Decentralized Ad-Hoc Groupware API and Framework for Mobile Collaboration

Dominik Buszko
Motorola iDEN Group
8000 W Sunrise Blvd.
Ft. Lauderdale, FL 33322
+1(954) 723-8811
Dominik.Buszko@motorola.com

Wei-Hsing (Dan) Lee
Motorola iDEN Group
8000 W Sunrise Blvd.
Ft. Lauderdale, FL 33322
+1(954) 382-0801
Dan.Lee@motorola.com

Abdelsalam (Sumi) Helal
University of Florida
CISE Department
Gainesville, FL 32611
+1(352) 392-6845
Helal@cise.ufl.edu

ABSTRACT

We describe a mobile collaborative system designed for wireless, ad-hoc collaboration. In recent years, mobile computing has emerged as a new discipline in the field of computer science. Due to advances in technology, portable computing devices have become more pervasive. From smart phones, and personal digital assistants (PDAs) running embedded operating systems, to portable computers running conventional desktop operating systems, these devices have increasingly provided communication capabilities that utilize wireless connections. With those communication capabilities firmly established, the next logical step is in the direction of greater *interactions* between mobile users equipped with such devices. However, conventional collaborative tools are ill suited for the demands of portable computers and mobile networks, especially in situations in which no fixed-network infrastructure is present. With these considerations in mind, we designed and implemented a collaborative environment and a framework API suited towards ad-hoc networks of small mobile devices. By creating such a framework, developers can easily take advantage of a decentralized and fault-tolerant collaborative environment, and rapidly develop custom collaboration spaces suited towards their specific needs.

Keywords

Mobile and wireless collaboration, decentralized groupware, ad-hoc collaboration.

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 [[5], [20]].

GROUP'01, ***Sept. 30-Oct. 3, 2001, Boulder, Colorado, USA.***

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 scenarios for ad-hoc groups include the following:

- 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 easily 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.
- A family of four spending their summer vacation hiking and camping in Yosemite National Park

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 interference, 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 and permanently fixed networks where the quality of service is much higher and the state of the network is permanent [13]. The key characteristic that makes them ill suited for ad-hoc networks is their single point of failure. 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.

Our framework is structured 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" allows the framework to function on a number of heterogeneous devices. [23]. 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 the session. Therefore, the goals for our project can be summarized as follows:

- Propose a framework for use by collaborative services in a wireless ad-hoc network.
- Provide a lightweight and flexible API that allows developers to rapidly create small, and highly optimized collaborative applications that can run on mobile devices.
- Implement the framework in Java for deployment over a number of heterogeneous devices.
- Develop services that will include text chat, shared whiteboard, shared images, and global positioning system navigation.
- Develop a sample application using these services to demonstrate the capabilities of the system.

2. CHARACTERISTICS OF THE MOBILE ENVIRONMENT

The ability to be mobile appeals to the user but presents a great challenge to the software developer since the environment of a mobile device can differ vastly from that of a stationary device. The mobile environment is different from a standard, fixed-connection environment due to the mobility of nomadic users and the mobility of its resources [[7], [13], [14]]. While most fixed-location devices have the luxury of consistent and abundant resources supplies mobile devices have no such guarantees. Compounding the problem are differences among mobile devices. As the current market stands, they range significantly and they contain an array of processor architectures and I/O devices resulting from both environmental influences and market considerations. As a result, mobile environment is defined by the following characteristics:

- *Low-bandwidth and high latency* - Network connectivity of mobile devices depends on radio frequency technologies to transmit and receive data. As a result, wireless networks generally exhibit low bandwidth, high latency, and high packet loss rate [3].
- *Low processor power* - Processor power becomes a limited resource, as mobile devices are designed to be portable and the weight and size of the system must be kept to a minimum
- *Small display size* – Most of the mobile devices are equipped with a small displays (most of the time monochrome) that are not suitable for displaying large amount of information or sophisticated user interface. This matter is further complicated by a lack of standards, similar to the ones available in the desktop environment.
- *Short battery life* - The power necessary for operation, easily obtainable and virtually limitless in a stationary device, is a scarce resource for most mobile devices. Upgraded power sources tend to compromise the weight and portability of the system thus limiting the use of power-hungry, high-performance processors.
- *Limited input methods* - The possible methods of data input currently available on the market include keyboard, pen-based, and voice. Some devices support varying combination of these methods to give the user the most flexibility [11], although they are still limited compared to their desktop counterparts.

In addition to the aforementioned constraints imposed by mobile computing, ad-hoc mobility (connection to peer wireless devices without a backbone network [11]) imposes several other constraints that can be summarized as follows:

- *De-centralized control* – in a decentralized system all hosts are considered equal. This eliminates a potential single point of failure. The problem with such a system is ensuring that the replicated data is synchronized. Synchronizing data in ad-hoc networks where network partitioning and merging is frequent is a challenging task.
- *Proxyless environment* - a proxy acts as a filter, through which all messages pass. However, with an ad-hoc system, there is no fixed server, and that makes establishing a proxy anther challenging task.
- *Fault-tolerance* - Disconnection in ad-hoc network is not a failure event, and rather a mobility behavior. Nevertheless, it is perceived by the system as a failure event. More flexible fault-tolerant methods are needed to cope with this new "failure model".

3. ARCHITECTURE FOR AD-HOC WIRELESS COLLABORATION

3.1. Requirements

In designing the framework, which we call YCab, our 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. 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.

3.2. System Design

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.

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.

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 [[10], [19]]. 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 [9]. Other applications such as text chat and shared whiteboard are within the paradigm of collaboration on a small device.

3.3. Support for Decentralized Control

One of the main features of a mobile, ad-hoc collaborative system is that the duties of maintaining the session space are distributed among the session participants. Therefore, unlike collaborative systems designed for a fixed environment, mobile systems must include provisions for decentralized control that is as transparent to the end user as possible. After all, no user wants to be burdened with managing the session, especially if such a task can be accomplished with minimal user intervention.

3.4. Support for Ordering of Messages

In distributed systems, the ordered delivery of unicast and multicast messages is ensured by an event replication mechanism that usually relies on some notion of logical clocks. In some instances, the underlying hardware aids in the replication process, by providing built-in synchronization mechanism. Ordering of messages could seem to be difficult to achieve in ad-hoc networks. However, we have looked carefully into the characteristics of the wireless network we chose to use and have found out that to the opposite, message ordering is easy to achieve in ad-hoc networks.

Specifically, the IEEE 802.11 standard for wireless LAN networks has several unique features that make it suitable for an ad-hoc environment [4]. The support for broadcast and multicast capabilities that is specified in the standard is very useful in designing collaborative systems. In such systems, there is frequently a need to deliver information to several clients simultaneously. In addition, the 802.11 standard is a collision avoidance protocol (CSMA/CA). Therefore, when one of the devices is transmitting, no other devices can transmit at the same time. This is guaranteed by the request to send (RTS) and clear to send (CTS) message pair exchanges that implement collision avoidance in 802.11. While this might seem like a serious issue for networks consisting of a large amount of clients (tens of collaborators), its impact is not felt in smaller groups (up to 15 collaborators). The multicast capabilities combined with the collision avoidance protocol result in messages being ordered the same on all clients.

3.5. Target Platform Summary

Our initial test setup consisted 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 was established using Lucent WaveLAN cards. However, due to inadequate support for multicast in the Windows CE drivers of several Wireless LAN cards, an alternative platform was selected. The alternative platform consisted 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. Although we were forced to evaluate YCab on an alternative platform, we believe that it would perform equally satisfactorily on our initial target platform.

4. YCAB: API AND ENVIRONMENT

4.1. Description of the API

The YCab API was developed using Sun's Java API [22]. The API conforms to Personal Java 1.1 specifications, so that it can run on many portable devices. 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 2**.

At the simplest level, the user can use the YCab application provided with the API. At a more complex level, the developers can develop their own application using the provided services in order to fully customize the application for the target device. 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. 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. 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 into 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.

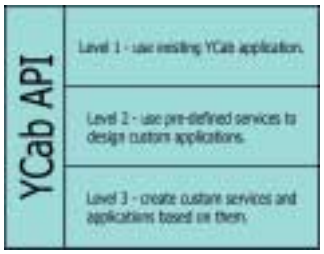

Figure 2 - Three levels for YCab API use

4.2. Description of the Client

4.2.1. GUI Description

Figure 3 - YCab application in session

A YCab application consists of a single window (see **Figure 3**). That contains both window components as well as a menu bar. The YCab window is actually divided into separate areas. Each one of these areas belongs to a 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 sample 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.

The Session menu contains actions that are related to the application. It allows the collaborator to join or leave a session, and terminate the application.

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 if state recovery can be performed.

In addition to these standard menus, each application may also have menus that are associated with each service. Developers are free to add any menus they wish, or they can add any of the following predefined menus:

- Restore State – 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.
- Optimization – sets the optimization level for the service: *Real Time*, *High Bandwidth*, and *Low Bandwidth*. The optimization mode refers to the service's usage of network resources.
- Control – controls the execution of a Threaded Service. This option is used to start and stop a given service.

4.2.2. 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. Such an arrangement 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 components (**Figure 4** describes the architecture):

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

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.

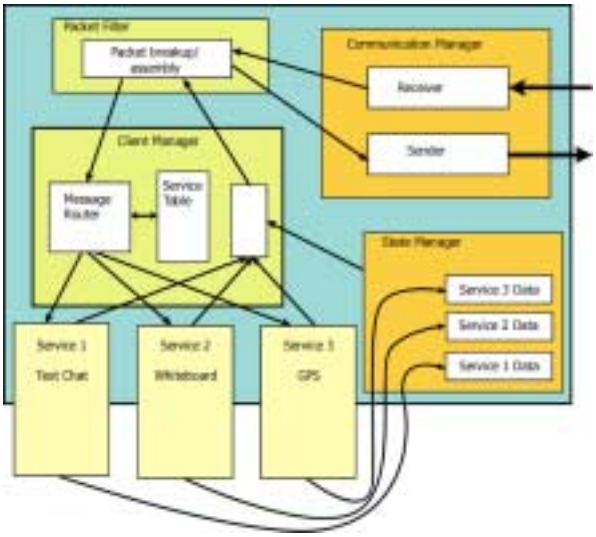

Figure 4 - Architecture of YCab client.

There are two main features that we believe distinguish our solution from other collaborative systems. One, the system is entirely decentralize and does not require a separate client in order to support the session. All the clients run exactly the same software. And second, all the communication between the clients is done via multicast packet. There is no one-to-one communication between the collaborators. Even though each client receives every message that is sent in the session, it only processes the messages that are addressed to that client.

4.3. Description of Other Ycab Components

4.3.1. Core Message

CoreMessage is the basic communication unit used by the YCab API. Each CoreMessage consists of the header and the payload. The payload contains 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 (see **Table 1**).

Table 1 - 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.

4.3.2. Communication Manager

The Communication Manager is responsible for providing asynchronous communications between the clients (see **Figure 5** for the design of the Communication Manager).

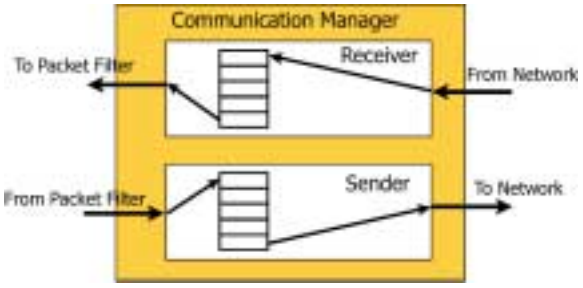

Figure 5 - Architecture of Communication Manager.

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.

4.3.3. Client Manager

At the heart of the client lies the Client Manager. This 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 6**). 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 all components.

The Client Manager is the "glue" that keeps the environment together. 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.

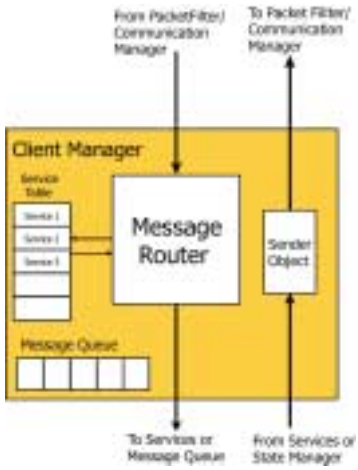

Figure 6 - Architecture of Client Manager.

4.3.4. Message Router

The Message Router is responsible for routing all messages to appropriate services. It runs in a continuous loop that performs three basic steps (see **Figure 7**).

- 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.
- 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.
- Update state information – the Message Router determines if the service for which the message is addressed has state recovery enabled. If it is enabled, 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* - 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 and are processed once the state recovery is completed.

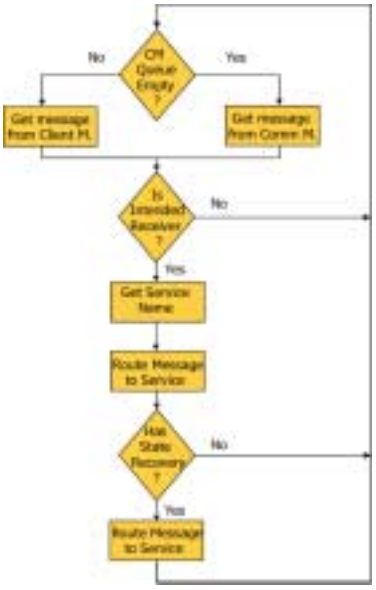

Figure 7 - Message router algorithm.

4.3.5. The Packet Filter

At times, the client may want to send a CoreMessage that is larger than the maximum allowed datagram packet (e.g. an image). Since we did not want to place limitations in the CoreMessage size, we included a PacketFilter that resides between the Client Manager and the Communication Manager (see *Figure 8*).

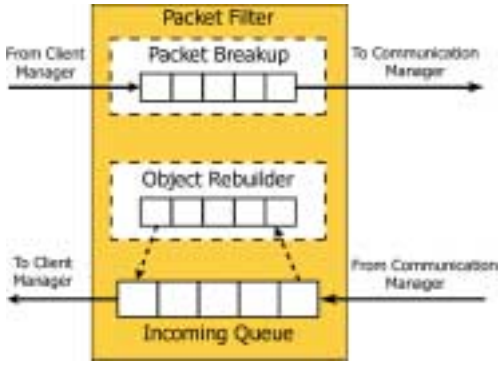

Figure 8 - Architecture of Packet Filter.

During initialization, the Packet Filter sets its filtering size to the maximum packet size as provided 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, which checks the size of the CoreMessage. If the CoreMessage exceeds the maximum packet size, it is broken up into a series of smaller CoreMessages. 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. Each message is then sent out to the other clients using the send method provided in the Communication Manager.

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. Because other clients in the session are also sending messages simultaneously, the series of partial messages can be interleaved with messages not belonging to the series. These messages must be queued in the background until the first partial message is fully reconstructed. 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.

4.3.6. The 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. In addition, the manager maintains a list of all services registered, so that it can quickly access relevant information. *Figure 9* shows the architecture of the State Manager.

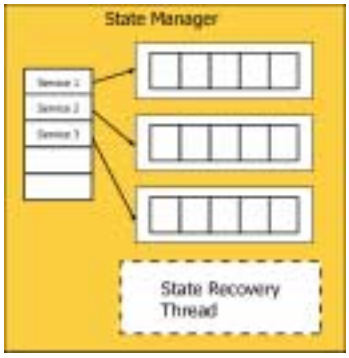

Figure 9 - Architecture of State Manager.

The operation of the State Manager is transparent to the user. To save session information, each service must register itself with the State Manager. 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 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 addressed 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 so that the receiving client can distinguish between regular messages. Once state recovery has finished it sends out a state recovery end announcement message to both the receiving client and itself.

4.3.7. The Service and Threaded Service

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 contains all the service initialization procedures.
- The processMessage() contains 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. 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.

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 *ThreadedService*, which is an extension of the Service object, is included to support these daemon-like services. In addition to all the functionalities available in the Service object, the ThreadedService provides additional control over the Threaded service, namely start and stop.

4.4. YCab Protocols

4.4.1. 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. Because there is no central server in the session that allows 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 10**).

1. The New Client (NC) chooses a handle or screen name to and broadcasts a join session request along with the proposed client name to the members within the session.
2. The clients in the session reply to the join session request by sending the new user a JOIN_SESSION_REPLY message containing their respective names and ranks.
3. The NC checks the JOIN_SESSION_REPLY 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 rank in the session.
4. After waiting a specified timeout period, the new client sends a join session message 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.
5. All clients process the join session message and add the associated client name and rank into their peers list.

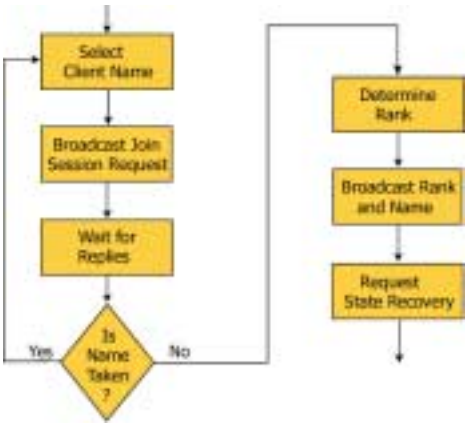

Figure 10 - Procedure for joining a session.

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.

4.4.2. Leaving a Session

A client leaving the session sends a leave session message to the other members. The other members remove the client from their respective peers list. If the client contains ThreadedServices, they are stopped first, before the client is terminated.

4.4.3. State Recovery

The state recovery mechanism enables clients to be brought to a state that is consistent with other session participants. A Session Coordinator is required during the State Recovery process. If a Session Coordinator is not present, the leader election algorithm is used to designate a new coordinator. Figure 15 provides a brief overview of the state recovery algorithm used within YCab, assuming the a Session Coordinator is present.

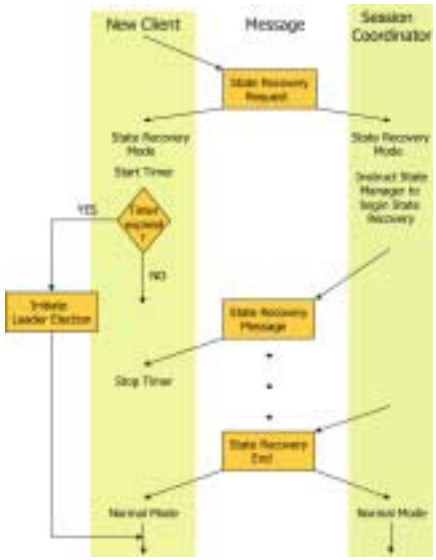

Figure 11 - State recovery algorithm.

4.4.4. Leader Election

During the normal course of a session, the session coordinator may become disconnected from 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. **Figure 12**outlines this process.

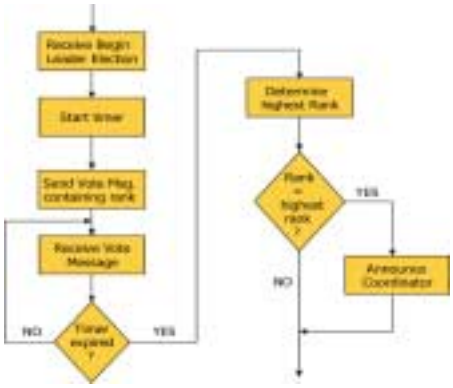

Figure 12 - Leader election algorithm

4.4.5. Pinging

During the lifetime of a typical collaboration session, many members may voluntarily choose to leave but some may be involuntarily disconnected. In such case, both the disconnected member and the rest of the members will have an incorrect "view" of the session state. These situations not only leads to miscommunication and confusion, but also destroy the semantics of collaboration. The following steps occur throughout the duration of a session:

1. A 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 is received, a ping response 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.

5. YCAB SERVICES

As a part of the YCab API, we have implemented several services that either play an intrinsic role in the API, or are provided as an example of what can be done with YCab. YCab services are divided into two groups: required services, and optional services. The required services are an intrinsic part of the YCab environment and are always included in the client. These services and their operation are for the most part transparent to the user. The optional services are services that are not required for the YCab client to function properly. These modules provide additional functionality, and they can be used to quickly design custom applications. Here is a list of the services:

- Leader Election Service – is a required service that provides the implementation of the leader election protocol. This service performs the process required to elect a session coordinator.
- Session Manager Service – is a required service that cooperates with the Client Manager to provide session management services.
- Session Ping Service - is an optional service that provides pinging services for discovery of clients in the session. The pinging of session members occurs in the background and as such does not require a lot of input from the user.
- Text Chat Service - is an optional service that implements a simple text chat service. A text chat service enables one to send text messages to other session participant, similar to instant messenger (see **Figure 3**).
- Whiteboard Service – is an optional service implements a shared whiteboard. The Whiteboard service provides a shared drawing surface that is used to visually exchange information with other session participants (see **Figure 3**).
- Image Viewer Service - is an optional service used to share local images among the session participants. This service only supports image formats that are platform independent, namely JPEG and GIF formats (see **Figure 3**).
- GPS Service – is an optional service provides services for working with GPS devices. It is used to improve the session awareness by graphically displaying the locations of session participants (see **Figure 3**).

Client Info Service – is an optional service that provides session and state information for the client. It monitors for any changes to the client and the session, and updates the display accordingly (see **Figure 3**).

6. API OPTIMIZATIONS

When implementing the API, its goals, namely small footprint, portability, and efficiency, were constantly kept in mind. Those criteria were used to try to optimize the API without sacrificing its usability. The optimizations can be found on two

levels. The first level is the environment and the API itself. Here a conscious effort has been made to provide the most functionality in the smallest package. Currently the entire system occupies about 170KB of memory and can be further reduced by providing compressed packages or removing services that are not used in a given application. Where network communication is involved, a substantial effort has been made to transfer as little data as possible in order to accomplish the required task. These optimizations involve both the types of objects transferred between the clients, and the optimization of the protocols. The second level of optimizations stems from the services itself. The Service API includes provisions for implementing different levels of optimizations. While these optimizations are not required, the developers are encouraged to use these facilities.

7. CONCLUSIONS AND FUTURE WORK

The evaluation of the YCab application proved to be quite successful in both the fixed network environment, and the wireless mobile environment. The fixed network simulated ideal conditions, in which a permanent, high-bandwidth connection is present at all times with very little, if any, packet loss. Although under most circumstances packet loss was not a problem, it did occasionally occur during the state recovery stage when a significant amount of information was sent in order to bring the new client up to the state of the session. While occasionally losing individual data packets usually does not present a problem, losing control packets may cause the system to remain in a certain state. One situation that caused packet loss to occur was during rapid sending of messages. Apparently, due to network latency, packet loss was quite severe, so in order to account for it, a send delay was added to the Communication Manager. Ideally, the client should adjust the send delay automatically, according to the network conditions. Using the calculation to adjust the send delay, YCab could self-adapt to the network conditions in order to reduce packet loss without manual configuration.

An area in which the YCab can be further improved is further reduction of the code footprint. This would enable YCab to be ported to smaller and less powerful devices such as MID which include cell phones and other handheld devices. 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.

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. Although creating such a protocol may increase the rate of message transmission, certain services may require this type of delivery mechanism.

8. REFERENCES

[1] Ackerman, M.S., "Everything You NEED to Know About Collaboration and Collaboration Software," *In proceedings of JCSE 97*, Boston, MA, USA, 1997.

[2] Begole, J., Rosson, M.B., & Shaffer, C.A., "Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems," *ACM Transactions on Computer-Human Interactions*, Vol. 6, No. 2, June 1999, pp95-132.

[3] Burridge, R., "Java Shared Data Toolkit User Guide," Version 1.5, April 1999.

[4] Cheng, L., & Marsic, I., "Wireless Awareness for Multimedia Applications," DARPA Contract No. N66001-96-C-8510, Department of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ, USA.

[5] Cravotta, N., "Wireless standards vie for your app," *EDN*, May 13, 1999, pp60-72.

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

[7] Guerlain, S., Lee, J., Kopischke, T., Romanko, T., Reutiman, P., & Nelson, S., "Supporting collaborative field operations with personal information processing systems," *Mobile Networks and Applications 4*, 1999.

[8] Ling, J., Helal, A., and Elmagarmid, A., "Client-Server Computing in Mobile Environments," ACM Computing Survey, June 1999

[9] Mandviwalla, M., & Olfman, L., "What Do Groups Need? A Proposed Set of Generic Groupware Requiremens," *ACM Transactions on Computer-Human Interaction*, Vol. 1, No. 3, September 1994, pp 245-268.

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

[11] NetMeeting, http://www.microsoft.com/windows/netmeeting/.

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

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

[14] Satyanarayanan, M., "Fundamental Challenges in Mobile Computing," School of Computer Science, Carnegie Mellon University, to appear in IEEE Computer.

[15] Seal, K. & Singh, S., "Loss Profiles: A Quality of Service Measure in Mobile Computing," NFS grant number NCR-9410357, Department of Computer Science, University of Carolina, Columbia, SC.

[16] Spellman, P.J, Mosier, J.N., Deus, L.M., & Carlson, J.A., "Collaborative Virtual Workspace," *In proceeding of GROUP 97*, Phoenix, Arizona, USA, 1997.

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

[18] Whalen, T., & Black, J.P., "Adaptive Groupware for Wireless Networks," *In proceedings of MOBICOM 99*, New Orleans, LA, 1999.

[19] Yavatkar, R., Griffioen, J., & Sudan, M., "A Reliable Dissemination Protocol for Interactive Collaborative Applications," *ACM Multimedia 95 – Electronic Proceedings*, San Francisco, CA, November 1995.

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

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

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

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