# A Robust Multi-Server Chat Application for Dynamic Distributed Networks

by

Michael C. Hamler
B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 2004

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 3, 2004

Author___________________________
Department of Electrical Engineering and Computer Science
September 3, 2004

Certified by____________________
Clifford J. Weinstein
Group Leader, Information Systems Technology Group, MIT Lincoln Laboratory
Thesis Supervisor

Certified by______________________
Roger I. Khazan
Staff Member, Information Systems Technology Group, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by_____________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

This work is sponsored by the United States Air Force under Air Force contract F19628-00-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

# A Robust Multi-Server Chat Application for Dynamic Distributed Networks

by

**Michael C. Hamler**

Submitted to the
Department of Electrical Engineering and Computer Science

September 3, 2004

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

## ABSTRACT

This thesis presents the design and implementation of a robust chat application for dynamic distributed networks. The application uses a decentralized client-server communication model and a reliable communication service to make it robust. The application turned out to be a critical support tool for the MIT Lincoln Laboratory teams participating at the Joint Expeditionary Force Experiment, 2004. The thesis also describes an implementation of several tools for monitoring and analyzing performance of the application.

Thesis Supervisor: Clifford J. Weinstein
Title: Group Leader, Information Systems Technology Group, MIT Lincoln Laboratory

Thesis Supervisor: Roger I. Khazan
Title: Staff Member, Information Systems Technology Group, MIT Lincoln Laboratory

Dedicated to
Mom, Dad and my Brother Dion
For Always Believing in Me

## Acknowledgements

I need to thank Roger Khazan and Cliff Weinstein for their support throughout this last year. Roger, you've been a great teacher, colleague and friend. Thanks for helping me get through this. I'd also like to thank Steve Goulet for his insight and his detailed knowledge of operating systems, both Windows and Unix.

Thank you, to all my friends and family for supporting me all these years and cheering me up when things didn't look so good. I guess we've finally made it.

Special thanks to Flora Lee, for putting up with me and supporting me while I wrote this thesis.

## Table of Contents

## List of Figures

## List of Tables

# 1 Introduction

This thesis presents the design and implementation of a robust chat application for dynamic distributed networks. Such networks are often comprised of a dynamic set of nodes, which have varying connectivity to other nodes and are possibly mobile. Because of this dynamism, network partitions and topological changes are very common as nodes join or leave the network and links fail and recover. Traditional chat applications typically assume highly reliable network connections in order to function properly. If deployed in a dynamic distributed network, traditional chat applications may fail ungracefully, causing great delays in communication and significantly restricting their usefulness. For example, this situation happened during the Joint Expeditionary Force Experiment in the summer of 2002 [1].

Our chat application provides a robust tool for collaborative textual chat. We describe an application as robust when it is able to tolerate unreliable network conditions, such as those presented by a dynamic distributed network. Our chat application uses a decentralized client-server communication model and a reliable communication service to make it robust. Client-server systems allow multiple clients to connect to a single server, which coordinates communication between those clients. A decentralized system allows multiple servers to work together to form a flexible network of connected components.

In a dynamic distributed network we refer to a subnet of reliably connected computers as a site. Our application capitalizes on these reliable subnets by placing a chat server and multiple chat clients in each subnet. This way, every chat client can always reliably connect to a chat server within its subnet, even if the larger network environment is experiencing link failures or even network partitions. When sites are connected to each other, the clients within those sites can communicate as if they were all connected to the same site. When sites are disconnected from each other, the clients within each site can still communicate with each other. The specific type of a dynamic distributed network that motivated this thesis is comprised of a relatively small number of reliable subnets, inter-connected with each other by unreliable links. Although this network contains only a small number of reliable subnets, the set of nodes in each subnet may be large and dynamic.

In addition to being robust, our application provides useful user interface features, such as site awareness and connection awareness, which provide the user with information about the topology and connectivity of their subnet. This information allows users to act effectively and adjust to the variable availability of other users.

To test our theories on robust collaboration we implemented a prototype of our chat application in Java. This made the application easily portable, which is an important feature for applications in large network environments where multiple different operating systems may be present. We also implemented several support tools that allow us to easily monitor and analyze the behavior of our application.

## 1.1 Background and Motivation

Collaborative applications, such as textual chat and shared whiteboards, have recently been recognized as valuable tools for supporting military operations [1]. Unfortunately, traditional applications for collaboration are not designed for the dynamic environments in which military operations are executed. Partially motivated by poor performance of traditional chat applications in the Joint Expeditionary Force Experiment (JEFX) 2002 [1], the MIT Lincoln Laboratory Information Systems Technology Group developed a research program to meet the need for robust collaborative applications. The chat application presented in this thesis was developed as part of this program.

Our chat application runs on top of the Robust Collaborative Multicast service (RCM) developed at MIT Lincoln Laboratory. RCM is a communication service designed to "support collaborative applications operating in dynamic, mission-critical environments" [2]. Collaborative applications often require a highly reliable and low-latency communication service to ensure participants can communicate effectively with each other. At the same time, another desirable feature is that participants see a globally consistent ordering on messages. RCM implements reliable communication between sites and globally orders messages according to a certain property, called Intermittent Global Order (IGO). RCM is designed to be a generic communication service. It has a simple interface that allows client applications to send and receive messages and receive connection status updates from the service.

Our chat application handles intra-site client-server communications and the server-server interactions necessary to operate a textual chat application. To perform reliable inter-site communication, the chat servers use RCM. The chat application takes full advantage of the connection status information that RCM provides and presents this information to the user in a simple and intuitive way.

The primary motivation for developing our chat application was to demonstrate a prototype system capable of supporting robust collaboration in an airborne command and control environment, such as the environment of the Multi-Sensor Command and Control Constellation (MC2C). The MC2C environment is a network-centric control structure designed to promote the easy flow of information in the battlefield [1]. As Figure 1 shows, the MC2C environment is a type of a dynamic distributed network consisting of multiple mobile air and ground sites.

In addition, the prototype was meant to be a test application for the RCM service and a platform for experimenting with new ideas related to the user interface, such as site awareness and connection awareness (these are discussed further in sections 1.3.4 & 1.3.5, respectively).

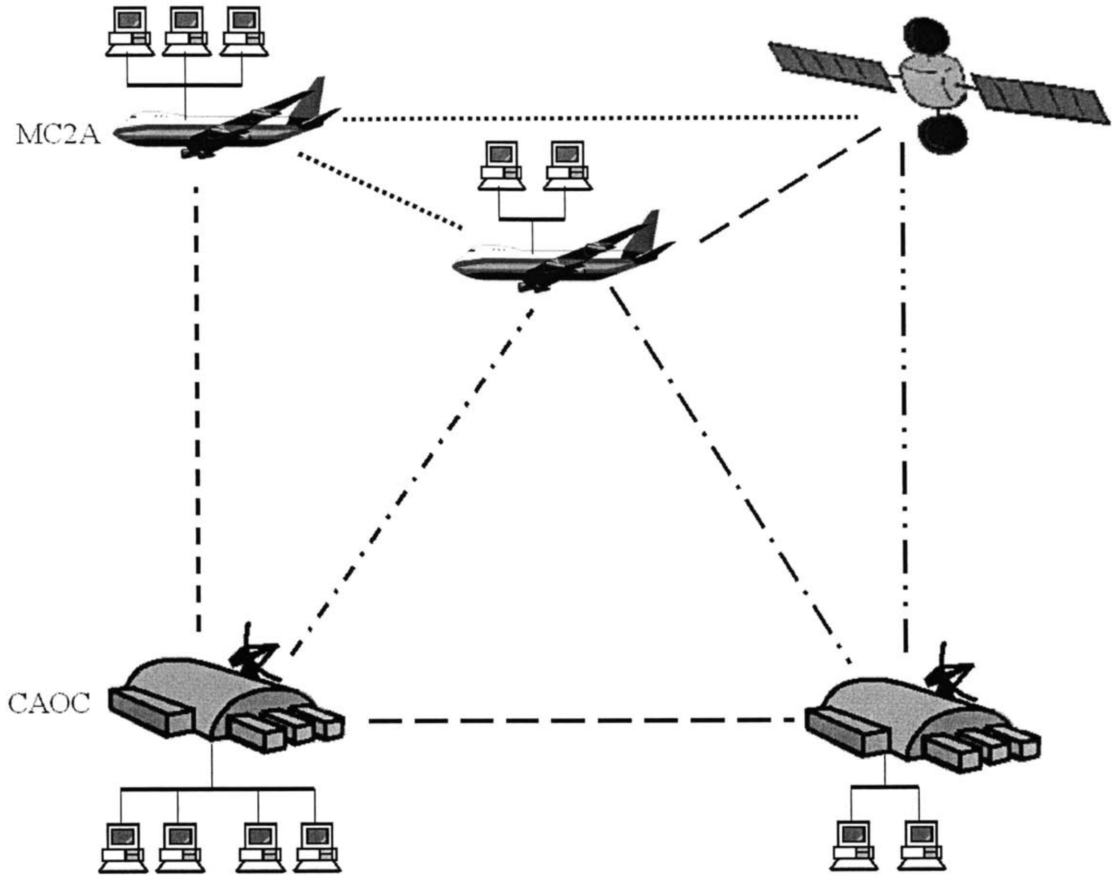


**Figure 1: The MC2C Environment. There are multiple sites, each with multiple chat clients. Two of the main components of the MC2C environment are the Combined Air Operations Center (CAOC) and the Multi-mission Command and Control Aircraft (MC2A).**

The original goals of this project were limited to demonstrating robust collaboration concepts and implementing a prototype robust chat application; the goals did not include any deployment plans. However, the project coincided with a greater effort by the MIT Lincoln Laboratory to develop various prototype MC2C technologies and demonstrate them at the Joint Expeditionary Force Experiment (JEFX) 04. Roughly a month before JEFX 04, we had an opportunity to show our chat application to one of the teams involved in this effort; the team found it to be a much better alternative to the communication tool that they were using at the time and started using it. Then, when JEFX 04 started, the other MIT Lincoln Laboratory team switched to using our chat application as well. The application operated robustly despite frequent problems in the underlying airborne network and turned out to be a critical support tool for the MIT Lincoln Laboratory teams at JEFX 04 [3][4][5][6].

## 1.2 Thesis Overview

This thesis paper is arranged into seven chapters, plus a glossary and a list of references, which appear at the end of the document. Chapter 1 is the introductory chapter and discusses the background and motivation for the robust chat application project. It also gives a brief overview of the paper and highlights the tangible results of the thesis. Chapter 2 describes related work in the field of collaborative textual chat. It discusses previous attempts at robust chat and several mainstream applications for collaborative chat. Chapter 3 describes the intra-site and inter-site network conditions of the JEFX environment, in which our chat application prototype was deployed. Chapter 4 explains the architecture of the robust chat application and describes several important protocols. Chapter 5 describes the implementation details of important chat application components. It also discusses the implementation of several valuable test and analysis tools, including the chat bots, data loggers, and monitor tools. Chapter 6 discusses the evaluation of the robust chat application, including results and feedback from the JEFX 04 flights. The test-bed architecture and test-schemes are described. The data gathered using the test tools and is also discussed. Chapter 7 describes possible avenues of future work related to this thesis. Several improvements for the robust chat application and test tools are suggested, in addition to some new robust collaboration projects.

## 1.3 Goals

The following sections outline the goals we strove for while designing the Robust Chat Application. The first three goals are design related and cover the modularity, extensibility and reliability of the application. The remaining goals are interface related and cover the two primary characteristics that set the Robust Chat Application apart from other chat applications.

### *1.3.1 Decoupled Communication Service and Chat Application*

In designing our application we wanted to separate the inter-site communication details as much as possible from the intra-site applications that would be using it. Therefore, RCM was designed by the project team to provide robust communication among sites [2]. The chat application uses a very simple interface to connect to RCM, which manages the inter-site connections and communication. To function properly, the chat application needs only minimal information about the status of the inter-site connections. This modularity will allow future modifications to RCM and the chat application to happen independently.

### *1.3.2 Extensibility*

Extensibility describes how easy it is to extend or modify an application. In developing this application we wanted to plan for future development, so we designed and implemented our application to be modular, polymorphic and otherwise flexible. Modularity allows us to exchange different parts of the application without rewriting other parts, while polymorphism allows parts of the application to behave differently based on their input. An example of the polymorphism in our application is evident in the way it handles messages. Although all messages are transmitted using the same processes, the handling of individual message types is specialized. These design considerations will make future additions and modifications easier to implement. Section 7.2 discusses possible future applications that would benefit from this extensibility.

### *1.3.3 Reliability of Delivery*

In a military environment all information exchanged by users, should be treated as critical to mission success. It is therefore unacceptable to not account for all the messages sent over the network. In designing our application, we want to guarantee that every message sent between two connected sites will arrive and that the messages will be in order. If for any reason there is a gap in the message sequence, possibly from a long-term disconnect, the application will notify clients of the missing messages and will attempt to deliver them later if they ever arrive.

Messages sent from clients to servers are also reliably delivered. Although clients and servers are generally connected over a reliable network, client-server disconnects may still occur. If the client had unsent messages at the time of disconnect it will attempt resend them when it reconnects to the server.

### *1.3.4 Site Awareness*

Site Awareness describes application features, which allow users to quickly determine at which sites other users are located. This means all users from a particular site are identified as being at that site. For the purpose of tactical communications this presents clear advantages. Users on a site aware system are immediately aware of the difference between inter-site communication problems and intra-site communication problems. If users become disconnected because their entire site is experiencing communication difficulties, this will be apparent since all the users from that site will be affected in the same way.

### *1.3.5 Connection Awareness*

Connection Awareness is knowledge of the state of the underlying network. Although the chat application does not handle the network connections between sites, it is aware of their status at all times and will make this information available to the user. By making connection awareness an explicit feature of the chat application we give the user an advance warning about the state of the network. Traditional chat applications expect the network to be very reliable, which make connection awareness unnecessary. Therefore, when these applications are placed in a dynamic network environment they lack the capability to provide the user with valuable status information about the connectivity of the network.

## 1.4 Thesis Contributions

The contributions of this thesis can be summarized as follows:

- Designed, implemented, and tested a robust chat application prototype for dynamic distributed networks

- Designed and implemented support tools and scripts to help analyze and test robust chat application and RCM
- Provided the robust chat application prototype and technical support for MIT Lincoln Laboratory research teams involved in the JEFX 2004 effort

# 2 Related Work

In this chapter we evaluate existing chat applications.

## 2.1 Internet Relay Chat (IRC)

Internet Relay Chat, commonly referred to as IRC, is one of the oldest chat applications available. Created in 1988, IRC is designed as a distributed multi-server system, where clients communicate through the servers. This means, clients never directly connect to one another to communicate. To chat with other users, a user must join a channel, which is the IRC equivalent of a chat room. Just as in a chat room, any message sent to an IRC channel is relayed to every user in that channel [8].

The strength of the IRC system lies in its availability. The IRC protocol is free, so there are many client and server implementations available for public use, which have been tested and improved through years of peer review. This also means the system is more extensible than most, since IRC defines a group of protocols rather than a strict implementation.

IRC servers can handle network disconnects elegantly, removing channel members who have been disconnected and eventually reconnecting once the network heals itself. This capability was deemed necessary, since IRC uses the Internet to communicate. As the Internet continually evolves, network paths between servers change or become congested, causing temporary disconnects between IRC servers [8].

The main drawback to IRC is that every server needs to know about every other server and user in order to operate correctly [8]. This means in a dynamic distributed network, where the configuration of the network is constantly changing, IRC servers may have difficulty adapting.

## 2.2 Jabber

Jabber is an XML-based instant messaging platform originally developed by Jeremie Miller in 1998. Jabber is a decentralized client-server based system, much like IRC. Clients generally communicate with servers in order to send messages to other clients. Although clients can exchange information directly, these exchanges must be brokered by a server. Like most chat applications, Jabber allows users to create and join multiple chat rooms [9].

The addressing structure in Jabber uses the same format as email. A username is of the format *nickname@server.com*, where *nickname* is the identifier for a particular user and *server.com* is the jabber server that user is connected to. Unlike many chat applications, jabber allows multiple users to use the same *nickname* at the same time. In order to distinguish between such users, the system adds a string, called a *resource*, to the end of the username. The *resource* is designed to make each username unique [9].

When a Jabber client sends a message to another Jabber client, the Jabber server will decompose the username of the destination user to determine that users server address. The Jabber server will use the server component of the username to query a DNS server for the other server's IP address. The server will then send the message to the other server, which is responsible for passing the message to the appropriate user, based on the *nickname* and *resource* information in the message [9].

The Jabber server's reliance on DNS servers for IP information means they require more overhead than many other chat applications. In the MC2C environment, DNS servers would be of little use, since the changing topology would cause cached DNS entries to frequently be invalid.

# 3 Environment

In this chapter, we describe the general characteristics of the MC2C environment and the JEFX 04 environment, in which we tested our prototype application. In general, the MC2C environment consists of between two and ten sites. A site may be a ground location or an airplane and may contain up to a hundred clients. We assume the intra-site network is reliable and high bandwidth; often this will take the form of an Ethernet. The inter-site links are unreliable and exhibit varying bandwidth capabilities.

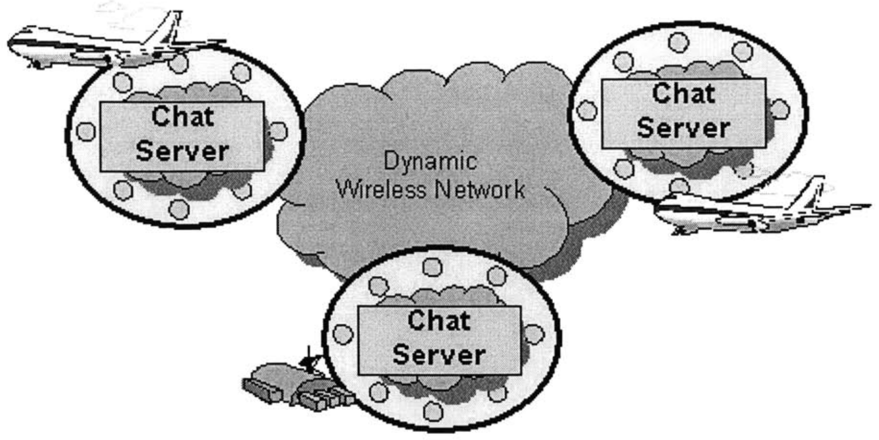


**Figure 2: MC2C Environment Model.**

Figure 2 shows a model of the MC2C environment. At a high level, the MC2C environment can be modeled as clusters of well-connected nodes (the sites) separated by a network with unreliable performance. The following sections describe the network configuration for JEFX 04.

## 3.1 JEFX 04 Intra-site Communications

The trailer and airplane subnets are divided into red (classified) and black (unclassified) subnets. Each subnet is composed of a router, an Ethernet switch, and several personal computers (PCs). The red and black subnets are separated by encryption devices, which are certified to encrypt classified information for transmission over unclassified networks. The trailer Ethernet switches contain 48 10/100 Mbps ports, which the PCs and routers connect to. Routers act as the ingress and egress points for the subnets they are in. Figure 3 depicts the general intra-site network configuration for JEFX 04 [10].

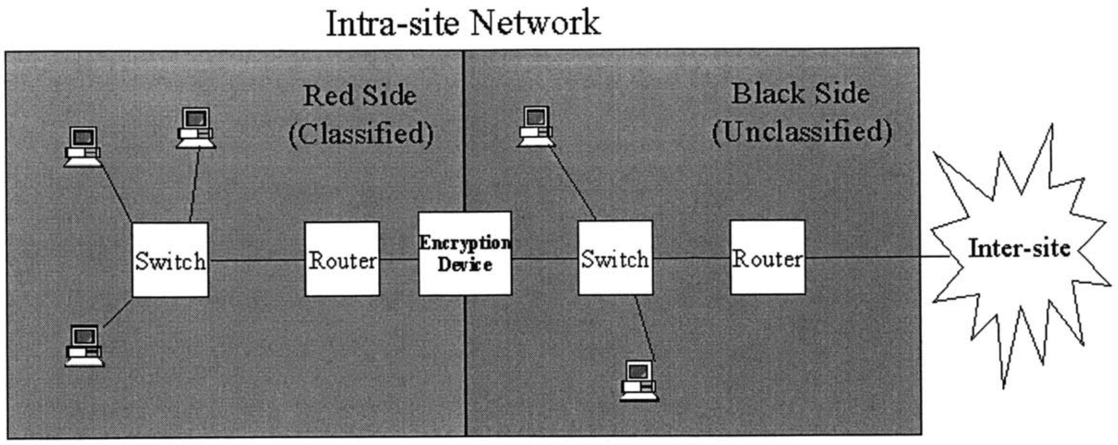


**Figure 3: Intra-site Communications.**

## 3.2 JEFX 04 Inter-site Communications

There are four inter-site links available for JEFX 04: TCDL, Connexion by Boeing (CBB), Inmarsat, and Iridium. Information on each of these links is listed in Table 1. During missions, special routing software monitors the links and chooses the best ones available. The TCDL link is the preferred link, because of its high bandwidth and low latency, followed by CBB and then Inmarsat. The fourth link, Iridium, is reserved for high priority traffic and network management, so applications will generally not be able to use it. During the test flights, the chat application was run in various network modes. Sometimes the application was dedicated to a single link, while at other times it was set up to use all available links [10].

| Link Name | Type of Link | Beyond Line-Of-Sight? | Bandwidth (Kbps) | Delay (ms) |
|---|---|---|---|---|
| TCDL | Point-To-Point; Air-Ground or Air-Air | No | 10,700 Full Duplex | 3-5 |
| CBB | Geostationary-Earth-Orbit (GEO) Satellite | Yes | 50-60 to trailer 300+ to plane | 900 |
| Inmarsat | GEO Satellite | Yes | 128 | 1800 |
| Iridium | Low-Earth-Orbit (LEO) Satellite | Yes | 2.4 | 2000-5000+ |

**Table 1: Inter-Site Link Information**

In Table 1, the links are listed in order of preference. This is primarily a result of their latency and bandwidth capabilities. At around 10 Mbps and with a 3-5 ms latency, the TCDL link is by far the fastest connection available. However, this high speed is a tradeoff for availability. The TCDL connection is a Line-of-Sight (LOS) link, which means that it implements a direct point-to-point connection between the two ends of the link. The directness of the connection is what allows TCDL to be fast, but it also means that objects placed between the two end points can interrupt the link. During the JEFX 04 flights, the airplane would fly in a circular pattern, which meant it would perform big turns from time to time. Turning would cause the wings to momentarily block the TCDL connection, subsequently causing the connection to go down [10].

Like the TCDL link, satellites tradeoff speed for availability. However, unlike the TCDL link, the satellite connections accept long message latencies and low bandwidths in exchange for better availability. Since satellite communications are indirect, there is a much smaller chance of the signal being blocked and therefore their availability is higher [10].

# 4 Design

This chapter discusses several key protocols of the chat application and other important design issues.

## 4.1 Architecture Overview

A high-level overview of the chat application architecture is pictured in Figure 4. At the highest level, the chat application consists of a set of sites connected by unreliable links. Each site consists of a single chat server and multiple chat clients. The two components of a chat server are the RCM Server object and the ChatServer object. The RCM Server handles inter-site communications and provides the ChatServer with status information about inter-site connections. The ChatServer handles intra-site communication and handles chat-related operations. When the ChatServer needs to send messages to other sites it uses the RCM Server. Conversely, if the RCM Server receives a message from another site, it passes the message to the ChatServer.

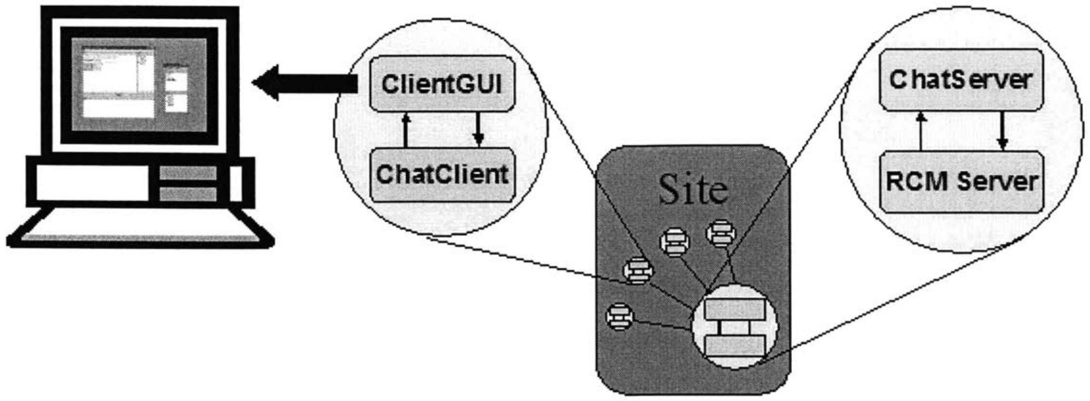


**Figure 4: High Level Overview of the Chat Application Architecture.**

The chat client is composed of the ChatClient object and the ClientGUI. The ChatClient manages the network connection to the ChatServer. The ClientGUI is an assembly of graphical objects that display the Graphical User Interface (GUI) for chat application users. When a user performs an action on the GUI, the GUI converts that action into a message and passes the message to the ChatClient for transmission to the chat server. When the ChatClient object receives a message from the chat server it passes the message to the ClientGUI.

## 4.2 Message Flow Through the Application

This section describes the flow of a text message through the chat application. Figure 5 illustrates the flow of a message in a three-site topology. For this example, we assume that each site contains two clients, each with a different user logged in. The users are named *A* through *F*. We will also assume that users *A* and *D* are in the same chat room, while the other users are not. User *D* is sending a text message to the chat room using the chat application.

1 When *D* enters a message into the input window of his graphical chat room interface, that input is caught by his ClientGUI.

2 *D's* ClientGUI takes his input and packages it into a chat message for his ChatClient.

3 *D's* ChatClient sends the message to *C* and *D's* ChatServer

4 *C* and *D's* ChatServer passes the message to their RCM Server for transmission to other sites

5 *C* and *D's* RCM Server sends the message to *A* and *B's* RCM Server and itself, but not *E* and *F's* RCM Server.

6 *A* and *B's* RCM Server passes the message to their ChatServer. *C* and *D's* RCM Server passes the message to their ChatServer.

7 *A* and *B's* ChatServer sends the message to just *A's* ChatClient. *C* and *D's* ChatServer sends the message to just *D's* ChatClient.

8 *A's* ChatClient passes the message to his ClientGUI. *D's* ChatClient passes the message to his ClientGUI.

9 *A's* ClientGUI displays *D's* message to the chat room. *D's* ClientGUI displays his message to the chat room.

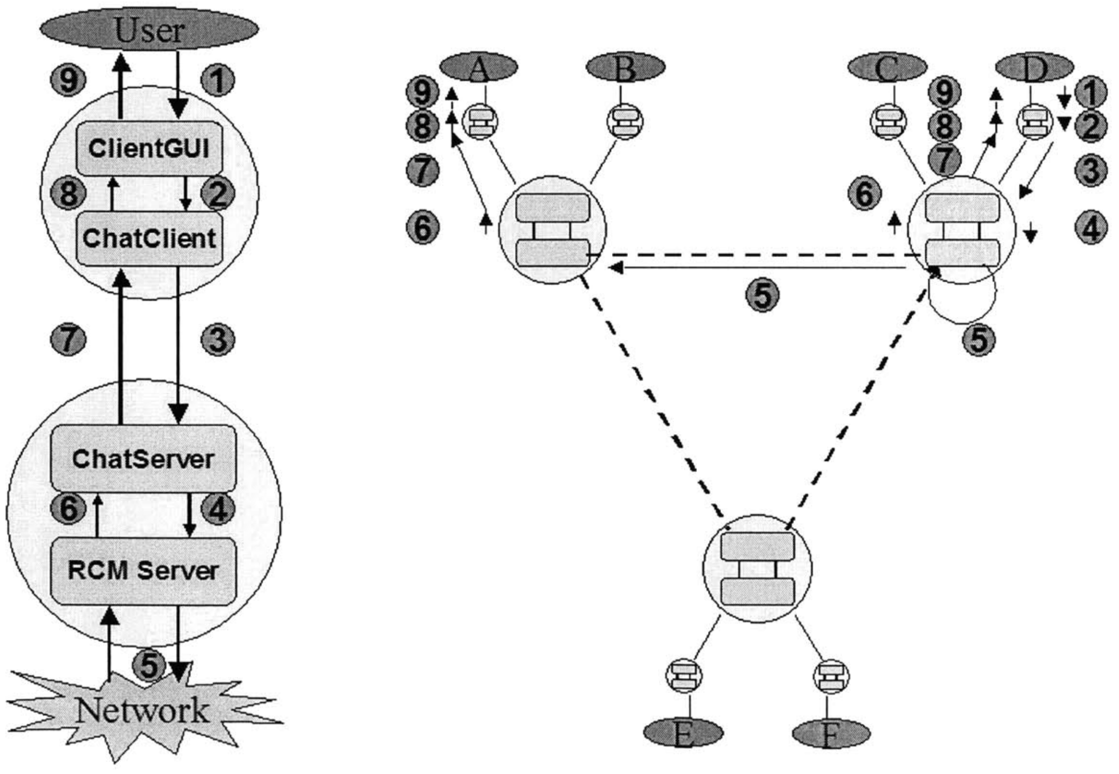


**Figure 5: Message Flow through the Chat Application.**

In step 5, *C* and *D's* RCM Server sends the message to itself to simplify the ordering process. By treating message from itself the same way it treats messages from other sites the RCM Server can order all messages the same way.

Steps 5 and 7 illustrate one of the key advantages of using the multicast protocol (described in section 4.6): It minimizes network traffic. In step 5, *C* and *D's* RCM Server does not send the message to *E* and *F's* RCM Server, since neither *E* nor *F* is a member of the chat room for which the message is intended. This makes reliable communications easier, as we no longer have to guarantee deliver to sites that do not actually need the message. In step 7, the same principle is at work. Users *B* and *C* are not members of the chat room, so the ChatServers only send the message to *A* and *D*.

## 4.3 Client-Server Login Protocol

The login protocol describes the steps taken by a chat client and chat server when the chat client attempts to log in. In designing this protocol we assume that the client and server have agreed a priori on which port the server will listen for clients.

### *4.3.1 Client Actions*

Before attempting to log in we assume the client has obtained the username and password for the user attempting to log in. The client must also know the IP address of the server the user wishes to log in to. The client then performs the following actions:

```
1      Attempt to connect to server
2      After connect succeeds
3      Send <u, p> to server, where u = username, p = password
4      Wait for <r> from the server, where r = Rejected or Accepted
5      If (r == Accepted)
6           Add OldQueue to end of OutQueue
7           Login succeeded
8      Else
9           Login failed
```

If any one of the above steps fails or the client disconnects from the server, the login process is considered a failure. OutQueue represents the list of messages that have yet to be sent to the server, while OldQueue represents any stored messages that have not been sent from a previous connection. OldQueue will only contain messages if the client is disconnected before it can log off from the server (see Section 4.4.1).

In the current implementation the wait in line 4 is indefinite, which could cause the client to become unresponsive. However, in practice this is not the case, since the client and server are on a reliable network. If the server crashes after accepting a connection, the client will receive an exception and close the connection.

### *4.3.2 Server Actions*

Servers listen for client connections on a predefined port. When a server receives a connection attempt on this port it takes the following actions:

```
1       Listen for connections on a predefined port
2       Accept connection
3       Start Timeout timer
4       Wait for <u, p> from the client, where u=username, p=password
5       If (Timeout timer reaches 0 before <u, p> arrives)
6            Send <r> to client, where r = Rejected
7            Close connection
8            Login failed
9       Else
10           If (<u, p> are a valid login/password combination)
11                Add Client to list of logged in clients
12                Send <r> to client, where r = Accepted
13                Login succeeded
14           Else
15                Send <r> to client, where r = Rejected
16                Close connection
17                Login failed
```

As in the case of the client, if any of the steps fails the login process is considered a failure.

The purpose of the Timeout timer is to prevent the server from overloading on login attempts. If every login attempt were allowed to stay open indefinitely it would lead to “ghost” logins. These are login threads for which the client has disconnected for some reason, but the login thread has not noticed the disconnect and therefore continues to wait for a message from the client. These threads constitute wasted resources for the chat server. The Timeout timer prevents “ghost” logins by allowing all login threads to only live for a finite amount of time.

Unlike the chat client, the chat server does not store unsent messages during a client-server disconnect, so no old messages are sent to the client after a successful login. Section 4.4.2 discusses how the server deals client-server disconnects in detail.

## 4.4 CLIENT-SERVER DISCONNECTS

The disconnect protocol describes the steps taken by a chat client and chat server when the connection between them is severed. These disconnects are generally much less frequent than disconnects between sites; nevertheless, they may still occur even in highly reliable networks. Therefore, a robust system must be designed to be able to handle such problems.

### *4.4.1 Client Response*

A client-server disconnect will cause the link between the client and server to close unexpectedly, which will generate an error in the client if it subsequently or concurrently attempts a read or write operation. Thus, the client will generally notice a disconnect either while trying to send a message to the server or while receiving a message from the server. In either case the response is the same:

```
Stop any sends/receives that have not completed
OldQueue = unsent messages from OutQueue
Clear OutQueue
Formally disconnect from the server
Atempt to login to Server
```

As before, OutQueue represents the list of messages that have yet to be sent to the server and OldQueue represents any stored messages that have not been sent from a previous connection. In the event of a disconnect, OldQueue is used to store the unsent messages from OutQueue and OutQueue is subsequently emptied. This prevents unsent messages from getting lost when a disconnect occurs. The messages in OldQueue are placed in OutQueue again after the client successfully reconnects (see section 4.3.1).

We chose to store messages from OutQueue in OldQueue to simplify the login process. During the login process, the client needs to send messages to the server, however, none of the old messages may be sent until the login process is complete. Also, if the login process fails and some login messages remain in the OutQueue, these messages should not be resent after the next successful login. Storing the messages that should be resent separately prevents them from mixing with the login messages.

In line 4, "formally disconnecting" refers to any cleanup actions the client must take in preparation for a reconnection attempt. In effect, the client must return to the state it would be in after a normal disconnect. After this cleanup has taken place, the client can once again attempt to log in to the server and resume operating normally.

### *4.4.2 Server Response*

On the server side, any error while sending or receiving messages from a client is treated as a disconnect and handled the same way a client logging off would be. The channel to the client is closed and the client is removed from the group of logged in clients. This zero-tolerance approach to errors allows the client and server to recover as quickly as possible from any error. As soon as the client has been logged off, it can reconnect to the server and resume normal communications.

## 4.5 Server-Server Disconnects

This section describes the protocol that the servers use to handle server-server disconnects, which are anticipated to occur relatively frequently in the MC2C environment. Figure 6 depicts the three possible states for inter-site links, from the chat application point of view. When two sites are well connected the link is in the connected state. When the RCM Server notices the link isn't behaving correctly, the link moves to the suspected state. At this stage the RCM Server attempts to reestablish a link to the other site. The disconnected state indicates that the RCM Server was unable to reestablish a working link and the two sites are no longer in communication with each other.

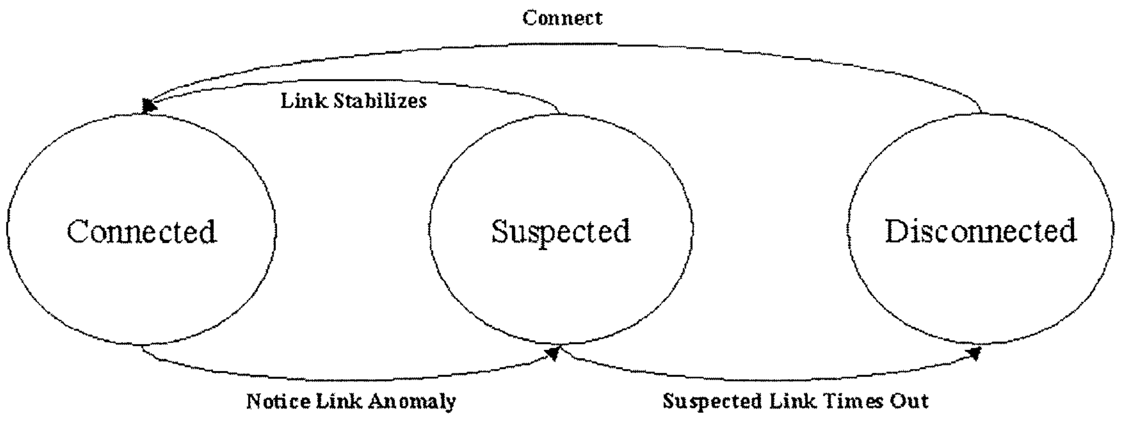


**Figure 6: State Diagram for Site Connections.**

### *4.5.1 Short-Term Disconnects*

When an inter-site link disconnects for a period of time short enough for RCM to bridge it, it is called a short-term disconnect. The exact length of these disconnects is configurable and depends on the systems running the chat servers. During a short-term disconnect, RCM notifies the chat server of a suspected link outage. The chat server forwards this information to all of its clients, which in turn inform the chat application users of the suspected link. The site that the link connects to is, however, still considered connected and the system will still attempt to deliver messages to that site. If messages cannot successfully complete transmission RCM buffers them for retransmission later. If the suspected link returns to a connected state, the system guarantees that all the messages that were sent to the site while the link was suspected will be delivered.

If the suspected link remains unresponsive beyond the specified short-term time limit period, the link is reclassified as disconnected, making it a long-term disconnect.

### *4.5.2 Long-Term Disconnects*

An unbridgeable link disruption is called a long-term disconnect. During a long-term disconnect the disconnected site is no longer considered connected, as opposed to during a short-term disconnect. In order to simplify the design all disconnected sites are treated as unknown sites: RCM no longer guarantees message delivery and the server does not maintain the client lists for disconnected sites.

## 4.6 Multicast vs. Broadcast

In designing our application we had to decide between using multicast and broadcast transmissions. With broadcast each message is sent to all connected sites, regardless of whether those sites host the recipients of the message. With multicast, each message is only sent to the sites that host the recipients of the message.

Broadcasting is the simpler approach to implement and offers some benefits, which multicast does not. Broadcast is easy to implement since every message is automatically sent to everyone. Broadcasting messages also allows the system to use message stability algorithms, which improve the reliability of the system.

Multicast messaging involves sending messages to only those nodes that need to see the message. The greatest advantage of multicast messaging in the MC2C environment is the security aspect. Multicast messaging is inherently more secure than broadcast messaging because multicast messaging has access control built in. If a system broadcasts messages, it must trust that only the intended recipients will view the messages. In a multicast environment, nodes that should not have access to a message are not given the opportunity to view the message.

In addition, multicasting uses unreliable links more effectively than broadcasting. By only sending messages over links when necessary, bandwidth is never wasted by sending large messages that will just be thrown away.

We decided to implement multicast messaging in our chat application prototype because of the benefits to security and efficiency were deemed more valuable than the simplicity offered by the broadcast design.

# 5 Implementation

The chat prototype is written in Java™ (version 1.4). The following sections describe the implementation of several key components of the application: the chat server, chat messages, chat client and the graphical user interface (GUI) for the client. In addition, we will also cover the implementation of important test software, such as the chat bots, data loggers, monitors and the debug class.

## 5.1 ChatServer

The ChatServer object implements the abstract concept we refer to as a chat server. The ChatServer consists of five permanent threads and a variable number of short-lived LoginThreads. There are three main flows of control through the ChatServer, all of which are pictured in Figure 7. The first corresponds to the process of accepting new clients and the other two represent the two flows of messages through the application: Clients-to-RCM and RCM-to-Clients. We will first describe each of the components in more detail and then discuss the control flows, which govern their behavior.

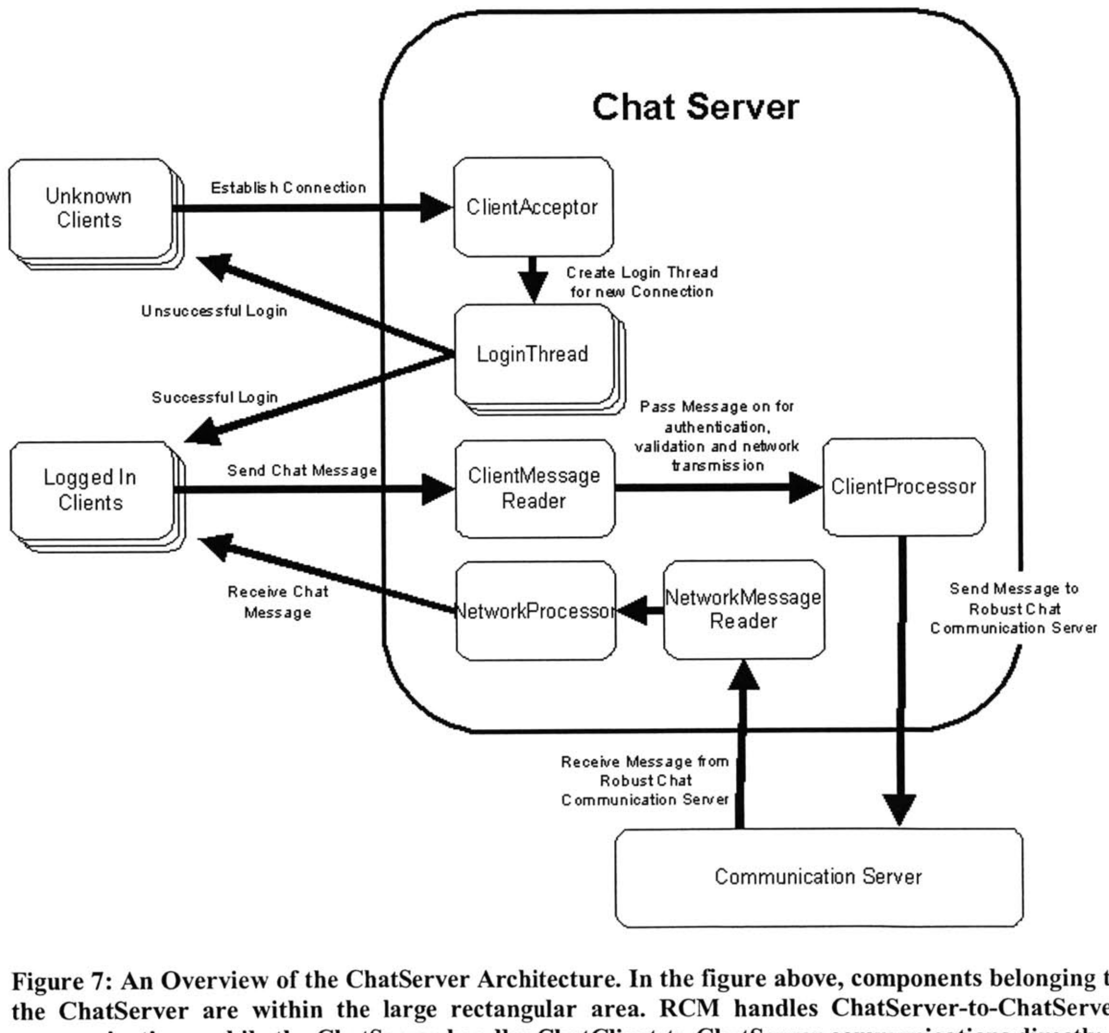


**Figure 7: An Overview of the ChatServer Architecture. In the figure above, components belonging to the ChatServer are within the large rectangular area. RCM handles ChatServer-to-ChatServer communications, while the ChatServer handles ChatClient-to-ChatServer communications directly.**

### 5.1.1 Component Overview

This section describes the major components of a ChatServer object. While a chat server may be comprised of many threads, the ChatServer object itself is merely a container to hold those threads and mediate their interactions with each other. Table 2 briefly describes each type of thread.

| **Component Name** | **Description** |
|---|---|
| ClientAcceptor | Accepts connection requests on the client port. Creates LoginThreads for successful connections. |
| LoginThread | Handles the login process for a single connection. Creates Clients for ClientMessageReader after successful login. |
| ClientMessageReader | Receives messages from all chat clients. Passes messages to ClientProcessor. |
| ClientProcessor | Makes sure chat clients have permissions needed to send messages. Passes messages to RCM. |
| NetworkMessageReader | Receives messages from all chat servers. Passes messages to NetworkProcessor. |
| NetworkProcessor | Handles chat room and chat server related messages. Forwards client-related messages to chat clients. |

**Table 2: ChatServer Component Descriptions**

### 5.1.2 Accepting Clients

The process of logging in a new client is pictured at the top of Figure 7. When an unknown client tries to connect to the ChatServer, the ClientAcceptor accepts the new connection and creates a new LoginThread to handle the remainder of the login process. During the login process the LoginThread receives all the messages the client sends, but does not forward any of them. This prevents the server from accidentally sending out a message from a client that is not logged in. The login process has two possible outcomes:

- If the LoginThread receives a valid LoginMessage before the login timeout expires, the login will be considered successful and the LoginThread will add its client to the list of logged in clients. After the LoginThread adds the new client, the ClientMessageReader receives any subsequent messages the client sends.
- In case an invalid LoginMessage is received or the timeout expires the LoginThread severs the connection to its client. The server does not save any information about

connections that result in failed logins, so those clients are effectively returned to the pool of unknown clients.

In either situation, the LoginThread will terminate upon completion of the login process. At any time there may be multiple LoginThreads in a ChatServer, each corresponding to a different client attempting to log in.

### *5.1.3 Handling Messages from Clients*

Handling messages from clients is pictured in the middle of Figure 7. When a client sends a message to the chat server, the ClientMessageReader receives it. The ClientMessageReader is the only thread that reads messages from logged-in clients. It accomplishes its task by using SocketChannels and Selectors from the Java™ New Input/Output (NIO) package. The objects work together to allow asynchronous communication. SocketChannels can perform non-blocking read and write operations and Selectors can "select" the SocketChannels that are ready to perform operations. In order to be selected a SocketChannel must first be registered with a Selector. During the registration process, the SocketChannel must specify which operations it wants to register for.

When a client has successfully logged in the ClientMessageReader registers the clients SocketChannel with the ClientMessageReader's Selector. The ClientMessageReader registers the SocketChannels of logged-in clients for read operations, which means that the selector will only choose channels that have messages that need to be read. While it waits for messages from clients to arrive, the ClientMessageReader performs a blocking select operation on its Selector. This causes the ClientMessageReader to block until one or more of the logged-in clients send messages. Once messages are available, the ClientMessageReader reads all the messages from the list of channels the Selector returns and then passes those messages on to the ClientProcessor.

The alternatives to reading client messages this way are to poll the logged in clients or to use multiple threads. Reading messages by polling involves cycling through the list of channels and checking whether each channel has a message available for reading. This process is very wasteful, since most of the channels polled will not have messages available for reading, but the ClientMessageReader would have to continually poll the channels to avoid missing a message. Spawning one thread to handle each client connection would simplify the process of reading messages from clients, however, it could also cause an explosion in the number of threads in a chat server. In the current implementation LoginThreads are the only type of thread that may occur more than once in a chat server. This simplifies the login process and since the login process is very short the number of LoginThreads is always low. However, the same is not true for the message reading process, which last as long as a client is logged in. Dedicating one thread to reading messages from a single client could easily triple or quadruple the number of threads in a chat server in a very short period of time. The overhead required to manage so many threads would slow the server down and make it unscalable. Performing reads with one thread (the ClientMessageReader) is clearly more efficient than either alternative.

The ClientProcessor checks client permissions before sending the message to the network. Checking client permissions refers to the process of making sure the client that originated a message has the correct permissions it needs to send that type of message. If, for example, a client who is not a member of a particular chat room attempts to send a message to that room, the ClientProcessor would reject the message based on that information. The chat server attempts to reject unauthorized messages, before sending them to the network, to minimize the cost of transmitting and ordering messages that will ultimately be rejected.

After checking the clients' permissions, the ClientProcessor sends messages to RCM for transmission to all relevant chat servers. These are chat servers that need to receive the message in question. There are certain messages sent by the system, which must be sent to all connected chat servers. In that case every server needs to see the message. However, this is not necessarily the case for messages sent by users to chat rooms. For example, text messages for a chat room are only sent to those sites that have a client who is a member of that room. This reduces the site-to-site communications, which are the weakest link in the MC2C environment.

### 5.1.4 Handling Messages from the Network

Handling messages from the network is pictured at the bottom of Figure 7. When the chat server receives a message from RCM, the NetworkMessageReader is the receiving thread. After receiving a message the NetworkMessageReader will make sure the message's sequence number is correct, which indicates whether the message is correctly ordered compared to other messages from the same site. After checking the NetworkMessageReader is done it passes the message on to the NetworkProcessor.

The NetworkProcessor handles most chat room and client related tasks in the chat server. Although some tasks, such as adding a new client or closing a disconnected client, may be started by other threads the process is always completed by the NetworkProcessor. The reason the NetworkProcessor handles many of these tasks is to avoid race situations between servers. Messages sent through RCM are guaranteed to have an absolute ordering. This means even if two different servers produce conflicting requests at exactly the same time, the system will order them such that it is clear to both servers which of the two requests should be honored and which should be rejected.

Since the NetworkProcessor completes many of the tasks in the chat server it also sends the most messages to chat clients. The NetworkProcessor originates messages notifying clients of new users, disconnected users, new rooms, and closed rooms. If the NetworkProcessor does not recognize a message type it simply forwards it to the chat clients. The NetworkProcessor behaves this way because it makes the system more extensible. If the NetworkProcessor does not recognize a message it assumes the client application added a new message, which it should know how to handle. For a detailed discussion of this mechanism see section 5.2.2.

## 5.2 CHATMESSAGES

In our system the abstraction of a chat message (some information that one user wishes to share with other users) is embodied by the ChatMessage object. The following sections describe the structure and purpose of these messages.

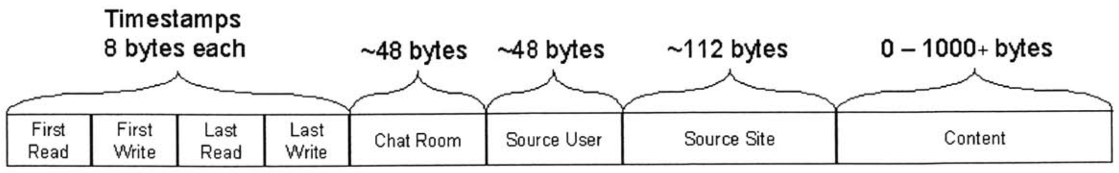


**Figure 8: The Basic Structure of a ChatMessage**

## *5.2.1 ChatMessage*

Figure 8 shows the basic structure of a ChatMessage. Any message sent by the chat application is derived from the ChatMessage object. This uniformity greatly increases the flexibility of the entire system, since any object derived from ChatMessage can be sent through the system. This flexibility is explored in more detail in section 5.2.2. The fields for a ChatMessage are described in Table 3.

| **Field** | **Description** |
|---|---|
| First Read | System time when the message was first deserialized |
| First Write | System time when the message was first serialized |
| Last Read | System time when the message was most recently deserialized |
| Last Write | System time when the message was most recently serialized |
| Chat Room | Name of this message's destination chat room. The originator of the message (either a chat client or chat server) must set this field |
| Source User | Name of the user who sent this message. If the message originated from a client, this field must be set. It is null for any message originating from a server |
| Source Site | Site information for the server that sent this message. The first site to send this message over an inter-site link must set this field |
| Content | The content of this message. Must be set when creating the message |

**Table 3: ChatMessage Field Descriptions.**

The first four fields in a ChatMessage are timestamps that are set during the transmission process. The client-server connections used to send ChatMessages are implemented using SocketChannel objects from the Java™ NIO package. The process of writing a message to a SocketChannel serializes the ChatMessage object. Serializing an object breaks it into its component data types, so it can be easily stored or transmitted. The first time this happens, the ChatMessage's *Write* timestamps are updated to reflect the time at which the object was serialized. Subsequent writes only cause the *Last Write* field to update. Conversely, the process of reading a message from a SocketChannel deserializes the ChatMessage object. This reassembles the component data types into a whole ChatMessage. When a message is first read from a SocketChannel it updates both of the *Read* timestamps to match the time at which the message is read. Subsequent reads only cause the *Last Read* field to update. None of the timestamps are used to order chat messages, however, they provide a useful tool for analyzing the message flow through the chat application.

The next three fields in a ChatMessage provide the application with information used to process the message. The ChatRoom field holds the name of the destination chat room for a ChatMessage. The ChatServer and ChatClient objects use this field to route the ChatMessage to the correct servers and clients. The *Source User* field holds the name of the user sending the message, if the message originated from a chat client. If the message originated from a chat server the *Source User* field will be null. Although many of the message types sent by clients require a *Source User* for the chat server to process them correctly (such as Login and Logoff messages), none of the messages sent by the server have this requirement. When a chat server receives a chat message from another server, the server can process the message based on the message type alone. The *Source Site* field holds the name and address of the server that sent this message. Chat servers use this information to identify the source of server-server communications. This field also allows the server to distinguish between messages from users with the same name who are connected to different servers.

The final ChatMessage field is the *Content* field. As the name implies, the *Content* field holds the cargo of a ChatMessage. A message can hold any serializable Java object. This is another feature, which makes ChatMessages very flexible.

In the current implementation the size of an average message is several hundred bytes. This is a very manageable size for local networks, however, for certain types of inter-site connections the message sizes can cause significant slowdown or even prevent transmission. We discuss the ramifications of message size further in chapters 6 and 7.

### *5.2.2 Message Specialization*

There are 14 types of messages in the current implementation of the chat application. The majority of these messages are server-to-client messages which, inform the clients of status changes, and server-to-server messages, which negotiate new connections and status changes with other servers. The fact that all these messages are derived from the ChatMessage object means that they can all use the same communication infrastructure designed to handle ChatMessages. This simplifies the task of handling different types of messages.

Another benefit of this common derivation is the possibility of extending the application with new message types, should this become necessary. The application can successfully transmit any new message type as long as that type is derived from the ChatMessage object. The UserWritingMessage is a good example of the application's capabilities. In essence the UserWritingMessage is nothing more than a ChatMessage where the content is a Boolean value specifying whether the originating user is currently writing or not. What makes this message type a good example is that the ChatServer and ChatClient objects know nothing about it. As long as all the ChatMessage fields are set correctly the message will arrive at all the correct destination clients, who are tasked with knowing what to do with the message. To see what the UserWritingMessage is used for, see section 5.4.3.

## 5.3 CHATCLIENT

This section describes the components and processes that govern the behavior of the ChatClient.

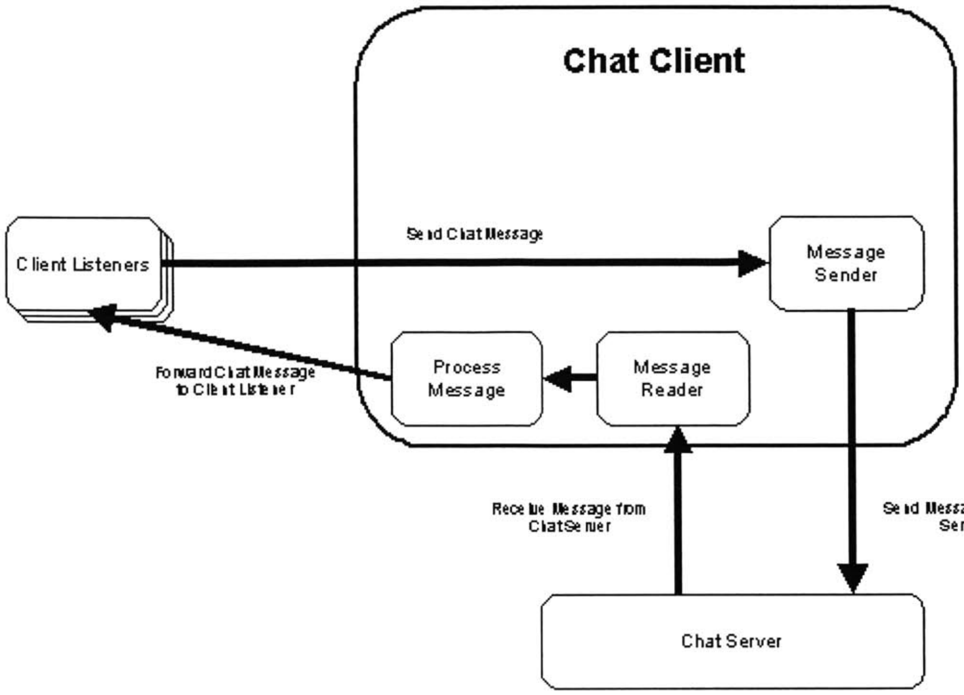


**Figure 9: An Overview of the ChatClient Architecture. In the figure above, components belonging to the ChatClient are within the large rectangular area.**

### *5.3.1 Component Overview*

Table 4 describes the components of the ChatClient object.

| **Component Name** | **Description** |
|---|---|
| MessageReader | Reads messages from the chat server and places them on the inQueue |
| MessageSender | Takes messages from the outQueue and sends them to the chat server |
| ChatClient | Takes messages from the inQueue and processes them. This usually entails forwarding a message to a specific ClientListener |
| ClientListener | Implements the interface that allows objects to use the ChatClient to send ChatMessages to the chat server |

**Table 4: ChatClient Component Descriptions.**

### 5.3.2 *ClientListeners*

ClientListeners control the actions of the ChatClient. The ChatClient manages network communications with the ChatServer, allowing the ClientListeners to function without detailed knowledge of the network environment they are operating in. In order to function properly the ClientListeners need only register with the ChatClient, by calling the ChatClient's addListener() method, and implement the four ClientListener methods: onMessageReceived(), getChatRoom(), onConnect(), and onDisconnect().

When a ClientListener registers with a ChatClient, the ChatClient will call getChatRoom() to determine the name of the chat room the ClientListener is interested in listening to. The onConnect() and onDisconnect() methods are callback methods to notify the listener of a disconnect between the ChatClient and the ChatServer. The onMessageReceived() method is a callback method the ChatClient will invoke when it receives a message for the chat room the ClientListener is registered for.

A ClientListener can be thought of as the inteface to a particular chat room. Although the current implementation of the chat application only allows textual chat rooms to exist, the ClientListener interface is flexible enough, such that any application that can send information in chunks can use the system. We discuss a few of the possible applications in Chapter 7.

### 5.3.3 *Handling Messages*

When a ClientListener is interested in sending a message to the chat room it is registered for it will invoke the sendMessage() method on the ChatClient. This places the ClientListener's message on the outQueue, where the MessageSender will eventually remove it from the queue and send it on to the ChatServer.

In the opposite direction, when a message arrives at the ChatClient, the MessageReader will place the new message on the inQueue for processing. When the ChatClient thread processes a message it searches for a ClientListener that has registered an interested in the message's destination chat room. If the ChatClient finds an appropriate ClientListener it invokes that ClientListener's onMessageReceived() method with the message as an argument.

## 5.4 ClientGUI

The ClientGUI package provides the user with an interface for the chat application. The three major components of the graphical interface are the ChatManager, ChatRoom and UserPanel objects. The ChatManager controls the login process for clients and displays the status of the chat application. The ChatRoom provides a simple graphical interface for textual chat. Finally, the UserPanel displays status information for sites and users, which enables features such as site awareness and connection awareness. The following sections describe each object in greater detail.

### *5.4.1 ChatManager*

The ChatManager (Figure 10) is the control panel for the robust chat application. When a user first starts the chat application the ChatManager will appear and both the Room Panel and the User Panel will be blank. The ChatManager title will simple say "Chat Manager."

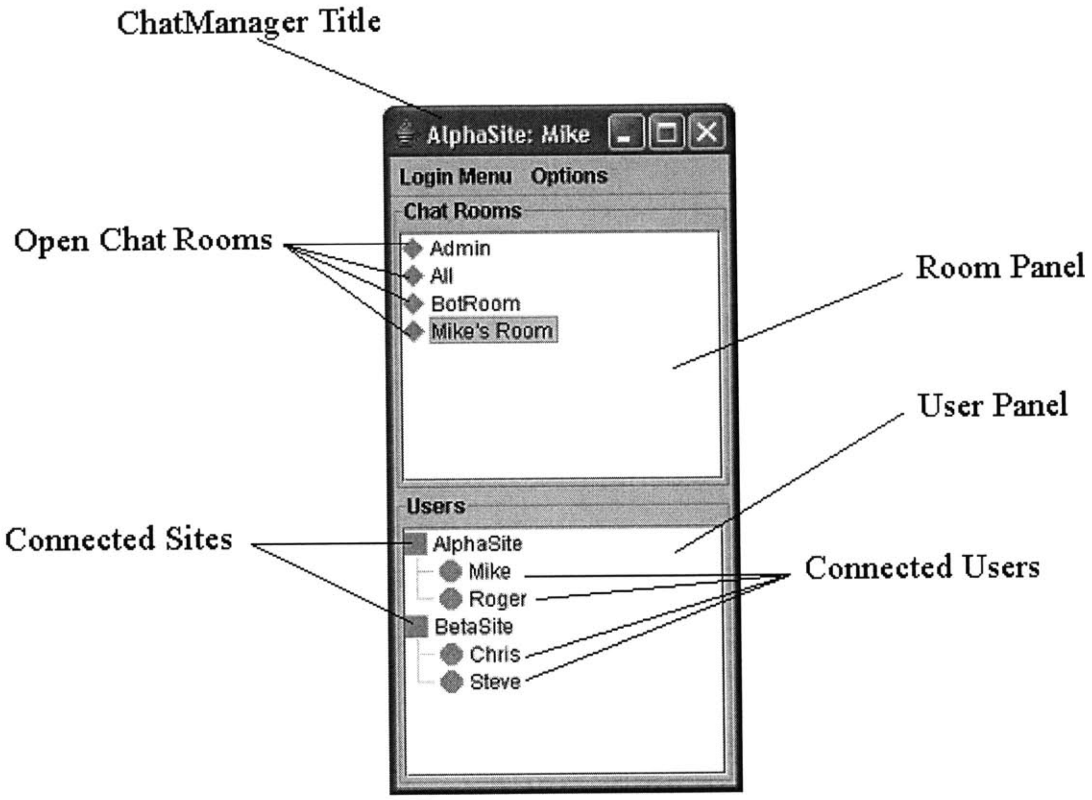
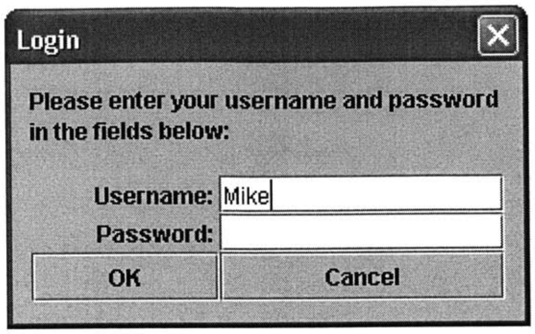


**Figure 10: ChatManager Interface**

The "Options" menu allows the user to configure the appearance of the ChatManager and to create new chat rooms. By default the ChatManager always shows both the Room Panel and the User Panel. By deselecting "Show Rooms" or "Show Users" the user can remove either the Room Panel or the User Panel, respectively. Removing one of the panels can make viewing a long list of rooms or users easier. Selecting "Start new Chat Room" from the "Options" menu will instruct the ChatManager to request that the ChatServer create a new chat room. This option is only available if the user is already logged in. If the ChatServer successfully creates the new chat room the Room Panel will update to include the new room and a ChatRoom window will appear for the requested room.

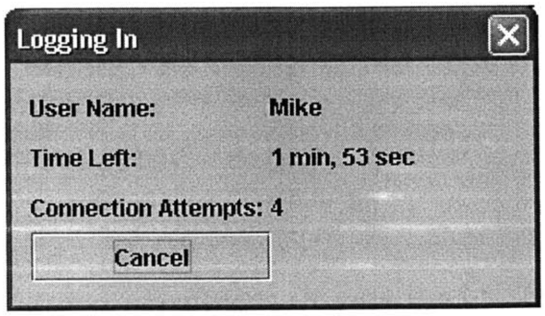


Please enter your username and password in the fields below:

Username: Mike
Password:

**Figure 11: LoginDialog. Prompts the user for their username and password.**

In the "Login Menu" the user may choose to "Log in" or "Log off." Logging in is only possible when not logged in and conversely logging off is only possible when logged in. Choosing the "Log in" menu item will cause a LoginDialog (Figure 11) to pop up, prompting the user for their username and password. After the user enters their information and clicks "OK" the ChatManager opens a ConnectorDisplay (Figure 12) to complete the log in. The ConnectorDisplay repeatedly calls the ChatClient's login() method until either the login completes successfully or the "Time Left" runs out. It also displays how many connection attempts have been made, so the user knows that application hasn't crashed. Selecting "Log off" will disconnect the client from the ChatServer and clear the Room Panel and User Panel.



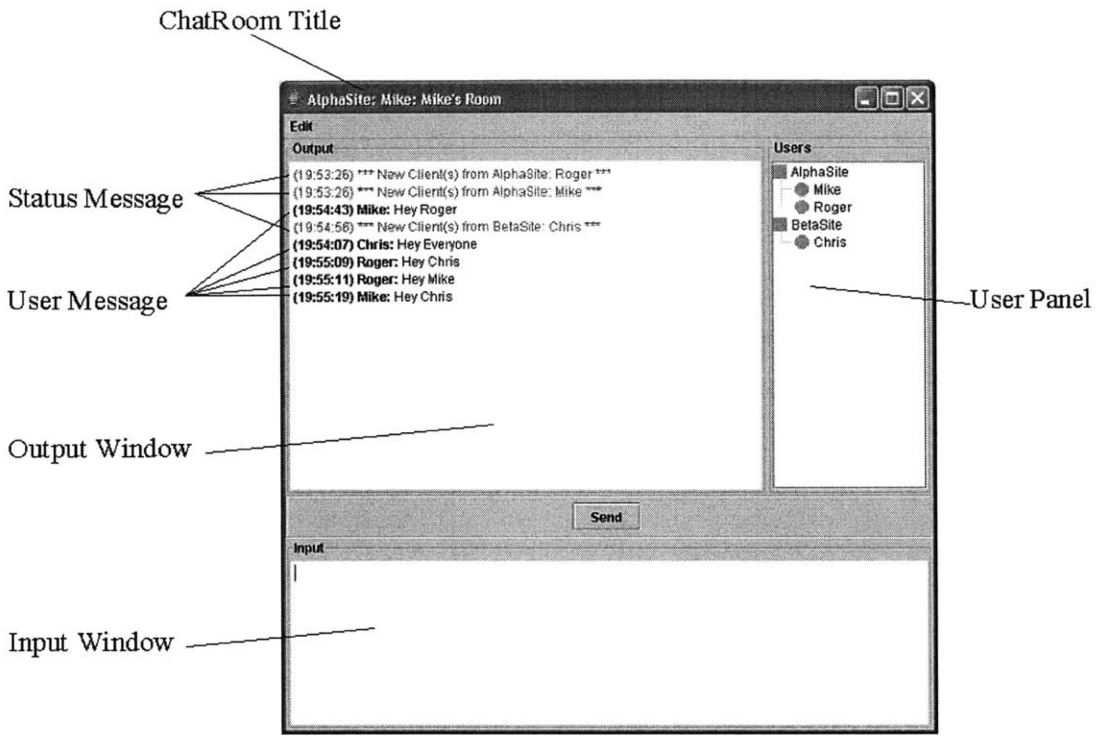
User Name: Mike
Time Left: 1 min, 53 sec
Connection Attempts: 4

**Figure 12: ConnectorDisplay. Attempts to connect to the ChatServer until the Time Left runs out.**

Successfully connecting causes the ChatServer to send a list of available chat rooms and connected users to the ChatClient. The list of available chat rooms will appear in the Room Panel, while the list of connected users will appear in the User Panel. Section 5.4.3 describes the User Panel in more detail. A successful log in will also cause the ChatManager's title to change from "Chat Manager" to "SiteID: Username," where SiteID is the name of the ChatServer that the client is connected to and Username is the name of the user who just logged in. Double clicking on a chat room in the Room Panel will cause the ChatManager to request that the ChatServer let the currently logged in user join the specified chat room. Successfully joining the requested chat room will cause a ChatRoom window to appear for the requested room.

### *5.4.2 ChatRoom*

The ChatRoom GUI allows a user to communicate textually with other users who are in the same chat room. The ChatRoom interface is designed to be simple, yet informative. The title of the ChatRoom window is in the format "SiteID: Username: Roomname," where SiteID is the name of the ChatServer the client is connected to, Username is the name of the user currently logged in, and Roomname is the name of the chat room this ChatRoom interface is connected to. The top left window in the ChatRoom interface is the OutputWindow where messages are displayed. To the right of the Output Window is the UserPanel, which displays the current chat room membership by site and username. For more information on the UserPanel, see section 5.4.3. Below both of these windows is the Send button and immediately below the Send button is the Input Window, where the user can enter input to send to all the users in the chat room. The Edit menu at the top left of the ChatRoom interface has a few of the basic text editor functions, which allow the user to cut, copy, paste, or select all the text from the various windows in the interface (cutting and pasting are only allowed in the Input Window).



```
(19:53:26) *** New Client(s) from AlphaSite: Roger ***
(19:53:26) *** New Client(s) from AlphaSite: Mike ***
(19:54:43) Mike: Hey Roger
(19:54:56) *** New Client(s) from BetaSite: Chris ***
(19:54:07) Chris: Hey Everyone
(19:55:09) Roger: Hey Chris
(19:55:11) Roger: Hey Mike
(19:55:19) Mike: Hey Chris
```

**Figure 13: ChatRoom Interface**

The Output Window displays text messages sent to this chat room. Every displayed message has a timestamp associated with it. The timestamp allows the user to gauge when the message was created. Although this can be helpful in determining the general flow of the conversation it doesn't necessarily help determine the ordering of messages, since the timestamps are set by the originating computer, which may not have the same clock value as the user's local computer.

There are two types of outputs in the Output Window: status messages and user messages. As Figure 13 shows, status messages are displayed in a bright red color to draw attention and distinguish them from user messages, which are displayed in the default black color. As their name implies, status messages are notifications to the user about changes in the status of the chat room, whereas user messages are text messages sent from other users.

There are currently four different status messages that the ChatRoom interface can display. Two are status messages to notify the user of another user joining or leaving the chat room. The other two notify the user that the ChatClient this ChatRoom is connected to has disconnected or reconnected to the ChatServer.

User messages are the result of user input. If a user enters text into the Input Window and subsequently clicks the Send button or presses enter, the text from the Input Window will be sent to all other users currently connected to the same chat room. When a ChatRoom receives a text message from a user it outputs the message in the format "***(Timestamp)* *Username*:** *text*." The *Timestamp* is system time on the originating computer, at which the message was created. *Username* is the name of the user who sent the message and *text* is whatever the user input and sent from his Input Window.

### 5.4.3 UserPanel

Figure 14 depicts a typical UserPanel. The UserPanel displays membership information for the Frame it resides in. If the UserPanel is in a ChatManager, it displays the membership information for the sites the user is connected. If the UserPanel is in a ChatRoom interface, it displays the membership information for the chat room.

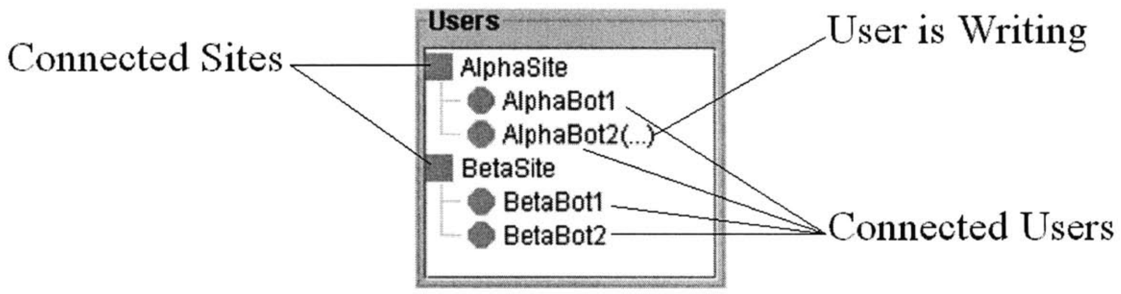


**Figure 14: The UserPanel Display.**

The ordering of users in the UserPanel is lexicographical by site and then lexicographical by the users within each site. This ordering scheme ensures that, when the same users connect from the same sites, the UserPanel displays them in the same way every time, regardless of the order in which the sites or users connected.

When residing in a ChatRoom interface, the UserPanel does more than just display the chat room membership information: It also displays user writing status. During the initial deployment of the chat application, users commented that it would be nice to have an indicator in the application that would let users know when other users were in the process of entering a message. As a result we added the UserWritingMessage: When a user starts entering text into the Input window of a ChatRoom interface, the ChatClient generates a UserWritingMessage, which is sent to all the members of that chat room. The user is considered to be writing if there is text in the Input Window. By removing all the text in the input window, either by sending the input or deleting it, the user is no longer considered to be writing, which generates another UserWritingMessage. When a ChatRoom interface receives a UserWritingMessage that indicates a user is writing, it places three ellipses in parenthesis next to the user's name. If it then receives another UserWritingMessage indicating that the user is no longer writing, it removes the ellipses and parenthesis from the users name. Adding the UserWritingMessage to the application took minimal effort to implement because of the flexibility inherent in the chat application architecture (see section 5.2.2).

If the ChatServer notifies the client that a certain site is suspected of being disconnected, the UserPanel will signal this state to the user by shading that site and all its users red. If the suspected site becomes well connected again, the UserPanel will restore the site and its users' original colors. Otherwise, the site will eventually be considered disconnected and the UserPanel will remove the site and its users from the display.

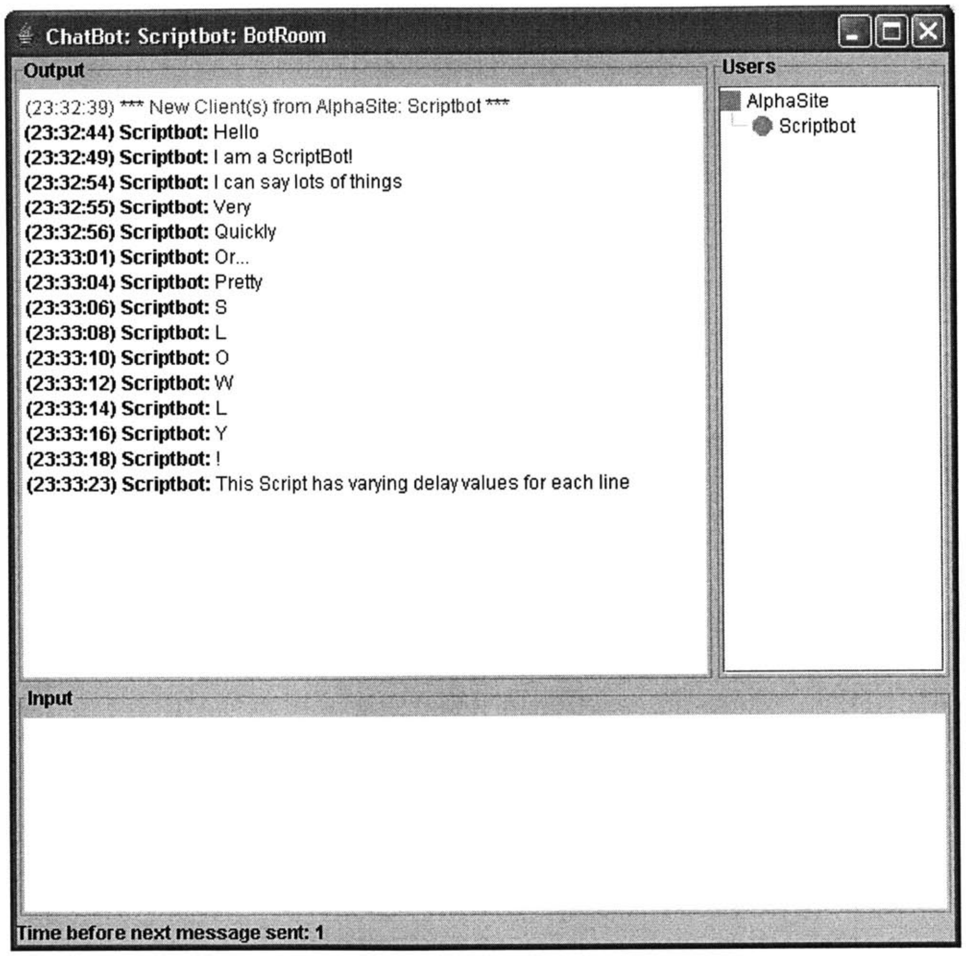
## 5.5 ChatBot

This section describes the ChatBot package. The package consists of the two chat bots written to help test the chat application and the BotDisplay, which outputs the actions of the chat bots. The bots test the chat application by sending messages to each other in a pattern that is meant to mimic rapid conversation between humans. In essence, the chat bots are small programs that use our chat application to communicate, just as users would.

In addition to being fully independent (capable of operating and testing the application without human intervention) the chat bots are also able to interact with humans. The bots can be configured to either ignore human user messages or respond to them the same way they respond to messages from other chat bots.

### *5.5.1 ScriptBot*

The idea for the ScriptBot was born of the desire to have a bot that could emulate human conversation. In order to come as close as possible to achieving this goal, we designed a chat bot that could read in a scripted dialog and send it to a specific chat room. The ScriptBot is capable of sending messages of varying lengths and at varying intervals, making it very adept at mimicking human response times (see Figure 15).



```
(23:32:39) *** New Client(s) from AlphaSite: Scriptbot ***
(23:32:44) Scriptbot: Hello
(23:32:49) Scriptbot: I am a ScriptBot!
(23:32:54) Scriptbot: I can say lots of things
(23:32:55) Scriptbot: Very
(23:32:56) Scriptbot: Quickly
(23:33:01) Scriptbot: Or...
(23:33:04) Scriptbot: Pretty
(23:33:06) Scriptbot: S
(23:33:08) Scriptbot: L
(23:33:10) Scriptbot: O
(23:33:12) Scriptbot: W
(23:33:14) Scriptbot: L
(23:33:16) Scriptbot: Y
(23:33:18) Scriptbot: !
(23:33:23) Scriptbot: This Script has varying delay values for each line
```

Time before next message sent: 1

**Figure 15: ScriptBot Output.**

### *5.5.2 ResponseBot*

The ResponseBot is a bot that alters its conversational speed depending on how many users it is talking to. It has the ability to send response messages in reply to messages originating from other users. When a ResponseBot receives a message from another user it will send a response message with probability p. If no response is sent the bot will take no action regarding the received message. To keep the ResponseBot's responses at a reasonable rate, p is initially set by taking the reciprocal of the number of users in the bot's chat room. After p is initially set, the ResponseBot will adjust it over time to achieve a desired response rate based on its *bored timeout* value.

The *bored timeout* value, t, is an input parameter that specifies how long it takes the bot to become "bored." If a ResponseBot does not send a message to the chat room before the timeout runs out it becomes bored and sends a boredom message to the chat room (see Figure 16). It then increases the value of p and resets the timeout. Alternately, if the bot sends a response message to the chat room before the timeout runs out, it decreases the value of p and resets the timeout. The rate of increase or decrease when adjusting p is governed by the *adjustment rate*, r. To adjust the value of p we either multiply it by r, in case p is decreasing, or divide by r, in case p is increasing.

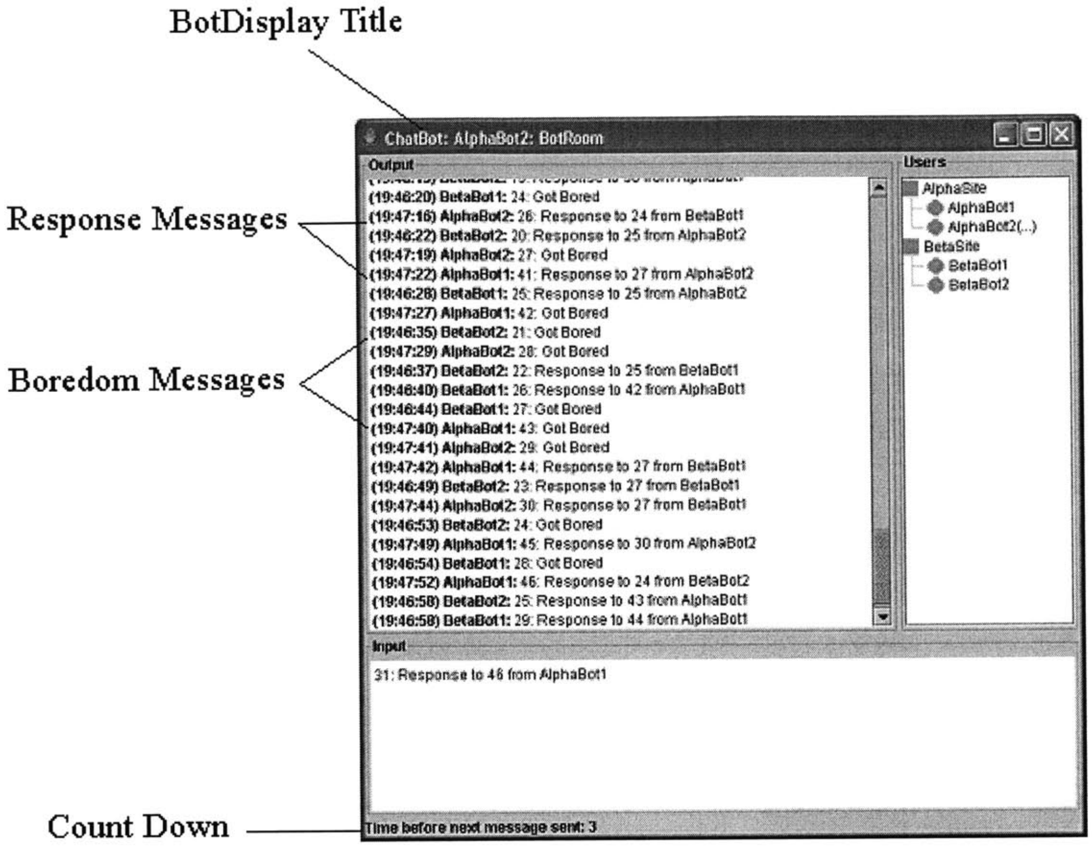
Another parameter that controls the ResponseBot's actions is called the *write delay*, d, and it determines the maximum time the bot will wait before sending a message. For example, if d is 5000 milliseconds, the bot will wait between 0 and 5000 milliseconds before sending any messages. The actual wait time is chosen from a uniform distribution of the possible values, which means the average delay will actually be 2500 milliseconds, or 2.5 seconds. The purpose of the *write delay* is to make the bot's responses slower and more randomly timed, just as human conversation would be.

To better understand the behavior of the ResponseBot we will work through a theoretical example. Assume that $r = 0.9$, $d = 5000$ ms, and $t = 10000$ ms. If the ResponseBot joins a chat room that initially has 10 members (including the bot) it will set $p = 1/10 = 0.1$. Each message the bot receives now has a 0.1 probability of producing a response message. If none of the messages the bot receives produces a response before it becomes bored (after 10 seconds), the bot will send a boredom message to the chat room and adjust p, such that $p = p / r = 0.1 / 0.9 = 1/9 \approx 0.111$. Now that the response probability is higher, the next message produces a response. The bot chooses a delay between 0 and d to make the response seem more natural and then sends the response message to the chat room. It then adjust the value of p, such that $p = p * r = 1/9 * 0.9 = 0.1$.

The initial value for p is not the equilibrium response rate for the ResponseBot, which is dependent on the rate of conversation in the chat room and the value of t. In equilibrium, p will be stable, so we can assume that the bot will send one boredom message for every response message it sends. Another way of stating this is to say that in equilibrium (probability of a reply, p) = (probability of becoming bored). If we assume that the rate of messages from other users is u, we can determine the equilibrium value of p. A message rate of u implies an average of u*t messages will reach the bot before it becomes bored. The probability of becoming bored is equal to the probability that the bot does not send a response to any of the messages it receives before it becomes bored. Thus, (probability of becoming bored) = $(1-p)^{(u*t)}$. Out equation now becomes:
$p = (1- p)^{(u*t)}$. We can simplify this equation further, to $p + p^{1/(u*t)} = 1$. At a message rate, u = 1/1000ms, and a *bored timeout*, t = 10000ms, the equation becomes $p + p^{1/10} = 1$, which is solved by the value p = 0.164917. This is equilibrium at which the ResponseBot will eventually settle if the conversation rate remains constant. Note, that since the equilibrium rate depends on the rate of conversation, it is continuously changing and impossible to set a priori without full knowledge of how the conversation will develop.

### 5.5.3 BotDisplay

The BotDisplay allows a user to monitor the actions of a chat bot. Figure 16 shows typical output from a group of ResponseBots. The BotDisplay is very similar to the ChatRoom interface, except it is purely a display object. It does not allow input from the user in any of the windows and the “Edit” menu and Send button have been removed. The title of a BotDisplay is formatted slightly differently from the title of a ChatRoom interface. The BotDisplay is in the format ChatBot: *Botname*: *Roomname*, where *Botname* is the name of the chat bot and *Roomname* is the name of the chat room the bot is currently in. The *Sitename* has been replaced by the word “ChatBot” to make it immediately recognizable that the window belongs to a chat bot.



```
(19:46:20) BetaBot1: 24: Got Bored
(19:47:16) AlphaBot2: 26: Response to 24 from BetaBot1
(19:46:22) BetaBot2: 20: Response to 25 from AlphaBot2
(19:47:19) AlphaBot2: 27: Got Bored
(19:47:22) AlphaBot1: 41: Response to 27 from AlphaBot2
(19:46:28) BetaBot1: 25: Response to 25 from AlphaBot2
(19:47:27) AlphaBot1: 42: Got Bored
(19:46:35) BetaBot2: 21: Got Bored
(19:47:29) AlphaBot2: 28: Got Bored
(19:46:37) BetaBot2: 22: Response to 25 from BetaBot1
(19:46:40) BetaBot1: 26: Response to 42 from AlphaBot1
(19:46:44) BetaBot1: 27: Got Bored
(19:47:40) AlphaBot1: 43: Got Bored
(19:47:41) AlphaBot2: 29: Got Bored
(19:47:42) AlphaBot1: 44: Response to 27 from BetaBot1
(19:46:49) BetaBot2: 23: Response to 27 from BetaBot1
(19:47:44) AlphaBot2: 30: Response to 27 from BetaBot1
(19:46:53) BetaBot2: 24: Got Bored
(19:47:49) AlphaBot1: 45: Response to 30 from AlphaBot2
(19:46:54) BetaBot1: 28: Got Bored
(19:47:52) AlphaBot1: 46: Response to 24 from BetaBot2
(19:46:58) BetaBot2: 25: Response to 43 from AlphaBot1
(19:46:58) BetaBot1: 29: Response to 44 from AlphaBot1
```

```
31: Response to 46 from AlphaBot1
```

```
Time before next message sent: 3
```

**Figure 16: The BotDisplay Interface.**

There is one addition to the BotDisplay, which is not present in normal ChatRoom interfaces. In the bottom of Figure 16 there is a count down message that indicates how long before this chat bot sends its next message. This count down can be very useful if a ResponseBot is set to respond very slowly or a ScriptBot has high delay values in its script.

Implementing the BotDisplay took little effort because it uses the same graphical components as the ChatRoom interface. This makes the BotDisplay easy to understand for users who know how the ChatRoom interface works and allows the BotDisplay to inherit improvements to the ChatRoom interface. The BotDisplay was actually developed before the UserWritingMessage was added to the application (see section 5.4.3), however, since the BotDisplay uses the same UserPanel as the ChatRoom interface, it automatically inherited the new functionality.

## 5.6 DataLogger

A DataLogger is an object, which implements the DataLogger interface. As its name implies, the DataLogger package is designed to record information about the chat server. Specifically, a DataLogger is supposed to collect useful runtime information about chat servers to help analyze behavioral patterns and problems. A DataLogger acts as an add-on to the chat server and requires that the server pass it the information it needs. To simplify the logging process and make them more modular, all DataLoggers take in the same four types of information. The four information types are physical connectivity status, logical connectivity status, sent messages and received messages, which are input by the chat server. This means exchanging DataLoggers may only require changing the type of the logger.

Although every DataLogger must take the same inputs, they are not required to output the same information. We chose to implement three different DataLoggers that demonstrate this class's flexibility. The RawDataLogger takes the input information and simply outputs the information into a formatted text file. The AnalysisDataLogger uses the inputs to calculate interesting information about the application. Finally, the MultiLogger wraps the abilities of the previous two DataLoggers into a single class.

The following three sections describe the three types of DataLogger in more detail.

### 5.6.1 RawDataLogger

The RawDataLogger records basic information about a chat server's activities. The input data is formatted in a standard way and then output to a log file with the .DAT extension. Every line produced by the RawDataLogger has the following format:

```
{timestamp}DELIM{thisServer}DELIM{otherServer}DELIM{dataType}DELIM{other}
```

The `{timestamp}` indicates the time at which this data is collected. `DELIM` is a placeholder for the delimiter in the log files. This delimiter is configurable, but defaults to a single tab character in the current implementation. `{thisServer}` is the name of the server that data is being collected from. `{otherServer}` is the name of the other server for which this data is relevant. In case the information is for a received message, `{otherServer}` will hold the name of the sender. If the information is for a sent message, `{otherServer}` will hold the name of the destination server. `{dataType}` indicates the kind of information a line is holding. Finally, `{other}` represents a list of one or more additional values that may be relevant for the given `{dataType}` value.

The four `{dataType}` values and the additional information that accompanies them is listed in Table 5:

| **{dataType}** | **{other}** |
|---|---|
| S<br>(send) | {seqNum} |
| R<br>(receive) | {seqNum}DELIM{clientSend}DELIM{serverSend}DELIM<br>{serverReceive} |
| P<br>(physical status) | {status} |
| L<br>(logical status) | {status} |

**Table 5: RawDataLogger Data Types.**

`{seqNum} is the sequence number` of this message from the originator's perspective. If `{seqNum}` is N then the originating server sent N-1 messages before this one. `{clientSend}` is the timestamp for when the originating server first sent the message. `{serverSend}` and `{serverReceive}` are timestamps indicating when RCM sent and received the message, respectively. If the `{dataType}` is P, then the `{status}` will be an integer representing either a connected or disconnected state. If the `{dataType}` is L, the `{status}` may also be an integer representing the suspected state, in addition to the connected and disconnected states.

### *5.6.2 AnalysisLogger*

The AnalysisLogger collects and logs analytically determined information about the chat server in a .ANL file. It takes in the same information as the RawDataLogger, but it produces four new {dataType} values: PU, LU, FE, and D. Every line produced by the AnalysisLogger has the same format as output from the RawDataLogger (see section 5.6.1). Table 6 lists the six {dataType} values the AnalysisLogger outputs and the associated {other} values:

| **{dataType}** | **{other}** |
|---|---|
| P (physical status) | {status} |
| L (logical status) | {status} |
| PU (Phys. Up %) | {%up} |
| LU (Logical Up %) | {%up} |
| FE (FIFO Error) | {lastIndex}DELIM{receivedIndex} |
| D (Data Rate) | {msgs/sec} |

**Table 6: AnalysisLogger Data Types.**

The {status} values are the same for the AnalysisLogger and RawDataLogger (see section 5.6.1).

When {dataType} is PU, {%up} represents the percentage of time the two servers ({thisServer} and {otherServer}) have been physically connected. Stated mathematically, {%up} = 100% * (how long the physical status has been "connected") / (how long the AnalysisLogger has been running).

When {dataType} is LU, {%up} represents the percentage of time the two servers have been logically connected. Short-term disconnects are considered logically connected. Stated mathematically, {%up} = 100% * [(how long the logical status has been "connected") + (how long the logical status has been "suspected" during short-term disconnects)] / (how long the AnalysisLogger has been running).

When {dataType} is FE, the AnalysisLogger has detected an error in the message sequence numbers, which indicates a possible gap in the messages. {lastIndex} indicates the sequence number for the start of the gap and {receivedIndex} indicates the sequence number for the end of the gap.

When {dataType} is D, {msgs/sec} holds the rate at which the chat server is receiving messages. The AnalysisLogger is configured to sample the data rate at a specified frequency. To calculate the rate, the AnalysisLogger counts the number of messages received between samples and divides by the amount of time that has passed between samples.

### *5.6.3 MultiLogger*

The MultiLogger is a wrapper object. This means it does not add any new functionality of its own, but it makes certain tasks easier. In this case, the MultiLogger makes using the RawDataLogger and AnalysisLogger simpler. The MultiLogger object acts as a container for a RawDataLogger, an AnalysisLogger and a MonitorActiveViewer (see section 5.7.1). The MultiLogger implements the DataLogger interface, so it takes the same input values as the RawDataLogger and AnalysisLogger. However, instead of processing the values itself, the MultiLogger simply passes them to the RawDataLogger and AnalysisLogger it contains. These two loggers can, in turn, be independently activated or inactivated, which allows the MultiLogger to function as either a RawDataLogger (AnalysisLogger inactivated) or an AnalysisLogger (RawDataLogger inactivated) or both (both activated) or neither (both inactivated).

The MultiLogger can also activate a MonitorActiveViewer, which converts the output from one of the loggers into a graphical monitor of the server status. The MonitorActiveViewer will use the log with the most data as its source, so if an AnalysisLogger is running, the .ANL file is used. Otherwise, if the RawDataLogger is the only logger running, the MonitorActiveViewer will use the .DAT file. If neither logger is running the MulitLogger will not start a MonitorActiveViewer, since there is no data source for it.

## 5.7 MONITOR

Monitors graphically display the network activity associated with a single ChatServer. Their primary value is as an analytic tool, to allow the server administrator to easily monitor a server's behavior and pinpoint problems.

We used the JFreeChart package to implement the graphical interface for our Monitors. JFreeChart is a free Java class library for generating charts available for free under the GNU Lesser General Public License (LGPL).

There are two types of Monitors: MonitorActiveViewer and MonitorLogViewer. The MonitorActiveViewer is a real-time monitoring tool that displays information about a chat server during execution. The MonitorLogViewer is a log-viewing tool that can visualize the chat server log. Both types of monitors may display up to six plots at one time (Figure 17). The different plots are described in Table 7. Every plot may contain multiple lines, with each line representing a different site. The lines are color coded, so it is easy to identify information from the same site on different plots. The horizontal axis that is shared by all the plots indicates the time at which data points were obtained. The two monitor types are described in further detail below.

| | |
|---|---|
| Physical Connectivity | The underlying link availability (as presented by the operating system. A value of 0 indicates no link availability, while a value of 1 indicates a working link. |
| Logical Connectivity | The link availability from the chat application's perspective. A value of 0 indicates a disconnected link, a value of 0.5 indicates a suspected link, and a value of 1 indicates a working link. |
| Physical Percent Available | The value at any point on this line is the percentage of time the underlying link has been available, up to that point in time. |
| Logical Percent Available | The value at any point on this line is the percentage of time the logical link has been connected, up to that point in time. The logical link is considered connected any time the link is connected or the link is suspected but returns to being connected without becoming disconnected. |
| FIFO Correctness | A spike on this line indicates an error in the message sequence numbers received by the server. Sequence numbers are expected to increase by 1 for every message received. When this is not the case it is generally the result of lost messages or a server that has restarted. The size and direction of the spike generally indicate the type of error. When a few messages are lost it results in a small positive spike, indicating that the sequence number received is slightly larger than the sequence number expected. A server restart will reset the sequence numbers for that server, resulting in a large negative spike. |
| Message Data Rate | Lines on this plot indicate the number of messages per second the server is receiving. |

**Table 7: Monitor Plot Descriptions.**

## 5.7.1 *MonitorActiveViewer*

The MonitorActiveViewer displays information about an active ChatServer. The MonitorActiveViewer obtains its data points from the output file of a running DataLogger. The display of a MonitorActiveViewer is a moving window of fixed size. In Figure 17, the size of the moving window is 60 seconds. Unlike a MonitorLogViewer, the MonitorActiveViewer will always display all 6 plots.

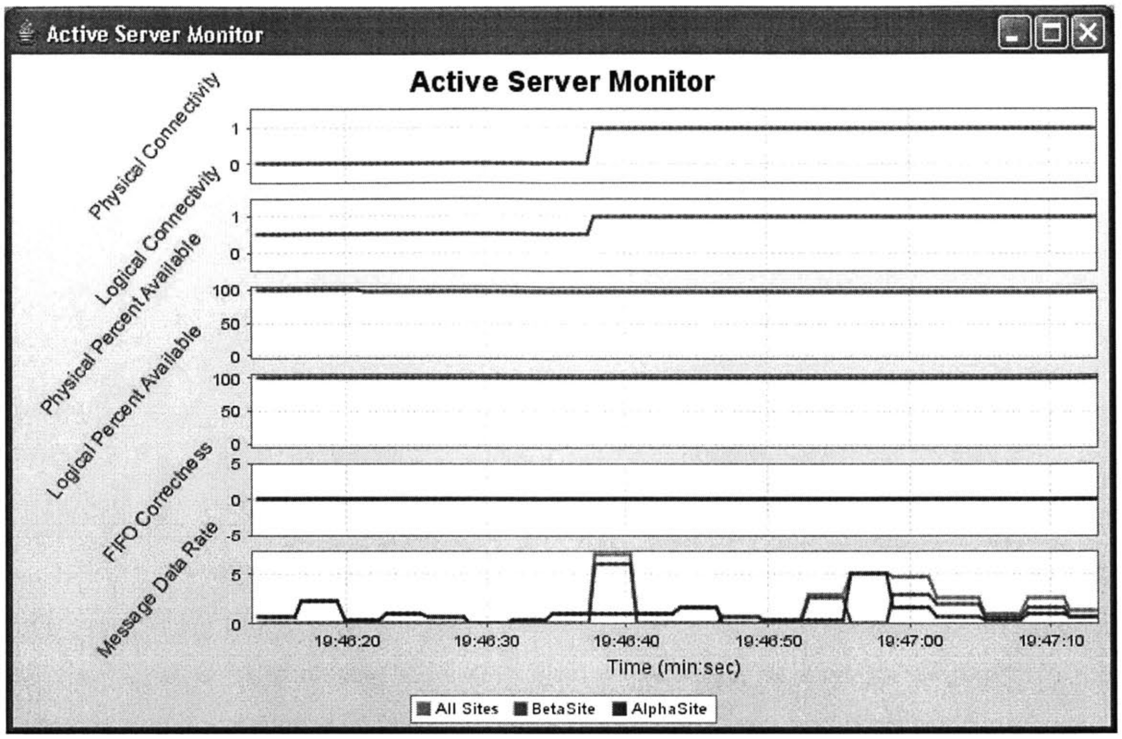


**Figure 17: MonitorActiveViewer.**

The colors in the MonitorActiveViewer are what we call the *Hash Color* for each site. We obtain the *Hash Color* of a site by taking the hash code of the Java™ String object that represents the site's name and converting that integer value into a color. In Java™, integers are 4-byte values, so we convert them into colors by taking the 3 lowest-order bytes and assigning them to be the red, green, and blue values of the *Hash Color*, respectively. Hash codes for Java™ Strings are designed to have low collision likelihood, so the probability that two *Hash Colors* will be the same is also low. Another advantage of using *Hash Colors* is that they don't require any ordering of the sites. Other coloring schemes that could easily be implemented include coloring based on the order the sites join, or coloring based on the alphabetic ordering of the site names. However, coloring based on the order the sites join would result in a different coloring scheme on every execution of the MonitorActiveViewer and coloring based on alphabetic order could require color changes at runtime, which may confuse the user. *Hash Colors* guarantee a uniform coloring scheme across any number of executions and with any number of sites.

### *5.7.2 MonitorLogViewer*

The MonitorLogViewer displays log files produced by a DataLogger after a run has completed (see section 5.6). At the top of the graph, the MonitorLogViewer indicates the name of the file being displayed (See Figure 18). Unlike the MonitorActiveViewer, the time scale in the MonitorLogViewer adjusts to fit the entire data set. This allows users to view in the entire data set before focusing closer on a particular area. The user can magnify an area of the graph by clicking and dragging the mouse of the desired area.

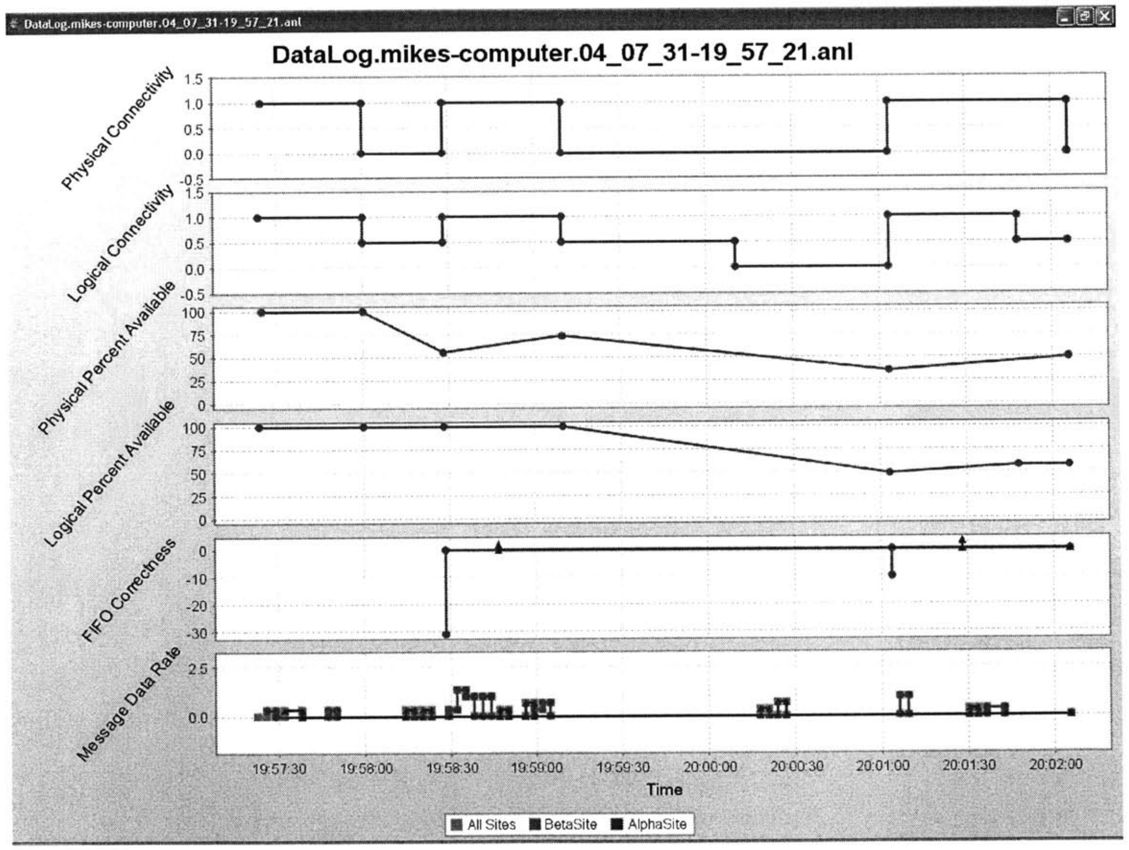


**Figure 18: MonitorLogViewer. View of a .ANL log file, produced by the AnalysisLogger object. The coloring scheme is based on Hash Colors. This graph is based on the same information as Figure 19.**

The user can select the coloring scheme to use for the plots at startup. The user can select *hash colors* or lexicographical colors. The reason for the choice is that, although it is unlikely for two sites to have the exact same color when using *hash colors,* it is fairly common that two sites will have names that are similar enough to produce *hash* *colors* that very closely resemble each other (at JEFX 04, the two chat servers were named "logba" and "logbt." The resulting *hash colors* were two very similar shades of green). Figure 18 displays a data set by using *hash colors*, while Figure 19 displays the same data set using lexicographical coloring.

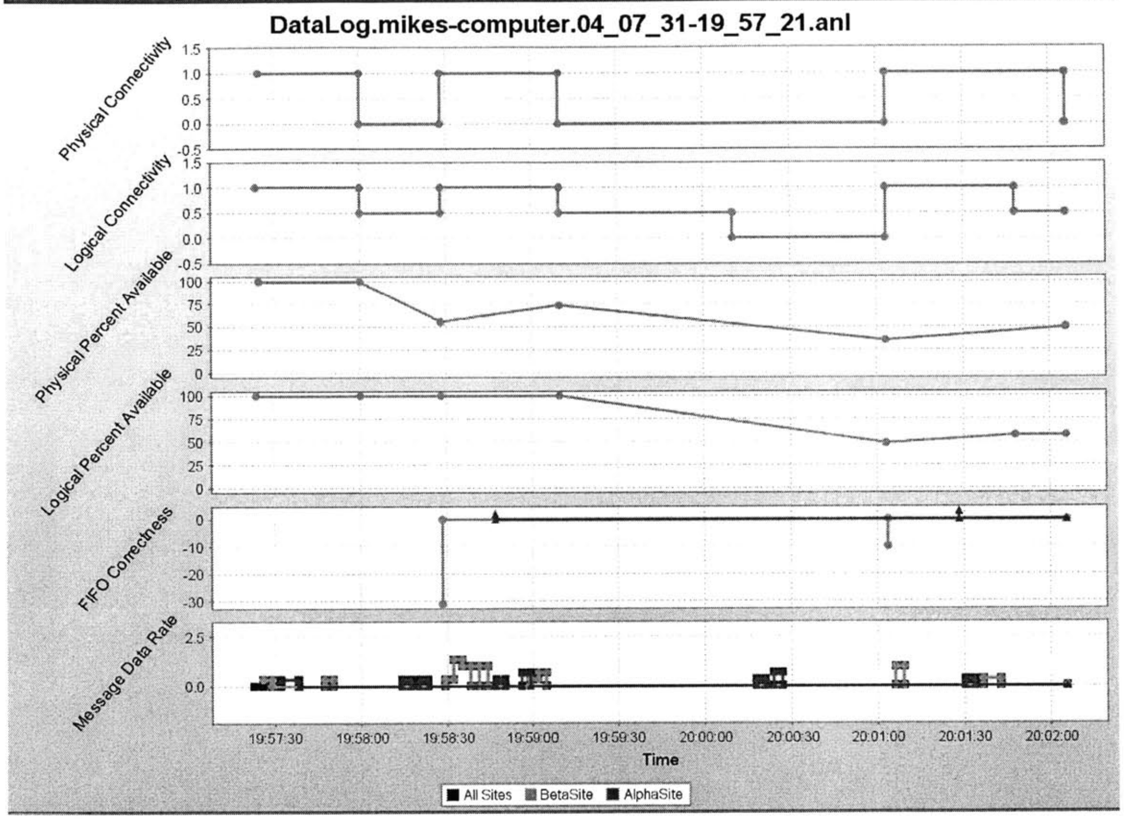


**Figure 19: MonitorLogViewer. View of a .ANL log file, produced by the AnalysisLogger object. The coloring scheme is based on lexicographic ordering. This graph is based on the same information as Figure 18.**

The MonitorLogViewer is also capable of customizing its display to the data set it must visualize. Figure 20 is the image the MonitorLogViewer created for a RawDataLogger log file. The only information in the log file that the MonitorLogViewer can visualize is the physical and logical connectivity information. The other data that the MonitorLogViewer is capable of displaying is only produced by the AnalysisDataLogger and is therefore not present in the RawDataLogger log. Instead of showing empty plots for unavailable data types, the MonitorLogViewer maximizes the sizes of the available plots, thereby effectively using the empty space that would otherwise be wasted.

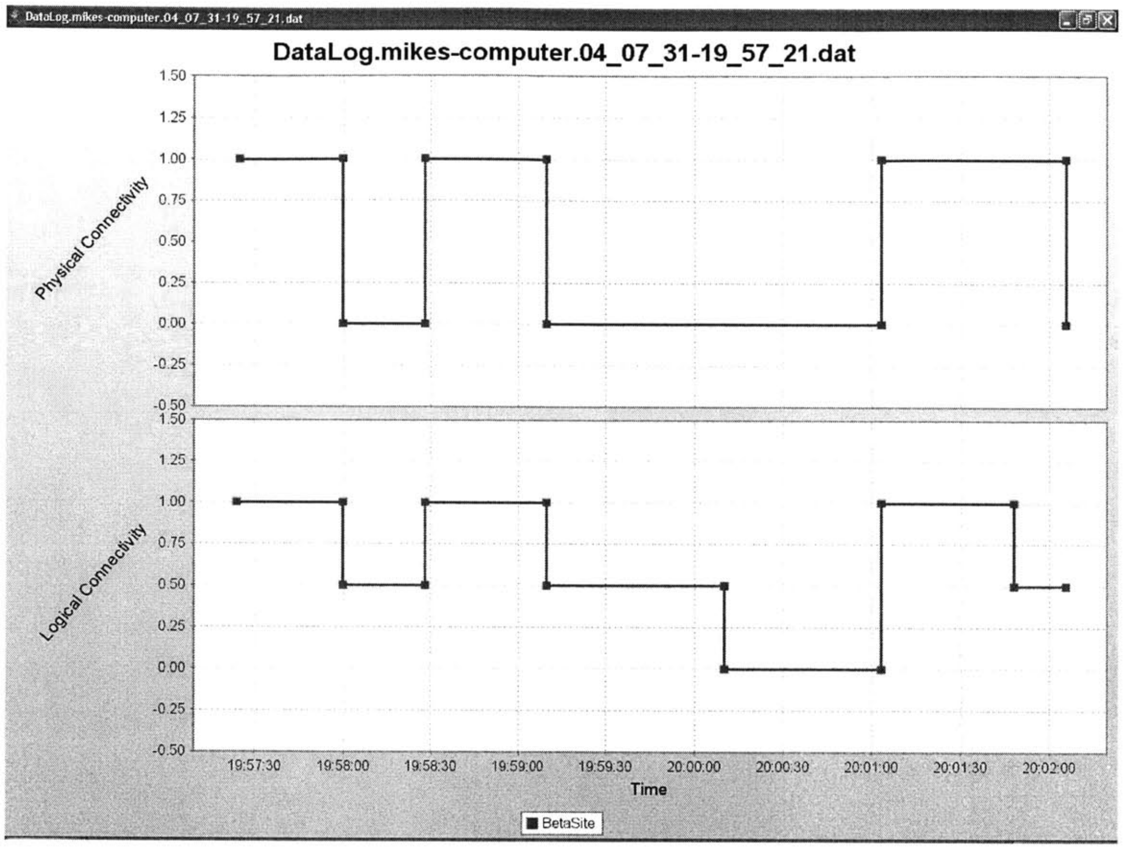


**Figure 20: MonitorLogViewer. View of a .DAT file, produced by the RawDataLogger object. The coloring scheme is based on Hash Colors.**

## 5.8 DEBUG

While writing and debugging code, developers will often place commands in their applications, which are purely intended to help remove errors from the code later on. These commands are said to produce debugging information. In the process of implementing the robust chat application we designed a standard method for outputting debug information. The following sections describe the features of this debugging system.

### *5.8.1 Debug Levels*

During program execution there are several phases when a developer may be interested in information about how the system is operating. Some of these phases include system startup, shutdown, errors, status changes, and normal operation. In order to expedite debugging efforts, it can be helpful to focus on only those debug messages, which occur during the same phase as the error being tracked down. To prevent information overload while debugging the chat application, we implemented the concept of *debug levels*. *Debug levels* allow developers to choose which kind of debug information they would like to see. Debug information that is not tagged at the same level as the requested information is not output at runtime reducing the amount of output the developer must study to find the error and the size of the output log files.

### *5.8.2 Timestamps*

All the system output from the chat application is timestamped to help pinpoint when errors occurred. Since the chat application functions as a distributed system, with components on many different computers working together, it can be very helpful to have a way to cross-reference information on one computer with information on other computers. Discovering bugs in distributed systems is a notoriously difficult task, which makes timestamping all the more valuable.

Timestamping is most effective when all the computers in the distributed system have synchronized clocks. In a test environment this is a feasible condition, so for debugging purposes the timestamps work very well in our test environment. In a real-world environment synchronizing the system clocks is generally not possible, but even in this situation the timestamps provide a rough guide, often narrowing the search for an error from hundreds of lines of output to just several dozen lines.

# 6 Testing and Evaluation

The following sections describe the tests and test environment we used to evaluate our chat application. First we discuss the setup of the test-bed. Then we discuss the results for several of the chat application components.

## 6.1 Test-bed Architecture

Many of our tests were conducted using a test-bed constructed by the project team; the test-bed is described in Prasad Ramanan's M.Eng. Thesis [11]. The basic layout of the test-bed is pictured in Figure 21. The test-bed consists of four computers running the Microsoft® Windows™ 2000 operating system, and a router running the Linux operating system. Each of the four Windows computers functions as a site for test purposes and runs one chat server. The four sites are named Commonwealth, Kingdom, Metropol, and Republic, while the router is named Ghetto. Data sent between chat servers is routed through Ghetto, which uses software to control the link availability between any pair of chat servers. The availability can be set independently for both directions on a link.

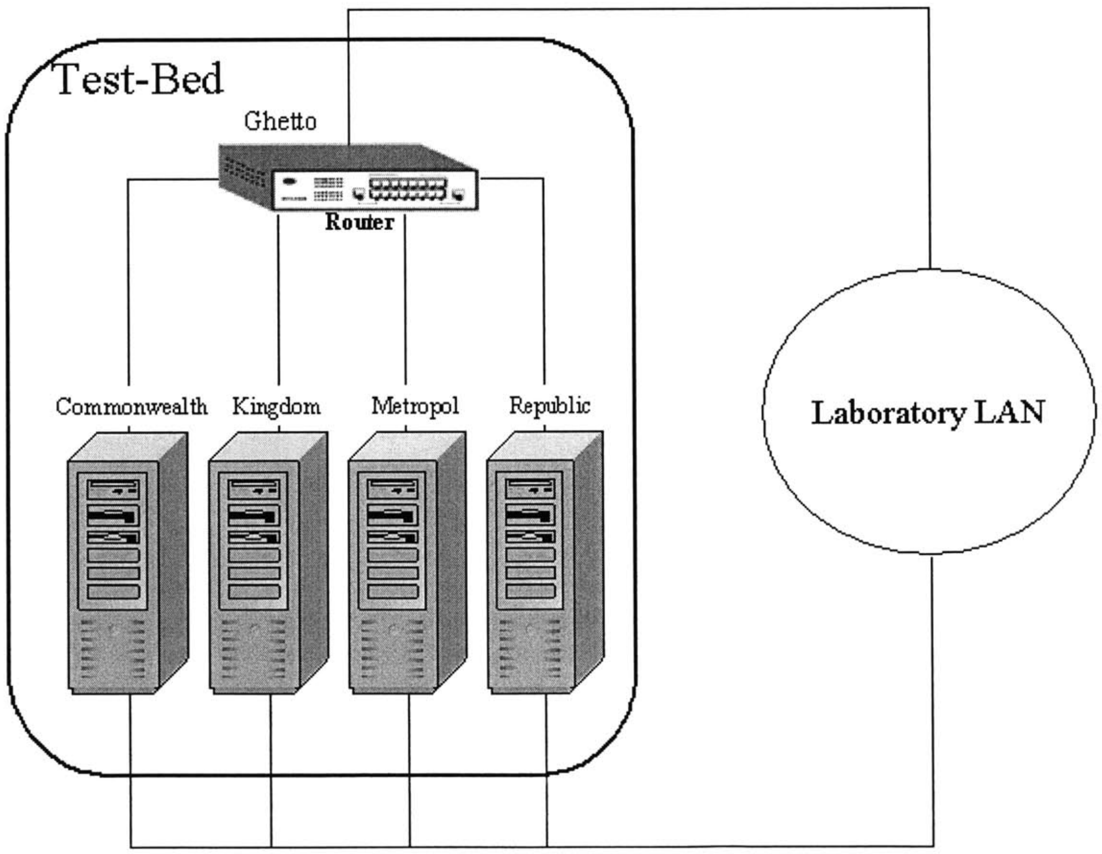


**Figure 21: Test-Bed Architecture.**

For tests focusing on inter-site connectivity chat clients are run on the same computer that is running their chat server. For tests focusing on intra-site connectivity chat clients are run on computers that are not running the chat server they are connected to.

## 6.2 Monitor Results

As we hoped, the monitor proved to be a useful analysis tool. We used it to view the output of DataLoggers and identified several behavioral problems. The Monitors also provided effective "summaries" of large data sets, by compressing many hours of collected data into a single graph.

The only drawback when using Monitors was the processor usage. The free graph rendering software package we use to implement the Monitor displays is not efficient at quickly rendering data. This causes the Monitors to use more processor time when either the data set size or the display size (i.e. the size of the picture being rendered) increase. For exceptionally large data sets this results in noticeable slowdown of the application.

## 6.3 JEFX 2004 Results

The results we gathered from the JEFX 04 flights were generally very encouraging. The system experienced very few behavioral problems and almost no errors. We collected application data from nine flights, which ranged in duration from one hour to over five hours.

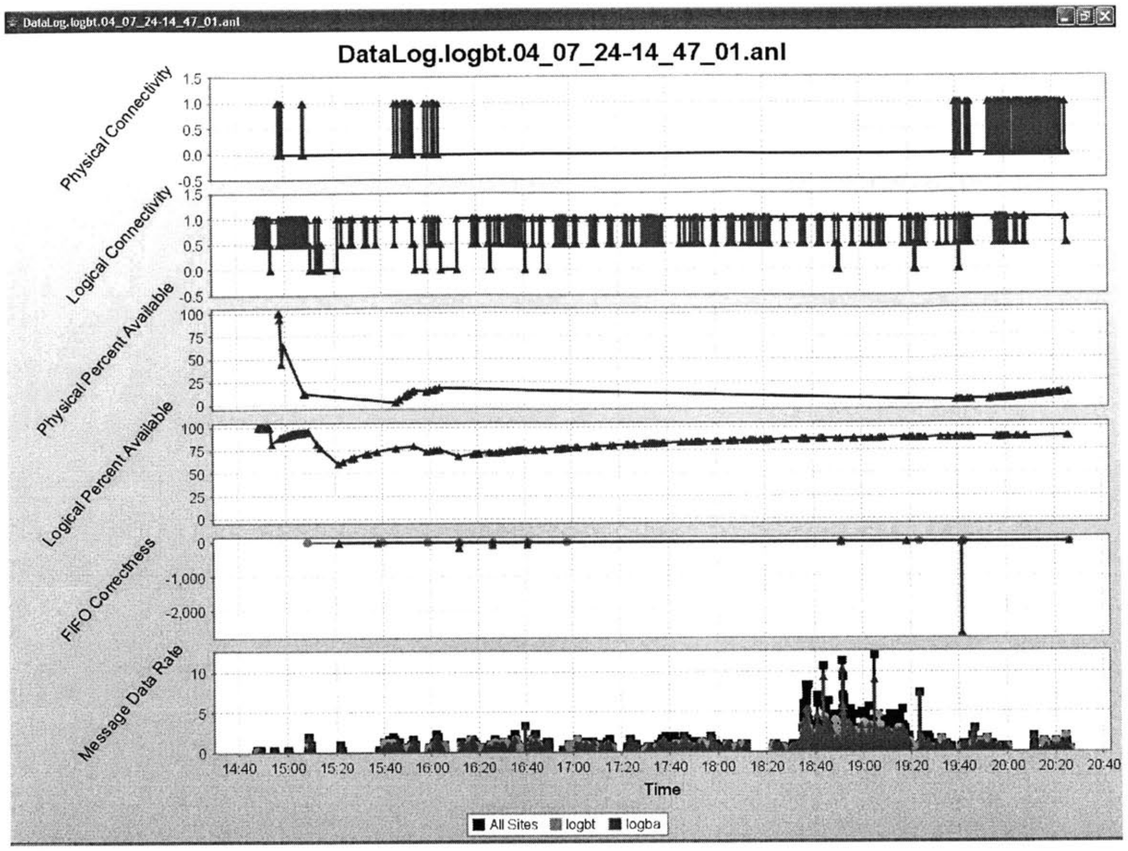


**Figure 22: JEFX 04 Flight Data.**

### *6.3.1 Flight Data*

Figure 22 displays some of the data collected during a JEFX 04 flight. The chat server that this data was collected from operated from right after the flight commenced, at 2:47pm, to right before the flight ended, at 8:26pm. The log indicates that the chat server was operational for over 5.5 hours (exactly 5 hr, 39min, 6sec, 797ms) without any major errors that could cause the server to crash. There were 12 long-term disconnects, of which only 4 were longer than 2 minutes. There were a total of 306 disconnects, which means the chat application successfully bridged 294 short-term disconnects. The average data rate for the run was around 0.72 messages per second, while the peak data rate reached 12 messages per second at one point. The chat application was available 89.74% of the time during the flight. The average availability of the network links, represented by the physical connectivity, is unfortunately inaccurate.

The operating system's ping command was used to gather information about the physical connectivity of the network. This approach worked well in the test-bed environment; however, it failed on the flights. We later determined that this method failed because the link latencies were much higher than expected. The timeout value for the ping command had been set based on the expected latencies, so the higher than expected latencies caused the ping command to timeout even during periods when the server was connected well. This had no effect on the functionality of the chat server, but it invalidated the physical connectivity data gathered using the ping command. Figure 22 indicates that the network links were connected 12.76% of the time, which is far below the actual value. Using the assumption that the links must be available for the application to have a logical connectivity value of 1, we determined the network link must have been connected at least 3 hours and 31 minutes during the flight. This is translates into a link availability of approximately 62.40%. The chat application was available 89.74% of the time, which means it outperformed the link availability by 27.34%.

### 6.3.2 Crew Feedback

The response from the JEFX crews that used the chat application was very positive. Several crewmembers commented that they appreciated the simple interface and overall robustness [3][4]. They appreciated that the system was easy to install and was robust enough to stay operational when other communications means broke down. Leonid Veytser, a crewmember who used the chat application on multiple occasions, commented that the application allowed them to "to keep communications between air and ground not just for [his team] but for other people that needed communications and there were no other means at the time" [4].

Senior Staff member, David Kettner, mentioned that the chat application was used to maintain situation awareness and coordinate activities between the ground team and airborne team members. He also used the tool to communicate with the airborne JEFX Test Director "from time-to-time." In his words: "The tool was used a lot" [5].

After one of the first flights with the chat application Senior Staff member, Stephen McGarry, noted that he was "very impressed with the performance of the [robust chat application] … We were [using] it almost exclusively for the 5.5 hour mission where the [airplane] was almost always beyond [line] of sight" [6].

Some of the crewmembers noted that our prototype system would have benefited from some optimizations, such as decreasing the size of chat messages. Smaller messages would have allowed the system to function using just low-bandwidth links, such as Iridium [3][4]. These suggestions will be addressed in future versions of the chat application; however, our current prototype chat application was designed as a basic foundation for a robust chat application. As such, we implemented the prototype with a focus on robustness and basic functionality, deferring some time-consuming optimizations for later.

# 7 Future Work

While working on the chat application we identified several areas that suggest new and continued directions for improvement.

## 7.1 Security and Authentication

As we have already mentioned, communicating information is essential in the MC2C environment. However, protecting this important information is arguably just as essential. Our current prototype chat application is robust enough to guarantee that errors introduced by the MC2C environment alone will not cause the application to fail. The system is also intended to be deployed in a network environment that provides basic security for transmissions: all communications are encrypted and the physical network components are secure. This does not mean that the system is immune to failing, though. Malicious users are a common source of system failures for non-secure systems. For our system to be thoroughly robust it must guard against such attacks by authenticating users and securing communications between system components (client to server and server to server). Authenticating users allows us to match usernames with real individuals. This removes any possible anonymity those individuals had and makes them accountable for their actions while using the chat application. Only in this way can we be sure that information has not been tampered with and users actually are who they say there are.

Security and Authentication are also fundamentally necessary to enabling access control capabilities.

## 7.2 Alternative Applications

As we mentioned before, one of our primary goals was to develop a highly extensible system. To capitalize on this extensibility, it makes sense to develop new applications that can easily use the system. Two ideas for such applications are a shared whiteboard and a distributed file transfer system.

A shared whiteboard allows users in different locations to collaborate by sending visual messages instead of textual ones. The idea is that several users would be able to draw on a shared "whiteboard" area and everyone would see the same resulting picture. There are already several commercial applications that support this kind of functionality, but building such an application on the foundation of the robust chat application would automatically make it robust enough for use in the MC2C environment.

A distributed file transfer system allows users to share information in any format they choose, by sending files instead of text or images. The file transfer system would allow users to send a file to multiple users at the same time instead of performing multiple point-to-point transfers. Large files could be split into smaller pieces, with each piece being sent as a message. The transfer system could weather short-term disconnects, since the robust chat application guarantees reliable message delivery; something that traditional file-sharing applications are not designed to do. The result of using the robust chat application as the foundation for a simple file transfer system would be a robust distributed file transfer system.

## 7.3 Chat Bot Improvements

While the chat bots already have the capability of responding to messages they receive, the implementation is still fairly rudimentary. In order to test the system more realistically it would make sense to improve the algorithm used to determine responses. Instead of making a probabilistic decision, it may be helpful to have bots output pre-scripted conversations, complete with questions and answers. The bots would then be expected to only send answers after they receive the proper questions. Bots that send questions could also become impatient and send new requests for information if the answer doesn't arrive in time. This approach would mimic natural conversation better than the current system does and could lead to new discoveries about the properties of the chat application.

# 8 Glossary

| | | |
|---|---|---|
| Chat Client | - | Client application that communicates with the chat server (not to be confused with the ChatClient object, which is an implementation of a chat client) |
| Chat Message | - | Message sent by the chat application |
| Chat Room | - | Abstract "room" that users may collaborate in |
| Chat Server | - | Server for the chat application (not to be confused with the ChatServer object, which is an implementation of a chat server) |
| User | - | Human user of the chat application |

## 9 References

[1] S.T. Dougherty, *JEFX 2002 ends with positive results*, in *Air Force Link.* August 9, 2002.

[2] R. Khazan, S. Lewandowski, C. Weinstein, et al. *Robust Collaborative Multicast Service for Airborne Command and Control Environment.* Military Communications Conference (MILCOM), Monterey, CA, October 31-November 3, 2004.

[3] J. Cooley, Flight Crew, MIT Lincoln Laboratory, Lexington, Mass., personal communication, July 28, 2004.

[4] L. Veytser, Flight Crew, MIT Lincoln Laboratory, Lexington, Mass., personal communication, July 28, 2004.

[5] D. Kettner, Senior Staff, MIT Lincoln Laboratory, Lexington, Mass., personal communication, August 5, 2004.

[6] S. McGarry, Senior Staff, MIT Lincoln Laboratory, Lexington, Mass., personal communication, July 23, 2004.

[7] R. Khazan, *Robust Chat for Network-Centric Air Operations.* MIT Lincoln Laboratory. Division 6 Seminar. May, 2004.

[8] C. Kalt, *Internet Relay Chat: Architecture,* ftp://ftp.irc.org/irc/docs/rfc2810.txt. April, 2000.

[9] Unknown, *Jabber User Guide*, http://www.jabber.org/user/userguide/index.html. August, 2004.

[10] J. Cooley, Flight Crew, MIT Lincoln Laboratory, Lexington, Mass., personal communication, August 6, 2004.

[11] P. Ramanan. *Robust Chat for Airborne Command and Control.* MIT Masters of Engineering thesis. August 2003.