DECENTRALIZED AD-HOC GROUPWARE API AND FRAMEWORK FOR MOBILE COLLABORATION

By

WEI-HSING (DAN) LEE

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2000

ACKNOWLEDGMENTS

I would like to thank Dr. Abdelsalam (Sumi) Helal for all his support throughout the project as well as my parents for their encouragement. I would also like to thank Dr. Joachim Hammer and Dr. Doug Dankel for serving on my supervisory committee. I would like to thank God for providing the opportunity and energy to pursue this thesis. Last but not least I would like to thank Harris Corporation, Krispy Kreme and Maui Teriyaki for their support.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

DECENTRALIZED AD-HOC GROUPWARE API AND FRAMEWORK FOR MOBILE COLLABORATION

By

Wei-Hsing Lee

December 2000

Chairman: Abdelsalam (Sumi) Helal
Major Department: Computer and Information Science and Engineering

Collaboration groupware technology has been confined to fixed network, desktop computing with no support for wireless or mobile environments. This thesis pushes the envelope of collaboration technology to support ad-hoc, mobile collaboration using resource-poor portable devices equipped with ad-hoc, wireless networks. The thesis surveys collaboration and mobile computing technologies and analyzes the requirements and application space for mobile collaboration. The main contributions of this thesis are the following 1) a decentralized collaboration framework and environment whose design and implementation emphasize portability, fault-tolerance, and failure recovery, and 2) an Application Programming Interface (API) that allows for the quick development and deployment of tailored collaboration spaces on wireless, portable devices.

CHAPTER 1
INTRODUCTION

In the past few years, the number of portable network devices has increased dramatically. Products such as laptops and personal digital assistants (PDAs) used to operate in a state of solitary confinement, but with the recent acceptance of the Internet and major developments in wireless networks, they are becoming increasingly connected [3,4]. As these products have become more pervasive, new application domains begin to emerge that incorporate and utilize these new communication capabilities in ways that were previously not thought of or were impossible to implement. One area of particular interest is applications allowing users of these connected mobile devices to collaborate in real time over wireless networks and, in particular, ad-hoc wireless networks. One of the biggest advantages of ad-hoc networks is that they can be quickly created without the need for a fixed network infrastructure. Some of the applications for ad-hoc groups include the following scenarios:

- Rescue efforts can be more easily coordinated in emergency situations and disaster areas where a wired infrastructure is not available.

- Groups attending a conference can share ideas and data anywhere by conducting "virtual" meetings.

- Military intelligence and strike teams can be more easily coordinated to provide quicker response time.

- Field survey operations in remote areas with no fixed infrastructure can be facilitated.
- A team of construction workers on a site without a network infrastructure can share blueprints and schematics.
- Staff and security of large events such as concerts, or sporting events can more easily coordinate crowd control and security.

Collaboration over ad-hoc networks differs in requirements from collaboration in a fixed-network environment. Ad-hoc networks generally consist of laptops or other mobile devices linked together through a wireless LAN to form an impromptu network. Since the availability of members within the network can change rapidly due to signal loss or other interferences, the state of the network becomes very dynamic. The dynamic nature of such a network plays a major role in designing the software used for collaboration. While many pieces of software do exist that fulfill collaboration needs, most of them are designed to run on stable fixed networks where the quality of service is much higher and the state of the network is permanent [15]. The key characteristic that makes them ill suited for ad-hoc networks is their single point of failure. The majority of existing solutions are based on a centralized architecture. This requires the collaborator to log into a central server, which is fine for a fixed network, but assume the same piece of software is run on an ad-hoc network. If the central server goes out of range or is experiencing high levels of interference, then the collaborative session will either freeze or cease to exist which is unacceptable for ad-hoc networks, where members can easily float in and out of range [21]. In addition to their single point of failure, the current generation of software is very resource-intensive in terms of both memory and processor

requirements. While both of these resources are abundant in some laptops, usage should still be conserved to extend battery life as well as to increase the range of target devices. Such resources, however, are scarce in smaller mobile devices.

This thesis presents a lightweight framework and application programming interface (API) based on a distributed architecture designed for mobile collaboration that eliminates the single point of failure. The discussion goes into detail about the motivations behind the design decisions and how to best leverage the technologies that are available today. The framework is structured so as to hide the complexities of implementing a distributed service from the application developer while providing an API with the flexibility needed to optimize and customize the functionality. The implementation of the API is Java-based, due to Java's object-oriented nature and code portability. By using an object-oriented language, new services or applications can easily inherit basic capabilities that allow them to be used in the framework. Since services are reusable, integrating previously developed services results in reduced development and deployment time, allowing programmers to rapidly create custom applications that fit specific needs. While the object-oriented nature of Java encourages code reuse, its code portability and the premise "write once; run anywhere" not only allows the framework to function on a number of heterogeneous devices, its compact code also reduces message sizes [18]. The framework focuses on providing a fault-tolerant environment allowing collaborators to float in and out of the ad-hoc network without causing disruption to other collaborators and the session. Additionally, an implementation of the environment is provided with sample services that can be used to build applications, as well as a sample

client. To understand the motivations behind the design decision, one must examine the underlying technologies and its limitations.

Thesis Objectives

The main objective of this thesis is to design a framework for wireless ad-hoc collaboration and to provide an Application Programming Interface (API) that would enable the developers to create custom services and applications using the framework [19]. The goals are summarized as follows:

1. Propose a framework for use by collaborative services in a wireless ad-hoc network.
2. Provide a lightweight API that allows developers to create small, and highly optimized collaborative applications that can run on mobile devices.
3. Provide flexible API for developers so as to facilitate rapid application development.
4. Implement the framework in Java for deployment over a number of heterogeneous devices.
5. Develop services that include text chat, shared whiteboard, shared images, and global positioning system navigation.
6. Develop a sample application using the provided services.

Structure of the Thesis

Chapters 1 and 2 provide background knowledge and fundamental necessary for an in-depth discussion of the API and the environment. They provide both introductory remarks about the scope of the thesis and the reasons for developing an API for wireless

collaboration. Additionally, descriptions of the design criteria that were used in the design of the API and evaluations of current hardware technologies available for implementation of a mobile collaborative system are also found. In-depth discussions of the API and environment are found in chapter 3 while chapter 4 provides tutorials on designing collaborative applications and develop custom services using the API. The thesis is concluded with a discussion of possible future work and improvements to the YCab API and environment.

CHAPTER 2
REQUIREMENTS FOR EFFICIENT WIRELESS COLLABORATION

What to Include in the API?

In designing the framework, the goal was to create a flexible yet small API that would enable developers to quickly create useful collaborative applications with a minimal learning curve. The challenge of balancing the flexibility of the API while keeping the code size small and efficient was not trivial. Adding flexibility to the API increases both code size and the learning curve. On the other hand, decreasing flexibility also decreases the learning curve and limits the range of applications that can be developed. To add to the already difficult situation, the framework needs to be fault-tolerant. To decide on what features to include, the potential target platform and the environment had to be carefully examined. By looking at the available resources, certain heavy features of the API could be easily eliminated from contention. This section examines potential applications useful in a collaborative system as well as target platforms and the environment for the API used to develop such system. To develop such API the desired application class must be clearly defined. The application class then defines the capabilities of the system, which in turn dictates the design of the API (see Figure 1).

Figure 1 - The type of applications and their capabilities drive the design of the API.

Platform Choices and Reasons

In deciding the target platform for the API the requirements and constraints had to be considered. Ad-hoc networks are generally created on the fly with little or no infrastructure support. In examining this situation, other factors came into play. Along with the lack of infrastructure, an ad-hoc network usually is formed with mobile devices that lack both battery power and processing power. To compound the situation, the network reliability is significantly lower than that of a fixed infrastructure and is subject to frequent disconnections. By carefully examining these constraints and requirements, a few potential target platforms may be eliminated from contention. Fixed network devices such as workstations, servers, and other Internet devices generally do no participate in forming ad-hoc networks. By eliminating these platforms, the user gives up the luxury of a high power processor along with a reliable network connection. Although these platforms are not targeted, it does not prevent the framework from reliably running on these devices. The main concern is to optimize the framework for devices that generally form an ad-hoc network. Some of the potential devices include laptops, PDA's, and other Mobile Information Devices (MID). Although there are large differences in the capabilities of these devices, a framework should be developed that works acceptably on all points of the spectrum. The following is a list of the potential devices:

1) Desktops – These devices operate in a fixed environment and as such do not have to deal with issues related to a mobile environment. While it would not make much sense to run an ad-hoc collaborative application in the fixed environment, there is nothing that would prevent one from doing so. However, such application would most certainly not take full advantage of the resources provided by those devices.
2) Laptops – These are the most powerful mobile devices available today. In terms of processing power and screen real estate, the capabilities of devices in this category rival desktop computers. The major drawback is the lifespan of the battery dictated by the weight constraints.
3) Jupiter class devices – Products such as Vadem Clio and Hewlett Packard's Jornada are examples of devices in this category. Compared to laptops, these devices have smaller screen real estate as well as less powerful processors, but along with those tradeoffs, there are distinct advantages, such as longer battery life and increased portability. Another major difference in these devices is the type of operating systems (OS) they use. For the most part, Jupiter class devices run small, but feature restricted operating systems as opposed to laptops that run desktop operating systems.
4) Palm-sized devices or PDAs - Compared to Jupiter class devices, palm-sized devices are substantially smaller and more portable. Although they provide the convenience of a small package they are generally less powerful than Jupiter devices and laptops. Some of the major tradeoffs are the screen size and the processing power. These devices also run alternative operating

systems that are optimized for smaller, less powerful devices. Some of those operating systems include Windows CE, Palm OS, and Symbian EPOC.

5) Mobile Information Devices (MID) – The computing power in devices of this category is severely limited. Cell phones and two-way pagers make up the bulk of the devices in this group. Screen space and processing power are at premiums in these devices. Additionally, these devices generally run proprietary operating systems, which are severely limited in features.

With the wide-ranging capabilities of mobile devices, it was essential to reduce the code size while at the same time increasing flexibility. While some of the platforms listed may have capabilities that rival those of desktops the goal was to develop a framework capable of deployment over a heterogeneous network. The main idea is to target the least powerful since upwards scaling is usually not an issue. To ensure usefulness, MID's were eliminated as a target platform because of their lack of screen real estate and processing power. Additionally, the user interfaces for MID's tend to be specialized and inflexible. Palm-sized devices however incorporate a substantially larger screen with a more intuitive and flexible user interface. They tend to include pointer devices that mimic mouse-based motion as well as more powerful processors. The major drawback to developing a framework for devices of this class is the diverse nature of their operating systems, which include Window CE, Symbian EPOC, and Palm OS [8,17,12]. Nevertheless, it was decided to support palm-sized devices as the minimum requirement for the collaborative environment, dubbed YCab.

Table 1 - Summary of mobile platforms.

	Mobile Information Devices (MID)	Handheld	Jupiter (Portable) class devices	Laptop	Desktop
Device size	Pager, Cell phone	Small, under 20 oz., very mobile	Fairly small, up to one pound	Fairly large, typically few pounds, notebook size	Large, heavy, fixed location
CPU power	Very limited and specialized	Limited	Low power	Powerful	Virtually unlimited
Display size	Less than 11 lines of text	Small, often text only, and monochrome	Medium, color	Large, high-resolution, color	Large display size, high-resolution, color
Memory	Typically under 4 MB and usually not expandable	Typically under 4MB, not always possible to expand	Typically 4 MB to 32MB, possible to expand	Typically in the range of 32MB to 256MB, fairly easily expandable	Unlimited, easily expandable
Network capabilities	Low bandwidth and high latency	Low bandwidth, high latency, usually highly integrated with the device, mobile networks	Low bandwidth, high latency, possible to tie into fixed network infrastructure, but not easily	Dependent on location, can easily be tied into fixed network, or highly mobile network	High bandwidth, low latency, fixed location
Location/ typical use	Extremely small and mobile	Small and extremely mobile	Fairly small and mobile	Fairly powerful devices, can be equally used in fixed and mobile environment	Large devices, fixed location
Input devices / navigation	Specialized keys.	Limited, small keypad and/or pen-based.	Usually pen-based, small keyboard, usually no mouse, limited voice	Typically full-size keyboard, mouse, voice	Keyboard, mouse, voice, etc.

In many instances the choice of operating system dictates the target platform. For example, choosing the handheld platform as the target platform automatically implies that desktop OS will not be used. Currently the choices for the OS can be broken down into three main categories. The most specialized operating systems reside on the palm

devices. Due to their small form factor and lack of resources, the operating systems typically lack the flexibility and features of their larger counterparts. An example of such an OS is Palm OS. A step up in terms of functionality and flexibility are operating systems, such as Windows CE and Symbian EPOC. These operating systems tend to reside in the larger and more powerful systems found in the Jupiter class of devices. In exchange for more memory and processor requirements, these operating systems typically allow a broader range of applications. Operating systems such as Windows 98/NT, Unix, and MacOS provide even more robustness and flexibility, but tend to have steep resource requirements. These systems are typically found on desktops and laptops. See Table 1 for a summary of operating systems. By targeting deployment of the framework over such an assortment of devices and operating systems, it was necessary that the framework be implemented as portable code. This brought up a different issue, mainly what type of development language to use.

Table 2 - Comparison of operating systems.

OS	Palm OS	Symbian EPOC	Windows CE / Pocket PC	Windows 98/NT/2000, Unix, MacOS
Memory footprint	Very small	Small to Medium	Small to medium	Large
Memory requirements	1MB – 4MB	4 MB – 32 MB	16 MB – 32MB	> 32 MB
Development/debugging tools	Some	Some	Some	Numerous

Implementation Language

With such a diverse assortment of devices the capabilities of their operating systems and hardware range dramatically. Since the market for mobile devices is still in its infancy, it cannot be predicted which platforms will survive and which will fail. As

the market matures, the devices on the market should start to share similar traits and capabilities. The possible languages taken into consideration for developing YCab were C, C++, and Java. See Table 3 for the summary of each language.

Table 3 - Comparison of C/C++ and Java environments.

Language	C/C++	Java
Code	Native, platform specific	Universal, platform independent
Development difficulty/speed	Fair/medium	Moderate/fast
Porting to other platforms	Difficult	Very little or no porting
Development/Debugging tools	Excellent	Excellent
Speed	Excellent/platform optimized	Poor to good, usually not platform optimized

Taking a look at the priorities for the YCab framework and given the turbulence in the mobile device market, a language was needed that allows for portable and secure code. Also the learning curve for developing applications using YCab should be minimal. In examining C and C++ as possible implementation languages, it became apparent that a significant amount of effort would be required to port the code to different devices. The learning curve of theses languages is also much greater than developing with a language such as Java. C and C++ do however offer the benefits of efficiency and speed found in native code. Java on the other hand, is slow and inefficient as compared to C and C++, but it does offer numerous benefits that cannot be taken lightly. Java by design is a platform independent language running on a virtual machine. The virtual machine acts as a common interface among all platforms allowing the code to run on a multitude of devices without the need to recompile the code. Although the requirement of a virtual machine adds to the memory requirements of the software, the generated byte code is quite small compared to the binaries generated by the C and C++ languages. While some might argue that the interpreted nature of Java is not suitable for low CPU

power devices, great strides have been made in recent months by Sun Microsystems and device manufactures to ensure that this will not present a problem [5]. Also, given the volatility of the mobile device market, more and more manufactures are beginning to implement a Java Virtual Machine (JVM) for their devices. Additionally, the object-oriented nature of Java not only allows the developer to more naturally create abstract objects but also promotes and facilitates code reuse. The elegance and simplicity of the language help to reduce the learning curve allowing applications to be developed rapidly and efficiently. The advantages of using Java clearly outweigh the disadvantages in a mobile environment and, therefore, Java was chosen as the language to used implement the YCab framework.

JVM Evaluation

For the purposes of this project it became apparent that the quality of Java Virtual Machine (JVM) would have significant impact on the performance of the system. As a result different Virtual Machines offered by different vendors were evaluated.

Microsoft's JVM for Windows CE claims to fully support Java1.1. However, the only version that could be located was a two-year old beta version. Apparently, Microsoft had ceased the development of JVM for Windows CE, due to the lawsuit from Sun Microsystems. While the JVM claimed full support for Java 1.1, tests showed that it had several problems that mainly related to the AWT components, such as excessive flickering and incorrectly rendered components. Additionally, the JVM had very poor performance, especially when it came to manipulating the GUI. The final problem was that most of the Java applications were simply unusable with this JVM.

Another product that was evaluated was CrEme from NSIcom. Similar to Microsoft's JVM it claimed full support for Java 1.1. In addition NSIcom claims that their implementation is highly optimized for different processor types. Test showed that although CrEme performed slightly better than Microsoft's JVM, the implementation suffered from the same problems related to AWT components. Additionally, the memory requirements for CrEme were quite significant. The 16MB of memory that Vadem Clio has is barely sufficient to run even the simplest Java applications. While this product is a step-up from the Microsoft's JVM, it is not enough to efficiently run Java applications on Windows CE platform.

Another Virtual Machine (VM) that was evaluated was Sun's PJava VM. Instead of supporting full Java 1.1 specifications, this VM supports PJava1.1 specification. PJava was designed for small devices and, therefore, its footprint is significantly smaller than the footprint of the other two VMs. The drawback is that Java applications that do not conform to PJava1.1 specification will not run successfully on that VM. Fortunately, PJava specifications contain the majority of the most widely used classes and, most importantly, include full support for networking and AWT. The early beta version showed remarkable increase in performance, even though the VM claimed to be highly unoptimized. Unfortunately, the early version had experienced the same AWT problems that plagued the other two virtual machines. These problems were fixed in the later releases of PJava VM and, as a result, proved a VM that was more than capable of supporting the intended collaborative system.

Additionally, one more JVM was briefly considered. Jeode VM from Insignia Solutions claims that its performance is close to native code. However, at the time of this

project Jeode was geared more towards embedded systems, and did not support AWT components (or any other GUI). Although, Insignia Solution promised AWT support in future versions, a version without GUI support does not provide a viable solution for collaborative systems.

	Microsoft JVM	CrEme	PJava VM	Jeode
Manufacturer	Microsoft	NSIcom	Sun Microsystems	Insignia Solutions
Version	Beta	1.0	1.0	1.6
AWT support	Yes	Yes	Yes	No
GUI support	Yes	Yes	Yes	No
Still in development?	No	Yes	Yes	Yes
Performance	Very poor	Poor	Good	Not tested

Network Adapters Evaluation

The available mobile devices in part dictated the criteria for the network adapters. The network adapters had to provide Windows CE drivers with UDP multicast support. The devices must also work with portable device and must provide PCMCIA interface. Finally, the card must support both ad-hoc modes and peer-to-peer networking. Unlike the desktop environment that has a plethora of network adapters, the Windows CE market is quite small so there were only few possible choices for network adapter hardware. The following network cards were evaluated as possible choices for the collaborative system: BayNetwork's Baystack 660, Proxim's RangeLan2, Aironet, and Lucent's WaveLan.

BayNetwork (Nortel) Baystack 660 – these cards work great in ad-hoc mode, but only with Win98/NT [9]. They use direct sequencing and fully support 802.11-network protocols. Unfortunately, no Windows CE drivers were available although future support was announced.

Proxim RangeLan2 – these network cards use frequency hopping and do not support 802.11-network protocol [13]. Proxim claims full support of Window CE. The assumption was that these devices would work with Windows CE in ad-hoc mode. However, there were several problems when trying to setup a test group. First, the Windows CE driver from the Proxim website did not work with the Vadem Clio. The driver documentation does not contain enough Windows CE specific instructions to solve this issue. Eventually, a "working" driver was located on the Vadem website. With the driver from Vadem's website the cards indicated that they are aware of other devices within the range. However, for the cards to work, one of the devices in the group has to act as a Master to synchronize frequency hopping. Unfortunately, the Windows CE driver does not allow one to configure Windows CE device as a Master device. Including a laptop running Win98/NT into the group can solve this problem. This laptop's sole purpose would be to act as a Master device (hoping that in the future revisions of the Proxim driver, it would be possible to configure Windows CE device as a Master). This issue was documented in the Proxim manual for the driver. However, once a laptop was placed in the group to act as a master the Windows CE devices could only communicate with the Master device (a laptop in this case) and do not support peer-to-peer communication. The only peer-to-peer communication is between a client (defined as a Station) and the Master. The only way to solve this problem is to have the Master be an access point, which then can be configured to provide limited peer-to-peer communication. Since the group is a mobile ad-hoc network, a fixed-location access point is not a possibility. In addition, any attempt to initiate communication from the laptop (Master) to any client failed because Windows CE drivers are suspended while

idle and cannot be awakened, by an incoming message. While, supposedly, this "feature" can be somehow disabled (it is not obvious and no specific instructions are provided in the documentation) it was not attempted due to the other issues with the network cards. Both the driver documentation and a Proxim technician confirmed those issues.

Aironet – these are direct-sequence 802.11 cards that claim to provide full support for ad-hoc networking and Windows CE devices [1]. The cards were tested at EduTech – a local reseller. Driver installation was fairly simple, but attempts to get the cards to work in ad-hoc mode proved unsuccessful. Despite the help of the local reseller and Aironet's technical support these issues were not resolved. The cards were finally configured in infrastructure mode (with a fixed access point) and a communication was established between the access point and a Clio (or a laptop). However, peer-to-peer communication between two Clio's (through the access point) could not be performed even though the software claims to support that feature. The only conclusion drawn was that possibly the latest version of the network cards with the latest firmware and drivers might solve this problem, but the latest versions were not available for testing.

Lucent WaveLan (also known as Orinoco) – these are another direct-sequence cards conforming to 802.11b standard. They support up to 11Mb/sec transfer rate, which would provide more than sufficient bandwidth. Such cards are well suited for implementing mobile multimedia delivery systems, but they experience a problem similar to the problem experienced with Aironet cards. Windows CE devices equipped with WaveLAN Turbo 11Mb cards and configured for ad-hoc mode do not receive multicast packets send by other devices in the ad-hoc group. After successfully installing and configuring Windows CE devices, the cards were tested by running client software on

one of the devices and server software on the other. The tests were performed by sending UDP packets from the server to the client. In this test the cards and the devices performed as expected. Next, the cards were tested using multicast IP. All the devices ran software that acted both as a server and a client. In this configuration each of the devices should receive its own packets as well as packets from the other devices. However, neither of the devices would receive the packets multicast by the other devices. The only multicast packets that were received were the ones sent by the device itself. However, when of the Windows CE devices was replaced with a laptop running Win98SE, the laptop would receive both the packets sent by itself and the packets sent by the Windows CE device. Strangely, the Windows CE device would still only receive the packets sent by itself and it would not receive packets that originated from the laptop.

For some reason, Windows CE devices do not receive multicast packets when in ad-hoc mode. Since there was not a fixed access point it was impossible to verify whether such problem exists in infrastructure mode. According to Lucent Technical support, the problems are due to poor networking support included in Windows CE. Further test were made that seemed to support that conclusion, although Microsoft denied that there are any issues with Windows CE networking. Despite extended efforts it was not possible to successfully put together an ad-hoc wireless system with portable devices that would be able to support wireless collaboration. Table 4 summarizes the network adapters evaluated for the purpose of this thesis.

Table 4 - Comparison of wireless network adapters.

	BayNetworks Baystack 660	Proxim RangeLan2	Aironet	Lucent WaveLan (Orinoco)
Network type	Direct Sequence Spread Spectrum/ 802.11	Frequency hopping	Direct sequence / 802.11	Direct sequence / 802.11b
Adapter type	PCMCIA	PCMCIA	PCMCIA	PCMCIA
Drivers	Windows	Windows, Windows CE	Windows, Windows CE	Windows, Windows CE, Linux
Bandwidth	2Mb	2Mb	2Mb	11Mb
TCP support	Yes	Yes	Yes	Yes
Ad-hoc mode	Yes	Yes, but requires Master device to synchronize communication; Windows CE device cannot be designated as a Master device	Yes	Yes
Peer-to-peer communication	Yes	No. The only peer-to-peer communication is between client and Master. Limited peer-to-peer possible with fixed access point	Yes. Claims peer-to-peer communication, however, we were not able to get it working	Yes
Multicast	Yes	Not tested	Not tested	Yes (works well with Windows machines, does not work with Windows CE devices even though it claims support)

System Design Issues

In designing the YCab framework, the actual operating environment played a large role in the system design. One of the main concerns was the fault tolerance of the system. In most fixed networks, the reliability of the network is relatively stable in comparison to an ad-hoc wireless network. YCab cannot depend on a reliable network and, therefore, a more fault tolerant design is required.

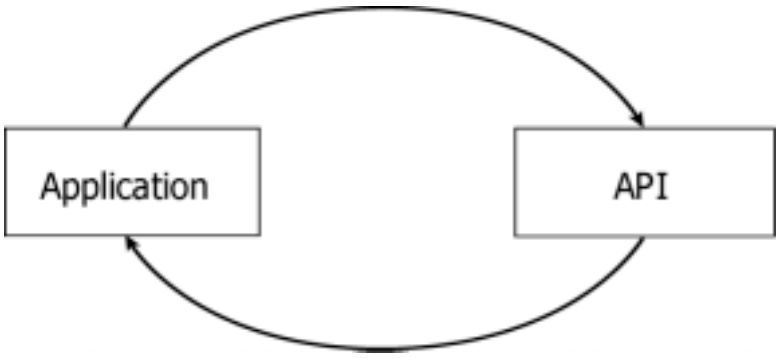

Figure 2 - Application and API

In most collaborative applications today, the participating client or session collaborator is required to log into a central server. Session information and control is disseminated from the server providing a centralized control node. While centralized control is not a problem for applications running on fixed networks, since reliability is superb, the same applications running on ad-hoc networks have very poor performance. The main reason is their single point of failure. Since nodes in ad-hoc networks tend to float in and out of range, applications relying on centralized servers will become quite useless if the server floats out of range. In this regard, centralized control is not optimal. A more successful approach is to distribute the control over all the nodes in the network. By distributing the control and data over multiple nodes, the single point of failure is eliminated, thus providing a more robust and stable application for dynamic networks. Since the target environment for YCab is an ad-hoc network, its architecture needs to avoid a single point of failure. By decentralizing control, a framework could be created that is capable of operating in a dynamic environment.

Table 5 - Comparison of centralized system and de-centralized system.

	Centralized system	De-centralized system
Overview	Single server acts as a focal point, clients connect to the server, the session cannot be started or continued without the server.	All clients execute the same code and communicate with each other.
Fault tolerance	Low – the server is a single point of failure. If a server leaves the session, the session cannot continue, unless a new server is brought up on another client.	High – since all clients execute the same code, the ramifications of one client leaving are significantly smaller than in a centralized system.
Session control (floor control)	The server is in the control of the session, if the server dies there is no session control unless another server is designated.	The session is in control of one of the clients. If that client dies, the control could be quickly and easily transferred to another client.

During the development period of YCab, the potential functionality of a collaborative application had to be determined. After looking at a wide range of collaborative applications a flexible API was chosen, whose capability ranged between Microsoft's NetMeeting and America Online Instant Messenger [2,20]. See Table 6 for a comparison of collaborative applications.

Table 6 - Comparison of collaborative applications.

Application	Microsoft NetMeeting	NCSA Habanero Project	America OnLine Instant Messenger
Capabilities	Full desktop and application sharing	Application sharing	Text messaging and image sharing
Architecture	Centralized	Centralized	Centralized
System Resource Requirements	Pentium 90, 16 MB RAM, Windows 95/98/NT	Unspecified, JVM requirement	Pentium, 16 MB RAM, Windows 95/98/NT, Mac

Since YCab is supposed to reside in a small, memory limited device, full desktop sharing is not a targeted application. The NCSA Habanero Project does support application sharing but does not support desktop sharing [10]. Other applications such as

text chat and shared whiteboard are within the paradigm of collaboration on a small device.

Target Platform Summary

The targeted platform for the development and testing of the API was selected based on criteria described in the preceding sections. The setup consists of several Clio portable devices manufactured by Vadem Inc. running Windows CE version 2.11. Each Vadem Clio has a NEC V4111 MIPS processor and 16MB of RAM. One of the best features of this device is its 9.4-inch, touch-sensitive screen. The communication link is established using Lucent WaveLAN cards. However, due to hardware problems described in section System Design Issues, an alternative platform was selected. The alternative platform consists of IBM Thinkpad 390 laptops equipped with 64MB of memory and a 14.1" display, running a combination of Windows 98 and Windows 2000 operating systems. The network cards used were the same as originally intended since they provide proper functionality under those operating systems.

CHAPTER 3
WIRELESS COLLABORATION API AND ENVIRONMENT

Overview

The goal of this project was to design and implement an API for collaboration in mobile environments. In addition, the environment to be used with the API was implemented. The environment defines the rules and protocols for the proper use of this API. The framework also takes care of the environment, and lets developers quickly develop and deploy applications on the target platform. Other criteria used for the design were the footprint of the design and its applicability to mobile devices, its portability across platforms, and its effectiveness in wireless environment.

Description of the API

The YCab API was developed using Sun's Java API [6]. Any platform that supports Java can use the API. At the minimum it requires a Java 1.1 compliant virtual machine (VM). The API also conforms to PJava 1.1 specifications, so that it can run on many portable devices. All the GUI components and client GUI are implemented using Java's AWT. While it is possible to use Swing and Java 1.2 specific features, any application using these specifications might not conform to the PJava specifications.

The YCab API provides developers with a high-degree of flexibility when designing custom applications. Due to the environment being already provided, application developers can work on several levels to design the desired application as shown in Figure 3.

At the simplest level, the provided YCab application can be used. This is a pre-compiled application that uses all the provided services. That option, however, provides limited flexibility when customizing the appearance of the application and its services. See section

Description of the Client

GUI Description

This section describes the functionality of the YCab application. This application is included in the API and can be used as a fully featured collaborative tool. This section explains how the application operates and describes the steps necessary to successfully participate in a session.

Although applications that were designed using the YCab API might differ from each other, they all contain the same basic features and functionality. The YCab API allows for creation of custom collaborative applications that will significantly differ from each other with all of them experiencing similar behavior and containing the same basic features. This section describes the basic behavior of the client as shown in Figure 4 - YCab application in session. For a description of a specific service, refer to an appropriate section.

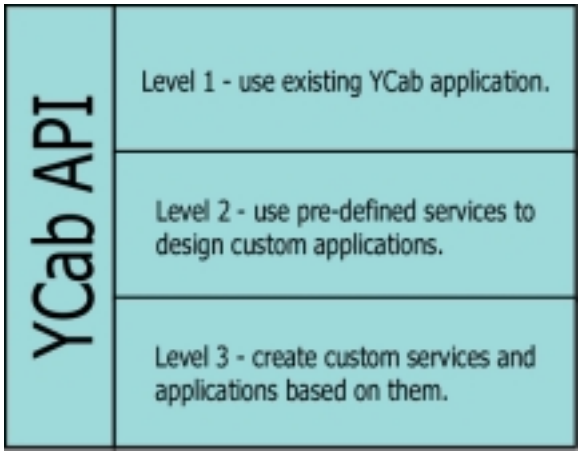

Figure 3 - Three levels at which the YCab API can be used.

To fully customize the application for the target device, developers can also develop their own application using the provided services. This requires knowledge of the client framework, but since pre-defined services are used it does not require extensive knowledge of the Service API. While at this level the developers are restricted to using the provided services (or services designed by other developers), they are free to customize the appearance of their application. They can specify which services are to be included in the application, and how they are to be arranged in the application. They have complete freedom to customize those services and the dimensions of the entire application. In addition they can enable or disable certain features of those services without the need to re-design each service. Features such as service optimization, state recovery, and others can be easily added to or removed from any service with just a simple method invocation. See CHAPTER 4 for a tutorial explaining how to create an application using predefined services.

While this level of customization might be sufficient for some applications, some developers might opt to develop their own services. This is where the power of the YCab

API becomes apparent. By using the Service API, developers can easily create powerful modules that can be later plugged in to any custom application. The skeleton Service class provided with the API takes care of integrating the service into the environment and takes care of communication between the clients without the need for the developer to worry about those issues and the protocols used. Any service that implements the services interface is guaranteed to work with the environment. While the default service skeleton provides significant functionality, the developers are not restricted to those functionalities. They can easily add more functionality, or extend or overwrite the existing ones.

Description of the Client

GUI Description

This section describes the functionality of the YCab application. This application is included in the API and can be used as a fully featured collaborative tool. This section explains how the application operates and describes the steps necessary to successfully participate in a session.

Although applications that were designed using the YCab API might differ from each other, they all contain the same basic features and functionality. The YCab API allows for creation of custom collaborative applications that will significantly differ from each other with all of them experiencing similar behavior and containing the same basic features. This section describes the basic behavior of the client as shown in Figure 4. For a description of a specific service, refer to an appropriate section.

Figure 4 - YCab application in session.

The YCab application consists of a single window. That window contains both window components as well as a menu bar associated with that window. The YCab window is actually divided into separate areas that are surrounded by gray borders. Each one of these areas is a separate module that is encapsulated in the application. These modules are called services, and each service performs a specific task. Not all services have window components. In the YCab application the services that are visible on the screen are (clockwise, starting in the upper-left corner): Client Info service, Image Viewer service, GPS service, Text Chat service, and Whiteboard service. In addition, some of the services have menus associated with them. These menus appear in the window menu bar and bear the name of the service. Although, the number of menus in

the menu bar will vary from application to application, each YCab application will always have the following menus: Session, About, and Session Manager.

Session Menu

This menu contains actions that are related to the application.

- Join Session –joins a session or creates a new session if one does not exist (see section Joining Session for description on how to join a session). This menu is disabled if the client is already in a session.
- Leave Session – leaves an existing session. See section Leaving Session for a description on how to leave a session. This menu is disabled if the client is not connected to any session.
- Quit – quits the application. Also leaves the session if the client is already in a session.

Help Menu

The about menu contains information about the application.

- About – creates a splash screen that displays the current version of YCab.

Session Manager Menu

The Session Manager menu is a menu associated with the Session Manager service. This menu is operational only when the client is participating in a session and provides services related to the session's process.

- Set as Coordinator – sets or unsets a client as the session coordinator. This toggle menu allows the user to designate the client as the session coordinator. Once the client has been designated the coordinator, a check mark appears next to the

option. Selecting this option again will indicate that the client is no longer the session coordinator.

- State Recovery – recovers the state for the client. Selecting this option will recover the state for all services that have state recovery enabled assuming that state recovery can be performed.

In addition to these standard menus, each application can also have menus that are associated with the services that are in the application. Although the designers of the services are free to include any functionality that they wish, there are three common menus that can be associated with each service: Restore State, Optimization Mode, and Control.

- Restore State – this option restores the state information for the given service. This is similar to the State Recovery menu in the Session Manager, except that the state recovery is performed only for that particular service. This option appears only if the state recovery has been enabled for a service and is especially useful if the state of one service has become corrupted while other services are still fully operational.
- Optimization – sets the optimization level for the service. When this menu is selected it expands to reveal three possible choices: *Real Time*, *High Bandwidth*, and *Low Bandwidth*. A checkmark appears next to the currently selected optimization mode. The optimization mode refers to the service's usage of network resources. Although, the implementation of this option is left to the service designer and is different for each service, the *Real Time*

option usually refers to the highest network bandwidth, while *Low Bandwidth* refers to the minimal network bandwidth requirements.

- Control – controls the execution of a Threaded Service. When this menu is selected it expands to reveal two possible choices: Start and Stop. This option is used to start and stop a given service.

Joining Session

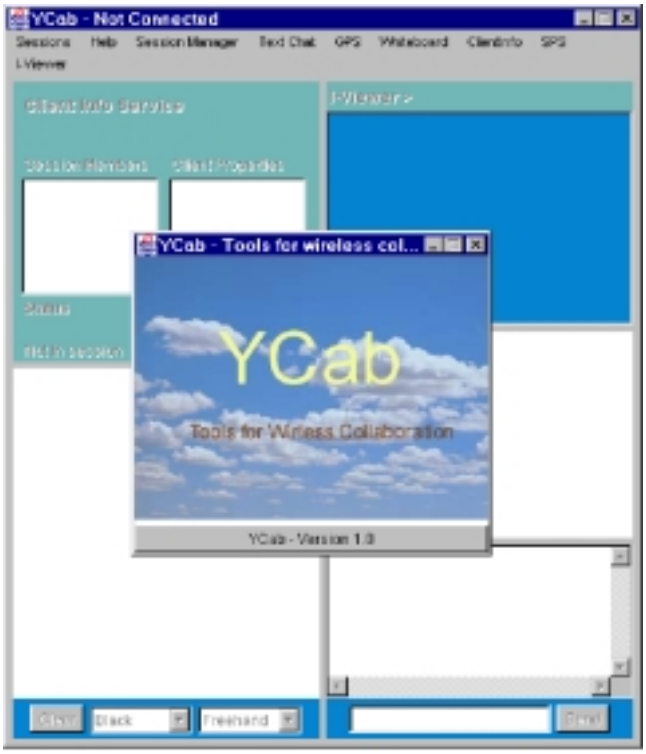

Figure 5 - YCab application immediately after it had been started.

After launching the YCab application, the application frame and the splash screen should appear as shown in Figure 5. The splash screen provides information as to what is the current version of the YCab application. Although in some cases YCab applications with different version numbers will be able to communicate with each other, the user should make sure that all clients run the same version of the software. Once the splash

screen has been closed, either by clicking on the YCab button on the splash screen or clicking the ‘X’ button on the splash screen, the client window appears with all the services disabled (their components are grayed-out and do not respond to user events). This is due to the fact that the client is not connected to the session and having the services respond to user actions would not make sense in this situation. Some versions of YCab can have the automatic displaying of splash screen disabled. In this case, the splash screen can be displayed by selecting About->Help menu from the menu bar. To enable the services either a new session has to be started or the client needs to connect to an existing session. To join the session the user selects Sessions-> Join Session from the menu bar. This creates the Connection Properties dialog window as shown in Figure 6.

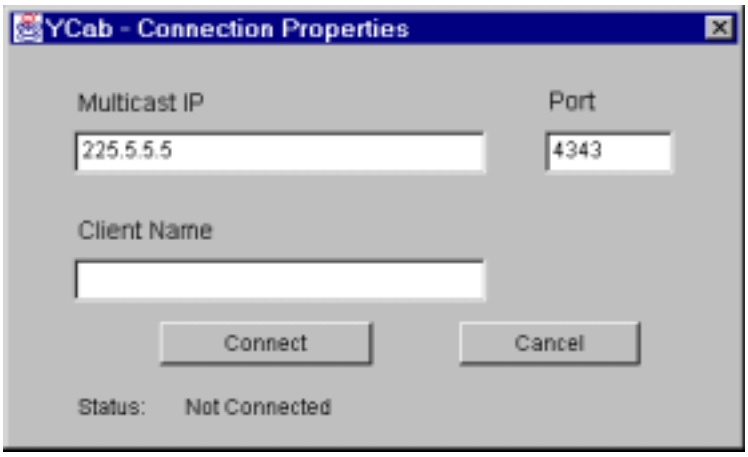

Figure 6 - Connection dialog window used to specify session properties.

This window allows the user to specify the multicast IP address for the session, the port number to be used for communication, and the name of the client. Most likely the Multicast IP and Port fields will have some predefined values. These can be changed or accepted as default. The final item that needs to be specified is the client name. The Client Name field contains the name of the client that is used to uniquely identify each

client in the session. It can contain any combination of letters and numbers as well as certain special characters. Once all the fields have been specified the user clicks on the "Connect" button to try to connect to the session. While the application is attempting to connect to the session, the Connection Properties dialog window will look like Figure 7.

Figure 7 - Connection Properties windows during the process of connecting to a session.

The status field indicates the progress of the connection. During the negotiation process the "Connect" button is disabled. If for some reason the client is not able to connect to the network, a message will appear in the Status field and the "Connect" button will be enabled again. At this point the user can try to reconnect. Similarly, if the user specifies an invalid client name or name that is already in use by another client, an appropriate message is displayed. On the other hand, if everything was successful, the Connection Properties dialog window disappears and screen should look like the one in Figure 8. Depending on the connection speed the client should establish connection with the session within few seconds of clicking on the "Connect" button.

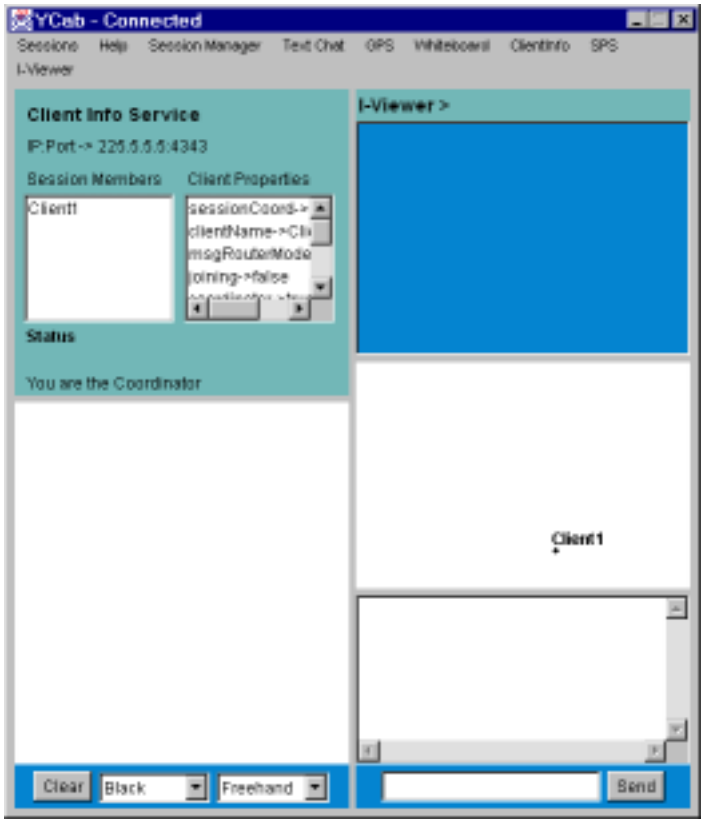

Figure 8 - YCab application immediately after new session has been created.

The title bar indicates that the client is in the session and the Session Members window lists the client's name. Also, all of the services have been enabled. At this point the client can fully participate in the session. This situation assumes that a brand new session has been created. Therefore, as indicated by the Client Info Service, the client is automatically designated as the Session Coordinator. In case the client is joining an existing session already in progress the screen might resemble the screen in Figure 4 and the procedure will be slightly different. After connecting to an existing session the status window in the Client Info service will indicate that the client is in State Recovery mode. This indicates that there are other participants in the session (as expected) and that the client is waiting to be brought up to the state consistent with the state of other clients.

Depending on the speed of the connection and the amount of state information, the state recovery process might take anywhere from just a few seconds to about a minute. In addition, the client's request for state recovery might result in starting a leader election process if a session coordinator was not present in the session. This will further delay the client's ability to participate in the session, but will not require any further actions on the collaborator's behalf.

Participating in a Session

Participation in a session depends on the services that are included in the application. In general, a collaborative session consists of a small group of clients, of which one of them is designated session coordinator. The coordinator has special privileges and is also responsible for managing the session. The coordinator is also responsible for restoring state information to new clients, although that process is transparent to the user.

Leaving Session

Once the user is finished participating in the session the client should gracefully leave the session. This way other session participants will be immediately notified of client's departure. There are two ways to leave a session.

One is to leave the session, but leave the application running. This way the application can later be used to re-join the same session or to connect to another session. To leave the session the user selects Sessions->Leave Session item from the menu bar. This option is only enabled when the client is connected to a session. Upon selecting this option the title bar of the application should indicate that the client is no longer connected. In addition, all the client's services are disabled to indicate that the client is

no longer able to participate in the session. Other session participants, if any, are at that time immediately informed of client's departure and the name of the client is removed from the session participant's list.

The second way of leaving the session is to terminate the application. This both leaves the session and terminates the YCab application. The user selects the "X" icon in the upper right corner of the application. This should bring up the dialog window as in Figure 9. This window confirms that the user is indeed trying to quit the application and the user did not just click on the "X" button by accident. Selecting "No" will returns the user to the application and any session that was in progress at that time. Selecting "Yes" results in the client leaving the session, using the procedure described above, and then terminating the application.

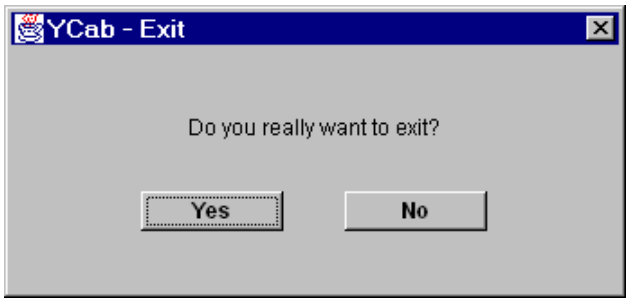

Figure 9 - YCab exit dialog window.

Client Design

YCab is divided into two parts, the framework API and the services. The framework implementing the environment has the responsibility of sharing information from the services with other members in the collaborative session over an ad hoc network. This separation of code allows developers to quickly and easily create services without having to create networking facilities, message routing, or state recovery.

Services used within the environment implement an interface that provides basic methods required by the framework. Some of these include calls to process a message and to store the state of the service. Since the environment requires many of the calls, they must be implemented within each new service developed. The framework consists of the following software components (Figure 10 describes the architecture):

1. Communication and service managers,
2. Session and election services,
3. State recovery manager, and
4. GUI components.

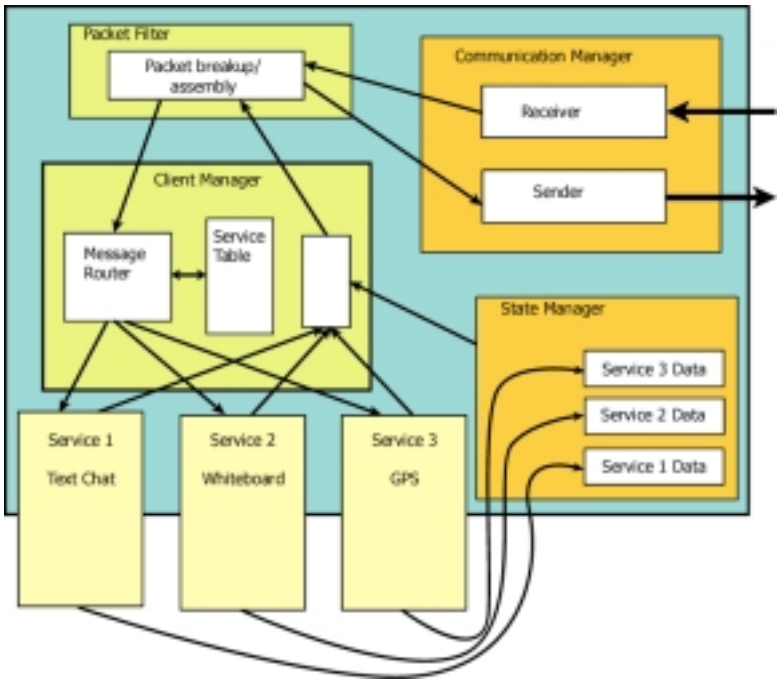

Figure 10 - Architecture of YCab client.

The client is partitioned into two major parts, the services and the framework. The main responsibility of the framework is to provide an environment for services on one client to communicate and share information with services on other clients in an ad

hoc network. To provide such an environment, the framework is responsible for implementing session registration protocols, leader election protocols, and state recovery protocols.

Description of Components

Core Message

CoreMessage is the basic communication unit used by the YCab API. Each CoreMessage consists of two parts: the Header and the Payload. The payload is of type Object and consists of the data that is associated with the message. The header contains message information, such as the sender, receiver, message type, sequencing information, and the payload type. Figure 11 shows the structure of CoreMessage class.

Figure 11 - Design of CoreMessage class.

The *Sender* field of type String contains the name of the client from which the message originated. The intended receiver for the message is contained in the *Receiver* field of type String. Even though all the clients receive all the messages, the *Receiver*

field is intended to specify the client that is supposed to process the message. In addition, pre-defined values exist in the Protocol class that can be used in the *Receiver* field. The summary of those values is found in Table 9 at the end of this section. The *Receiver* field can contain more that one client name if the message is to be sent to more that one client, in which case the pre-defined name separator separates the names. The method *createReceiverField()* located in the Header class takes two receiver names and returns a single String containing those two client names, in the format that is appropriate for the *Receiver* field. The field is always parsed for multiple names when the client receives the message.

The *Service* field of type String contains the name of the service that is the intended receiver of the message. It is necessary to route the message to the proper service. In most cases it will contain the name of the service from which the message originated, since the payload field will contain data specific to that service. However, no guarantee can be made about it, since cases exists where services of one type can send messages to services of other type.

The *Message Type* field of type *int* is used to specify the type of the message. This includes messages of type election message, state recovery message, plain data message, or others as listed in

The sequence information is actually returned to the user as a Sequence object, but is stored in the header as two fields: *MultiPacketNumber* of type *int* and *TotalMultiPackets* of type int. The Sequence object provides encapsulation for the sequencing information and provides a convenient facility for manipulating the data contained within it, while still occupying the minimum amount of memory. This

information is mainly used by the Packet Filter for sending and receiving messages that exceed the maximum size of a UDP packet. Individual services can also use the Sequence object for their own sequencing needs. The *Payload Type* field describes the contents of the Object in the payload. This is provided so that the services are not limited to a single type of payload. This field is optional and can contain a null value if a certain type of payload is expected by the receiving service. Table 7 is a summary of the field types in each CoreMessage and a short description of each field.

Table 8. The codes for different message types are contained in the Protocol class. CoreMessage destined for user services, such as text chat or whiteboard, are tagged as data messages (Protocol.DATA_MESSAGE). Other messages used by the framework are tagged appropriately.

Table 7 - Summary of CoreMessage fields.

Field	Type	Description
Sender	String	Name of the client from which the message originated.
Receiver	String	Name of the client to receive the message.
Service	String	Name of the Service to receive the message.
Message Type	Int	Type of the message.
Multi Packet Number	Int	Part of the Sequence object. Number of the packet in a sequence.
Total Multi Packets	Int	Part of the Sequence object. The total number of packets in a sequence.
Payload type	String	Type of data associated with the message.
Payload	Object	Data to be transferred.

The sequence information is actually returned to the user as a Sequence object, but is stored in the header as two fields: *MultiPacketNumber* of type *int* and

TotalMultiPackets of type int. The Sequence object provides encapsulation for the sequencing information and provides a convenient facility for manipulating the data contained within it, while still occupying the minimum amount of memory. This information is mainly used by the Packet Filter for sending and receiving messages that exceed the maximum size of a UDP packet. Individual services can also use the Sequence object for their own sequencing needs. The *Payload Type* field describes the contents of the Object in the payload. This is provided so that the services are not limited to a single type of payload. This field is optional and can contain a null value if a certain type of payload is expected by the receiving service. Table 7 is a summary of the field types in each CoreMessage and a short description of each field.

Table 8 - Summary of pre-defined message types for the Message Type field.

Code	Purpose	Value
DATA_MESSAGE	General message containing service data.	0
STATE_RECOVERY_MESSAGE	Messages sent out by State Manager containing state recovery information.	1
JOIN_SESSION	Messages announcing that a client is joining session.	11
JOIN_SESSION_REPLY	Message sent in reply to JOIN_SESSION_REQUEST message.	12
JOIN_SESSION_REQUEST	Message indicating that a client is requesting to join a session.	13
LEAVE_SESSION	Message announcing that a client is leaving session.	21
SET_COORDINATOR	Message announcing that new session coordinator has been selected.	30
START_DETERMINE_COORDINATOR	Message indicating beginning of leader election.	31

VOTE	Vote message for leader election algorithm.	32
STATE_RECOVERY_REQUEST	Message indicating a request for state recovery.	40
STATE_RECOVERY_END	Message announcing the end of state recovery messages.	42.
PARTIAL_MESSAGE	Indicates a partial message.	50
PING_REQUEST	Indicates ping message.	60
PING_RESPONSE	Message in reply to PING_REQUEST message.	61

Table 9 - Summary of the pre-defined values for the Receiver field.

Code	Purpose	Value
ALL	Indicates the message is to be received by all clients.	^
COORDINATOR	Messages addressed to the session coordinator.	%
SESSION_MANAGER	Messages to be received by the Session Manager service.	SMS
LES	Messages to be received by the Leader Election service.	LES

Communication Manager

The Communication Manager is responsible for providing asynchronous send and receive functions to the client (see Figure 12 for the design of the Communication Manager).

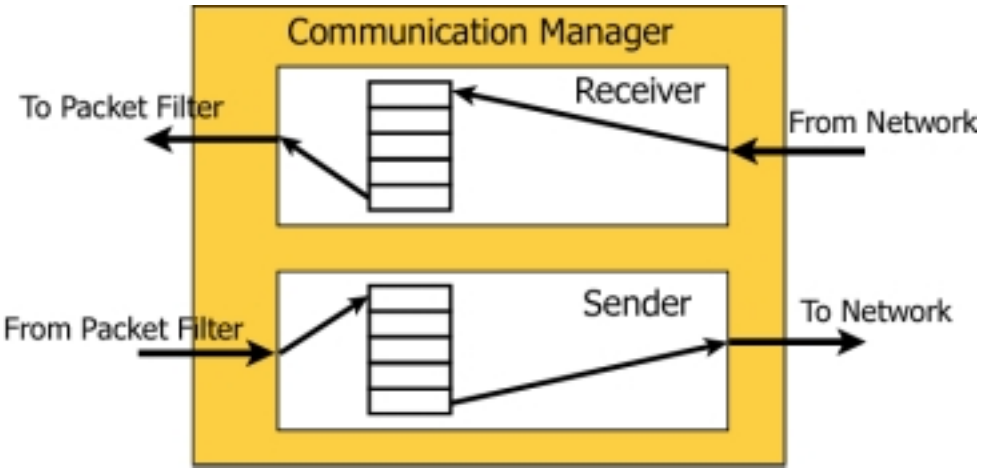

Figure 12 - Architecture of Communication Manager.

All outgoing messages, regardless of their origin, are sent out by the Communication Manager. Contained within the Communication Manager is a threaded Sender object. The threaded Sender object is used to send outgoing messages through a MulticastSocket supplied by the Communication Manager. This mechanism of using a threaded Sender requires thread synchronization between the “Producer Thread” and the “Consumer Thread.” In this case, the “Producer Thread” supplies the outgoing messages while the “Consumer Thread” is the thread inside the Sender performing the actual multicast. To avoid busy waiting and achieve thread synchronization, the Communication Manager includes an outgoing queue located in the Sender Object, which provided a two-fold benefit. The queue not only acts as a buffer, but it also allows the Sender Object to act as a monitor. During the course of normal operations, the Sender Thread removes messages that need to be sent from the head of the queue by accessing a synchronized method. If there are messages available to be sent, then the method returns. If there are no messages in the outgoing queue, the Sender Thread puts itself to sleep thereby relinquishing the lock on the Sender object. When a “Producer Thread” places

an object on the outgoing queue, it checks to see if the number of messages in the queue is greater than one. If there are messages in the queue, it notifies a waiting thread, which in this case is the Sender Thread. During periods when outgoing messages are sent rapidly, an increase of packet loss is exhibited which is most likely caused by bandwidth constraints, latency issues, etc. By slowing down the rate of sending, we experienced a significant decrease in the number of packets lost, so to accommodate for varying bandwidth and latency, a method to adjust the rate of transmission was added to the Sender. The default value for the send delay, as specified by the Communication Manager, is fifteen milliseconds, which means that at the minimum there is a 15-millisecond pause between consecutive messages.

Messages are placed in the outgoing queue as CoreMessage objects and, as such, must be packaged into a DatagramPacket. The Sender packages the CoreMessage into a DatagramPacket as an array of bytes using Java's serialization mechanism that allows easily copying of entire objects. The Datagram packet is then multicast to the group and the receiver performs a reverse procedure to de-serialize the receiver byte array into a CoreMessage object.

Since the Communication Manager is responsible for receiving as well as sending CoreMessage, it uses a threaded Receiver Object to listen on the specified IP and port for any incoming messages. As with the Sending end of the Communication Manager, there is a notion of "Consumer" and "Producer." In this case, the "Producer" is the Receiving thread that monitors for incoming messages and the "Consumer" is the thread originating from the Message Router. Since the goal is to have the Receiver thread do minimal processing, allowing it to read incoming messages quickly and repetitively, it simply puts

the DatagramPacket into an incoming message queue and notifies any waiting threads. The Receiver Object, but not the Receiver thread, handles the extraction of the data from within the DatagramPacket. The thread responsible for the processing actually originates from the Message Router. When an incoming message is requested from the queue, the Receiver extracts the byte array payload and casts the data as a CoreMessage. Since all messages used within the YCab framework use either CoreMessage or a child of CoreMessage, we can assume the casting is correct. The CoreMessage is then returned to requesting Object. If there are no items in the incoming message queue, the calling thread is put to sleep until it is notified by the "Producer" thread.

In addition to providing a send and receive mechanism for the client, the Communication Manager allows for an adjustable DatagramPacket size. The default size of an outgoing and incoming DatagramPacket is 8 kilobytes. Although this size may be suitable for simple text messaging, image and other data transfers usually require sending messages that span several packets. To reduce the number of packets required, the Communication Manager allows the DatagramPacket size to be adjusted up to the 64k limit.

Client Manager

At the heart of the client lies the Client Manager. This software component is responsible for managing services as well as keeping track of client properties such as the name, the rank, the coordinator status, etc. (See Figure 13). Additionally, it is responsible for instantiating the other managers, router, and filters. The Client Manager is also responsible for instantiating, initializing, and maintaining references to the following of software components:

1. Client Frame – is the main GUI framework for the applications based on the YCab environment.
2. Communication Manager – provides communication for the client. See section Communication Manager.
3. Packet Filter – handles messages for the communication manager. See section Packet Filter.
4. Message Router – routes messages to appropriate services. See section Message Router.
5. State Manager – provides state recovery information. See section State Manager.
6. Session Manager Service – manages the session.
7. Leader Election Service – handles election of session coordinator.
8. User defined Services, including both regular Services and ThreadedServices are the modules that provide functionality to the application.

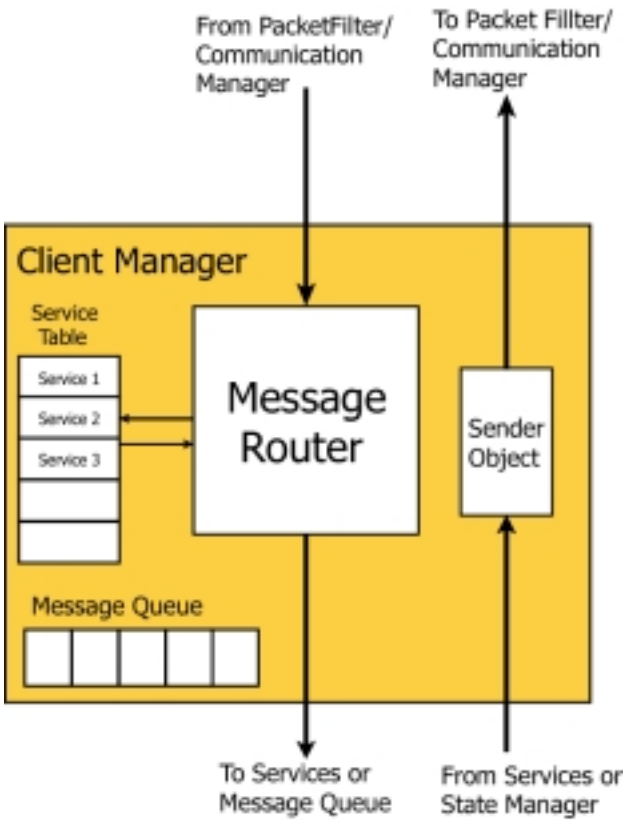

Figure 13 - Architecture of Client Manager.

In a sense the Client Manager is the “glue” that keeps the environment together. When the developer instantiates a Client Frame, a Client Manager is automatically instantiated as well. To minimize memory and processor usage when the client is idle and not connected to a session, references to the above-mentioned software components are created without instantiation. The only exception is the State Manager, since it is needed when services are added to the Client Manager. Optional services such as a Text Chat service and a Whiteboard are then added to the Client Manager’s service table. During this process, the services are requested to provide it a service name and whether state recovery is enabled.

Message Router

The Message Router is responsible for routing all messages to appropriate services. Every message that is received by the client must pass through the Message Router at least once. Because of this the Message Router needs to be as simple as possible and to have the minimum amount of logic. Message Router runs in a continuous loop that performs three basic steps (see Figure 14).

- Get message – The Message Router works closely with the Communication Manager and the Client Manager. All incoming messages are received by the Communication Manager and are stored in its internal queue. Both the Message Router and Communication Manager are synchronized on that queue, so it is a typical Consumer/Producer situation with the Communication Manager acting as a producer and Message Router as a consumer.
- Process message – The Message Router extracts the service name from the header of the message and then gets the reference to the service with that name from the table maintained by the Client Manager. Once the reference to the appropriate service has been acquired, the Message Router then invokes the *processMessage()* method on that service, so that the message can be processed.
- Update state information – the Message Router determines if the service for which the message is addressed has state recovery enabled. If the state recovery is disabled this step is skipped. Otherwise, the Message Router invokes the *updateState()* method on that service, so that the message can be added to the services state according to the rules supplied by the service.

The actual algorithm is somewhat more complicated due to the fact that the Message Router can be in one of the two modes: Normal Mode or State Recovery Mode. Those two modes are required to accommodate the state recovery mechanism and to preserve the ordering of messages. The ordering of messages is vital and must be identical on all clients to ensure consistency among all session participants.

- Normal Mode - all received messages are immediately passed to the appropriate service for processing.
- State Recovery Mode – in this mode the client recovers the state for its services. To ensure that all services remain consistent, data messages that are not marked as State Recovery Messages are queued by the Client Manager.

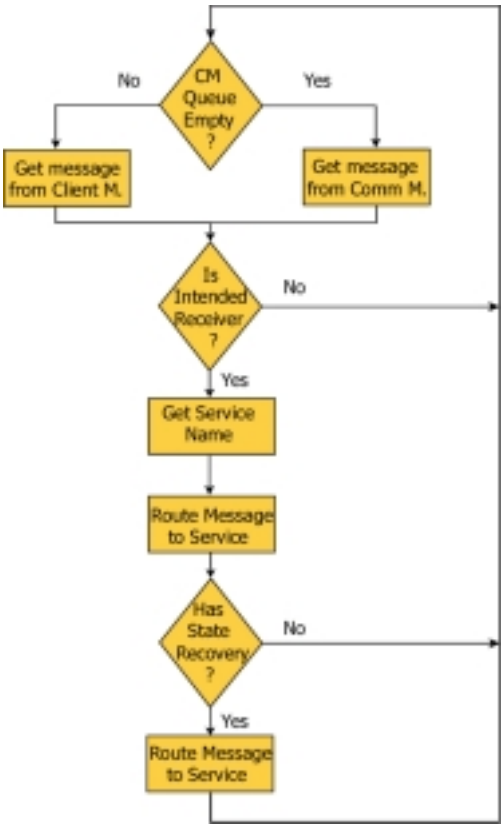

Figure 14 - Message router algorithm.

The State Recovery mechanism increases the complexity of the Message Router, because the ordering of messages must be preserved. Since messages are queued during State Recovery, as soon as the State Recovery is completed these messages must be processed before any new messages can be processed.

Packet Filter

At times, the client may want to send a CoreMessage that is larger than the maximum allowed DatagramPacket. For instance, an image sharing service may need to send images that exceed the 64k maximum size for DatagramPackets. Since we did not want to place limitations in the CoreMessage size, we included a PacketFilter Object that resides between the Client Manager and the Communication Manager (see Figure 15). Spanning the large CoreMessages over a series of smaller CoreMessages as well as reassembly of the spanned CoreMessage is the responsibility of the PacketFilter.

Given that all the messages need to be smaller than the maximum DatagramPacket size, all messages sent and received must pass through the PacketFilter. By providing an interface resembling the Communication Manager, the Client Manager can talk to the PacketFilter just as it would to the Communication Manager. To have all the messages pass through the Packet Filter, the Client Manager calls upon the Packet Filter to send and receive messages instead of the Communication Manager thereby eliminating all direct calls to the Communication Manager.

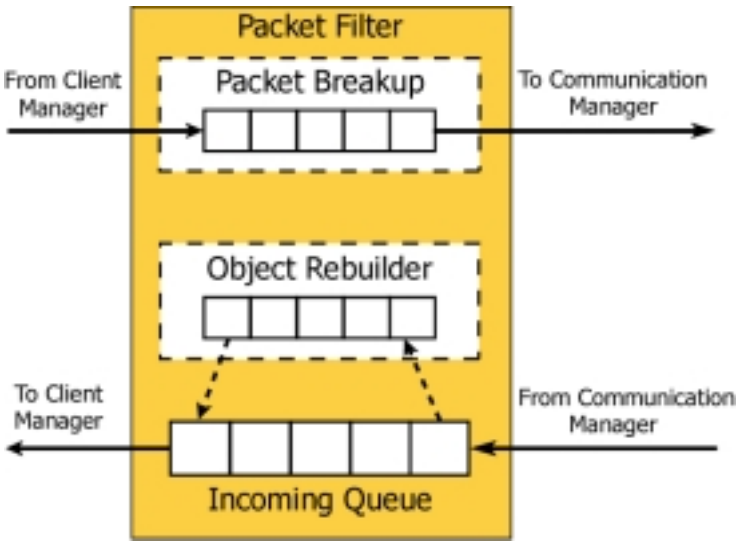

Figure 15 - Architecture of Packet Filter.

Construction of the Packet Filter requires a reference to the Communication Manager to provide a means of sending and receiving messages to and from the other clients. During construction, the Packet Filter sets its filtering size to the maximum packet size as returned by the Communication Manager. This filtering size is used to determine if an outgoing packet needs to be spanned over a series of smaller messages. In the course of normal operations, the Client Manager passes CoreMessages to the Packet Filter's send method, which checks the size of the CoreMessage. If the CoreMessage exceeds the maximum packet size, it is broken up into a byte array. The byte array is then sectioned off into smaller pieces and placed as payload into another CoreMessage. The Headers in these CoreMessages are tagged with the same randomly generated ID number as well as a packet number and then placed in a Vector. Additionally, the message type in the Header is tagged as a partial message. Each message in the Vector is then sent out to the other clients using the send method provided in the Communication Manager.

To shield the complexities of disassembling and reassembling messages from the Client Manager, the Packet Filter must also reassemble the partial packets into the original CoreMessage. One of the major issues with spanning a large message across a series of smaller messages is deciding upon and preserving the semantics of total message ordering. This design decision is to use the first incoming partial message of a large message as the ordering position for that message. In other words, when the Packet Filter receives the first partial message, no other messages are processed until that partial message is fully reassembled. Because other clients in the session are also sending messages simultaneously, the series of partial messages can be interleafed with messages not belonging to the series. These messages must be queued in the background until the first partial message is fully reconstructed. Upon receiving the first partial message of a series, the Packet Filter creates an Objec Rebuilder and stores it in an internal Hashtable with the randomly generated ID key found in the Header as the key value. The ObjectRebuilder takes the byte arrays stored within the partial messages and reconstructs the original CoreMessage. While partial messages are being reconstructed other messages are queued, however, since we assume that packet loss is a possibility in a wireless environment, certain partial messages should be discarded after a given amount of time. We assume packets are lost quite frequently in a wireless environment and since a partial message cannot be reconstructed if any of the messages are lost, we must make this a best effort procedure. Because lost packets are not automatically retransmitted, we do not want the Packet Filter locked into the message queuing mode so if the number of messages in the queue exceed the queue's threshold, the ObjectRebuilder associated with

the partial message is discarded and processing begins on the messages stored in the internal queue.

When the Message Router in the Client Manager requests an incoming message to process, it is always guaranteed to receive a CoreMessage to process and never a partial message. The getIncomingMessage() method in the Packet Filter first checks to see if there is no message currently in the reassembly. It checks the internal queue for messages, if a message is present and it is not a partial message, the message is returned to the Message Router for processing. If the internal queue is empty, the Packet Filter requests a message from the Communication Manager. If that message is not a partial message, the message is returned to the Message Router, otherwise an ObjectRebuilder is created for that message and the reassembly process begins. The Packet Filter then continually requests incoming messages from the Communication Manager until either the ObjectRebuilder receives all the necessary bytes required to reconstruct the CoreMessage or the threshold for maximum messages queued is reached. If the message is reconstructed successfully, it is returned to the Message Router for processing. If the queue threshold is reached, the ObjectRebuilder associated with the partial messages is discarded and the message is pulled from the internal queue. The message from the internal queue is then returned to Message Router for processing. By utilizing a Packet Filter, the Message Router and Client Manager are shielded from the complexities of message disassembly and reassembly.

State Manager

The duty of the State Manager is to maintain the state of the session. The state for each service consists of messages sent to that service. This way any new client that joins

a session can be easily brought up to date. The State Manager only keeps state information for services that have state recovery enabled. For each such service there exists one Service State object that contains the state information for that service. In addition, the manager maintains a list of all services registered, so that it can quickly access relevant information. Figure 16 shows the architecture of the State Manager.

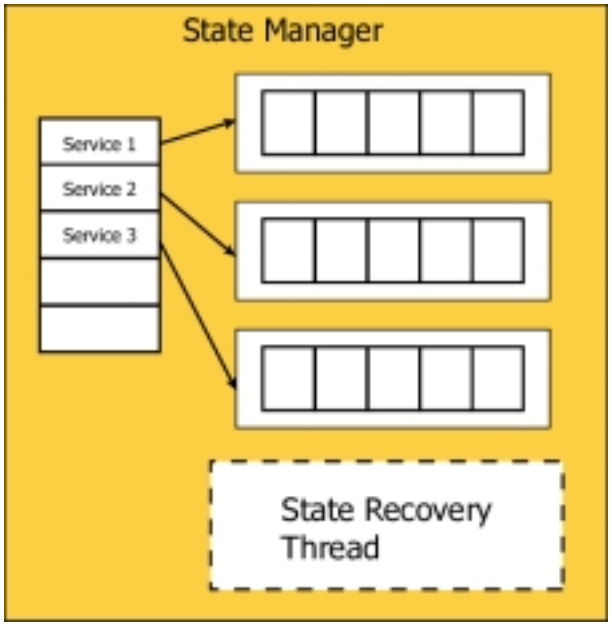

Figure 16 - Architecture of State Manager.

The operation of the State Manager is transparent to the user. To save session information, each service must register itself with the State Manager. When a service registers itself with the State Manager, the State Manager first checks whether the service is already registered. If not, an entry is added to the list, and a new ServiceState object is created. From that point on, the State Manager maintains the state information for that service. Once the Message Router has routed a new message to the appropriate service, it checks whether the given service has state recovery enabled. If so, the message is routed again to the service, this time for the purposes of updating the state. The Service class

provides a default procedure for maintaining the state of the service, which assumes that the state of the service consists of a contiguous number of messages (ranging from one to infinity). For more advanced services, the developers can provide their own implementation of state update procedures, using the API. The State Manager API provides facilities for managing state data for each service, such as updating the state, retrieving state info, or clearing the data.

When the State Manager receives a request for state recovery, it launches a separate thread that is responsible for retrieving state information for the requested service. It then sends all the data to the requesting client only. The state information is sent as a sequence of separate messages in the same format as received by the client. Each message is marked as a state recovery message (Protocol.STATE_RECOVERY), so that the receiving client can distinguish between regular messages. Once state recovery has finished it sends out a state recovery end announcement message (Protocol.STATE_RECOVERY_END) to both the receiving client and itself. At that point the recovery thread terminates and is ready to handle another request.

Service State

The ServiceState object is used to store state information for a given service. It is implemented by using Java's Vector objects. By default, ServiceState has unlimited capacity which is only bounded by the available memory. However, the capacity can be specified in which case once the capacity has been reached the messages are discarded on a FIFO basis. In this case the capacity refers to the number of messages that are retained. All messages are stored in the order in which they are received, so that the ordering is preserved.

Service

The Service class as defined in the YCab API is not an integral part of the environment. In fact, the Service class cannot be instantiated to produce a ready-to-use object. Instead the Service class is an abstract class that provides the basic framework for a service. It provides the basic functionality for a service and provides guidance as to what methods must be implemented to provide a fully functional service. This class provides a way to quickly develop a service without extensive knowledge of the YCab framework by providing default implementations for common tasks such as state recovery or service initialization. The idea behind the service is to allow for implementing modular collaborative applications. Such modules, called Services, can then be developed for specific tasks and easily added and removed from the application to create very customizable applications, that do not require extensive programming expertise to design.

For the most basic implementations the user has to implement two methods: init() and processMessage().

- The init() method should contain all the service initialization procedures (such as creating the service GUI and setting initial values) and should be called once from the constructor.
- The processMessage() should contain the procedure for dealing with incoming messages. The Message Router invokes the method once it has determined that the message belongs to a particular service.

Several other methods have been defined that help adding certain features to the service such as state recovery information and optimization information. Each service can implement state recovery on three different levels as shown in Figure 17.

Figure 17 - Different modes of State Recovery.

The default behavior of a service does not include state recovery. This scenario assumes that the service either does not care about its state or the nature of the state is such that its data is time sensitive, i.e. the data becomes invalidated quickly and as such it does not make sense to store it. The second level is by enabling the default way of providing state recovery. In this scenario the service uses the default version of the *updateState()* method. The *updateState()* method provides the procedure for preserving the state of the service, and it assumes that the state of a service can be defined by a contiguous sequence of messages. The default *updateState()* method also has some flexibility of further defining exactly what amount of messages constitute the state for each service. Any positive integer can be specified as the boundary for the amount of messages, so that for a value of 1, a single message will be the state for the service (e.g., the Image viewer). A special value of –1 is used to indicate that all the messages received to date constitute the state information. Finally, the developers can design custom updateState() methods that extend or overwrite the functionality of the method.

The service skeleton does not implement any optimization per se. However, the service interface includes provisions for setting the service into different optimization modes. It is then up to the service designers to provide functionality for each of the modes.

By default a service does not have any GUI components. To develop a GUI the developer needs to be familiar with Java's AWT. Each service contains a Panel object and a Menu Object that can be used to add custom components to a service.

ThreadedService

While some services only require processor time when an incoming message is processed or if invoked by the user interface, other Services may need to perform background tasks. Services such as one to ping clients in the session or to broadcast GPS coordinates require most of the processing to occur as a background service, similar to a daemon. By creating these services as a separate thread, these services can run as a daemon, processing messages and sending messages asynchronously from the Message Router. The ThreadedServiceObject is included to support these daemon-like services.

The ThreadedService Object is a natural extension of the Service object. Since threaded services are autonomous to a certain extent, they need to process messages asynchronously from the Message Router thread. The situation that occurs here is similar to that of the Communication Manager. Recall, the Receiver thread of the Communication Manager also runs asynchronously from the Message Router thread. While the Service class provides certain pre-implemented methods, the processMessage() method is left abstract and becomes the responsibility of the service developer to implement. The Message Router calls upon the processMessage() of each Service to

process the incoming message. In the case of the ThreadedService, the processMessage() method is implemented to provide an asynchronous method of communication between the Message Router thread and the ThreadedService thread. When an incoming message is received for a ThreadedService, the Message Router will call the processMessage() method, just as it would with a non-threaded service. However, unlike a non-threaded Service, the ThreadedService's processMessage () places the message in a queue and notifies any waiting threads if the queue was previously empty. The waiting thread will most likely be a ThreadedService thread that has called the getIncomingMessage() method. The getIncomingMessage() method is called by the thread within the ThreadedService that processes messages. If a message is available to process, the message is returned, otherwise the calling thread is put to sleep until notified by the processMessage() method. Since the actual processing of the message is performed by the ThreadedService thread, the Message Router is available to route other messages.

In addition to implementing the Runnable interface and providing the processMessage() method, the ThreadedService provides simple start() and stop() methods, a thread, as well as a simple boolean switch value. After the client joins a session, the Client Manager initializes and starts all the necessary managers. The Client Manager then checks its service table for ThreadedServices. If a ThreadedService is found, the start() method is called to start the threads within the ThreadedService. The basic implementation of start() only starts the included thread so if additional threads are added, overriding the start() method will probably be necessary. After the client has left a session, the Client Manager will again check the service table for ThreadedServices. If a ThreadedService is found, the Client Manager calls the stop() method to stop the

threads within the service. In the included implementation, the stop() method simply sets the boolean switch to false, allowing the thread to terminate on its own. There is a potential issue with this simple implementation, however. Suppose a thread is sleeping until incoming data is available. If the stop() method is called to terminate the thread, the sleeping thread will not terminate until it is brought out of sleep by an incoming message. This potential situation can be solved by overriding the stop() method. The overriding method could interrupt a thread instead of using a simple boolean value switch to terminate the thread.

During the course of normal operations during a session, the user may wish to manually stop and start a thread. The ThreadedService Object also includes predefined menu items bound to the start() and stop() methods of the service. These are optional and can be easily added to the menu component. Since the methods run as a background service, they need to be started after the client joins a session and stopped after the client leaves.

YCab Protocols

This section describes the main protocols that occur in the YCab API. These protocols define the procedures event propagation, joining and leaving session, state recovery, leader election, and others.

Creating and Joining a Session.

During the course of a session, many new clients may request to join in the collaboration event. Although we are involved in a multicast environment, simply writing to and reading from the specified multicast IP is not enough. For a new client to participate effectively, it must be brought up the same state as other members of the

session. For example, if a shared whiteboard is present, then the image present on the whiteboard of the new client should be the same as that of other members in the session, after state recovery occurs. While there is no central server in the session that allows for this to happen, one of the members in the session will be given the responsibility of bringing the new client up to speed. This task is given to the coordinator of the session. Along with the state recovery responsibility, the coordinator may have other connotations within the application, an example of which is floor control of the session. The following procedure occurs when a new client joins an existing session (see Figure 18).

1. The New Client (NC) chooses a handle or screen name to use during collaboration within the session. The user enters the proposed screen name.
2. The NC broadcasts a join session request (Protocol.JOIN_SESSION_REQUEST) along with the proposed client name to the members within the session. Although sending the proposed client name explicitly in the payload is not necessary, it is done so for consistency.
3. The clients in the session reply to the join session request by sending the new user a Protoco.JOIN_SESSION_REPLY message containing their respective names and ranks.
4. The NC checks the JOIN_SESSION_REPLY payload for the proposed name. If the proposed name is used then the handshake is terminated. Otherwise, the names returned in the JOIN_SESSION_REPLY are added to a peers list, along with their rank. The NC also determines the highest client ran in the session.

5. After waiting a specified timeout period, the new client sends a join session message (Protocol.JOIN_SESSION) containing the NC proposed client name as well as the highest rank number plus one. In this case, the client with the lowest rank number is the oldest client in the session. If this rank is equal to 1, it indicates that the NC is the first client in the session. As such the client skips the rest of the steps, terminates the handshake protocol, and jumps straight into the session.
6. All clients process the join session message (Protocol.JOIN_SESSION) and add the associated client name and rank into their peers list.

At this point the client has successfully joined the session. However, for the client to participate in the session it must first be brought up to the state consistent with the state of the session. Therefore NC initiates the state recovery mechanism as described in the section State Recovery.

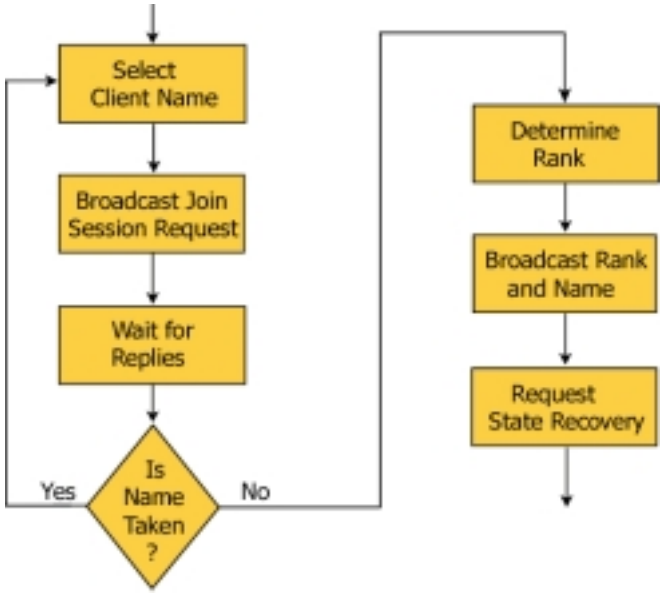

Figure 18 - Procedure for joining a session.

The above-described steps occur after starting the NC's managers and services. If a new client attempts to join a session but receives no replies, then the NC assumes it has started a new session. At this point, the NC sets itself as the coordinator with a rank of 1 and omits steps 7, 8, and 9. Other clients subsequently joining perform all the steps.

Leaving a Session

Leaving a session is much simpler than joining a session. A client leaving the session sends a leave session message (Protocol.LEAVE_SESSION) to the other members. The other members remove the client from their respective peers list. After sending out the leave session message, the Client Manager looks through the service table for ThreadedServices. If a ThreadedService is found, the Client Manager invokes the stop() method to terminate the threads. The Client Manager then terminates all non-essential managers and clears the contents in the State Manager.

State Recovery

The state recovery mechanism enables clients to be brought to a state that is consistent with other session participants. This applies to clients that are new session participants, were temporarily disconnected from the session, or for some other reason are inconsistent with the other clients in the session. State Recovery requires that a Session Coordinator (COORD) be present in the session. The new client (NC) then closely coordinates with the coordinator to ensure that NC is up to date with the other clients.

NC initiates State Recovery by sending a State Recovery Request message. The message is addressed to the COORD and NC (the client itself) and contains a list of Services that require State Recovery. It is important to understand that the act of sending

out a State Recovery Request does not imply the beginning of the State Recovery process. The State Recovery actually begins when COORD and NC process the State Recovery Request message. Since multicast guarantees that all multicast group participants receive the messages in the same order, this ensures that no messages will be lost.

1. NC broadcasts the State Recovery Request (REQ). The message is addressed to the COORD and NC and contains the list of services that require state recovery.
2. REQ is received by all clients simultaneously, but is only processed by the COORD and NC. The Message Router routes the message to the Session Manager service in both the COORD and NC.
    a. Upon receiving the State Recovery Request the COORD switches its own Message Router into State Recovery Mode. This effectively "freezes" the current session state and ensures that any new incoming messages will not be processed until NC has been fully recovered. In this mode all new data messages are queued by the Client Manager (the Communication Manager is not aware that the client is in state recovery mode). The coordinator processes the REQ message and extracts from it the name of requesting client and the list of services to recover. It then instructs its State Manager to send all the state information for those services to the requesting client. All the messages sent by the State Manager are marked as State Recovery Data Messages (Protocol.STATE_RECOVERY_DATA) and are only

addressed to NC, so that other session participants will not process them.

b. NC also switches its own Message Router into State Recovery Mode. In addition NC starts a timer with a pre-defined delay. This timer is used to ensure that NC receives state recovery information within a reasonable period of time. If State Recovery Messages are not received within that time, it indicates that the session coordinator is not present in the session and NC initiates a Leader Election to establish a new session coordinator. During State Recovery Mode all messages marked as State Recovery Data Messages are passed to an appropriate service for processing. All Data Messages are queued by the Client Manager.

3. While the State Recovery is in progress, all session participants other than the COORD and NC, process messages normally.
4. Once the COORD is finished sending out state recovery data, it sends out a State Recovery End (STATE_REC_END) message addressed to the COORD and NC. To improve the reliability of the system the State Recovery End message is actually broadcast three times. This ensures that both clients will return to Normal Mode even if one of the messages is lost. When the STATE_REC_END message is processed by both the NC and COORD, the Message Router is switched from State Recovery Mode back to Normal Mode. However, at this point it is likely that there are messages in the internal queue of the Client Manager that accumulated during state recovery

process. Therefore, the Message Router always checks the Client Manager queue to see if it contains any messages. If a message is found it is removed from the queue and processed by the Message Router. This process continues until no messages remain in the Client Manager queue. This ensures that both NC and COORD "catch up" to the current state of the session. At this point the State Recovery has been completed, and Message Router can resume normal operation and process messages buffered by the Communication Manager.

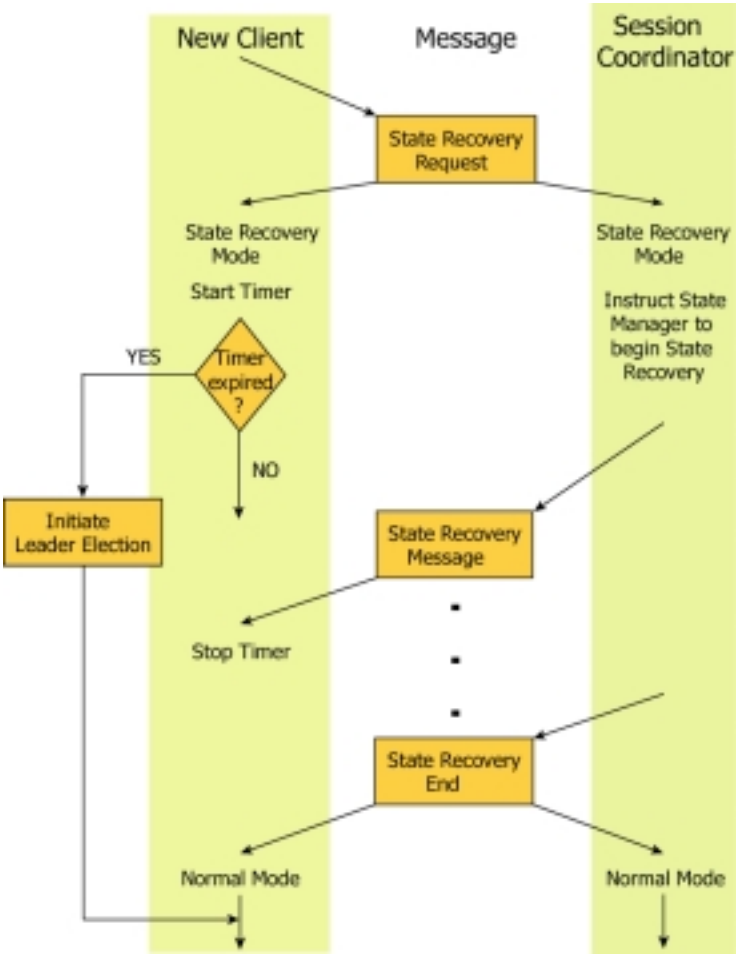

Figure 19 - State recovery algorithm.

It should be noted that while the state recovery process has no effect on the messages that originate from the NC, i.e. other clients continue receiving and processing messages from NC. However, the NC does not see its own actions since the State Recovery Messages have the priority and all other messages are queued by the Client Manager.

Leader Election

During the normal course of a session, the session coordinator may become disconnected from the session or perhaps voluntarily leave the session. Because of the distributed nature of the YCab framework, the remaining clients in the session can continue to collaborate without any adverse affects. Although some services may require a session coordinator for floor control, the absence of a session coordinator will not adversely harm the session. Eliciting a leader election to determine the next appropriate session coordinator easily rectifies the situation. While most services will not need the notion of a session coordinator, the process of bringing a new client up to the state of the session does. In the course of joining a session, a NC requests state recovery from the session coordinator. If no session coordinator is present and the rank of the NC is not one, then the NC calls for a leader election from the current members of the session. The following steps occur during leader election.

1. The NC sends a “Start Determine Coordinator” message to the members of the session.
2. After receiving the “Start Determine Coordinator” message, members of the session, including the NC, start internal timers. A “Vote” response including the client name and client rank is sent out to all members of the session.

3. Incoming “Vote” messages are checked for rank and client name. The lowest ranking number is stored as the current winner.
4. After the timer expires, the client’s rank is checked with the current winner’s rank.
5. If the winner’s rank is lower than the client’s then the election process is over for this client. It then waits for an incoming “Set Coordinator” message. If the current winner’s rank is higher than or equal to this client’s rank, then the timer is restarted. This number of timer restarts is configured by the developer, but defaults to two. The election process ends when either the client’s rank is no longer the lowest or the number of timer restarts has reached the configured value.
6. If the current winner’s rank is equal to the client’s rank, then the current winner’s name is compared to the client’s name. If they are equal, then the client sends a “Set Coordinator” message with its name and rank to the members of the session.

After receiving a “Set Coordinator” message, the NC again requests the session coordinator to recover its state.

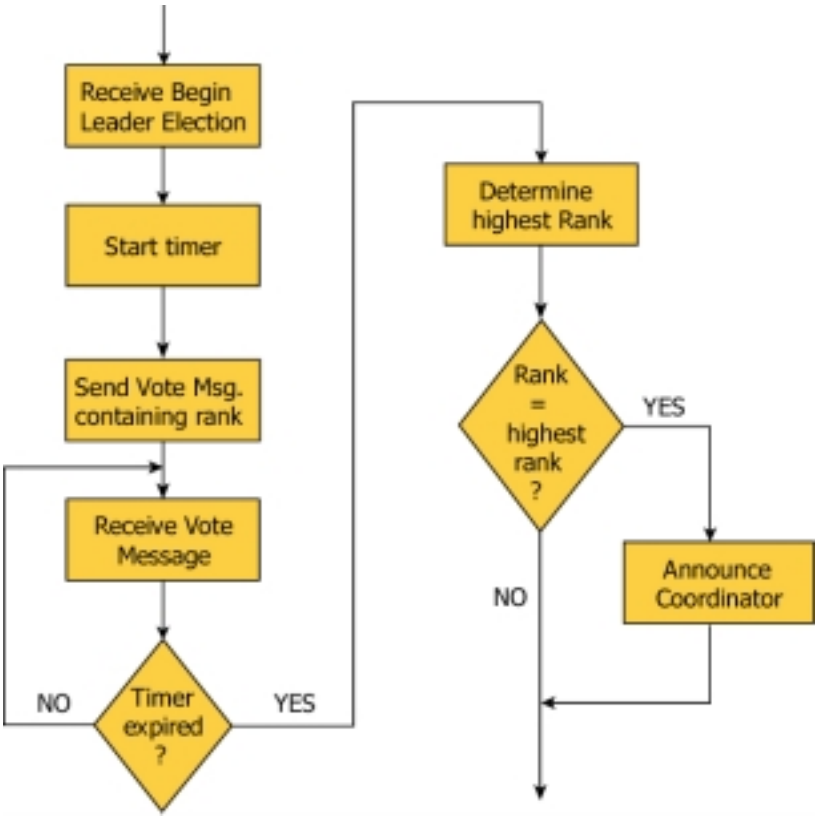

Figure 20 - Leader election algorithm

Pinging

During the lifetime of a typical collaboration session, many members may voluntarily choose to leave but some may be involuntarily disconnected. In a wired network environment, the frequency of involuntary disconnects and packet losses are minimal compared to an ad-hoc network environment. As a member is voluntarily leaving a session, a “Leave Session” message is sent out to alert other members in the session. This usually clears the leaving client’s name from the peers list of members still in the session, however, messages may still be lost. In certain cases, some members will have cleared the leaving member’s name while others do not. This leads to incorrect perception of the session state. Conversely, a client may become involuntarily

disconnected from the session. In this case, both the disconnected member and the rest of the members will have an incorrect “view” of the session state. These situations not only lead to miscommunication and confusion, but also destroy the semantics of collaboration. To prevent these situations from occurring, the YCab framework includes a protocol for pinging clients in the session. The following steps occur throughout the duration of a session:

1. A ping request (Protocol.PING_REQUEST) message is sent out to the session members.
2. The sending client waits for a specified time interval to allow other members in the session to reply.
3. The list of members that replied to the ping is compared to the peers list.
4. Members in the peers list who did not reply are subsequently removed. Additional members who replied to the ping, but do not exist in the peers list, are added.
5. If a ping request (Protocol.PING_REQUEST) is received, a ping response (Protocol.PING_REPLY) is sent out.

By automatically removing and adding members from the peers list, the user and services will perceive a more accurate representation of the session state. For example, if a service displays a listing of the members in the session, the user may notice other members in the session drifting in and out of range. Although not all the clients may have the same state information, the user will at least be aware of the situation. The protocol for pinging is implemented as a Threaded Service by the Session Ping Service.

CHAPTER 4
DEVELOPING APPLICATIONS USING YCAB API

This section assumes that the reader already knows how to use the YCab application and is familiar with how the YCab environment works (CHAPTER 3 provides the description of the client software). This section describes how to create an YCab application, using pre-defined services that are included in the API.

The first step in designing an YCab application is to import the necessary YCab packages – specifically the *client* package. The following line needs to be placed on top of the client code (this assumes that the directory containing the YCab API has been added to the system's classpath).

```
import client.*;
```

All YCab applications extend the ClientFrame class. ClientFrame class contains all the managers and the GUI components required for the application. Therefore, all YCab applications must extend the ClientFrame class. For the purposes of this tutorial the application will be called SampleApp as defined in the code below.

```
public class SampleApp extends ClientFrame
```

To provide custom functionality to the application, the default constructor of the ClientFrame class must be overwritten. Since ClientFrame is just an extension of Java's Frame class any methods that are applicable to Frame class can be used to customize the appearance of the application. However, the first step that must be performed is to call the constructor of the ClientFrame class (the super class) in the SampleApp's constructor.

This initializes all the managers so that the application can successfully participate in a session. Once this is done, the new application can be instantiated and displayed on the screen. For this purpose main() method is added to the class so that the application can be run. The code to accomplish this is shown below.

```
import client.*;

public class SampleApp extends ClientFrame {
    public SampleApp() {
        super();
    }

    public static void main(String[] args) {
        SampleApp t = new SampleApp();
        t.setVisible(true);
    }
}
```

At this point the code for the most basic client is complete and can be compiled and run. The result is an empty frame as shown in Figure 21.

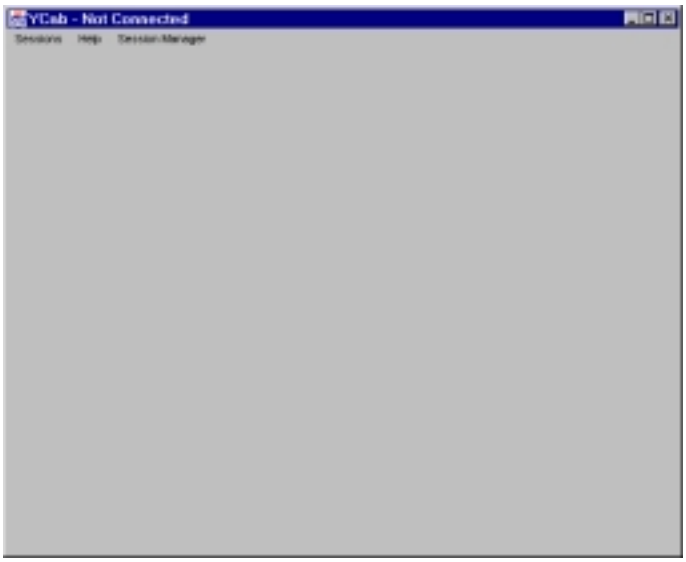

Figure 21 - YCab application without any optional services.

Although there are no services in the application (other than the required services), the application can participate in a session. After specifying the IP address and

port number for the session, along with the desired client name, the title bar of the application should change to “YCab – Connected.” At this point the application is a member of the session, although it does not provide the user with a way to interact with other session participants. Therefore the next step is to add services that add interactivity to the application. Before adding the services the call to the super class constructor can be changed, by using the constructor with two parameters. Those parameters are used to specify default values for the IP address and the port number, in that order, so that they do not have to be entered every time the collaborator brings up the Connection Info dialog window. Even though the default values are already entered in the Connection Properties dialog windows, they do not prevent the user from changing them to some other values if necessary. This feature is only provided for convenience. The new constructor invocation is shown below.

```
super("225.5.5.5", "4343");
```

The next step is to add interactivity to the application, which is accomplished by adding services. The YCab API comes with six optional services that can be used to provide desired functionality to the application: Client Info, Text Chat, Whiteboard, GPS, Image Viewer, and Session Ping. For example, the Text Chat service that is implemented by the TextChatService class enables session participants to communicate with each other via text messages. To use the pre-defined YCab services the *service* package needs to be imported.

```
import service.*;
```

Each service must be added to the frame and registered with the application in the application’s constructor, so that the service can correctly interact with the application. First a new instance of TextChatService class is created. The name for the service can be

specified as a parameter to that constructor. Once the service has been instantiated it must be registered with the application, so that the service can send and receive messages. The code to accomplish those steps is shown below.

```
TextChatService textChat =
     new TextChatService("Text Chat");
addService(textChat);
```

Here the addService() method performs the majority of work. It registers the new service with the Client Manager, so that the service can send and receive messages. Then addService() determines whether there are any GUI components associated with the service. Any window and menu components that were discovered are then incorporated into the application. Below is the code for the application, and Figure 22 demonstrates the resulting application.

```
import client.*;
import service.*;

public class SampleApp extends ClientFrame {
    public SampleApp() {
        super("225.5.5.5", "4343");
        TextChatService textChat =
               new TextChatService("Text Chat");
        addService(textChat);
    }

    public static void main(String[] args) {
        SampleApp t = new SampleApp();
        t.setVisible(true);
    }
}
```

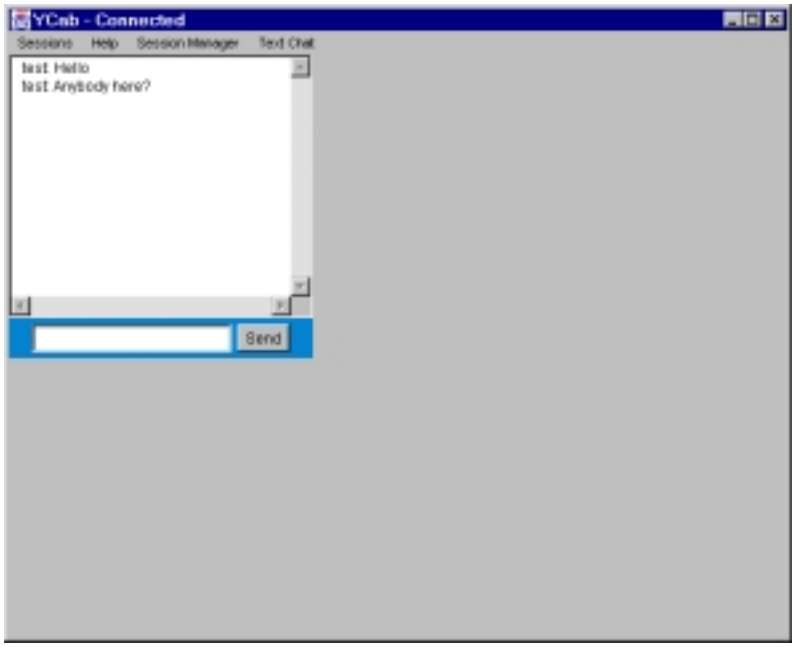

Figure 22 - Unoptimized YCab application with a single service.

At this point the application has the desired functionality. If several instances of that application are simultaneously run and connected to the same session, then whatever is typed in the text field in one of the clients will be propagated to all the clients. Figure 22 shows that one more menu has been added to the menu bar. When the addService() method is invoked, this method determines if there are any GUI components associated with the Service. Furthermore, the GUI components for that service are divided in two types: GUI window components that are displayed on the screen, and GUI components that are included in the menu bar. These two types of GUI components are independent of each other, and each service can have any combination of those components. In this case the Text Chat service contains both types of GUI components. The GUI window components are added to the application window, and the menu components are added to the application's menu bar. The name of the menu corresponds to the name of the

service specified in the constructor. If no name is specified in the constructor, the service that is created is automatically assigned the name “No Name.”

Even though the application is working correctly, it does not have the optimal appearance due to the fact that the size of the service does not match the size of the application’s frame. Also, by default the service is placed in the upper-left corner of the application. This can be remedied by specifying the initial location of the service, and either the size of the frame, or the size of the service. For the purposes of this tutorial all three actions are demonstrated. To change the size of the application’s frame Java’s setSize() method of the Frame class is used. To set the frame size to 300 pixels by 300 pixels, the setSize method is invoked with two integer parameters, the width of the Frame, and the height of the Frame, in that order.

```
this.setSize(300,300);
```

The next step is to set the size of the service and place it in an appropriate location inside the application’s window. One way is to invoke the method setServiceSize() with four arguments. The first two arguments specify the x- and y-coordinates for the upper-left corner of the service. The next two arguments specify the width and the height of the service, in that order[1]. To improve appearance of the application, a 5-pixel wide border will be added around the service. Therefore the service will be placed at location 5,5 and have width and height of 290 pixels. The following line of code accomplishes that and, the resulting application is shown in Figure 23.

```
textChat.setServiceSize(5, 5, 290, 290);
```

[1] Java’s coordinate system places location 0,0 in the upper left corner of the frame. The x-coordinate increases from left to right, and y-coordinate increases from top to bottom.

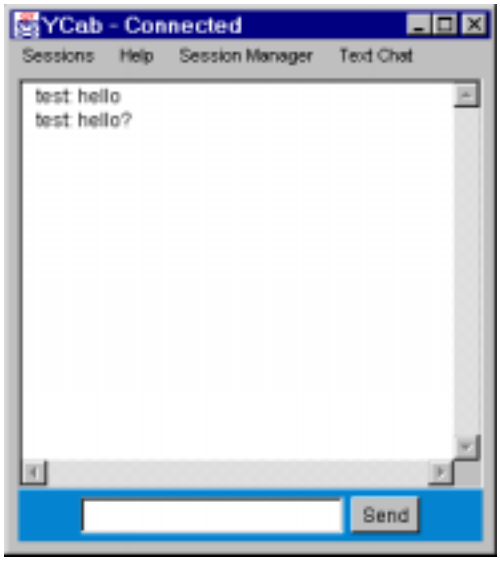

Figure 23 - Optimized YCab application with a single service.

At this point the text chat application is fully functional and its appearance optimized. Its functionality can be further enhanced enabling the service's pre-defined features. All services provide support for state recovery and bandwidth optimization. In addition, services of type Threaded Service (see section **ThreadedService**) provide support for controlling the execution of the service. Finally, custom services can add other features that are appropriate for the service. For example, state recovery functionality (see section State Recovery) can be easily added to the text chat service. With state recovery enabled, any session newcomers can receive the session's history for that service. The amount of history depends on how the service is configured. All that needs to be done to add state recovery information to the Text Chat service is to invoke the enableStateRecovery() method. This method takes one or two arguments, depending on the context in which it is used. The first argument is a boolean value *true* or *false*, which indicates whether the state recovery should be enabled or disabled. The second, optional argument specifies the number of messages to be retained by the State Manager

for that particular service. If the second argument is omitted then all messages received by that service since the beginning of the session will be retained (this is limited only by the available memory). To add state recovery to the Text Chat service such that the last 15 messages that were displayed on the screen are retained, all that needs to be done is the addition of the following line of code in the application code.

```
textChat.enableStateRecovery(true, 15);
```

After recompiling the application, any new client that joins the session automatically receives up to 15 of the most recent messages when connecting to the session. In addition, enabling the state recovery mechanism automatically adds the "Recover State" option to the service's menu. This option allows the collaborator to request state recovery for that service at any time. Adding other pre-defined functionalities to services follows the same procedure. Below is the final code for the application.

```
import client.*;
import service.*;

public class SampleApp extends ClientFrame {
    public SampleApp() {
        super("225.5.5.5", "4343");
        this.setSize(300,300);
        TextChatService textChat = new
            TextChatService("Text Chat");
        textChat.setServiceSize(5, 5, 290, 290);
        textChat.enableStateRecovery(true, 15);
        addService(textChat);
    }

    public static void main(String[] args) {
        SampleApp t = new SampleApp();
        t.setVisible(true);
    }
}
```

As can be seen from the above example creating an application using pre-defined services is fairly simple. Using the above steps developers can use any of the pre-defined YCab services, or any other services developed by other third parties, to quickly create a fully functional application. However, not every desired application can be developed with just the pre-defined components. In situations where the pre-defined components do not provide all the necessary functionalities it is necessary to develop a custom service.

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

An area in which the YCab can be further improved is reducing the code size. Although YCab already has a very small footprint there are rarely used methods and unused objects that can be easily eliminated. For example, the Service Object contains multiple send methods, and while it provides conveniences and flexibility to the developer, most are rarely used. By reducing the code size even further YCab can potentially be ported over to smaller and less powerful devices such as MID's which include cell phones and other handheld devices. In addition to removing rarely used methods and objects, the GUI most likely will have to be rewritten to support Java Micro Edition's graphics toolkit, LCDUI.

While scaling down YCab to fit on smaller devices is an option, increasing functionality and target Service domains can also be a viable option. By targeting a wider range of services such as multimedia services, developers can potentially create voice and video enable collaboration software, given the hardware resources is available. Additionally, inclusion of location aware services such as a true GPS service can also aid in collaboration awareness. Also, by increasing the numbers of prefabricated components included with YCab developers can easily and rapidly creating their own custom application using a level 2 style of development.

While the system was being evaluated it was discovered that additions could be made to further improve clients fault tolerance. Further optimizing the protocol, and possibly implementing a subclass of CoreMessage, a reliable message transmission protocol could be achieved allowing services that required guaranteed packet delivery to be developed. Although creating such a protocol may increase the rate of message transmission, certain services may require this type of delivery mechanism. Reliable connection could be used in situations where it is absolutely necessary to guarantee packet delivery.

During testing phase, it was discovered that the system could to provide even greater awareness of the session and session participants by creating a mechanism to notify the user of packet loss. Such mechanism could at the minimum alert the user that a part of the information is missing and the client risks not being consistent with the state of the session.

LIST OF REFERENCES

[1] Aironet, http://www.aironet.com/.

[2] America Online, http://www.aol.com

[3] De Silva, D. & Pearson, S., "A Wireless Effort Developing Java Applications for Embedded Devices", Java Report, April 2000, pp43-58.

[4] Forman, G. & Zahorjan, J., "The Challenges of Mobile Computing", Computer Science & Engineering, University of Washington, March 9, 1994.

[5] Gage, D., "Java Dreams: Smart light switches?", http://www.zdnet.com/, September 21, 1999.

[6] Java Developer Connection, http://java.sun.com/.

[7] Kaashoek, M.F., Pinckney, T. & Tauber, J.A, "Dynamic Documents: Mobile Wireless Access to the WWW", *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[8] Microsoft Corporation, http://www.microsoft.com

[9] Netwave Technologies, Inc., http://www.netwave-wireless.com/.

[10] NCSA Habanero, http://havefun.ncsa.uiuc.edu/habanero/.

[11] Nunamaker, J.F, "Collaborative Computing: The Next Millennium", Computer, September, 1999, pp66-71.

[12] Palm, Inc., http://www.palm.com

[13] Proxim, http://www.proxim.com/.

[14] Royer, E. & Toh C.K., "A Review of Current Routing Protocols for Ad-Hoc Mobile Wireless Networks", Dept. of Electrical & Computer Engineering, University of California, Santa Barbara.

[15] Satyanarayanan, M., "Fundamental Challenges in Mobile Computing", School of Computer Science, Carnegie Mellon University.

[16] Seal, K. & Singh, S., “Loss Profiles: A Quality of Service Measure in Mobile Computing”, Department of Computer Science, University of Carolina, Columbia, SC

[17] Symbian, http://www.symbian.com

[18] Webb, W., “Embedded Java: an uncertain future”, EDN, May 13, 1999. pp 89-96.

[19] Webopedia, http://webopedia.internet.com/TERM/A/API.html.

[20] Windows NetMeeting 3 SDK, http://www.microsoft.com/netmeeting/.

[21] Whalen, T., & Black, J.P., “Adaptive Groupware for Wireless Networks”, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada.

[22] Yavatkar, R., Griffioen, J., & Sudan, M., “A Reliable Dissemination Protocol for Interactive Collaborative Applications”, Department of Computer Science, University of Kentucky, Lexington, KY, USA

BIOGRAPHICAL SKETCH

Wei-Hsing was born on December 5th, 1975, in Kaohsiung, Taiwan. After moving to the United States and attending high school in Daytona Beach, FL, he enrolled at the University of Florida in the fall of 1993. He received his Bachelor of Science degreein the field of molecular biology. The following year he continued his education through post-baccalaureate work in the field of computer science. In the fall of 1998, Wei-Hsing (Dan) started to pursue a master's degree in the Department of Computer and Information Science and Engineering at the University of Florida while paying for school though teaching assistantships. His research interests include mobile information devices and distributed systems.