as documented in the previous test section (5.2.2.1).

5.3.2.5 CONCLUSIONS

5.3.2.5.1 Evaluation conclusions

Overall, I thought the interfaces were good. The lack of clear labels or space to spread out screen elements meant that the interfaces were initially unclear and daunting to some of the evaluators. All of the evaluators agreed that the guessability was the single largest weakness of the interfaces. However, they acknowledged that once explained how the interfaces worked, they did find them easy to use.

The file transfer application was the one interface that received the most complaints – the usability of the interface was generally not popular with the evaluators who found it unclear. They could use it after training and explanation, but found that it required concentration and effort – which would have made them reluctant to use the application for their own use. While the other interfaces could be improved with the minor changes and improvements discussed above, the file transfer interface does need to be totally redesigned to reflect the findings here.

Another comment made by the evaluators was that the small canvas size meant that the whiteboard application was too small to be a useful collaborative design tool. While this is a more an

aspect of the hardware than of the system produced, it is a valid usability point. As a commercial end-user product, the application would likely fail with such a restrictive workspace. However, as an educational system developed to demonstrate the potential for peer-to-peer free design work, it was still a valid application to produce.

Obviously with a test group of only three subjects, this evaluation cannot be an in-depth or statistically reliable study of the usability of the interfaces. However, I believe that the use of Dix's principles did help to mitigate this. By providing prompts and directing comments to specific areas and concepts it increased the value of their comments, making them broader. Without the list of principles to guide consideration, such a narrow sample group could have ended up being indicative of only three people's competence with a software application – not of the interface's usability.

5.3.2.5.2 Post-test Interview Data

The following list details a summary of the evaluator's comments after using the applications, based on a heuristic approach[37] to critiquing interface implementations:

- **Simple and easy to use** – Evaluators acknowledged that the interfaces were simple and clear. They did not display irrelevant or rarely used information, and were easy to use once understood.

- **Minimised the amount of information the user needs to remember** – Evaluators agreed that they wouldn't need to refer to any user manual to use the device effectively, and needed to remember very little to use it. (*However, this was thought to be more a result of the simplicity of the applications rather than interface design*).

- **Consistency throughout the interface** – Evaluators agreed that the interface was consistent, in both visual style and operation.

- **Good feedback to the user** – Evaluators generally felt that the interfaces provided effective feedback to the user, although commented that feedback in the file transfer application was slightly unclear at times

- **Clearly marked exits** – The interfaces rely upon the standard windows-style control boxes to close the applications, which most of the evaluators thought was straightforward enough.

- **Good error messages** – Evaluators acknowledged that the error messages were clear and helpful. They did not contain excessive or unnecessary detail, and included clear and helpful suggestions of how the user could correct the error, or avoid it recurring.

- **Minimise the possibility for the user to perform an error** – Evaluators acknowledged that the interfaces provided effective basic protection against most possible errors – for example, disabling of the user and group name text fields while logged in was recognised as helpful.

---

[37] *Heuristic Evaluation* – developed by Jakob Nielsen and Rolf Molich

- **Lack of help or documentation** – The evaluators noted the absence of any online or interactive help, or accompanying documentation. While this would have been helpful if I was intending to deliver the applications to real-world users, I do not believe that producing this would have been relevant to this project.

The evaluators also made several general suggestions for possible additions to the project, not directly about the interfaces, which are discussed in the **Further Work** section (6.2.1.2.4, 6.2.1.2.5).

5.4 RESULTS

I believe that overall, testing and evaluation revealed my project to be a relatively stable and reliable system. It correctly manages peer groups, and allows group collaboration and communication, and all application features function correctly.

Strengths identified during testing included that the chat-room successfully handled a large number of users even while under heavy network load, with no noticeable delays or lags. The file transfer application was demonstrated reliably even with very large files, with acceptable delays for transfer time. The retransmission functionality worked perfectly, with logging confirming the requesting and receiving of the correct missing packets.

Strengths identified during evaluation included the usability of the application interfaces that was found to be acceptable by all of the volunteers involved in the testing. They efficiently and clearly provided the underlying functions. There were some learnability concerns – where parts of the interface needed to be explained to some testers, however this was to be expected in such a small interface where there is limited space for explanatory guides. The lack of tooltips and other such hardware restrictions added to this problem.

One of the biggest weaknesses I identified during testing was the sacrifice of reliability since the implementation of the caching of output pipes. While this worked well while users were connected, and left a peer group by correctly informing other group members first, when peers were killed off or disconnected, the handling of their pipes was not ideal. This is looked at in detail in the following **Further Work** section (6.1.2).

Another weakness was the performance of the whiteboard. The evaluation showed that it normally produced an accurate representation of user's sketches on the remote peers under development environment conditions. However, it was often very slow at doing this – with a noticeable lag between drawing on the sender, and the line appearing on the receivers. Performance deteriorated further when tested on the target platform. This is discussed further in the **Further Work** section (6.2.1.1.2).

The services layer evaluation (5.3.1) revealed that more work is needed on the services layer. It is still quite slow and struggles to support a streaming connection to more than one other user. While fast enough for chat-room type applications, a more efficient approach would still be beneficial. The evaluation (with results such as round-trip times of over three seconds) demonstrated that there is a bottleneck in the system somewhere, and further work is needed to locate it.

6. FURTHER WORK

6.1 IMPROVING JXTA

6.1.1 INTRODUCTION

One thing that I originally did not like with the JXTA platform was the way that it provided one-way connections with other nodes – uni-directional "pipes", which an application could either send through or receive from. This introduces another layer of complexity for the application writer – who must create and manage both an input and an output pipe in order to communicate with another user.

I decided that I would look at the benefits of extending JXTA to include a two-way pipe.

6.1.2 EXISTING APPROACH

The JXTA approach used so far in this project is that each peer (such as Peer X in the diagram below) creates an *InputPipe*, where it can receive incoming messages. It then publishes the presence of this input pipe so that other users know of it's existence. Other users (such as Peers A – D) can then create an *OutputPipe* based on that InputPipe, so that they can send messages to that peer.

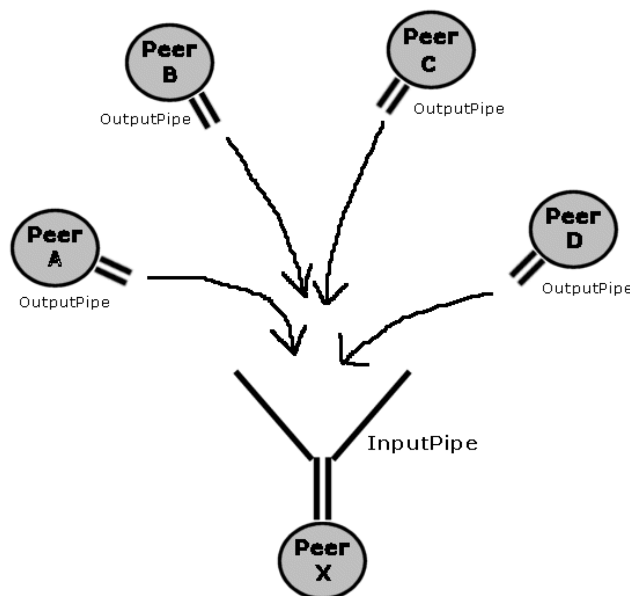


Figure 19 - Existing use of pipes in JXTA

As Figure 19 shows, in some ways the use of the term *InputPipe* is somewhat misleading in that it suggests a single connection between peers. In reality, it is more like a funnel that any other peer can throw messages down.

The biggest problem with this approach is lack of convenience for the services layer – having to manage two uni-directional connections for each user. It is rare that a user would ever want a peer to only send or only receive data from another peer, so I initially thought that it would be more convenient to have a single connection that could be used for both.

The other problem is that the output pipes are created based upon pipe advertisements and not the input pipes themselves. When the user's pipe advertisement expires, it can publish a new advertisement. This is then discovered by remote peers who (unless they open and examine the pipe advertisement elements and compare against currently cached expired adverts) will treat this as a newly discovered peer, and create a new OutputPipe based on it.

Basing the OutputPipes on adverts instead of the InputPipes themselves also led to problems – such as not knowing when the InputPipe is no longer there. The pipes are an abstraction – providing a virtual connection that hides the complexities of network transport and routing. As a result, an OutputPipe has no way of knowing if the InputPipe to which it sends is still there. Likewise, an InputPipe has no way of knowing if the OutputPipes sending to it are still active. In an attempt to provide a protocol set that does not require any particular transport protocol or implementation, the pipes provide a virtual connection without any ability to sense the link status. Instead, the services layer must rely upon the presence or absence of the adverts upon which the pipes are based.

As a result, I included the EXIT message in the services layer as discussed previously. In this way, a user informs all local peers before they leave a peer group, so that they can close the pipes to them. This worked well to a point, but is not a foolproof approach. If a peer exits in any way other than expected (such as network failure, or any other sudden or unexpected crash or event which ends the application) and the local peers are not informed, the users have no way of knowing that the user has left until they try and send a message to them.

A user leaving without sending an EXIT message has to be considered, particularly when designing for a mobile application running across a wireless LAN where users might move out of range of other peers without intentionally leaving a peer group.

Even when the user tries to send a message, the absence of the receiver is not immediately obvious. When creating an output pipe you specify how long you want the send operations to block for. This is a difficult choice when dealing with a mobile network with potentially variable properties. Too short a time and the pipe could give up before creating a connection, and too long a time leads to the system seemingly stalling when trying to send a message to a user who is no longer there. Without having an actual connection to gauge the connection status by, the system has no way of knowing if the user at the other end is still there.

This was not so bad initially while I was testing the chat application, as the creation of the output pipe does require some communication between the peers, to establish routing and other information. As a result, trying to create an output pipe based on a cached advertisement of an input pipe that no longer exists, did give more immediate feedback, making it clear that the user was no longer there. However, as discussed previously, opening a new output pipe for every message introduced too high an overhead when used on the whiteboard and file transfer applications, which led to me introducing the cache of output pipes. While reusing output pipes did lead to quite noticeable performance increases, it did make the services layer less stable as a result.

The reuse of pipes when you have no way of knowing if the user that the pipe points to is still there led to increased number of errors and stalling when users leave without informing. As a result, the services layer keeps reusing a stored output pipe either until they are explicitly told not to, or

when using it leads to exceptions. My initial requirements for the applications stated that they should support users who just sit in a chat-room or peer group to listen, but not wanting to take an active part in the discussion. Relying on trying and failing to send a message to users to know if they are really there does not fulfil this requirement adequately – such users will not know.

Creating a two-way connection between peers potentially solves these problems. Instead of caching potentially out-of-date connection information, and reusing it to create an actual connection, you create an actual two-way link between two peers, allowing for greater feedback and quicker error detection. With unidirectional pipes, you cannot get direct feedback from the pipe you are communicating with. The input pipe you are sending messages to cannot reply. In fact, just because the remote input pipe may be functioning is no guarantee that the remote user has an active output pipe.

With a single InputPipe that anyone can send information down, you have no way of knowing who is currently sending to you. It is a chaotic situation, with anyone potentially throwing information at you. As discussed previously, if you want to ignore certain users, you are still forced to receive and process messages that they send you, only able to discard them once they have been processed sufficiently to identify the sender from message headers. By preventing the user from having any control over the actual connection, the user cannot control who it receives from and when. Using two-way pipes, instead of a global input pipe means that a user has total flexibility over whom it listens to. It can listen to all, poll a few, listen to them individually in turn, or any other approach or combination.

Another problem is reliability. During development of the services layer, and the testing of the applications, I noticed that when the network is under relatively heavy loads, messages did go missing. This was especially evident from the file transfer application, where missing messages were noticeable (leading to file sizes being too small and files being corrupted). This led to the introduction of the file handler into the services layer discussed previously, which managed the retransmission of missing file segments.

The uni-directional specified in the JXTA protocols does not specify any guarantee of delivery. Instead of implementing this reliability in the services layer, creating a new type of pipe would also enable me to create enhanced pipes with greater reliability.

6.1.3 REQUIREMENTS ANALYSIS

6.1.3.1 OVERVIEW

The main requirements for my modification of the JXTA layer were provide an alternative connection and message transport implementation to that which already exists, which tackles the problems discussed above.

To summarise the primary requirements, they were to:

- Provide a more convenient tool for the service or application writer, avoiding the need to manage both input and output connections separately
- Provide a more connection-oriented approach so that the user is better informed if an active connection is open without having to try sending a message

- Solve reliability issues to prevent messages going missing, and handle retransmission if they do

6.1.3.2 CHOICE OF TRANMISSION PROTOCOL

Many of the decisions made when implementing JXTA described above are a result of the fact that the developers have tried to keep it as open as possible. It does not enforce the use of any particular language, network transport layers or hardware types. As a result, it does not take advantage of the properties of any individual network type.

However, my project is not bound by such constraints, and so I was free to consider different network implementations. I decided to use TCP as my transmission protocol, as it provides a reliable, connection-oriented service. This makes it well suited to solving many of the problems described above. For example, the reliability means that TCP will handle any problems with missing messages as described above. In addition, the connection-oriented approach is well suited to such applications such as chat-rooms, and collaborative idea-storming whiteboards, where users will log in (thereby opening a connection) before wanting to send or receive data. Connectionless protocols such as UDP are less appropriate – while they do avoid the overhead of setting up a connection that TCP requires, this is not a problem where users will expect to log in and establish a connection anyway.

I did briefly consider UDP as an option because the relatively small amounts of data being transferred means that the speed and reduced overheads of UDP messages would be beneficial. To be honest, the use of UDP would probably be more suitable for the whiteboard application, where reconstructing large messages and requesting retransmission of missing messages isn't as important as the ability to receive data in pseudo-real-time. The streaming nature of the whiteboard network requirements is perhaps better suited to UDP than TCP, but I decided that TCP would be a better choice for the project as a whole. Using TCP means that I can take advantage of the reliability and connection managing that the protocol provides. While it would be unsuitable for the JXTA developers to have enforced the use of TCP/IP, I believe that it makes an appropriate implementation for my project.

6.1.4 DESIGNING THE EXTENSION

6.1.4.1 DESIGNING THE PIPES

I considered a number of approaches to implementing a bi-directional pipe. One approach that I considered was to provide a "wrapper" to the existing InputPipe and OutputPipe functionality. This could be done by producing a single class that used the InputPipe and OutputPipe classes to provide functionality, but holding a pair together with a single convenient interface.

However, I discounted this approach because, while it dealt with the requirement for a more convenient interface for the application writer, it did not really tackle the other problems identified.

I therefore decided to avoid the existing JXTA connection libraries. As they are implemented in a "network-agnostic" way, they do not really make the most out of the beneficial properties of a TCP network. Instead, I decided to implement the connections myself, using the networking libraries provided in Java. While this could be argued to break what principles JXTA is designed with, I believe that this is still an acceptable thing to do. TCP connections are a suitable solution for my project, as discussed above, and providing an implementation of this as an extension to the existing JXTA functionality shouldn't be considered to be breaking JXTA. As long as the existing unicast,

network-independent pipes are still available and function correctly, then my work can sit alongside this, providing as an optional choice of pipe type for the user if their network supports it

Despite this, I did want to aim my extension to be as consistent with the existing JXTA conventions and approaches as possible, which is reflected in the design discussion shown below.

6.1.4.2 DESIGNING A PROTOCOL

The use of InputPipes and OutputPipes in JXTA enforces the use of a service-layer "protocol" for establishing a connection between peers. The tables below show an overview of the protocol that the design requires.

RECEIVER	SENDER
1.) Create InputPipe	
2.) Listen for incoming messages	
3.) Publish InputPipe's presence	
	4.) Discover InputPipe advert
	5.) Create OutputPipe based on advert
	6.) Send message
7.) Receive message	

Figure 20 - Existing connection protocol

As Figure 20 shows, the sender cannot send a message until it has an output pipe to send through. It cannot create an output pipe without an input pipe advert to aim the pipe at, and it cannot have an input pipe advert unless the receiver has created an input pipe ready to receive incoming messages.

In effect, the use of unicast pipes does in part ensure that users can only send a message when a connection has been established correctly – preventing an application from skipping any stages in the protocol by requiring that each step rely on a previous step.

My implementation obviously required such a protocol, to define the sequence of communication between peers. My initial approach is shown in a UML diagram attached to this report in the Appendix[38]. I chose a UML sequence diagram to display the design, as it best shows the order of interactions between peers.

6.1.4.3 INITIATING A CONNECTION

The biggest problem that this design tried to address is that even with two-way pipes, there has to be one user who initiates the connection. My approach was to create a connection manager, which I called SocketController. The role of the SocketController was to continually listen out for remote peers who want to open a connection to it. It listens out for connection requests on a known port number. By naming one permanent socket[39] that every peer has and listens to, it allows remote peers to open a connection to the SocketController – an initial, temporary connection to the peer.

[38] Appendix I – "JXTA Extension: Initial Protocol Design - Connecting"

[39] The socket created to listen for connection requests is shown on the diagram as ServerSocket, while incoming connection requests are sent from ClientSocket. This is in keeping with Java terminology, reflecting considerations for implementation.

Once this connection is established, the remote peer can ask the SocketController for it's own dedicated socket. The SocketController creates a new connection, then gives the remote peer the connection information it needs to disconnect from the SocketController and connect to it's own new permanent port. Once disconnected, this frees up the SocketController to receive and process connection requests from other peers.

6.1.4.4 CREATING AN INTERFACE

As Appendix I also shows, my initial design placed the public interface for the two-way pipe constructor in the PipeService class. This was largely a result of my aim to keep my extension consistent with the existing JXTA functionality. The methods to create the existing pipe types are all contained in the PipeService class, as shown in the excerpt from the Javadoc below.

InputPipe	createInputPipe(PipeAdvertisement adv) create an InputPipe from a pipe Advertisement
InputPipe	createInputPipe(PipeAdvertisement adv, PipeMsgListener listener) create an InputPipe from a pipe Advertisement
OutputPipe	createOutputPipe(PipeAdvertisement adv, long timeout) create an OutputPipe from the pipe Advertisement
void	createOutputPipe(PipeAdvertisement adv, OutputPipeListener listener) registers a listener for a NetPipe.

http://platform.jxta.org/java/api/net/jxta/pipe/PipeService.html

The PipeService class provides the service layer with an API to the JXTA pipes. To conform to this approach, rather than allow the user to create a TwowayPipe by directly using a constructor method in the TwowayPipe itself, it made sense to add a createTwowayPipe method to PipeService.

In this way, the TwowayPipe is simply added as another type of JXTA pipe that the user can choose from, rather than being a distinct stand-alone class.

In reality, I was aware that most of the actual constructor methodology will actually reside in the connection manager class SocketController, however I felt that requiring the services layer writer to know of and understand the implementation of the pipes at such a level was unnecessary. The services layer should not need to know of the existence of the SocketController class. The use of this should be handled by the PipeService interface, unseen to the services layer programmer.

6.1.4.5 RECEIVING CONNECTIONS

As described, the SocketController creates a two-way pipe when it (say Peer X, for example) receives a connection request from a remote peer (Peer Y). As these are two-way connections, this means that X does not need to request a connection to Y, as one already exists, initiated by Y itself.

This meant, as well as the services layer keeping track of which remote peers it requests connections to, it also needed a method of finding out what connections are created at the request of other peers – so as not to duplicate them.

One approach to this that I considered was to specify a service layer interface that the SocketController calls when it creates a TwowayPipe, to inform it that the connection is available. However, I felt that enforcing a core-layer to services-layer interface would be too restrictive. While it might have provided an effective solution in this instance, I wanted my modifications to the JXTA protocol layer to remain as generic as possible. Although it is quite unlikely that my extension will be used by other applications, I still felt that a more open approach was suitable and any such interface might be unsuitable for other applications.

Instead, I decided to make the connection manager class SocketController responsible for maintaining a record of which peers it has an active connection to. In this way, if a peer requests a connection to a user with whom a connection has already been requested and established, the SocketController knows not to create a new one, and simply returns a handle to the existing connection. This means that the services layer will not be aware of a connection's existence until it wants to use it itself, but this is quite appropriate. If the services layer does not want to use a connection, there is no need why it must know of its presence anyway.

6.1.4.6 CLOSING CONNECTIONS

Making the SocketController responsible for maintaining a list of active connections had an implication on how I close the connections when no longer required. My initial approach to closing pipes is attached to this report in the Appendix[40]. The existing JXTA approach to pipes is that the services layer can close pipes directly once no longer required. The PipeService simply provides an interface to the constructor methods.

However, as discussed above, as the nature of two-way pipes meant that the construction of a pipe may not always be initiated by the services layer itself, some record of established connections is required – which I decided to make a responsibility of the SocketController class. This means that allowing the services layer to directly close pipes could leave SocketController records out of date.

I did consider implementing a two-way pipe "close" method, which informs its parent SocketController before closing. However, once created, I did not think that it was appropriate for a pipe to remain in communication with its "parent". Instead, I implemented the close method as an additional PipeService functionality – responsible for closing the pipe and informing the SocketController. This is slightly different to the existing JXTA approach, but this is partly a result of the uni-directional nature of pipes used – that there is no such need for notification. I felt that a close method still fitted in well with the general ethos of the PipeService's responsibilities – providing an interface to the pipe's API. It is reasonable to consider that closing a connection is a part of this.

6.1.5 IMPLEMENTING THE EXTENSION

6.1.5.1 GENERAL APPROACH

My first step was to download the source code for the JXTA platform. I downloaded the Java binding to maintain compatibility with the services and application layers already developed. The JXTA implementation is quite large – with over a thousand files.

My first task was to orient myself with the structure of the source code and the existing pipe implementations – to find where to make my modifications and extensions. This was a good way of

[40] Appendix J – "JXTA Extension: Initial Protocol Design - Disconnecting"

learning more about how JXTA works – and how the existing InputPipe and OutputPipes that I had been using are implemented. I learnt a lot about the way that the pipes are implemented – and gained a greater understanding of why they work as they do.

My approach, as discussed before, was to use the Java `socket` abstraction to provide TCP sockets. These are a powerful programming concept, giving the opportunity to create application protocols to be used in place of the existing JXTA protocols.

As discussed previously, I wanted to implement my protocols in a way consistent with the approach already present in JXTA. As a result, I decided to make my TwowayPipe class implement the defined JXTA InputPipe and OutputPipe interfaces. In this way, JXTA applications can use my new type of pipe in the same way that they use the existing unicast pipes – implementing at least the same methods. The exception to this is the close method, as discussed before, which must come from the PipeService to ensure that the SocketController can maintain a list of active pipes.

6.1.5.2 PORT NUMBERS

My first idea for implementing the sockets was for the SocketController to listen on a "well-known" hard-coded port – so other peers always know where to send connection requests. However, such an approach is flawed in two ways – firstly, it is impossible to know on any one machine if the "well-known" port number I specify will be available, and secondly this would restrict JXTA in a major way: currently, multiple JXTA peers can run on a single machine by specifying a different port number when configuring the platform. By making the SocketController always use the same port number, I would prohibit this, meaning that only one instance of JXTA can run on a single machine.

Instead, I decided that the SocketController should use a dynamically allocated port number – finding any available port number at run-time. This port number can then be incorporated into the pipe advertisement that is published by the peer. In this way, when a remote node receives the advert, this will include the IP address and port number, allowing it to easily open a client socket to the SocketController. This is reflected in the design sequence diagram shown in the appendix[41].

6.1.5.3 PEER MESSAGING

As my implementation includes the design of a new messaging protocol, it also required the creation of new messages to cater for the connection requests and replies sent to and from the SocketController.

The connection request message needed to include enough information for the SocketController to decide if a connection already exists, if any local ports are available, and who the requesting user is.

The reply message (if the request is accepted) needed to include enough information to allow the connection initiator to be able to connect to the newly created TwowayPipe spawned by the SocketController connection manager.

I decided that these messages would best be implemented using XML structured text documents. I chose these because this would be consistent with the existing messaging format used in JXTA, and because it provided a good platform-independent approach to transferring information between

[41] Appendix I – "JXTA Extension: Initial Protocol Design - Connecting"

machines. This required the use of XML creators and parsers to generate and interpret XML documents – with tags separating each message element.

An example of the message formats is included in the appendix attached to this report[42].

Notice that I avoided the use of "usernames" in these messages. The concept of a username is an application-level concern, and is not appropriate in the protocol layer. Instead, I used machine endpoint addresses and the unique JXTA peer IDs to record information about established connections, and identify distinct remote nodes.

6.1.5.4 MESSAGES

Another responsibility of the JXTA PipeService class discussed previously is that it provides an interface to the JXTA message API. There are two standard representations for messages in JXTA, XML and binary, depending on the transport layer being used by the node. I decided to continue using XML for my messages.

Reading through the code I found that JXTA messages contain a lot of headers, containing a variety of information such as hop counts, routing information, addressing and so on. As a message passes through the protocol stack, each level adds elements to the message

Discovering this made me consider whether it was wise to use existing JXTA messages as they were. By replacing the connection and transmission protocols, I made this network information obsolete. Therefore, each message may have been unnecessarily inflated with routing information and other such headers. I considered a number of approaches to dealing with this, including:

- Cloning messages

One possible approach was to have the services layer use JXTA messages as normal. Then, have the sender extract the message data elements and send them as data through the socket, letting TCP package the data adding headers and transport information as required.

The receiving peer can then use these message elements to reconstruct a replica of the original message, by adding them to a new blank message. In this way, the receiving peer creates a **clone** of the original message, passing it on to JXTA as normal.

I rejected this because it seemed inefficient to deconstruct and reconstruct every message – and it would be pointless to use JXTA messages if I would not actually be transporting them.

- New message type

Another approach I considered was to use a new, custom Message type for use with TwowayPipes. In this way, I could design it to suit the transmission approach of my sockets. However, I was reluctant to introduce a new type of message because of my intention to conform to the existing JXTA approach.

[42] Appendix K – "JXTA Extension: XML Messages"

The existing implementation uses a single message type for all three types of pipe (Unicast, UnicastSecure, and Propagate), and I was reluctant to break this to implement my new pipe. As a result, I rejected this approach.

- Serialising message

Another approach I considered was to serialise the JXTA Message and send it through the socket as an object. In this way, the application programmer can continue to use standard JXTA messages regardless of the type of pipe they decide to use. However, the current Message implementation is not serialisable, and providing a serialisable implementation would have required creating a new message type, which I wanted to avoid.

- Existing message type

I rejected all these approaches, as I wanted to try to use the existing Message class. Further investigation into the Message implementation revealed that as long as I do not pass the Message to any of the existing JXTA transport methods, the actual overhead added is quite low – so any performance implications are negligible compared to the convenience of allowing the user to continue using standard JXTA message processing methods. Therefore, I designed a way of passing a byte array representation of the message through the Java socket, allowing me to remain consistent with the JXTA use of Messages.

6.1.5.5 ADVERTISEMENTS

My design discussed previously required a change to the information that the user publishes about themselves. For a peer to create a TwowayPipe to a remote user, it needs to negotiate a new connection with the remote user's connection manager class, SocketController. This meant that the peer needed to know the IP address and port number that the SocketController was listening on. My design stated that this information would be included in the advertisement published by all peers.

When trying to implement this, my initial intention was to include this information in the PipeAdvertisement already being published by the Services layer. However, I found that this was not possible – PipeAdvertisements are not extensible at run-time. The only way to store this information in the PipeAdvertisement was to alter the PipeAdvertisement class and re-compile. As my requirements stated that I did not want to damage the existing JXTA platform, and allow the existing functionality to be used alongside my additions, I rejected this approach.

Instead, I decided to create a new type of advertisement – a TwowayPipeAdvertisement - capable of storing the connection information that I needed. To do this, I needed to define the class for the new advertisement. The default behaviour of the new advertisement was defined in an abstract class, providing a simple placeholder for the PipeServiceImpl Id and its name. However, this simply defined behaviour, without processing the required XML documents. That had to be done in another class that implements the abstract TwowayPipeAdvertisement class. Pipes are named with their XML advertisement.

I also had to alter the JXTA platform configuration to include the new type of advertisement into the hash table of keys generated at initialisation time – this allowed classes such as the AdvertisementFactory, responsible for creating new advertisements, to recognise the new advertisement type.

6.1.5.6 LISTENING

One problem encountered is how the application listens for incoming messages. The approach of having a single InputPipe does have one major advantage – it means that the services layer need only listen to one endpoint for incoming messages. It knows that all incoming traffic will come to that one point, which does a lot to simplify the handling of listening for messages.

However, a peer with multiple bi-directional pipes must listen to multiple input endpoints. One listener is no longer an option. My initial work showing interaction between a sender and receiver – only two users! – failed to highlight this problem during the design stage.

I identified two possible approaches to this problem:

- Leave the problem to the services layer

One possibility could be not to enforce any particular solution on the application writer, and let them handle the problem. In this way, different applications could implement an approach that bests suits their messaging requirements – implementing a listening thread for each pipe, polling each pipe in turn, or something different.

While this way provides the greatest flexibility for the application writer, it does impose an additional complication on the use of this pipe type – which might negate the convenience benefits of avoiding uni-directional pipes.

- Implement a single pipe listener – with all two-way pipes routing incoming messages through it

The use of a single connection manager – the SocketController class – does lead to the consideration of an overall TwowayPipeService. In fact, the services layer must call `createTwowayPipeService` (to create and start the SocketController) before being able to create and use TwowayPipes. Thinking of the two-way pipes as a pipe group does lead to the consideration of a single incoming message router implemented at the protocol layer – allowing the service layer to continue listening at a single point regardless of the type of pipe used.

This would have made the task of converting the existing project Services layer to use my new type of pipe much easier – instead of listening to an InputPipe, the Services layer can listen to a single incoming message router.

My final decision was to implement a listening approach in the services layer – it would not be appropriate for the core protocol layer to enforce any one approach to dealing with multiple listening points, even if it would make the conversion of existing code more convenient.

Having decided where to implement an approach to listening to multiple pipes, I next needed to decide how. This could have been implemented in a number of ways, including those mentioned above (multiple threads, polling etc.). A low level approach (such as the select() function in C) could be to let the kernel do the work, suspending a listening process and specifying a group of sockets to listen to – getting the process to wakeup when I/O is ready from any of the sockets. Such an approach could potentially be less costly than active polling – which could be too processor intensive for mobile devices.

However, the absence of a select()-type in Java[43] means that such an approach would have been difficult to implement. I did consider implementing it using a more event-driven approach…

Instead, I decided that Java's provision of Thread-control methods meant that a multiple-threaded message listener/router would be more appropriate. My main concern is that a large number of active threads could be very costly and cause performance problems. However, this is not really related to the number of peers – as each thread will be suspended, blocking on the socket and waiting for input, the number of threads should not provide too much of a problem.

The biggest potential problem is that if there is a lot of network traffic with lots of messages being received from multiple users, then the running of too many threads could cause performance problems.

Despite this, I decided that using a thread for each pipe was still the best approach. I thought that it was unlikely that any such performance impacts would be an issue with the sort of number of pipes that I would be opening. If I was considering the system supporting perhaps fifty or more connections at a time then the number of threads might have caused me to consider a different approach, however this was not the case. The biggest overhead in using threads is in starting and closing them – as these threads will be running for a considerable length of time, the performance concerns of thread-startup is not a major concern.

6.1.5.7 OTHER CONSIDERATIONS

Other implementation considerations include the option of breaking up large messages into segments and sending individually. However, the Java socket class provides an abstraction of the TCP socket techniques in a manner that is invisible to the programmer. I decided that I would try allowing this to handle the breaking up of large messages and evaluate it's performance, before trying to do this manually.

Another thing that I considered was the implementation of a send queue for the TwowayPipe class, to store outgoing messages before sending. However, again I decided to wait and see how the Java socket abstraction would cope before implementing further safeguards – as it should provide adequate queuing.

6.1.6 EXTENSION PERFORMANCE

The performance of the extension was compared with the original in two main ways:

- Modifying the services layer to enable the applications to be used with the new pipes
- Using test applications created previously (5.3.1.3) to evaluate speed and accuracy

6.1.6.1 APPLICATION PERFORMANCE

The performance of all applications was noticeably improved. The chat application seemed faster and more responsive. The file transfer application no longer needed to use the retransmit functionality that it relied upon so heavily before, leading to significant reductions in overall transfer

[43] Note that Java Specification Request JSR51 (http://jcp.org/jsr/detail/051.jsp) does include proposals to add such a method, so this may change. However, there is no such method at this current time.

time. The whiteboard application operated noticeably faster, with the lag discussed previously (5.4) being greatly reduced.

The removal of having to wait for new pipes to be created (or the pause when getting an output pipe out of the connection store) means that the sending of messages is faster. The pipes are also quicker to react to a user leaving – with catch statements the connection reset exception when a peer leaves.

6.1.6.2 SERVICES LAYER PERFORMANCE

The test applications used in the evaluation of the Services layer (5.3) provided a way to examine the improvement over the original services layer.

6.1.6.2.1 Accuracy

The use of TCP/IP meant that the pipes created were reliable. After running all of the evaluation applications shown previously (5.3.1.3.1), I was able to confirm that the services layer consistently supported a 0% message loss service.

6.1.6.2.2 Speed

The use of a direct connection rather than having to send a message through the JXTA layers seemed to be much quicker. The average message round-trip time was nearly half that of the implementation using JXTA pipes – working out to 75ms (compared with 145-155ms using JXTA – 5.3.1.3.2.1).

6.1.7 EXTENSION CONCLUSION

I believe that this extension was a worthwhile addition to this project. To return to my aims for this extension, they were to:

- Provide a more convenient tool for the application writer

  I have done this – by providing the writer with both access to the new pipe through the PipeService as normal, and creating an automated connection manager which can handle incoming connection requests, I believe that I have begun to implement a more intuitive approach pipe implementation.

- Provide a more connection-oriented approach

  I have done this – by using Java sockets, I was able to implement functionality to detect when a connection is reset by a peer. By dealing with the exceptions that `Socket` provides, it is possible to monitor the connection without having to keep explicitly sending messages through it.

- Solve reliability issues

  I have done this – through my choice of TCP/IP which implements a reliable connection-based approach. Retransmission of missing messages is now handled by the lower levels, providing a much more reliable service to the application writer.

In addition, as discussed above (6.1.6), the new socket-based Two-way Pipe is faster and more reliable than the uni-directional JXTA pipes. The performance of all applications was noticeably improved as a result.

The design of such an implementation would be inappropriate for inclusion into the JXTA project because of the restrictions to using TCP. However, for this project it was a helpful and educational approach to solving some of the inherent problems with JXTA pipes.

In integrating my design into a large networking library, and implementing my changes within a software package of over 1500 files, I was able to learn about and demonstrate a different type of software development. I needed to gain a broad understanding of the functionality and structure of the JXTA platform, and how my code could fit into that. It was very educational to work on a far larger scale than I could by working only with my own code.

In addition, by setting up and implementing TCP sockets and connection managers for myself, I learnt a lot about network programming, which I found to be a fascinating area.

Due to time constraints, I was unable to complete the implementation of the extension. The remaining work is detailed in the following section on **Further Work** (6.2.2).

6.2 FURTHER WORK

6.2.1 ORIGINAL PROJECT

6.2.1.1 IMPROVEMENTS

As with any project of this nature, there are inevitably parts of the work that I would do differently now if I were to start again, as I have continued to learn more about the way the project area. This section details some such ideas that I had, but that time constraints prevented me investigating further.

6.2.1.1.1 Whiteboard messaging

The whiteboard could have implemented the sending of messages more efficiently – discovering a list of local users and sending data to each on individually proved to be an inefficient. One approach that I would have liked to investigate was to alter the use of output pipes.

Instead of creating multiple OutputPipes, with one output pipe for each intended receiver, I could create a new output pipe capable of broadcasting to multiple peers. In this way, the sender can simply write it's messages to a single point, without needing the overhead of maintaining a list of users to send to or sending each message multiple times.

Any peers wishing to receive this "broadcast" could then connect a type of InputPipe to this output pipe, and receive the messages for as long as they have the pipe connected. The sender would not have to concern itself with knowing which peers are listening, just throwing messages down its output pipe. Any other peers would then be able to attach/detach input pipes to that output at will.

This seems more appropriate for an application like the whiteboard, where you always want to send to all users in the peer group, and the overhead of working this out every time is an unnecessary overhead.

It would be interesting to measure the performance implications of exchanging multiple OutputPipes and a single InputPipe for a single OutputPipe and multiple InputPipes. I believe that the sort of behaviour witnessed in the testing of the whiteboard application, where sending messages was causing a bottle-neck, would be better suited by such an approach.

6.2.1.1.2 Whiteboard graphics

As mentioned in the **Testing Results** section previously (5.4), the whiteboard performance was poor at times. I believe that there are two main reasons for this: firstly, the services layer is operating too slowly to support a streaming, real-time type network. In fact, the services layer is operating slower than I expected, but my investigation failed to reveal the reasons for this. Secondly, the whiteboard is sending too many messages – operating at a level of accuracy that is too high for the requirements. Every move of even a pixel results in a new message being sent. Some attempt at aliasing – to approximate the user's sketch and reduce the number of messages sent, is required. However, time restrictions prohibited investigating this further.

6.2.1.2 ENHANCEMENTS

There are also a variety of additional improvements and enhancements that could have been made if time had permitted. The original brief was to produce a suite of basic applications to demonstrate the network services produced, and the potential of peer-to-peer mobile networking. The applications and services layer produced do broadly satisfy that goal, but the usability could still be greatly enhanced. Examples of some of the enhancements that I considered, but did not have time to produce, are detailed in this section.

6.2.1.2.1 Publishing

The services layer produced in this project only makes use of publishing the presence of pipes – allowing for the creation of connections between peers. However, the system could be improved by publishing other things, such as groups.

Currently, to communicate with other peers, all users must enter the same group name when creating a group. By all entering a pre-specified agreed group, communication can then begin. However, there is no way of communicating with users without first knowing this group name. Publishing and searching for available groups could present a user with a list of currently active groups that they can join – which would improve the usability of the applications.

6.2.1.2.2 Discovery

The services layer uses a "busy-waiting" approach to discovery – periodically sending out remote discovery messages, and recreating a list of users in the current peer group.

This is quite a costly approach, with initial implementations leading to a heavy load placed on the system processor. This was improved by reducing the frequency of sending discovery messages, at the expense of introducing a potential delay before discovering the arrival of new users if the discovery attempts are made too infrequent.

However, the problem is that discarding a list of known users and creating a new one, however infrequently, is an inherently inefficient approach.

As I learnt more about the JXTA platform during the **Further Work** section, I found that a potentially better approach could have been to implement an event-handling methodology. I could have sent out a discovery query, and defined a listener to handle replies. With each reply, the listener could simply check if I already know that user, and add them to my list if I do not.

This would have removed the need to repeatedly run the discovery query – in effect, instead of keep asking "Who's there?" and compiling a new list every time, I could have said "Let me know if you get here" and waited for replies. This would also have removed the need for running a thread to do this.

This approach could also have been used elsewhere throughout the services layer. For example, when waiting for incoming messages, the current services layer uses the **BabelListener** class, which also implements a thread. The thread blocks on the input pipe, waiting for incoming messages. This could also have been implemented using events – defining a listener to respond to incoming message events, when creating the pipe.

Again, this would have removed the need for additional threads, and provided an effective way of responding to incoming messages.

I did not use this approach originally because I was still quite new to the JXTA platform when I began the design and implementation work. I did not know that JXTA supported the use of listeners like this, and as I was also quite new to Java, I did not know what events and listeners were to look for them in the JXTA implementation.

As a result, I manually implemented an approach to this using threads which sleep and block while waiting for input. While this is probably not as efficient as the use of events and listeners would have been, I believe that it was still worthwhile as I did learn a lot from the process.

6.2.1.2.3 Whiteboard functionality

The current functionality of the whiteboard is quite basic – intended to demonstrate the potential of a mobile collaborative design tool, but not intended to be a final perfect application. There are a variety of functions that could be added, such as the ability to erase sections of an existing drawing or clearing the canvas completely.

For example, erasing could have been implemented quite simply – by providing a toggle for the foreground colour used for painting – between the user's paint colour and the background colour.

Although time constraints precluded a variety of such additions, while they would have provided a gloss for the final application, I do not believe that they would have demonstrated either any of my coding or design ability or potential for such applications that is not already evident from the applications as provided.

6.2.1.2.4 Multiple applications

One comment raised by two of the evaluators during the evaluation of the user interfaces (discussed previously – 5.3.2.1) was the lack of a central front-end. Instead of starting each application individually, they felt that a single menu where the user could select which application to run would be a nice enhancement.

This would be a relatively simple modification. While I did not feel that the benefits justified the time that it would require to implement, this is something that I would have liked to add if time had allowed.

Another request from the evaluators was the ability to switch between applications while in a peer group. This would be more complicated, but was an interesting suggestion that I would have liked to try. It would provide a huge boost for usability to allow users to discuss something on a chat-room, agree on a file to swap, and then switch to the file transfer mode to do so.

The services layer should support this with minimal modification – it contains a single flag for each application to tell if they are active or not (allowing it to decide what to do with incoming messages) and a handle to each application. Allowing it to set more than one flag to true at any one time would be a trivial modification. Supporting the redrawing and maintaining of the interfaces, and being able to create a new application without a new services layer and telling it how to access the existing services layer, would be more complex, but perfectly possible. By modifying the initialisation procedures of the applications, I think that this would be quite straightforward, and I regretted not thinking of this approach from the start.

6.2.1.2.5 User list flickering

Another comment raised by the evaluators during the user interface evaluation (discussed previously – 5.3.2.1) was about the way that the user list is maintained. The user list is refreshed by the application every few seconds. However, the way that this is done, by emptying the list and refilling it, led to the user list appearing to flicker every few seconds. Two of the evaluators reported that they found this distracting.

If I had more time, I would have liked to implement an approach where the user list is *maintained,* with users who have left being removed and new users added – rather than discarded and recreated. As discussed previously (4.4.2.3), it was implemented this way as I felt that this was the most computationally efficient approach. However, I had failed to consider the aesthetic implications of this decision, and if time allowed I would reconsider this.

6.2.2 PROJECT EXTENSION

Time constraints meant that I was unable to finish the **Further Work** implementation. The current state of the system is that it supports users creating and joining peer groups, and allows two-way communication between users. It can support all three of the applications, and the SocketController correctly manages the creation and access to the new TwowayPipes.

However, I have not completed the functionality to handle when users leave peer groups. The connection manager, timer and listening threads are not correctly closed down, and other users do not always deal with a user leaving correctly. While this means that it cannot support a reliable application, it was usable enough to support some use of the chat-room, whiteboard and file-transfers. This was enough to draw the conclusions discussed previously (6.1.7).

7. CONCLUSIONS

7.1 OVERVIEW

This section provides an analysis of the project as a whole, and looks at how well I met my original objectives. It also provides an overview of what I think I have learnt and achieved from the project.

7.2 ANALYSIS

7.2.1 MEETING PROJECT OBJECTIVES

The **Introduction** (1.2) outlines my reasons for doing this project, and the goals that I hoped to achieve. To review these, I believe that my project successfully satisfied the goals set. The services layer and suite of applications demonstrate how peer-to-peer, server-free networking can provide a viable solution for mobile networking, and demonstrate how the use of collaborative network applications can add value to the use of mobile computing devices.

The services layer and applications produced show an understanding of the implications of using a peer-to-peer topology, and produce a stable, effective collaborative work environment. The applications and the evaluation of the usability of their interfaces, demonstrate an understanding of the implications of developing applications for wireless, mobile devices.

The previous section describing **Further Work** (6.2.1) acknowledges that there is a lot of work that could still be done on this project. While I realise this, I believe that this project was still a success as an educational exercise. I learnt a lot from doing the project in a variety of areas (some of these are included in the following list – 7.3), and my discussion of the remaining areas demonstrates that I have a sufficient understanding of the areas that I would be able to satisfactorily solve these remaining issues if the time allowed.

7.2.2 EVALUATING MY APPROACH

I believe that the use of JXTA as a platform was a good choice, as working with it provided an excellent way to learn about developing peer-to-peer networking and a flexible foundation from which to build my project. I especially think that working on modifying and enhancing an existing system of considerable complexity and size uses different sorts of skills to producing a wholly original smaller system, and I think that this was an interesting addition to the project.

I think that my approach of doing some development in all three layers was very effective – giving me a broader overview of the issues involved in developing network software and in the differences in developing for different network topologies.

I think that the use of formal software development techniques was perhaps a little over-enthusiastic, with a lot of time spent producing lengthy deliverable documents and working through formal development processes that delayed the start of implementation considerably.

7.3 MAIN RESULTS

I believe that I have learnt a lot from this project. The following list highlights some of my achievements in completing this project:

7.3.1 REQUIREMENTS ANALYSIS

- Demonstrated detailed requirements study using UML notation
- Studied the requirements of a new user interface, with quite different requirements to traditional desktop machines
- Investigated uses of mobile computing, and networking
- Demonstrated good use of formal software development procedures

7.3.2 DESIGN

- Demonstrated detailed design study using UML notation
- Designed stable network services layer
- Designed effective file transfer application – relatively reliable and able to deal with missing packets
- Designed reliable multi-user chat room and instant messager
- Designed multi-user collaboration idea storm network whiteboard
- Demonstrated use and knowledge of P2P topology and it's implications – in design and implementation
- Designed effective GUI interfaces suitable for mobile devices
- Identified shortcomings in Sun's JXTA platform – made changes, and introduced new classes and methods

7.3.3 IMPLEMENTATION

- Learnt Java – first time used, including use of:
  - Multithreading – using threads for timers, incoming message listening and other
  - Streams – using a variety of stream types with files, sockets and other
  - File I/O – reading and writing variety of XML and custom files
  - AWT interfaces – implementing complex GUIs using variety of GUI elements
  - Networking – using Java TCP/IP networking libraries including sockets

- Creating and throwing exceptions – using exceptions for error-handling
- Event handling and creating custom listeners
- Interfaces – defining interfaces and implementing implementations
- API – implementing design using restrictive API's for mobile devices

- Learnt JXTA – have become very knowledgeable about the JXTA platform – an excellent networking and peer-to-peer tutorial
- Learnt and used a variety of new tools – log4j, JavaCheck, emulators, tcpdump and other network traffic analysers etc.
- Use of make and other self-written scripts to automate compiling and building

7.3.4 TESTING AND EVALUATION

- Made effective use of Fagan peer code-inspection techniques and Coding Standards definitions to produce high quality code
- Applied Dix's usability principles and heuristic evaluations in a detailed evaluation of the produced application graphical user interfaces
- Designed and created test automated applications to evaluate the performance of system – stressing the network and collecting statistics

7.4 SUMMARY

I believe that this project was largely successful, demonstrating the points that I set out in my original concept and providing opportunities for a lot of learning in networking and software development as I did so.

8. REFERENCES

8.1 OVERVIEW

All references are included in the report body as footnotes.

A full list of the other articles, webpages and documents used while researching this project is too long to include here, but I have included an overview of a cross-section of them in the following subsection.

8.2 BIBLIOGRAPHY

8.2.1 P2P

- 2001 P2P Networking Overview

  "The Emergent P2P Platform of Presence, Identity, and Edge Resources"
  By Clay Shirky, Kelly Truelove, Rael Dornfest & Lucas Gonze
  http://www.oreilly.com/catalog/p2presearch/chapter/ch01.html

- P2P and XML in Business

  A look at the use of XML in P2P application design
  By Brian Buehling
  http://www.xml.com/pub/a/2001/07/11/xmlp2p.html

- Peer-to-Peer with Visual Basic and XML

  An introduction to creating P2P applications for the PocketPC
  By Christian Forsberg
  http://www.microsoft.com/mobile/developer/technicalarticles/peerevb.asp

8.2.2 WIRELESS LANS

- File swapping for wireless devices?

  A look at Endeavors Technology's file-swapping software for wireless devices
  By Ben Charny
  http://news.com.com/2009-1033-251256.html

- 80211-planet

  News and articles about wireless LANs
  http://www.80211-planet.com/

- Wireless P2P Gaming

  A look at using wireless peer-to-peer LANs for applications such as mobile games
  By Christian Forsberg
  http://www.microsoft.com/mobile/pocketpc/columns/wirelessp2p.asp

- Are Device Independent Wireless Internet Applications possible?

  A look at the implications of creating applications for wireless LANs
  By Keith Bigelow
  http://www.onjava.com/pub/a/onjava/2001/06/11/device_ind.html

- The Palm Computing Platform

  An evaluation of the Palm Pilot as a platform for developing wireless applications
  http://www.wirelessdevnet.com/channels/pda/training/palmoverview.html

- An Introduction to Palm Programming

  An introduction to developing applications for the Palm OS platform
  http://www.wirelessdevnet.com/channels/pda/training/palmdevelopment.html

8.2.3 JXTA

- JXTA Protocols Specification

  http://spec.jxta.org/v1.0/docbook/JXTAProtocols.html

- JXTA Platform API javadoc

  Downloadable JXTA javadoc for offline use
  http://download.jxta.org/stablebuilds/javadocs/jxta_doc.zip

- Hello JXTA!

  A walkthrough of a P2P application
  By Raffi Krikorian
  http://www.onjava.com/pub/a/onjava/2001/04/25/jxta.html

- Learning the JXTA Shell

  A tutorial
  By Rael Dornfest
  http://www.openp2p.com/pub/a/p2p/2001/04/25/learning_jxta_shell.html

- JXTA Takes Its Position

  An analysis of the framework
  By Rael Dornfest
  http://www.openp2p.com/pub/a/p2p/2001/04/25/jxta_position.html

- PeerGroup Tutorial

  Small tutorial for creating and joining PeerGroups in JXTA's Java implementation
  http://platform.jxta.org/java/tutorial/PeerGroupSmallTutorial.htm

- The JXTA command shell

  An introduction to configuring, using and extending the JXTA Shell
  By Sing Li
  http://www-106.ibm.com/developerworks/java/library/j-p2pint2/

- The JXTA Story

  An introduction to the JXTA platform and the major protocols
  By Sing Li
  http://www-106.ibm.com/developerworks/java/library/j-p2pint1/

- Project JXTA

  An introduction to the JXTA platform
  By Richard Dragan
  http://www.pcmag.com/print_article/0,3048,a=20102,00.asp

- JXTA White Papers

  Variety of JXTA technical articles
  By Various
  http://www.jxta.org/white_papers.html

8.2.4 JAVA

- Java 2 Platform Standard Edition API Specification

  Java language reference
  http://java.sun.com/products/jdk/1.2/docs/api/index.html

- Writing applications for the J2ME Personal Profile

  An introduction to using the Personal Profile
  By Ed Ort
  http://wireless.java.sun.com/personal/questions/gui/

- Think small: Java on Compaq's iPAQ

  An introduction to developing Java applications for the Compaq iPAQ
  By John Zukowski
  http://www.javaworld.com/javaworld/jw-10-2001/jw-1026-devices.html

8.2.5 OTHER

- Using UML

  Software Engineering with Objects and Components
  By Perdita Stevens and Rob Pooley

- Productivity Improvement Through Defect -Free Development

  Using the Fagan Defect-Free Process
  By Michael Fagan Associates

APPENDICES

PROJECT DELIVERABLES A1

PROJECT RESEARCH A1

Note: *These pages contain a contents listing for the Appendix.The appendices themselves are contained in a separate document, which should be attached to this report.*

The Appendix was bound as a separate document for convenience, because of the large size of the two documents combined.

9. PROJECT DELIVERABLES

DOCUMENTS PRODUCED AS A PART OF DEVELOPMENT

APPENDIX A. Chat Application: Application Layer Requirements

APPENDIX B. Chat Application: Application High Level Design

APPENDIX C. Chat Application: Services Layer Requirements

APPENDIX D. Chat Application: Services Layer Design

10. PROJECT RESEARCH

DOCUMENTS PRODUCED AS A PART OF DEVELOPMENT

APPENDIX E. Chat Application: Requirements Investigation

APPENDIX F. Chat Application: High Level Design work

APPENDIX G. Coding: Implementation Journal

APPENDIX H. Coding Standard

APPENDIX I. JXTA Extension: Protocol Design – Connecting

APPENDIX J. JXTA Extension: Protocol Design – Disconnecting

APPENDIX K. JXTA Extension: XML Messages

APPENDIX L. Code Listings: Original Project Work – *moved to CD-ROM*[44]

APPENDIX M. Code Listings: JXTA Extension Work – *moved to CD-ROM*

---

[44] The CD-ROM attached to this project contains a copy of all the code produced in this project.