The Java Talker Project

Computer Science 690 Project
University of New Haven

Jeffrey K. Brown
November 24, 2003

Table of Contents

I. Introduction to Talker Communication

Chat rooms are a means of text based communication that take place across the Internet. Chat rooms are an appealing means of communication since they attract people with similar interests. They are also desirable because they are generally free and allow for an open exchange of ideas. Most of the time, chat room communication is public and available to all of the users on the system. Often, chat room software provides the capability to send private messages between users.

Talkers are another implementation of the chat room concept. Like their time-based, role-playing game Multi-User Dungeon (MUD) cousins, Talkers have generally been built in C for Unix-based systems using the Socket API. Users of Talkers typically connected to the system using telnet and sent plain text data directly to the Talker's port. There are also specialized client programs that have features like macros, shortcuts, account and password management, and address book features. Talkers provide messaging types that I break down into 4 main classifications. Online Public messaging is the typical message. Everyone can see the message. Online Private messaging is a typical private message from one user online to another user online. Talkers have Offline Public messages, in the form of a message board. Users on the system can read the message boards long after a message's originator has logged off. The other message classification is the Offline Private message. This message is implemented as a sort of online mailbox that can be reviewed by the recipient even after the sender has logged off the system.

One very popular Talker implementation was the NUTS Talker. NUTS is short for Neil's Unix Talk Server, and was built by Neil Robertson in 1992. The NUTS system was designed to fill the niche between the rustic Unix talk feature and the overly complex Internet Relay Chat or IRC chatroom world. (Robertson, 1) The NUTS system fit this niche quite well and Talkers became very popular. In addition, Robertson made his NUTS source code available to the public domain, gaining it even more popularity. Many users with Unix accounts could download, customize and run their very own NUTS systems. University students, separated from their high school friends, often utilized Talkers to maintain communication for free between universities.

In 1995, I started as a Computer Science undergraduate at the University of New Haven. Being naturally curious about computer systems, especially the Unix system, Charger, I explored what I could, utilizing the Unix talk program to communicate with friends at their respective universities. I was introduced to Talkers in late 1995 as a better means of communicating with multiple friends at the same time, since Unix talk would only support individual connections between two users. In mid to late 1996, I began utilizing some of my newly learned C programming skills by customizing and adding features to a NUTS Talker belonging to a friend.

Presently, I host my own Talker called Dark Ages, which is built upon a NUTS 2.3 release. I implemented many new features, customized code, but never delved very deep into the underlying socket communication that was taking place. It was always my goal to figure out and understand the socket communication, but because it was just a hobby, neither the need nor the urgency was there.

Today, there are far more user-friendly and available means of real-time text communication than NUTS Talkers. NUTS' interfaces were very user-friendly in 1996, when it was essential to remember keyboard commands to use a PC. Advancements in graphical user interfaces, most notably in Microsoft Windows, and the overwhelming popularity of “dumbed-down” Internet applications like America Online have led to NUTS interfaces being perceived as difficult to use. Additionally, commercial services like Yahoo! and MSN have added graphics, sound, and video to enhance the chatroom experience. For these reason, the popularity of Talkers has dwindled, but still remains popular among many people.

My main objective in creating this project was to learn about, understand, and implement a NUTS-like Talker server of my own design. Since a reasonable opportunity to accomplish my goal of understanding socket communication had presented itself, I took it immediately. I also wanted to create a chat room interface that was easy to use which would be compatible with the NUTS-like system, so NUTS fans would still be happy, but users unfamiliar with the NUTS commands would neither be turned away nor intimidated by their complexity. Finally, like Neil Robertson, I would like to create a system that others can build more interesting and complex applications from using my system as a base. Documentation of the base system has always been difficult to find, and a project of this sort would allow me to provide some to the world. The Talkers of the past would run only on Unix systems. There are many people running Windows systems, and Java would enable me to build a Windows compatible Talker. Utilizing Java's claims of platform independence could gain my new system a foothold where Talkers never had one before. Reengineering the Talker server in Java would allow me to accomplish these goals.

II. Project Challenges: Problems of Talkers and C-Based Socket Software

Because of the availability of much more powerful, feature-rich, and user friendly Chatroom software, Talkers and other C-based Socket software of its generation are antiquated by today's standards. Users must often remember keyboard commands while newer systems offer friendlier graphical user interfaces. Software of that generation is also limited by old constraints, like the Socket libraries and forking calls being available on Unix systems only. User and System data is stored in file databases, which can be cumbersome for Administrators to maintain. Finally, security issues like unencrypted communication and supsceptibility to Buffer Overflow attacks plague these systems. This section is intended to shed light on some of the challenges that this project will attempt to address.

When taking a look at other Chatroom software, most notably America Online chatrooms and Yahoo! chatrooms, I immediately notice the ability to get multiple sources of information on the screen at once. You are able to open a user profile in a window while you continue to chat in another. Because of the medium Talker software uses, (i.e. the sockets), information is provided on a single interface. If you wanted to see a user profile, you would get it on the same interface used for online messaging, message boards and area descriptions.

Talker software often uses Unix-specific libraries, like the Socket API. Computers running Microsoft Windows could not run Talker software because there were no standard Socket libraries available. Additionally, Unix systems could run system calls like fork(), which would allow them to create a server process in the background and return the system to the user. Windows systems also lacked this feature.

Typically, NUTS and other Talker systems were arranged in a directory structure containing many files. Files in one directory would represent message board data, while others would represent configuration files. Other directories contained User settings and mailboxes. Usually the Talkers were programmed to deal with each of the files when needed, but if one wanted to change the format of the user files, scripts to convert the old format to the new needed to be written or every file would need to be edited manually.

Finally, today's society demands security and privacy. Talkers were designed to facilitate communication, not secure and reliable communication. As a result, they have a few security issues that need to be addressed. First, messages are sent between the user and the server as unencrypted plain text. This communication could be captured on a network and viewed by users other than the intended recipient. I have provided a demonstration of this in Appendix K. Stability is another aspect of Talker Security. Talkers were generally written in C, which means pointers and array sizes come into play. Malicious users would often search for commands and strings with fixed buffer sizes in an attempt to overflow the buffer and bring down the Talker. Additionally, students running Talkers did not always understand memory management, and memory leaks often contributed to Talker downtime as well. There are also newer languages available today that offer automatic buffer sizing and garbage collection.

Some of the challenges of this project will be to address some of the issues discussed above and provide solutions to them. Even if I am unable to overcome every challenge, I would like to provide some of my knowledge and experience gained from this project to the readers if they would like to address any of these issues in the future.

III. Project Objectives: Deliverable Solutions to Problems

The main objective in this project was to build a Talker Server. To maintain compatibility with the current NUTS base that Dark Ages Talker presently hosts, my design goals suggested I make it seem as much like NUTS as possible. At the same time, I was driven by my goals outlined in the Project Challenges section above. I think the blending of the reimplementation of a Talker and the tackling of some new challenges made an interesting project. Some of the original objectives evolved from their proposal form, but for the most part, the majority remained the same. There were even new challenges and objectives that revealed themselves along the way that were addressed. In this section, I will cover the main Project Objectives and describe them in detail.

The first objective was to implement a new Talker Server using Java. I wanted to implement the same sort of commands and functionality into the new Talker Server as in NUTS Talkers. This would allow existing Talker users to continue to use the same friendly interface with which they were already familiar. This new Talker Server would be a framework on which to build new features and functionality in the future.

The next objective was to implement the socket server portion of the Talker server. The Java Socket API deals with the socket communication at a higher level of abstraction than the C Socket API. This makes the Socket communication take place using easier to understand code. I wanted to take advantage of this as part of the project. Once I was able to get the socket portion of the Java Talker Server working, the objective was to implement the 4 types of messaging, Online Public, Offline Public, Online Private and Offline Private.

I wanted to replace the file system database with a relational database for storing user and area information. In addition to being easy to manage, relational databases are more interesting, less complex and less cluttered than many individual files and directories. The relational database also allows integration with other languages and systems.

Java is often touted as being a write-once, run everywhere language. I wanted to utilize Java to create a Java Talker server that would run on Linux and Windows, solving the Unix only problem.

For the users who are not particularly skilled in NUTS commands, I wanted to provide an easy to use interface using a Java Applet and some web scripting. The applet interface would also allow multiple types of data on the screen at one time.

The next objective would be to improve the documentation of the system. Since I would be designing the system from the beginning, I would be in a perfect position to create good documentation. I was interested in utilizing the Javadoc tool to build web-formatted documentation.

The final objective would be to provide more information and suggestions about how to handle other Talker challenges in the future.

IV. Project Technology

The Java Talker project utilized many different types of technology. I implemented a new Talker Server utilizing Java to achieve the multiplatform socket server. I added a database to store user and area information, provided a web based interface using PHP and Java Swing Applets. Finally, I was able to use Javadoc to create some web-based documentation. In this section, I will discuss some of the major technologies in use within the project.

The first technology that I will cover is Socket Programming. I define a socket as an abstraction used by a computer program to talk to a network. Socket Programming is the implementation of these sockets in network applications. Most modern computer languages and operating systems provide a means of connecting to and communicating with the sockets. This means of working with the sockets is called the Socket Application Programming Interface or Socket API. The Java Talker uses the Socket API to allow users to connect to the system and communicate over a network.

Implementing a Talker using the Java Socket API instead of the C Socket API used by NUTS Talkers proved to be considerably less complex. This is because Java abstracts the Socket calls at a much higher level than the C Socket API. The Java Socket API allows you to create a simple ServerSocket and start accepting connections in two calls, while creating the same server socket in C takes about 12. C requires you to set many low level parameters even if you are creating a simple implementation. Java's Socket API classes have multiple methods for each call so you can define lower level parameters if they are needed. The Talker class will demonstrate how the Java Socket API works. I have provided in Appendix J, an example of how the C Socket API works.

Since Talkers are multiuser systems, I needed to make sure the Java Talker could support multiple users, too. Once again, Java provided the means to be considerably less complex. System calls to read data from the sockets are blocking, or no processing can continue until the call completes successfully. In other words, User 2 can not send a message until User 1 has, even if User 1 has nothing to say. To solve this blocking problem, NUTS Talkers rely on File Descriptors to keep track of connections. File Descriptors are sets of flags that indicate whether or not a file or stream are ready to be read or written. When a new connection is made, the socket is added to an array and linked to a File Descriptor set. An infinite loop continually traverses the set and checks to see if any File Descriptors are set to be read, and if detected, reads and processes the input. The building and managing of the File Descriptor set can be cumbersome, but it works in the NUTS Talkers. Because Java is multithreaded, we are able to put each new connection into its own thread and use the ready() method to check and process input on each individual socket. Once again, Java has the additional methods needed to make low level changes if needed, possibly utilizing file and socket descriptors, but in general the high level of abstraction takes care of the low level details for the programmer.

Another technology being used is the Talker technology in general. Since Talkers already exist and their interfaces are reasonably well defined, I needed to make sure I implemented some of the more important commands. I will discuss specific commands implemented in more detail in a later section. Talker commands start with a period (.). The system identifies and distinguishes commands from communication with the leading period. Online messaging commands deal with chat-type communication. There are online public commands like .say, .emote, and .rev that are used to send messages intended for all in a public area. There are online private message commands like .tell and .remote that send messages intended for the individual. Offline messaging commands implement the message board and mailbox types of messages. Offline public messages are used with the .read and .write commands to perform message board features. Finally, offline private message commands like .smail and .rmail implement mailboxes.

A third technology would be the addition of the Relational Database. User data was historically stored in a file system directory with multiple files for different types of user data. Area information was stored in a second directory. Message boards in a third. The Java Talker, on the other hand, is utilizing the database to store this information. The database that we are using for the Java Talker is a MySQL database. We are connecting to it using the JDBC, or Java Database Connectivity methods. The Java Talker database consists of a table to store User, Area, Mail and Message Board data. Because multiple technologies can talk to the database, in the case of the Java Talker, the Talker itself and the Chatlet web interface, we are able to share data more easily than if we had to write special code to open the filesystem and pull out data from the file system data store. I have provided an Entity Relationship Diagram of the Java Talker database in Appendix I for more information.

The Chatlet Web Interface uses several different technologies as well. The Chatlet Interface is written in the web scripting language, PHP, with a Java Applet to control the socket communication with the Java Talker. The PHP interface runs as a module of the web server software installed on the system. I was attempting to get an interface that would look the same on all systems running it. PHP provided the capability to create dynamic web pages displaying HTML and still link my Java Applet for the Talker connection. The Applet itself is written using Swing, the native Java windowing toolkit. The combination of HTML and Swing output would give the Chatlet a similar feel on nearly all systems. The Chatlet's applet has two threads. One thread reads output coming from the Java Talker. The second thread reads keyboard input from the user and sends it to the Java Talker.

Because Java technology can be used to write programs on multiple types of operating system, I addressed the issue of NUTS Talkers being available on Unix and Linux systems only. My implementation of the multiplatform Java Talker required me to use a multiplatform database. For this, I chose MySQL, using the Connector/J JDBC libraries provided by MySQLAB. In addition, since the Java Socket API talks to the Java Virtual Machine to get the system network interface control rather than special operating system drivers, like it would if I used the C Socket API or Winsock (another Socket API), the socket section works as well on Windows as it does under Linux. Java also brings to the table the automatic memory management. Rather than needing to allocate and deallocate memory, Java provides the memory when needed and cleans up using a process called Garbage Collection. Another added benefit of the memory management is that Buffer Overflow attacks that would cripple C based Talkers do not impact the Java Talker, since Java will automatically allocate more memory if a String buffer was nearing its maximum size. Finally, Java provides a great tool, Javadoc, which creates web-based documentation.

V. Java Talker Design Alternatives

When I began thinking about how to implement the Java Talker and the Chatlet interface, I looked at it three key alternatives. My goal was to have communication taking place through the Talker Server interface be compatible with the Chatlet web interface and vice versa. I wanted the ability to create an easy-to-use interface for the masses, but retain the familiar NUTS-style interface. The three alternatives, each of which I will discuss in this section, were 3rd Program Controlled Messaging, Java Talker Controlled Messaging and Database Controlled Messaging. For each alternative, I will discuss the advantages and disadvantages of each. In the end, I ultimately decided upon Java Talker Controlled Messaging.

The first alternative was Database Controlled Messaging. This alternative required a separation between Java Talker and Chatlet Interface. Cross-system communication necessitated the use of a communications table. When input was detected on either interface, it would be added directly to a communications table. The communications table would then be read every few seconds for new input. If new input was detected in the communications table, each respective interface would process the input and provide the communication to the user connected to that interface in the correct format. The advantage to this approach is that each interface could be very different from each other. The disadvantages of this would be that I would need to implement any commands twice, once for each interface. Additionally, because the database would be used for communication, there would be a lot of database activity, which would use a lot of resources.

The next alternative was 3rd Program Controlled Messaging. Like the Database Controlled Messaging, communication between the two types of interface required the use of a communications table. Each interface would place input received onto the Input Table. The difference is that a 3rd Program would be running on the server, continually monitoring the Input table for new entries. The external program would then convert the input data to the appropriate output for each interface and place it on the Output Table. Then, each interface would monitor the output table, and pass the data along to the user. This alternative was desirable because it reduced the amount of programming. In other words, I would only need to implement each command on the 3rd Program. The program would handle the interface differences. The disadvantage to this approach was again, the database activity. The communication table creates enough database activity, since every message sent to and from the system has to start on the communications table. Since we have another program continually monitoring the tables, like the individual interfaces do already, this would create twice the number of database hits.

The final approach was the Java Talker Controlled Messaging. This alternative did not use a communications table, so the database was not going to slow the system down this time. The database is only used to store user settings, offline messages and area information. The Web Interface connects directly to the Java Talker and sends messages received to the Talker in the same way as a direct socket connection would do. This means that you could still use the legacy NUTS commands if you wanted, but I can also link Chatlet events to talker commands and get the same functionality. Finally, because the Java Talker controls the messaging and handling of commands, there is no need for double coding. I ultimately decided upon this approach.

VI. Development Environments

I used several different development environments to build the Java Talker. This is mostly due to the fact that I worked on the system at different times of day at different places. Most of the work on the Java Talker was done on my home systems, but on occasion, I would work on the system after hours while on my work systems. In this section, I will briefly get into the computers and development software that I used on each piece.

Chimaera is my main home computer. It is running SuSE Linux 8.0, and is powered by a Pentium 2. I used this system to develop the C socket server, which I use to demonstrate the differences between Java and C Socket APIs. I also had Java Development Kit and MySQL installed here to work on the Java Talker server. Most of the work done on Chimaera involved the C Socket implementation.

Moose was the primary development environment for the Java Talker. For the first three quarters of the project, Moose was running SuSE Linux 8.0. As I neared the end, I decided to install a UML diagram program, which resulted in major system problems. Fortunately, I backed up my software beforehand, so when I was forced to reload the system, I was able to pick right up where I left off. After this experience, I switched to Debian GNU Linux. I also had MySQL with Connector/J libraries for Java Development Kit 1.4.2 installed for developing the Java Talker and Chatlet. Finally, I had the Apache Web Server with PHP 4 modules for the Chatlet. environment. I also used Moose to develop the setupDatabase program that builds the Java Talker database. Moose is powered by a Pentium MMX processor.

Viper is a Pentium 2 system running SuSE Linux 8.0. I used this system to work on and test the C socket implementation. I also installed Java 1.3 on this system to compile the Java Talker and test to see differences in compatibility with different web browsers and Java Clients.

Griffin is my office computer, a Pentium 3 running Windows 2000. I used this system to work on the Windows 2000 version of the Java Talker. I utilized Griffin for Windows 2000 development prior to linking the MySQL database. Griffin was installed with Java Development Kit 1.3.

Pegasus is my home computer running Windows 2000. I used Java Development Kit 1.4.2 with the Netbeans IDE for the Windows 2000 development with the Windows 2000 based MySQL database. I also tested the Chatlet under Windows 2000 environment on Pegasus prior to adding the PHP interface.

Dark Fantastic Network is the production environment for the Java Talker. Dark Fantastic Network is running Slackware Linux on a Pentium 3 based system. Also installed is Apache Web Server and MySQL database. Since I have my Internet hosting provided here, I designed the Java Talker to be compatible on this system.

Finally, for testing purposes, I requested friends with different levels of Windows, different distributions of Linux, a FreeBSD and Apple Macintosh computers running OS9 and OSX. Peculiarities discovered by the multiplatform testing were modified to make it work equally well on all systems.

VII. Documentation

I was interested in creating some better documentation for Talkers, and hopefully in designing the Java Talker and explaining how it works through this project, others may take what I have learned and build onto it. In this section, I will discuss the forms of documentation I provided.

The Java Talker and Chatlet program listings contain comments at interesting points throughout the programs, explaining what is happening at different times. I also wanted to utilize the Java Development Kit's online documentation generator, Javadoc to create documentation. By reading specifically formatted comments, Javadoc is able to populate a web-based template and create web pages with code documentation.

In addition to the code-level documentation, I have included some UML class diagrams so that we can understand the relationship between different classes and activity diagrams to show how things communicate. Finally, I created an Entity Relationship diagram of the Java Talker database to aid in the understanding of the Java Talker system. All of these documents can be found in the Appendix.

VIII. Project Software Components

There are four primary components of the Java Talker project. The Java Talker Server, the Java Talker Chatlet, the SetupDatabase database builder and the Java Talker startup script. In this section, I will dissect the different components of the project and discuss them.

First, I will cover the Java Talker Server. The Java Talker Server is the largest part of the Java Talker project. The Java Talker has 3 principal parts, the UserConnection array, the Talker Server, and the TalkerTimer.

Talker Server

The Talker Server part of the Java Talker has two stages, the Initialization Stage and the Accepting Connection Stage. All of the Java Talker Server's parts are tied together by the Talker Server. In addition, the Talker Server's class contains most of the processing for the individual commands called by the UserConnections.

In the Initialization Stage, the Talker Server makes the initial connection to the database. Using the database connection, the Talker determines whether or not the system will support encryption. This is done so the system will correctly authenticate users, whether to use encryption for password checking or not. Then, we initialize the UserConnection array, Area array and the TalkerTimer. Finally, the Talker Server creates a ServerSocket and starts listening on a port. At this point in time, the Talker Server ends the Initialization Stage and starts the Accepting Connection Stage.

The Accepting Connection Stage calls the Accept method of the ServerSocket. This is in a loop that will continuously accept new connections. The Accept method blocks, so the system will sit and wait for a single connection. When the connection arrives, the processing continues and the connection is assigned to a temporary socket. Next, we create a UserConnection object and assign it to the new connection referenced by the socket. Then, the Talker Server resizes the UserConnection array to make room for the new UserConnection and adds it to the array. Finally, the Talker Server turns over all responsibility for the socket and connection to the UserConnection array and goes back to listening for new connections.

When the system is going to shut down, the close() method of the ServerSocket is called. Because the accept() method blocks, we are unable to have the ServerSocket continuously check for a shutdown command. When the close() method is called, the ServerSocket accept() method immediately results in an Exception. In the Exception handling section, we have a break statement, which will terminate the loop. Some final cleanup is done, and the Java Talker Server ends.

UserConnection Array

The UserConnection Array is an array that grows and shrinks with the number of UserConnection objects that are connected. This is done by creating a temporary array of the new size and copying objects from the old array. When a new user is connecting, their UserConnection is added to the array. When a user is disconnecting, their UserConnection is not copied to the new array and is deleted.

Each UserConnection has three main parts, the Setup Section, the Authentication Section and the Command Processing Section.

The Setup Section deals mostly with the handling of the Socket Communication aspects of the UserConnection. First, we assign the socket from the ServerSocket accept() in the Talker Server to the current UserConnection. Next, we create a BufferedReader to handle all input from the socket and a PrintWriter to handle all output to the socket. Next, we connect to the Database and start the UserConnection thread.

The Authentication Section is utilized in two main ways. If a user already has an account, the user will provide their username and password in this section. The username and password will be compared against the username and password in the database, and if successful, the data in the User record will be assigned to the UserConnection. If the user does not already have a user account, the Authentication section lets the user create a new account in the database. Once the user account is created, the user is authenticated and let onto the system. Users connecting are announced to the other users on the system.

The Command Parsing Section deals with user input and how the system reacts to it. First, there is a main loop. In this loop, the system checks to see if there is any input on the socket using the ready() method. If there is input, it is read, otherwise, the UserConnection goes to sleep for a period of time and checks again. The input is then Tokenized. The first token of input is compared to the names of each command. If the token does not match the commands, the input is treated as regular communication and is written to the other users in the area. If a match is found, the user's level is compared to the minimum level required to execute the command. If the user satisfies the level requirement, the rest of the tokens are passed to the method in the Talker object corresponding to that command. After the input has been handled, the loop repeats. If the quit command is received, the loop breaks, which results in the user's settings getting saved to the database. The UserConnection terminates and the UserConnection array closes the socket and resizes the array.

Talker Timer

TalkerTimer deals with connection times of the users. The primary function of the TalkerTimer is to save resources by terminating connections that may be disconnected. Every 60 seconds, TalkerTimer runs through the array of UserConnections and increments their logon time and idle time. Next, the idle times are compared against thresholds to determine whether or not the user should be disconnected. If a user executes a command or sends any input to the system, the idle time is reset to zero. Usually, if a UserConnection hasn't responded in 60 minutes, the UserConnection is considered dead and the objects representing it are terminated.

Chatlet

Chatlet is the client of this Client-Server system. Chatlet is a Java Swing JApplet that provides a simpler and more friendly interface to less computer savvy users. Chatlet has three parts, the Chatlet Applet, the SocketPrinter and the Web-side Functionality.

The Chatlet Applet portion simply sets up the client. We Initialize the JApplet interface and connect to the server. We read the applet parameters and pass them to the Java Talker Server. We send a control code to the Java Talker to disable the printing of the ASCII color codes and then pass the username and password parameters for logging in. There is one main scrolling JTextArea and a JTextField. Data received from the SocketPrinter is displayed in the JTextArea. Data read from the JTextField is sent to the SocketPrinter.

The SocketPrinter works in a similar manner as the Talker Server's UserConnection. Input from the Chatlet Applet. The SocketPrinter reads and writes user data on the Socket.

The Web-side Functionality provides some of the common features in a friendly web page format for less Talker savvy users. Initially, when a user connects to the Chatlet page, a username and password box is displayed. The user must type a username and password and be authenticated against the database. If authenticated, the username and password are provided via parameters to the Chatlet Applet. In addition to providing the username and password, hyperlinks to area message boards are provided. Links to usernames provide the output of the .examine command when clicked. Lastly, there is a link that will let a user review all talker mail.

SetupDatabase

SetupDatabase is a program that I wrote to build and populate a database with which to run the Java Talker. In order to use it, you must have access to a MySQL database with the ability to create, update, insert and delete tables and rows. Additionally, the database username and password should be set in the setupDatabase program along with the database name. These should be consistent with the names that are used in the Java Talker Server and Chatlet source codes prior to compilation. The program creates the essential tables and populates certain tables with data. The Help table is populated with records for providing Online Help. The Area table and Messages tables are populated with some default areas and a welcome message. Finally, the User table is populated with a default user for initial administration purposes. It is recommended that anyone using this program to build their databases change the password of the default user after setting it up.

Startup Script

The Startup Script became necessary to start a Java Talker instance and allow a user, specifically me, to log out of my shell account while the Java Talker continued to run. The Java Talker startup script is for Unix and Linux based systems. It uses two different utilities, the AT scheduler and the Perl Interpreter to pull the time from the system and then use it to schedule the Java Talker to start. It is necessary to start Java servers on Linux and Unix in this manner because the Java Interpreter does not allow processes to fork, or spawn new processes. If you start the Java Talker and then attempt to disconnect from the command line shell that you started it from, you will either cause the Java Talker to terminate because its parent shell is closing, or your connection will not close, pending the termination of the Java Talker server. Since I must rely on a less-reliable Internet service, I can not keep a continuous connection to the server open. The Startup Script is a workaround for this. We use the AT scheduler to schedule the Java Talker to start in one minute from the time called. The AT scheduler creates a new shell instance and starts the Java Talker as a new process in that shell. The Java Talker Server is linked to the Unix system's main processes by way of the AT scheduler. If the Unix system is shut down, the Java Talker will go down, but otherwise, it should remain running. Shutting down the Java Talker after having started it in this manner is done by connecting to the Java Talker and issuing the .shutdown command. If necessary, the Talker Server can also be terminated forcibly by issuing the Unix command, kill.

The Startup Script requires access to a Perl Interpreter, since the program is written in Perl, the Java Interpreter to launch the Java Talker and the AT scheduler, to schedule the Talker. The Startup Script does a few things to accomplish its task. Because it uses the AT scheduler, we have to include the Java Talker Server's command in the AT call. I decided to have the Startup Script generate a new script and have AT call that one. It pulls the system time and then figures out one minute from that time, and executes the AT call to occur at that time. The Perl source code for the Startup Script can be found in Appendix E.

IX. Java Talker System Functionality

In this section, I will discuss the user experience and command functionality. The Java Talker Server and the Chatlet Interface are the only parts of the Java Talker that will be covered in this section. First, I will discuss the Talker Server. There are two main sections concerning messaging commands and environmental commands. Secondly, since the Chatlet interface can use all of the commands as well, I will only discuss unique Chatlet features in the Chatlet section.

Java Talker Commands

There are two types of commands on the Java Talker, the Messaging commands and the Environmental commands. Environmental commands deal with viewing the status of users and areas, viewing personal settings and information, and changing virtual areas. Messaging commands deal with communication between users. There are four types of messaging, Online Public, Online Private, Offline Public and Offline Private. I will discuss the different types of commands in detail in their appropriate sections following.

Online Public Messaging commands send messages that are readily available for other users. The most popular command is the implied .say command. By implied, I mean that if you don't explicitly specify a command, the .say command is called. Any parameters passed to the .say command are passed directly to the other users in the virtual room of the calling user. .Emote works in the same manner. Emote is different because it takes a third-person view of the user, so one could use the emote command to show emotion or action. The .review command shows the last 20 lines of public conversation that took place in the area. The .afk, or Away from Keyboard, command displays a message to the other users in an area that a user will not be listening for a short period of time, because that user is presently going away from the computer. Finally, .cbuffer is used to remove all of the public communication from the buffers used by .review. This is used to make public messages private.

Online Private Messages are implemented using the .tell and .remote commands. These commands are used to send messages privately to users online. Tell works like the .say command, except you pass it a parameter specifying the user to whom you would like the message to go. Other users on the system do not receive this message. Remote works like .emote, only you show action or emotion to the specified recipient only.

Offline Public Messages are sent and stored in Areas. Users connecting to the system at different times still have access to the message board structures on the Talker. The .read and .write commands allow users to read and write messages to a message board in a public area. Additionally, the .topic command can set topics to rooms so users will know what conversation types took place in a particular room. The .wipe command is used to delete messages from message boards. This is also to make public messages private by removing them.

The last set of Messaging commands is the Offline Private messages. These commands are implemented as a mailbox. If a user is sent an Offline Private message, it goes into the mailbox table that the recipient can read messages from.

The .smail command is used to send messages from one user to another, while the . rmail command is used to read a user's own mail. The .dmail command deletes mail messages from the system.

The other type of command is the Environmental commands. The most popular Environmental command is the .quit command, as most users tend to log off the system after a while. The .help command provides access to the Online Help system, which accepts a keyword and provides specific help relating to that topic. Users can find out information about individual users by using the .examine command. Examine shows a user's age, gender, homepage, email address, profile and other information. The .set command allows users to change their own age, gender, homepage, email address and other profile information. The .who command displays to a user a list of other users connected to the Java Talker system, while .idle shows the idle time of all of the users. Users can get information about the area or virtual room they are in by using the .look command. The .go command is the command that lets users change public areas. Finally, the .version command displays information about the particular build of the Java Talker.

Chatlet Specific Features

The Chatlet Web Interface provides a command line with which a web user can send talker commands, so all commands that work when connected directly to the Java Talker, through port 2122, continue to work. The Chatlet Web Interface provides some additional features that are more appealing to less technical Talker users. The Chatlet provides the capability to authenticate using a web browser interface, read web-based message boards, examine individual user stats and read their mail.

The Chatlet Authentication takes place by providing an HTML form with text and password boxes. Upon submitting both username and password through these boxes, the Chatlet checks the database and authenticates the user. If authenticated, the Chatlet will pass the username and password to a Java Applet, which connects directly to the Talker server. The Chatlet will also generate links needed to view email, message boards, and user information.

Chatlet Message boards take all of the messages from the message table and display them in a neat HTML Table. The message sender's username is linked to a hyperlink of the User record. Clicking on it will bring up the examine interface.

Online Mail works in a very similar manner to the Message boards. The mail messages are displayed in an HTML table. Unlike Message boards, where we pass the area name to the message board methods to get the messages, we pass the username and password to the Online Mail methods by way of an HTML Post. This is because HTML Get requests are written directly to the web server logs. Post requests pass parameters by way of STDIN to the address, and are not written to the server logs. Writing usernames and passwords to the web server logs is a sure way to get them compromised.

Finally, the Examine User interface is the web based version of the online .examine command. User information like name, webpage, photo and profile are displayed in a neat HTML table. In the future, we may be able to take advantage of the HTML functionality and create more customized user pages and embed images and hyperlinks.

X. Chatlet Web Interface and Socket Interface Comparison

The Chatlet Web Interface is a new addition to the typical Talker implementation. The Chatlet uses web based technology to provide access to information that has historically been available through only the socket interface. In this section, I will compare the Chatlet Web Interface features to the Socket Interface.

First, the Chatlet provides access to the Socket interface. All of the commands available to the Socket Interface are also available on the Chatlet. The commands can be sent to the Java Talker by way of the Input Line, which is beneath the main Talker window on the Chatlet Applet. Commands are sent in exactly the same way, with the leading period, and are displayed in very much the same way. I decided not to implement colored text on the Chatlet Interface at this point in time. I ran into a few problems with the user interface, which I will cover later. I felt that these interface issues were more important to solve than adding additional features.

The next feature is the Examine command. This command allows users to view information about other users. When connected to the Socket interface, a user would need to type .examine Jkb, in a situation where they wanted to look up information about the user, Jkb. The information returned from the Talker would be printed out to the socket. The Chatlet allows a user to click on a hyperlink for the username of the user to be examined. The user's information is opened in a new window, so conversation may continue while the second window contains the profile information.

The .read command is also implemented as a Chatlet command. The Chatlet Read takes the contents of the message board and displays them as a neat HTML table. One of the advantages to displaying the data as a HTML table in a new browser window is that the message board is now printable. Unless you have a specialized Talker client that permits cut, paste, or print operations, this was not easily done prior to this.

.Rmail is implemented in the same manner as Chatlet Read. The difference between Chatlet Rmail and Chatlet Read is that the Read is done by sending the room name by way of an HTML Get request. The by product of this HTML Get is a line in the Web Server logfile containing the entire Get request. Since the Rmail is intended to be a Offline Private message, I wanted it a little more secure than the Offline Public message, Read. By using a Post, the request parameters are not written to the web server log. Additionally, one could guess by looking at the address line of the Chatlet the way to view the mail files and simply change the username being sent to the system and attempt to read other users' mail. For this reason, I was forced to retain the user's password to have that sent via POST as well. Only a correct username and password combination will reveal a user's Java Talker mail.

Finally, the Chatlet Web Interface provides an HTML means of authentication. Since users of other Internet services are familiar with Username and Password text boxes, I added one to the Chatlet Web Interface. When successfully authenticated, the username and password are sent directly to the Chatlet Java Applet and automatically log the user in. They are also stored in hidden HTML fields for the user to check mail if needed.

XI. Compatibility and Testing

In this section, I will talk about the different types of systems that the Java Talker system and components will work on. Where applicable, I will also discuss how the components were tested.

The first component is the Java Talker Server. In order to run the Java Talker Server on a system, you will need to have installed at least version 1.4.1 of the Sun Microsystems Java Runtime Environment. This version is required because I have compiled it for that that version. I have also compiled it successfully for 1.3. The Java Performance Tuning book discussed the performance advantages for using 1.4 over 1.3, so for that reason, I recommend it be run on a 1.4 Java Virtual Machine. The Java Talker uses a library called Connector/J, which is a JDBC driver for the MySQL database distributed by the MySQLAB company.

The Java Talker Server has been tested on SuSE Linux, Debian Linux and Slackware Linux. I have also tested the Java Talker server on Windows 2000 Professional and Windows 2000 Server. Testing the system was an ongoing process. As I implemented new features, I would go back and test existing commands and features to make sure they still worked. During my ongoing testing, I tested the system in a number of ways. First, I would create four connections to the Talker Server. I would have three users in a main room, with one in another room. One user would send a command to the Java Talker system. I would check the desired output for the one user on that one user's connection. The second user was the recipient of commands that involved a second user. This was usually only needed for Private Messaging commands. The third user was to watch the interactions taking place in the area. If messaging that was supposed to be private or personal-environmental, the third user would definitely not see messages. Finally, the fourth user, in the other virtual room would watch for messages extending their virtual area boundaries. In addition, I had the Java Talker console open, which is basically the Talker system log open to watch for exceptions and system messages. Then, to perform testing, I would run though the list of commands, trying variations in an attempt to confuse the system and reveal problems. At various points during development and presently, I have the latest version of the Java Talker Server available on the Internet and have provided a way for Dark Ages Users interested in testing the system to get access and perform their own tests. For the most part, the Java Talker Server performed as expected. There were a few instances where bugs were uncovered. Since the testing was ongoing, however, they were fixed almost immediately.

The MySQL database contains the user and area information about the Java Talker. The requirements for the MySQL database relating to this project were for a Windows or Unix based system, since it supported both. I was able to get the MySQL database installed on both Windows 2000 and Linux. Since the MySQL database is a third party product, I relied heavily on the fact the MySQL organization does their own testing. I did, however, test the database tables by sending bad data to the database. This was necessary to make the Java Talker perform its own data integrity checking prior to it being sent to the database. After putting the data checks into the Java Talker, using the escapeUserInput() method, there were no input related problems on the MySQL database.

The testing I described in the Java Talker Server section was done to both the Socket Interface and the Web Chatlet Interface. Using the Socket Interface, I tested using Windows Telnet and the popular Talker Client, Gmud. From Linux systems, I utilized the Linux telnet command. The Web Chatlet Interface was tested using the same techniques.

Because the Chatlet offers additional functionality to the normal Socket Interface, I tested the Chatlet's functionality on different web browsers using different Java Plugins. For the most part, Java Plugins with versions higher than 1.4.1 worked without problem with the Chatlet Applet part of the Web Interface. Java 1.3 versions often produced Security Exceptions. Because of this, I require the minimum Java Plugin to be 1.4.1.

The web scripting end of the Chatlet, was only tested on Linux systems. This is because I did not locate a PHP interpreter for the Windows Internet Information Server, although I am sure they probably exist. Since PHP is similar to Microsoft's ASP, I am fairly confident, that if needed, an ASP implementation of the web scripting end could be implemented.

Finally, the Talker Startup Script is for Linux and Unix based systems. The script is written in Perl, so we need a Perl interpreter. In addition, the AT scheduler must be available to the script, since it uses AT to schedule the Java Talker's startup. If these are unavailable, the Startup Script will not work.

XII. Project Challenges

During this project, I encountered some notable challenges that I will discuss in this section. Some of these challenges were good learning experiences, while others were simply stumbling blocks.

One of the challenges that I worked on for most of the project's duration was one of performance. Early in the Java Talker's development, I happened to be looking at the system performance indication and noticed that the CPU utilization was evenly divided among the Java Talker processes. Upon further investigation, I found that the processes were considered “Nice Processes”, or processes that yield the CPU when needed by other applications. Satisfied with this explanation, I continued working on the Java Talker. Several weeks later, I kept thinking about it until I decided this was not a good enough explanation. Even with a few users connected to the system, I was using considerable CPU. After implmenting checks to reduce the load on the system, like a check to make sure users can have at most one connection and checks for idle time, I purchased a book about Java Performance Tuning and read about different ways to improve performance and find bottlenecks. I discovered a big problem in the UserConnection data structure. This problem was directly related to the resolution of another challenge. I will discuss this next challenge and explain how the solution created this problem.

I was attempting to implement a way to have the system manually disconnect users and conserve resources. At this point in time, I was also concerned with the resource utilization of the Java Talker Server. Additional inspiration was provided by a friend who insisted that Java was wasteful and sluggish, and I was determined to prove him wrong. I reasoned that if I could detect whether or not a user was disconnected, I could disconnect that user and free up the resources in use by its corresponding UserConnection object. Unfortunately, Java 1.4.2 does not yet have the capability to detect whether a socket is still open on the client end of the connection. I decided for another approach. I created a class called TalkerTimer, which keeps track of the amount of time a user is connected to the system, and the amount of time between input, or idle time. Using the idle time as a threshold, I would check the UserConnection record for idle time, manually disconnect the UserConnection's socket, and delete the UserConnection. When I programmed the TalkerTimer, I found that it did not work. If a UserConnection received the call to be terminated, the connection would remain open and the object instantiated until the Enter button on the client was pressed. At that point, the connection would immediately close. Upon further investigation, I found that several of the Socket methods block. Blocking is when a program does not permit any other input until the present system call is satisfied. In the UserConnection object, I had a call to in.readLine(), which is the BufferedReader reading a line from a socket. If no data is detected, it waits until some data is detected and then processes it. In order to solve this problem, I made an additional check. I added a call to the BufferedReader ready () method, which returns true if there is data available to read. When ready() was satisfied, I would call the blocking readLine() method. After implementing this, TalkerTimer worked perfectly, disconnecting idle UserConnections immediately after the idle thresholds were reached.

Prior to fixing this problem with the blocking on the readLine() call, I did not really notice the performance of the Java Talker server. This was probably due to the fact that when the UserConnections blocked, the CPU did not continue processing until input was detected. The resolution of this problem left nothing to keep the CPU from processing, so it kept looping and using up the remainder of the available CPU. When I determined that this was taking place, mostly from learning in more detail about blocking calls from the Java Performance Tuning book, I asked the UserConnection threads to sleep for a fraction of a second at every iteration, and my CPU utilization problem was solved.

After solving the CPU utilization problem, I took on the challenge of better use of memory. The Java Performance Tuning book also discussed ways in which to optimize the use of memory in ways that made sense. For instance, if I was instantiating objects in a loop, I was continuously allocating memory, setting the pointers and letting the Java garbage collector clear the memory. I found two major instances in the UserConnection array of this wasteful use of memory. I reworked those parts of the program to use memory more efficiently, and upon execution of the Java Talker Server, I was rewarded with a smaller memory footprint.

I encountered two more challenges when I started testing the multiplatform features of the Java Talker. I installed the MySQL database on my Windows 2000 system and compiled the Java Talker. Things compiled without problem. When I started the server and connected to it to test the commands, I got an Exception, NoSuchMethod, on the call to getGeneratedKey(). The getGeneratedKey() method was a new method in Java 1.4.2 that provides the key after an INSERT operation on a database table that has an automatically incremented integer for the primary key. Apparently, the Windows 2000 version of Java 1.4.2 did not have it implemented. I worked around this issue by removing the call to getGeneratedKey() and replacing it with a SELECT call, asking for the primary key on a row that matches the other fields. The other challenge on Windows 2000 was the lack of the SQL encrypt() method. I was unable to encrypt the passwords or decrypt existing passwords. To get around this, I wrote a method that checks to see if encrypt() is implemented in the database, and if found, the Java Talker uses it. Otherwise, passwords are stored in plain text. On Windows 2000 systems, I leave the security end of the passwords up to the server administrators.

Since I am involved with computer security at my job, I was paying attention to challenges relating to computer security. I did run across a few instances. When dealing with the CPU utilization problems, I realized that many malicious user's connections could effectively choke out other processes on the CPU. By implementing some of the performance tuning recommendations and solving the blocking issue, I was able to overcome this challenge. I also found that certain commands that write to the database, like .write, .smail, and .set could be used to attack the database by using SQL Injection attacks. SQL injection attacks take place when a user sends a special character with meaning in SQL. MySQL uses the single and double quote in addition to the escape character to bound strings. If a malicious user included these characters in their .write, .smail, and .set calls, they could corrupt data, grant themselves privileges and overwrite harmless data with malicious. In my experimentation, I was able to use the SQL injection to inject into a harmless description change a change that would grant Talker Administrator privileges to a low level user. Prior to utilizing the SQL injection to perform a sophisticated attack that granted Administrator privileges, I was able to crash the Talker Server by sending bad data to the database access. I responded to this challenge by creating a method, escapeUserInput(), which escapes all special characters going into the database. After implementing this method, the SQL injection flaws went away. Finally, the last security challenge that I tackled was the exposed Mailbox password on the Chatlet Web Interface. I found that if I had a hyperlink that passed the user's password to the mail reader script, the username and password for the user were written to the web server log. Anyone with access to the log could potentially capture Talker usernames and passwords and commit identity theft crimes. This problem was solved by changing the mail request to a post request, so nothing was written to the web server log. Other security issues like the encryption not being implemented on Windows MySQL and the Plain Text, unencrypted Internet communication would have to go unsolved for this project.

I even had a physical disaster impact the Java Talker project. I decided to install a program called Umbrello, which allowed me to easily create UML diagrams. I attempted to install the software on the same SuSE Linux system on which I was developing the Java Talker. I unknowingly replaced some important dependencies in the process. This unfortunately corrupted the system administration tool, the Java executables and other critical system processes. I did have a backup handy, and I was still able to connect through telnet to the computer and retrieve my work, but this did set me back, as I was forced to recreate my development environment. I turned this into an opportunity, however. I decided to install a new distribution of Linux, Debian GNU Linux. This simply provided a new platform on which to test the Java Talker Server.

Another challenge involved the fact that the Java language does not permit processes to fork, or spawn child processes and leave them. Java wants users to use the built in multithreading when additional processing is needed. Because of this, I needed to write a special Startup Script for the Java Talker. Otherwise, I would have to leave my Internet connection open at all times to keep the Java Talker running.

Less exciting challenges appeared, too. I found the test version of the Talker would crash every few days. I wasn't able to find any problems on the development version, but every time I would replace it, after a few days, the test version would be crashed. I researched this phenomenon on the Connector/J website and found that Connector/J automatically closes the database connection after about 24 hours. I created a method called ConnectDB(), which restores the connection if it is unavailable. After this, I needed a way to create user accounts. I created a special mode in the Authentication section of the UserConnection that would walk a user through creating a new account. Finally, as a final challenge, I added color to the socket interfaces, and stripped any color codes from the Chatlet.

Finally, I had a problem with the Chatlet Web Interface. I originally wanted to implement a scrolling text box that would allow me to scroll back to review conversations, and copy text to the clipboard. Unfortunately, I was never able to get the scrolling to properly work. Often, the scrollbars would jump back to the beginning or skip lines. Ultimately, I ended up removing the scrollbar features, but I was unhappy that I was unable to overcome that challenge.

XIII. Conclusion

The Java Talker Project and all of its parts proved to be an interesting challenge with tangible results. I learned quite a bit about socket communication and performance tuning. I got to experience the design, planning, development and implementation of a large software development project. I encountered successes, failures and setbacks. In the end, I was able to produce a software system that I can utilize with my personal pursuits.

I believe that the Java Talker is an improvement over the classical NUTS based Talker. Reimplementing the system in Java can provide the opportunity to integrate newer technologies if I can see a proper application for them. I understand sockets and network communication at the Java Talker level and can add new functionality and explain to others how the system works. The system is also more robust, as Java brings to the table its garbage collection and buffer management. The system stores most of its data in a database, which can allow data integration with other applications. By making the system multithreaded, the Java Talker is scalable on machines with multiple processors, however, I've never had an occurrence where a single processor was not enough, under normal conditions. Finally, because I understand the system in its entirety, I created a store of web-based documentation using Javadoc.

Even though I feel the Java Talker Project was a success as a whole, I was disappointed with the outcomes in a few different areas. Two shortcomings that I encountered were related to planning. Initially, my project description was somewhat open-ended. Despite having a pretty good idea what I wanted, the boundaries of the project were vague. For this reason, I continued implementing new features and working toward a finished product, when I was looking for a framework upon which to build a finished product. Eventually, I had to just stop implementing new ideas. I had to address the requirements of the project and decide what I needed to finish for the project, and what was beyond the scope of it. I will discuss some of these new features in the next section. The other shortcoming was the length of time I took. Originally, I wanted to have the project complete by the middle of September. Now it is November when I am finishing the project. I believe I underestimated the workloads for classes that I took while also doing the project. I also believe that getting sidetracked on minor details attributed to vague project requirements used up time.

If I could go back to May, when I began the project, I would offer myself some advice. I would advise myself to take a little more time defining the specifics of what I wanted to deliver. I would also attempt to set milestones on the achievements. Even though this would probably result in fewer features being available for the user, I would have delivered everything that I defined and on time, rather than leaving the completeness of the project to interpretation over a month after the original delivery date.

XIV. The Java Talker Future

The Java Talker has satisfied the requirements that I set for this Graduate Student project. However, after I receive my grade on the project and ultimately receive my degree, the Java Talker's life will not be over. Even as the Java Talker's scope crept out of control, I continued having new ideas about things that I could implement to improve the system. In this section, I will discuss new ideas that I will one day implement.

The first set of ideas involve the Chatlet Web Interface. I really liked the ease of use that the HTML interfaces provided. I think it would be a very good idea to have the capability to change user settings through an HTML form, and have the Java Talker database updated directly with the settings. Additionally, I think it would be nice to add buttons to the Chatlet Applet that link to Talker commands. These two ideas would further abstract the NUTS commands from less technical users. I also would use the user administration HTML form to allow new users to be created without requiring them to understand the complexities of the NUTS setup commands. Finally, I would like to have another attempt at fixing the Chatlet Scrolling TextArea. I was never really satisfied with the implementation of it and maybe in the future, I will be able to solve this problem. At the same time, I could add color to the Chatlet as well.

From a system administration point of view, I think it would be valuable to have some web-based Talker administration interfaces. In particular, interfaces that would let Talker Administrators access Area records or Message Boards and purge messages or update area descriptions. I would also like to have messages printed to a System Log table. Using this System Log table, Talker administrators could view system information while connected to the Talker itself. Finally, I would like to add more flexibility and options to running the Talker. This would possibly be done using an XML configuration file. Using this file, an administrator could specify ports, databases, whether or not to use encryption, or other implementation-specific settings. In addition, I would like to add additional JDBC database drivers. Like the Java Socket API, the JDBC calls use a higher level of abstraction than MySQL's C libraries. For this reason, we could simply add a JDBC driver for PostgreSQL, Oracle or Microsoft SQL Server, and then tell the Java Talker to use the new JDBC Driver
Not everyone has access to MySQL, but databases like PostgreSQL, Oracle and Microsoft SQL Server have JDBC drivers available. Possibly the database choice can be made in the XML configuration file.

Even after reimplementing the Talker concept in Java, the communication takes place in the same unencrypted fashion as always. I think this could be addressed by creating a Java Talker specific client that can send a public encryption key to the Java Talker server and encrypt the communication based on the public and private keys. Using the Chatlet Web Interface on an SSL supported server will, however, encrypt the user communication. While on the topic of security, the Java Talker database stores its mail data in plain text in the mail table. It might be a good idea to consider encrypting these stored messages. In addition, on the Windows 2000 implementation of the Java Talker, the database does not use encryption for passwords. I think it would be a good idea to pursue this feature with the MySQL development list or find an open source database that will allow the use of encryption functions on Windows 2000. Finally, to make the system more robust, I would like to utilize the return values from many of the Talker methods to determine whether or not additional error checking needs to take place.

To integrate the Java Talker with other systems, I may consider supporting some other means of messaging, like Java SOAP. I also have considered using JavaServer Pages or Java Servlets instead of PHP for the Chatlet Web Interface.

Finally, I will ultimately move the existing user base on the Dark Ages Talker from the NUTS Talker platform to the Java Talker. When I determine the additional commands and features that should be implemented and actually implement them, I can replace the existing NUTS Talker with the Java Talker. Then, at some point, I would like to release the Java Talker source code to the open source community, where it may hopefully continue to grow and become a valuable learning experience for someone else as well.

XV. References

Bell, Douglas; Parr, Michael. Java for Students, 3rd Edition. New York: Prentice Hall, 2002.

Booch, Grady; Jacobson, Ivar; Rumbaugh, James. The Unified Modeling Language User Guide. Boston: Addison-Wesley, 1999.

Flanagan, David. Java Examples In a Nutshell. A Tutorial Companion to Java in a Nutshell.   Cambridge: O'Reilly and Associates, Inc., 1998.

Flanagan, David. Java in a Nutshell: A Desktop Quick Reference, 2nd Edition. Cambridge: O'Reilly and Associates, Inc., 1998.

Gay, Warren W. Linux Socket Programming By Example. Indianapolis: Que Publishing, 2000.

Gittleman, Art. Advanced Java: Internet Applications, 2nd Edition. El Granada: Scott/Jones Publishers, 2002.

Gittleman, Art. Objects to Components with the Java Platform. El Granada: Scott/Jones Publishers, 2000.

Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd Edition. Upper Saddle River: Prentice Hall, Inc., 2002.

Laurie, Ben; Laurie, Peter. Apache: The Definitive Guide, 2nd Edition. Cambridge: O'Reilly and Associates, Inc., 1999.

Lerdorf, Rasmus; Tatroe, Kevin. Programming PHP. Cambridge: O'Reilly and Associates, Inc., 2002.

Lewis, John; Loftus, William. Java Software Solutions: Foundations of Program Design. Reading: Addison-Wesley Publishing Company, 1998.

Liang, Y. Daniel. Introduction to Java Programming with JBuilder, 2nd Edition. Upper Saddle River: Prentice Hall, 2002.

MKS Software. “File Descriptors: Synopsis of select() Function.”. http://www.mkssoftware.com/docs/man3/select.3.asp (03 Aug 2001).

Nemeth, Evi; Snyder, Garth; Seebass, Scott; Hein, Trent R. Unix System Administration Handbook, 3rd Edition. Upper Saddle River: Prentice Hall PTR, 2001.

O'Neil, Patrick; O'Neil, Elizabeth. Database: Principles, Programming, and Performance, 2nd Edition. New York: Morgan Kaufmann Publishers, 2001.

Pantham, Dr. Satyaraj. Pure JFC Swing: A Code-Intensive Premium Reference. Indianapolis: Sam's Publishing, 1999.

Peterson, Larry L; Davie, Bruce S. Computer Networks: A Systems Approach, 2nd Edition. New York: Morgan Kaufmann Publishers, 2000.

Robertson, Neil. “NUTS home page”. neil@ogham.demon.co.uk. http://www.ogham.demon.co.uk/nuts.html (16 Dec 1997).

Shirazi, Jack. Java Performance Tuning, 2nd Edition. Cambridge: O'Reilly and Associates, Inc., 2003.

Silberschatz, Abraham; Galvin, Peter B. Operating System Concepts, 4th Edition. Reading: Addison-Wesley Publishing Company, 1995.

Skoudis, Ed. Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses. Upper Saddle River: Prentice Hall, Inc., 2002.

Sobell, Mark G. A Practical Guide to the Unix System, 3rd Edition. Reading: Addison-Wesley Publishing Company, 1995.

Somerville, Ian. Software Engineering, 5th Edition. New York: Addison-Wesley Publishing Company, 1996.

Wall, Larry; Christiansen, Tom; Schwartz, Randal L.. Programming Perl, Second Edition. Cambridge: O'Reilly and Associates, Inc., 1996.

Walton, Sean. Linux Socket Programming. Indianapolis: Sam's Publishing, 2003.

Wu, C. Thomas. An Introduction to Object-Oriented Programming with Java, 3rd Edition. Boston: McGraw Hill Higher Education, 2004.

Yahoo! Chat. http://chat.yahoo.com/ (23 Nov 2003).

Yarger, Randy Jay; Reese, George; King, Tim. MySQL & mSQL. Cambridge: O'Reilly and Associates, Inc., 1999.

Appendix

Table of Contents:

Appendix A:

Java Talker Server Source Code

This Appendix contains the source code used to build the Java Talker Server.

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.*;
import java.sql.Connection;
import java.text.*;

/** UserConnection Object
        @author Jeffrey K. Brown
        <p>The UserConnection is a threaded object that continually monitors an input
        stream of a socket from the Talker and prints to an output stream of a socket.
        Input to the UserConnection from the user is checked for command names which
        determine what type of communication is taking place.  The different commands
        execute different methods in the Talker object.
        </p>
*/

class UserConnection extends Thread {
        /** This is the Socket on which the connection to the Talker is made. */
        public Socket socket;

        /** This is a BufferedReader that reads the InputStream from
        socket and passes it to the Talker. */
        public BufferedReader in;

        /** This is a PrintWriter used to print messages from the Talker
        to the user's socket. */
        public PrintWriter out;

        /** This is used by the Talker to identify the position in the
        UserConnection array the current UserConnection has. */
        int index=0;

        /** This serves as a link back to the main Talker object, so the
        individual users can share the set of data in Talker */
        Talker t;

        /** This String is used to store SQL queries prior to passing
        them to the database. */
        String sql;

        /** This variable is used by the Talker to identify which virtual
        area or room the user is in. */
        int area;

        /** This is used by the UserConnection to determine whether or
        not to terminate its internal thread. */
        boolean quit = false;

        /** This is a boolean used to designate a connection that has
        been dropped because the user has logged on again */
        boolean replaced = false;


        /** This is a boolean which helps the system determine if the
        PrintWriter out is ready to be written. */
        boolean outset=false;

        /** This is a boolean that helps determine whether or not to
        break out of the login loop and into the normal Talker loop. */
        boolean authenticated=false;

        /** This is the User Name */
        private String name = "Unnamed";

        /** This is the User's description */
        private String desc = "A Generic Desc";

        /** This is the User's password */
        private String password = "";

        /** This is the number of minutes of the current connection. */
        private int minutes = 0;

        /** This is the number of minutes since the last input from the user. */
        private int idle = 0;

        /** The level is used to designate what commands the user has access to. */
        private int level = 0;

        /** The profile is a place where the users can put a
        few paragraphs describing themselves. */
        private String profile = "This user has not yet set a profile";

        /** This is used to keep a running count of the total
        number of minutes on the system. */
        private int totaltime = 0;

        /** This is used to tell males from females.*/
        private String gender = "Unknown";

        /** This is used to show the user's age */
        private int age = 18;

        /** This stores the user's email address. */
        private String email = "me@noneofyourbusiness.com";

        /** This stores the URL of the user's homepage */
        private String homepage = "darkages.darkfantastic.net/";

        /** This stores the URL of the user's picture */
        private String photo = "darkages.darkfantastic.net/none.jpg";

        /** This is used by the system to determine whether or not to print colors */
        private int colorpref = 2;

        /** This is used to store the last location a user signed on from. */
        private String lastsite = "This user has never been here before";

        /** This variable is used to store the date the user last signed on. */
        private String laston = "Never";

        /** This is used to determine whether or not the user
        is connected through the Chatlet. */
        private boolean chatlet=false;

        /** The tell[] array stores the last 20 online private
        messages for reviewing. */
        String[] tell = new String[20];

        /** This boolean shows the user is away from the keyboard if true */
        boolean afk=false;

        /** If a user is away from keyboard, and this is true, an
        online private message was received. */
        boolean afktell=false;

        /** A user can specify a reason why they are going AFK if
        they desire.  It's stored here. */
        String afkmsg = "ZZ... Z... z...";

        /** The quit() method sets the quit boolean, which is used
        to terminate the UserConnection Thread. */
        public void quit() { quit = true; }

        /** addMinute() increments the running count of minutes and
        idle minutes for timing purposes. */
        public void addMinute() { ++minutes; ++idle; }
```

```java
/** The getIdle method returns the number of idle minutes.
        @return integer number of idle minutes */
public int getIdle() { return idle; }

/** The getTime() method returns the numbef of minutes the
user has been connected.
        @return integer number of minutes connected. */
public int getTime() { return minutes; }


/** isAuthenticated() returns a boolean that lets the calling method know whether
or not a UserConnection is authenticated or not.  Often used by methods printing output
to multiple UserConnections.
        @return boolean*/
public boolean isAuthenticated() { return authenticated; }

/** UserConnection Constructor.  In the constructor, we initialize a lot of the variables
used by the UserConnection.
        @param t The UserConnection constructor takes a link back to the calling Talker object.
*/
public UserConnection(Talker t) {
        int i=0;
        this.t = t;
        area = 0;

        for (i=0; i<20; ++i) tell[i]="";
}

/** setArea is called to change the virtual area of a user to a new one.
        @param area is an integer, which stores the index in the Area
        array of the new virtual area. */
public void setArea (int area) {
        this.area = area;
}
/** getArea is called by methods in Talker that need to determine where the user is located.
        @return area, an integer index of the virtual room in the Area array
        in which the user is located. */
public int getArea () {
        return area;
}

/** getIndex is called by methods in Talker that need to know the Index of
        this particular UserConnection.
        @return index, an integer index of this UserConnection in the
        UserConnection array. */
public int getIndex() {
        return index;
}

/** setIndex is used by the Talker to set the UserConnection's index. It is called when resizing
the UserConnection array, or when the user initially connects.
        @param index an integer index for the UserConnection to take on. */
public void setIndex(int index) {
        this.index = index;
}

/** getPort() is used to determine which port the UserConnection is connected from.
This is used along with GetSite and GetInetAddress.
        @return integer representing the port. */
public int getPort() {
        return socket.getPort();
}

/** getInetAddress() returns the name or IP address from which the UserConnection originated.
        @return InetAddress of the site's DNS name or IP address */
public InetAddress getInetAddress() {
        return socket.getInetAddress();
}
```

```java
        /** getSite() returns the port and address from which the user has connected.
                @return String containing the site and port. */
        public String getSite() {
                String output = ""+getInetAddress()+", Port: "+getPort();

                return output;
        }

        /** getSocket() returns the Socket on which the UserConnection is made.
                @return Socket of the user. */
        public Socket getSocket() {
                return socket;
        }

        /** getTotalTime() returns the Total number of minutes the user has logged on the system,
         including the minutes of the current UserConnection.
                @return integer containing the number of minutes. */
        public int getTotalTime() {
                return totaltime + minutes;
        }

        /** getAfk() returns a boolean showing whether or not the user is AFK.
                @return boolean designating the AFK status, true or false. */
        boolean getAfk() {
                return afk;
        }

        /** setAfk() is a toggle that either marks a user AFK or not.  This is called by the .afk command.
                @param afk boolean whose value replaces the current afk boolean value. */
        void setAfk(boolean afk) {
                this.afk = afk;
        }

        /** setAfkMsg() sets the message that is displayed when a user sends an online
        private message to an AFK user.
                @param msg is a String containing the message */
        void setAfkMsg(String msg) {
                afkmsg = msg;
        }

        /** getAfkMsg() returns the message to be displayed while the user is AFK.
                @return afkmsg is the String containing the AFK message. */
        String getAfkMsg() {
                return afkmsg;
        }

        /** setAfkTells() marks the UserConnection as having received online private messages while AFK.
                @param afktell is the boolean with the afktell status. */
        void setAfkTells(boolean afktell) {
                this.afktell=afktell;
        }

        /** getAfkTells() returns the boolean value with the afktell status, either the user has
        received tells while AFK or not.
                @return afktell which is the status. */
        boolean getAfkTells() {
                return afktell;
        }

        /** setEmail() sets the UserConnection email value to the string passed as a parameter.
        If the parameter string is longer than 60, it is truncated.
                @param email a string containing the new value for the email address. */
        void setEmail(String email) {
                if (email.length()>60) {
                        this.email = email.substring(0,60);
                }
                else {
                        this.email = email;
                }
        }

        /** getEmail() returns the value set for the UserConnection email address.
                @return email which is the the String containing the email address. */
        String getEmail() {
                return email;
        }
        /** setHomepage() sets the UserConnection homepage value to the value passed as a
        parameter.  If the new value is longer than 60 characters, it is truncated.
                @param homepage String containing the homepage value. */
        void setHomepage(String homepage) {
                if (homepage.length()>60) {
                        this.homepage = homepage.substring(0,60);
                }
                else this.homepage = homepage;
        }

        /** getHomepage() simply returns the UserConnection homepage value.
                @return homepage which is a String containing the homepage value. */
        String getHomepage() {
                return homepage;
        }

        /** setPhoto() sets the UserConnection photo value to the one in the parameter.
        If the new value is longer than 60 characters, it is truncated.
                @param photo contains the new value for the photo */
        void setPhoto(String photo) {
                if (photo.length()>60) {
                        this.photo = photo.substring(0,60);
                }
                else this.photo = photo;
        }
        /** getPhoto() simply returns the UserConnection photo value.
                @return photo, the String containing the value for photo. */
        String getPhoto() {
                return photo;
        }

        /** setDesc() sets the value of the parameter to the UserConnection desc.
        If it is longer than 40, it is truncated.
                @param desc which is a string containing the new description value. */
        void setDesc(String desc) {
                if (desc.length()>40) {
                        this.desc = desc.substring(0,40);
                }
                else this.desc = desc;
        }

        /** getDesc() returns the value of the UserConnection desc.
                @return desc which is the UserConnection description value. */
        String getDesc() {
                return desc;
        }

        /** setAge() sets the UserConnection age to the parameter value.
                @param age the integer representing the new age value. */
        void setAge(int age) {
                this.age = age;
        }

        /** getAge() returns the UserConnection age value.
                @return integer containing the age. */
        int getAge() {
                return age;
        }

        /** setLevel() is called by the .promote and .demote commands to change the UserConnection's
        level to the one in the parameter.
```

```
                @param level is an integer containing the new value for level. */
        void setLevel(int level) {
                this.level = level;
        }

        /** getLevel() returns the value of the UserConnection's level.
                @return level which contains the value of the UserConnection level. */
        int getLevel() {
                return level;
        }

        /** getUserName() returns the value of the UserConnection's username.
                @return name, which is a String containing the username. */
        public String getUserName() {
                return name;
        }

        /** setUsername() sets the UserConnection's username value.
                @param name is the new value for UserConnection name.*/
        public void setUserName(String name) {
                this.name = t.escapeUserInput(name);
        }

        /** getUserDesc() returns the UserConnection desc value.
                @return desc which is a String containing the UserConnections description. */
        public String getUserDesc() {
                return desc;
        }

        /** setPassword() sets the UserConnection password to the value passed.
                @param password String containing the new password. */
        void setPassword(String password) {
                this.password = password;
        }

        /** getPassword() returns the UserConnection password.
                @return password is the UserConnection password. */
        String getPassword() {
                return password;
        }

        /** setProfile() sets the UserConnection profile to the value passed to the method.
                @param profile is a String containing the new Profile. */
        void setProfile(String profile) {
                this.profile = profile;
        }

        /** getProfile() returns the String value of the user profile.
                @return profile is a String containing the profile. */
        String getProfile() {
                return profile;
        }

        /** setGender() sets the UserConnection gender to the value passed as a parameter.
        This is called from the .set command, which only permits "Male", "Female", or "Unknown".
                @param gender containing the new gender to be set. */
        void setGender(String gender) {
                this.gender = gender;
        }

        /** getGender() returns the value set for gender in the UserConnection object.
                @return gender is a String containing the gender. */
        String getGender() {
                return gender;
        }

        /** addTell() adds online private messages to the tell[] array.  The tell array stores
        20 messages.  If there are more than 20, the oldest ones are dropped off the end to
        make room for the new.
```

```
                @return boolean indicating whether or not there was an error.
                @param line which is a String containing the online private message to
                add to the tell buffer. */
        public boolean addTell (String line) {
                int i=0;

                for (i=0; i<20; ++i) {
                        if (tell[i].equals("")) {
                                tell[i] = line;
                                return false;
                        }
                }

                for (i=0; i<19; ++i) {
                        tell[i] = tell[i+1];
                }
                tell[19] = line;
                return false;
        }

        /** showTell() displays the contents of the tell buffer.
                @return boolean indicating whether or not there was an error. */
        public boolean showTell () {
                int i=0;
                println("~FYYour Private Messages:~N\n");

                for (i=0; i<20; ++i) {
                        if (tell[i].equals("")) break;
                        println(tell[i]);
                }
                println("");

                return false;
        }

        /** clearTell() clears the tell buffer by replacing the online private messages with empty strings.
                @return boolean indicating whether or not an error has occurred. */
        public boolean clearTell () {
                int i=0;
                println("~FYTell Buffer cleared.~N");

                for (i=0; i<20; ++i) {
                        tell[i] = "";
                }
                return false;
        }

        /** getLevelName returns a string based on the level and gender of the UserConnection.
                @return String containing the value of the Level Name. */
        String getLevelName() {
                if (getLevel()==0) {
                        if (getGender().equals("Male")) return "Peasant";
                        else if (getGender().equals("Female")) return "Peasant";
                        else return "Visitor";
                }
                if (getLevel()==1) {
                        if (getGender().equals("Male")) return "Knight";
                        else if (getGender().equals("Female")) return "Lady";
                        else return "Subject";
                }
                if (getLevel()==2) {
                        if (getGender().equals("Male")) return "Cleric";
                        else if (getGender().equals("Female")) return "Mystic";
                        else return "Mage";
                }
                if (getLevel()==3) {
                        if (getGender().equals("Male")) return "Wizard";
                        else if (getGender().equals("Female")) return "Sorceress";
                        else return "Chanter";
                }
                if (getLevel()==4) {
                        if (getGender().equals("Male")) return "Prince";
                        else if (getGender().equals("Female")) return "Princess";
                        else return "Heir";
                }
                if (getLevel()==5) {
                        if (getGender().equals("Male")) return "King";
                        else if (getGender().equals("Female")) return "Queen";
                        else return "Royal";
                }
                return "Visitor";
        }

        /** getLevelName() returns a String containg the Level Name based on the
        specificed level and gender.
                @param level integer containing the level.
                @param gender String containing the value of the gender.
                @return String containing the Level Name */
        String getLevelName(int level, String gender) {
                if (level==0) {
                        if (gender.equals("Male")) return "Peasant";
                        else if (gender.equals("Female")) return "Peasant";
                        else return "Visitor";
                }
                if (level==1) {
                        if (gender.equals("Male")) return "Knight";
                        else if (gender.equals("Female")) return "Lady";
                        else return "Subject";
                }
                if (level==2) {
                        if (gender.equals("Male")) return "Cleric";
                        else if (gender.equals("Female")) return "Mystic";
                        else return "Mage";
                }
                if (level==3) {
                        if (gender.equals("Male")) return "Wizard";
                        else if (gender.equals("Female")) return "Sorceress";
                        else return "Chanter";
                }
                if (level==4) {
                        if (gender.equals("Male")) return "Prince";
                        else if (gender.equals("Female")) return "Princess";
                        else return "Heir";
                }
                if (level==5) {
                        if (gender.equals("Male")) return "King";
                        else if (gender.equals("Female")) return "Queen";
                        else return "Royal";
                }
                return "Visitor";
        }

        /** parseColors() is used to put color codes into the Java Talker.  Parsecolors is hard
        coded with the ASCII color codes.  When a user puts a color code, for example ~FR for
        foreground red, parseColors does a string replace of ~FR with the ASCII color code for
        red.  This is called by the print and println methods of UserConnection.
                @param line is a String containing the line to be parsed.
                @return line is a String containing the parsed line. */
        public String parseColors(String line) {

                String RED    = "\033[31m\033[1m";
                String GREEN  = "\033[32m\033[1m";
                String YELLOW = "\033[33m\033[1m";
                String BLUE   = "\033[34m\033[1m";
                String PURPLE = "\033[35m\033[1m";
                String SKY    = "\033[36m\033[1m";
                String WHITE  = "\033[37m\033[1m";
                String BRED   = "\033[41m\033[1m";
```

```
            String BGREEN = "\033[42m\033[1m";
            String BYELLOW= "\033[43m\033[1m";
            String BBLUE  = "\033[44m\033[1m";
            String BPURPLE= "\033[45m\033[1m";
            String BSKY   = "\033[46m\033[1m";
            String BWHITE = "\033[47m\033[1m";
            String NORMAL = "\033[0m";
            String BLINKY = "\033[5m";
            String UNDER  = "\033[4m";

            line = line.replaceAll("~FR", RED);
            line = line.replaceAll("~FG", GREEN);
            line = line.replaceAll("~FY", YELLOW);
            line = line.replaceAll("~FB", BLUE);
            line = line.replaceAll("~FP", PURPLE);
            line = line.replaceAll("~FM", PURPLE);
            line = line.replaceAll("~FS", SKY);
            line = line.replaceAll("~FW", WHITE);
            line = line.replaceAll("~BR", BRED);
            line = line.replaceAll("~BG", BGREEN);
            line = line.replaceAll("~BY", BYELLOW);
            line = line.replaceAll("~BB", BBLUE);
            line = line.replaceAll("~BP", BPURPLE);
            line = line.replaceAll("~BM", BPURPLE);
            line = line.replaceAll("~BS", BSKY);
            line = line.replaceAll("~BY", BWHITE);

            line = line.replaceAll("~UL", UNDER);
            line = line.replaceAll("~LI", BLINKY);
            line = line.replaceAll("~N", NORMAL);

            return line;
    }

    /** clearColors() is used to remove color codes from input into the Java Talker.
    For clients who don't support colors, this is a means of stopping the ASCII codes
    from being printed.  Color codes are replaced with empty strings, so the default
    text color is used.  This is called by the print and println methods of
    UserConnection.
            @param line is a String containing the line to be parsed.
            @return line is a String containing the parsed line. */
    public String clearColors(String line) {

            line = line.replaceAll("~FR", "");
            line = line.replaceAll("~FG", "");
            line = line.replaceAll("~FY", "");
            line = line.replaceAll("~FB", "");
            line = line.replaceAll("~FP", "");
            line = line.replaceAll("~FM", "");
            line = line.replaceAll("~FS", "");
            line = line.replaceAll("~FW", "");
            line = line.replaceAll("~BR", "");
            line = line.replaceAll("~BG", "");
            line = line.replaceAll("~BY", "");
            line = line.replaceAll("~BB", "");
            line = line.replaceAll("~BP", "");
            line = line.replaceAll("~BM", "");
            line = line.replaceAll("~BS", "");
            line = line.replaceAll("~BY", "");

            line = line.replaceAll("~UL", "");
            line = line.replaceAll("~LI", "");
            line = line.replaceAll("~N", "");

            return line;
    }

    /** println() prints lines using println to the PrintWriter (user output) that
    has either been passed through the parseColors, enabling ASCII text, clearColors,
```

```java
stripping out ASCII text, or nothing, enabling client-side color parsing.
        @return boolean indicating whether or not there has been an error.
        @param line which is the input line needing to be parsed. */
public boolean println(String line) {
        if (colorpref==2) {
                line = parseColors(line);
        }
        if (colorpref==1) {
                // Do nothing...keep color codes...
        }
        if (colorpref==0) {
                line = clearColors(line);
        }
        out.println(line);
        return false;
}

/** print() prints lines using print (no newline) to the PrintWriter (user output)
that has either been passed through the parseColors, enabling ASCII text,
clearColors, stripping out ASCII text, or nothing, enabling client-side color
parsing.
        @return boolean indicating whether or not there has been an error.
        @param line which is the input line needing to be parsed. */
public boolean print (String line) {

        if (colorpref==2) {
                line = parseColors(line);
        }
        if (colorpref==1) {
                // Do nothing...keep color codes...
        }
        if (colorpref==0) {
                line = clearColors(line);
        }

        if (chatlet==true) {
                // Chatlet doesn't like the manual buffer flush...
                out.println(line);
        }
        else {
                out.print(line);
        }
        return false;
}

/** loadUserData() makes a database access and reads a user record.  The values from
the user record populate the UserConnection object based on the authenticated user name.
        @return boolean indicating whether or not the loadUserData() method was
        successful */
boolean loadUserData() {
        t.connectDB();
        String sql="";
        try {
                sql = "SELECT description,level,profile,totaltime,gender,age,email,"
                        +"homepage,photo,colorpref,lastsite,laston FROM user WHERE "
                        +"username='"+name+"'";

                ResultSet rs = t.stmt.executeQuery(sql);
                if (rs.next()) {
                        desc = rs.getString(1);
                        if (desc==null) { desc = "A Generic Desc"; }
                        level = rs.getInt(2);
                        profile = rs.getString(3);
                        if (profile==null) { profile = "This user has not yet set a profile"; }
                        totaltime = rs.getInt(4);
                        gender = rs.getString(5);
                        if (gender==null) { gender = "Unknown"; }
                        age = rs.getInt(6);
                        email = rs.getString(7);
```

```
                                if (email==null) { email = "me@noneofyourbusiness.com"; }
                                homepage = rs.getString(8);
                                if (homepage==null) { homepage = "darkages.darkfantastic.net/"; }
                                photo = rs.getString(9);
                                if (photo==null) { photo = "darkages.darkfantastic.net/none.jpg"; }
                                if (chatlet==false) colorpref = rs.getInt(10);
                                lastsite = rs.getString(11);
                                if (lastsite==null) { lastsite = "Who knows..."; }
                                laston = rs.getString(12);
                                if (laston==null) { laston = "The other day"; }

                                return true;
                        }
                        else {
                                t.printSyslog("Unable to loadUserData for User: "+name);
                                return false;
                        }
                }
                catch (Exception e) {
                        t.printSyslog("loadUserData(): Exception caught trying to Load User Data");
                        e.printStackTrace();
                        return false;
                }
        }

        /** saveUserData() saves the UserConnection data to the database.  This method is
        called by the quit commands so information is saved when the user logs out.
                @return boolean indicating whether or not the saveUserData() call was
                successful. */
        boolean saveUserData() {

                t.connectDB();
                String sql="";
                String passstring;
                try {
                        password = t.escapeUserInput(password);
                        desc = t.escapeUserInput(desc);
                        profile = t.escapeUserInput(profile);
                        email = t.escapeUserInput(email);
                        homepage = t.escapeUserInput(homepage);
                        photo = t.escapeUserInput(photo);

                        if (t.getEncryptPasswords()) { passstring = "encrypt('"+password+"','PW')"; }
                        else { passstring = "'"+password+"'"; }

                        sql = "UPDATE user SET "+
                                "description='"+desc+
                                "',level="+level+
                                ",password="+passstring+
                                ",profile='"+profile+
                                "',gender='"+gender+
                                "',age="+age+
                                ",totaltime="+getTotalTime()+
                                ",email='"+email+
                                "',homepage='"+homepage+
                                "',photo='"+photo+
                                "',lastsite='"+getSite()+
                                "',laston='"+t.datestamp+", "+t.timestamp+
                                "' WHERE username='"+name+"'";

                        int sqlrows = t.stmt.executeUpdate(sql);

                        return true;
                }
                catch (Exception e) {
                        t.printSyslog("saveUserData(): Exception while saving data!\nSQL:"+sql);
                        e.printStackTrace();
                        return false;
                }
```

```
}

/** addConnection() is called when a new UserConnection is created and a new
Socket it linked to it.  This initializes the BufferedReader and PrintWriter,
starts the thread, and connects to the Database.
        @param s is the Socket over which the UserConnection communicates with
        the rest of the system. */
public void addConnection(Socket s) {
        socket = s;
        try {
                in = new BufferedReader (
                new InputStreamReader (socket.getInputStream()));
        } catch (Exception e) {
                t.printSyslog("Exception Creating BufferedReader");
                e.printStackTrace();
        }
        try {
                out = new PrintWriter (
                        new BufferedWriter (
                                new OutputStreamWriter (
                                        socket.getOutputStream())), true);
                println("~FGWelcome to Dark Ages Test System~N");
                println("~FGRunning Java Talker Version "+t.version+"~N");
        } catch (Exception e) {
                t.printSyslog("Exception Creating PrintWriter");
                e.printStackTrace();
        }


        outset = true;
        start();
        t.connectDB();
}

/** checklevel() takes a parameter that is the minimum level for a command.
Checklevel is put in an if-block, and if checklevel returns a true, the command
is executed.  If the level of the user is less than the parameter, checklevel
returns false and the user may not execute the command.
        @param test is an integer representing the minimum level for a command's
        execution.
        @return boolean representing whether or not the user's level compares to
        test and if true, the command is executed. */
public boolean checklevel (int test) {
        if (level < test) {
                println("\n~FRYou are not authorized to use that command~N");
                return false;
        }
        return true;
}

/** run() is the main method of the UserConnection.  In it, there are loops that
continuously read the socket and act based on the data read.  As long as the user
is connected to the system, the UserConnection thread is running.  The run()
method is divided into two main sections, the login section and the talker
section.  The login section simply reads the username, reads the password, checks
them against a database, and if they match, the user is let onto the system and
is put into the talker section.  Additionally, in the logon section, we take care
of creation of new accounts.  A user may put "New" as the username which triggers
the system to assist in creation of the new user account.  When completed, the
system creates the new account in the database and passes the new user into the
Talker section.  The Talker section continuously checks the socket for input.  If
it finds some, it reads the line of input looking for command keywords.  If a
keyword is found, the run method calls the method responsible for that command
with its parameters and goes back to reading the socket. */
public void run() {
        StringTokenizer st;
        String cmd = new String("");
```

```
                String loginname=null;;

                String cmd_who = new String(".who");
                String cmd_tell = new String(".tell");
                String cmd_go = new String(".go");
                String cmd_read = new String(".read");
                String cmd_write = new String(".write");
                String cmd_wipe = new String(".wipe");
                String cmd_topic = new String(".topic");
                String cmd_quit = new String(".quit");
                String cmd_shutdown = new String(".shutdown");
                String cmd_look = new String(".look");
                String cmd_emote = new String(".emote");
                String cmd_remote = new String(".remote");
                String cmd_review = new String(".review");
                String cmd_cbuffer = new String(".cbuffer");
                String cmd_rmail = new String(".rmail");
                String cmd_smail = new String(".smail");
                String cmd_dmail = new String(".dmail");
                String cmd_description = new String(".description");
                String cmd_set = new String(".set");
                String cmd_idle = new String(".idle");
                String cmd_examine = new String(".examine");
                String cmd_promote = new String(".promote");
                String cmd_demote = new String(".demote");
                String cmd_afk = new String(".afk");
                String cmd_remove = new String(".remove");
                String cmd_revtells = new String(".revtells");
                String cmd_ctells = new String(".ctells");
                String cmd_delete = new String(".delete");
                String cmd_nuke = new String(".nuke");
                String cmd_version = new String(".version");
                String cmd_help = new String(".help");
                String cmd_colors = new String(".colors");

                // These booleans are responsible for the script taking place in the login section.
                // At each step, the booleans are changed so that the next step may take place.
                boolean newuser = false;
                boolean printednameheader = false;
                boolean readloginname = false;
                boolean printedpassheader = false;

                //*****************************************************************
                //  Start of Login Section
                //*****************************************************************
                try {
                        // If the user has not authenticated, keep the user in the
                        // Login section.  If the user asks to quit, let him.

                        while ((authenticated==false) && (quit==false)) {

                                // This sleep call keeps us from using up all CPU
                                try { Thread.sleep(100); }
                                catch (InterruptedException ie) { continue; }

                                // If the PrintWriter is not initialized, continue.
                                if (outset==false) continue;


                                // First, we check to see if this is the New User section, if not,
                                // use this main section, which is for normal logins.
                                if (newuser == false) {

                                        if ((printednameheader==false) && (readloginname==false)
                                                && (printedpassheader==false)) {

                                                print("\n");
                                                print("~FGNew Users, Enter: New~N\n");
                                                print("~FGWhat is your Name?~N ");
```

```
        out.flush();
        printednameheader = true;
}
if ((in.ready()) && (printednameheader==true) &&
        (readloginname==false) && (printedpassheader==false)) {

        // Read the user Login Name, since we're passing it to the
        // database, escape it.
        loginname = t.escapeUserInput(in.readLine());
        readloginname = true;
        idle = 0;

        if ((loginname.length() > 0)) {

                // Properly format the username.
                loginname = loginname.trim();
                String fl = loginname.substring(0,1);
                fl = fl.toUpperCase();
String el = loginname.substring(1,loginname.length());
                el = el.toLowerCase();
                loginname = fl.concat(el);
                loginname = loginname.replace(' ','x');

                // If the login name is New, we set the booleans
                // to use the New User Setup Mode.
                if (loginname.equals("New")) {
                println("~FYEntering New User Setup Mode...~N");
                        newuser = true;
                        printednameheader = false;
                        readloginname = false;
                        printedpassheader = false;
                        continue;
                }

                // This will restart the login section, but
                //  disable the colors for the Chatlet.
                if (loginname.equals("Chatlet_coloroff")) {
                        printednameheader = false;
                        readloginname = false;
                        printedpassheader = false;
                        chatlet=true;
                        colorpref=0;
                        continue;
                }

                // This restarts the login section and then changes
                // the color parsing to leave the color tags intact
                // so that client-side color parsing may take place
                if (loginname.equals("Chatlet_coloron")) {
                        printednameheader = false;
                        readloginname = false;
                        printedpassheader = false;
                        colorpref=1;
                        chatlet=true;
                        continue;
                }

                // If the user types quit, we'll just quit them.
                if (loginname.equals("Quit")) {
                        println("~FYNow Quitting...~N");
                        quit=true;
                        break;
                }

                println("Your name is now "+loginname);
        }
        else {
                println("~FRInvalid Username~N");
                printednameheader = false;
```

```
                                                        readloginname = false;
                                                        printedpassheader = false;

                                                        continue;
                                                }
                                        }

                                        if ((printedpassheader==false) && (readloginname==true)
                                                && (printednameheader==true)) {
                                                print("~FGWhat is your Password?~N ");
                                                out.flush();
                                                printedpassheader=true;
                                        }

                                        if ((in.ready()) && (printedpassheader==true) &&
                                                (readloginname==true) && (printednameheader==true)){

                                                // Read Password, and Escape Input, since it will go
                                                // to the database.
                                                password = t.escapeUserInput(in.readLine());
                                                idle = 0;
                                                if (password.length() > 0) {
                                                        println("Your password is "+password);

                                                        // Depending on the system, (Windows 2000 or not),
                                                        // encrypt the password or not.
                                                        if (t.getEncryptPasswords()==true) {
                                                                sql = "SELECT username,password FROM user"
                                                                +" WHERE username='"+loginname+"' AND "
                                                                +"password=encrypt('"+password+"', 'PW')";
                                                        }
                                                        else {
                                                                sql = "SELECT username,password FROM user"
                                                                +" WHERE username='"+loginname+"' AND "
                                                                +"password='"+password+"'";
                                                        }
                                                        try {
                                                                String tname = "";
                                                                ResultSet rs = t.stmt.executeQuery(sql);

                                                                // If a username / password combination is
                                                                // found in the database, we have
                                                                // authenticated.
                                                                if (rs.next()) { tname = rs.getString(1); }

                                                                // If not, we have failed authentication,
                                                                // and we'll just start over.
                                                                else {
                                                                        //System.out.println("SQL: "+sql);
                                                                        t.printSyslog("Username and "
                                                                        +"Password Failure by '"
                                                                        +loginname+"' from "+getSite());
                                                                        tname = "nothing";
                                                                        println("~FRInvalid Username "
                                                                        +"and Password Combination.  "
                                                                        +"Try again.~N");
                                                                        printednameheader = false;
                                                                        readloginname = false;
                                                                        printedpassheader = false;
                                                                        continue;
                                                                }

                                                                // Okay, suppose we do properly
                                                                // authenticate.  Lets now check the
                                                                // system to see if this user is already
                                                                // signed in.  If so, the other is
                                                                // treated like a connection timeout,
                                                                // and the present UserConnection is
                                                                // authenticated.
```

```
        if (t.getUserIndex(tname)!=-1) {
                t.printSyslog("Multiple Instances of same User Logged In");
                println("~FRYou are already signed on!~N");
                println("~FRDropping Old Connection...~N");
                t.conn[t.getUserIndex(tname)].replaced=true;
                t.conn[t.getUserIndex(tname)].quit();
                name = loginname;
                authenticated=true;
                loadUserData();
                replaced=true;
        }

                                // Authenticate this UserConnection.

                                else {
                                        if (tname.equals(loginname)) {
                                                name = loginname;
                                                authenticated = true;
                                                loadUserData();
                                        }
                                }
                        }
                        catch (Exception e) {
        t.printSyslog("Invalid Username and Password Exception");
        t.printSyslog("Probably means SQLException, and DB Connection"
                +" is dropped");
        e.printStackTrace();
        t.printSyslog("Attempting to Reconnect the Database...");
        if (t.connectDB()==false) {
                t.printSyslog("Failed to Reconnect the Database...");
        }
        else {
                t.printSyslog("Database Successfully Reconnected.");
                println("Please try again.");
        }
        printednameheader = false;
        readloginname = false;
        printedpassheader = false;
        continue;
                        }

                }
                else {
                        println("~FRInvalid Username and Password.~N");
                        printednameheader = false;
                        readloginname = false;
                        printedpassheader = false;
                }
        }
}

// This is the New User Setup section.  If the user has
// authenticated in the login section, this section is skipped.

else {
        try {
                if ((printednameheader==false) && (readloginname==false)
                && (printedpassheader==false))  {
                        println("~FYNew User Setup reached!~N");

        print("\n~FGWhat username would you like to have?~N ");
                        out.flush();
                        printednameheader=true;
                }

                if ((in.ready()) && (printednameheader==true) &&
                (readloginname==false) && (printedpassheader==false)) {
```

```
        // Read the Loginname, and Escape it, since it
        // goes into the database.
        loginname = t.escapeUserInput(in.readLine());
        idle = 0;

        if ((loginname.length() > 0) &&
        (loginname.length() < 13))  {

                // Properly format the Username.
                loginname = loginname.trim();
                String fl = loginname.substring(0,1);
                fl = fl.toUpperCase();
String el = loginname.substring(1,loginname.length());
                el = el.toLowerCase();
                loginname = fl.concat(el);
                loginname = loginname.replace(' ','x');

                // If the user wants to quit, let him.
                if (loginname.equals("Quit")) {
                        println("~FYNow Quitting...~N");
                        quit=true;
                        break;
                }

                // Users can't have "New" as a username.
                if (loginname.equals("New")) {
        println("~FRYou can't use that for a Username!~N");
                        printednameheader = false;
                        readloginname = false;
                        printedpassheader = false;
                        continue;
                }

                // Next, check to see if that username has
                // been used already.
                if (t.isUser(loginname)) {
        println("~FRThat username is already in use.");
        println("Please pick another.~N");
        println("~FRIf you really are ~FY\""+loginname
                +"\"~FR, type Quit");
        println(" and log in the normal way.~N");
                        printednameheader = false;
                        readloginname = false;
                        printedpassheader = false;

                        continue;
                }
                readloginname = true;
        }
        else {
                println("~FRIllegal Username.~N");
                printednameheader = false;
                readloginname = false;
                printedpassheader = false;
                continue;
        }
}

if ((printednameheader==true) &&
(printedpassheader==false) && (readloginname==true)) {
        println("Your name is now "+loginname);
        print("\n~FGWhat Password would you "
                +"like to use?~N ");
        out.flush();
        printedpassheader=true;
}
if ((in.ready()) &&  (printedpassheader=true) &&
(readloginname==true) && (printednameheader==true)) {
```

```
                                                // Read and Escape the password.
                                                password = t.escapeUserInput(in.readLine());
                                                idle = 0;
                                                if (password.length() > 0) {
                                                        out.println("Your password is "+password);
                                                        try {

                                                // If the system does not support
                                                // encryption (Win2000), don't
                                                // encrypt the passwords.
                                                if (t.getEncryptPasswords()==true) {
                                sql = "INSERT INTO user (username, password) VALUES (\""
                                        +loginname+"\",encrypt('"+password+"','PW'))";
                                                }
                                                else {
                                sql = "INSERT INTO user (username, password) VALUES (\""
                                        +loginname+"\",'"+password+"')";
                                                }
                                                int sqlrows = t.stmt.executeUpdate(sql);

                                                                if (sqlrows == 1) {
                                                                        authenticated = true;
                                                                        name = loginname;
                                                                        saveUserData();
                                                                }
                                                        }
                                                        catch (Exception e) {
                                                                println("Please try again!");
                t.printSyslog("Invalid New User and Password Exception");
                t.printSyslog("Probably means SQLException, and DB Connection is dropped");
                                                                e.printStackTrace();

                t.printSyslog("Attempting to Reconnect the Database...");
                if (t.connectDB()==false) {
                        t.printSyslog("Failed to Reconnect the Database...");
                }
                else {
                        t.printSyslog("Database Successfully Reconnected.");
                        out.println("Please try again.");
                }
                                                                printednameheader = false;
                                                                readloginname = false;
                                                                printedpassheader = false;

                                                                continue;
                                                        }
                                                }

                                                else {
                                        println("~FRInvalid Username and/or Password?~N");
                                                        printednameheader = false;
                                                        readloginname = false;
                                                        printedpassheader = false;

                                                        continue;
                                                }
                                        }
                                }
                                catch (Exception e) {
                                        t.printSyslog("Exception Caught in New User Setup!");
                                        e.printStackTrace();
                                }
                        }
                }
        }
        catch (Exception e) {
                t.printSyslog("Error Authenticating:  Caught Exception");
                e.printStackTrace();
        }
```

```java
// Print announcement to system that another user has just signed on or reconnected

if ((quit==false) && (replaced==false)) {
        saveUserData();
        println("\n\n");
        t.printSystemMessage("~FYJOINING US: "+name+" "+desc+"~N", index);
        t.printSyslog(""+name+" signed on from "+getInetAddress());
        t.area[0].look(this);
}

if ((quit==false) && (replaced==true)) {
        saveUserData();
        println("\n\n");
        t.printSystemMessage("~FYREJOINING US: "+name+" "+desc+"~N", index);
        t.printSyslog(""+name+" signed on again from "+getSite());
        t.area[0].look(this);
        replaced=false;
}

//*****************************************************************
//  End of Login Section, Start of Talker Section
//*****************************************************************

try {
        while (quit==false) {

                // First Sleep so we don't eat all the CPU.
                try { Thread.sleep(100); }
                catch (InterruptedException ie) { continue; }

                try {
                        // If there is Input waiting to be read, read it.
                        if (in.ready()) {
                                String str = in.readLine();
                                String rest = "";
                                if (str.length()<=0) continue;
                                st = new StringTokenizer(str);

                                // Parse through the tokens and build two variables:
                                // cmd - the first token (or command) on the line
                                // rest - everything else on the line.
                                try {
                                        cmd = st.nextToken();
                                        rest = "";
                                        while (st.hasMoreTokens()) {
                                                rest = rest.concat(" ");
                                                rest = rest.concat(st.nextToken());
                                        }
                                        rest = rest.trim();
                                }
                                catch (Exception e) {
                                        t.printSyslog("UserConnection Run:"
                                        +" String Tokenizer Exception");
                                        e.printStackTrace();
                                }

                                // Any input from the user is enough to set the
                                // idle count to 0.
                                idle = 0;

                                // If a user was AFK, any input brings them back...
                                // Also print AFK info.
                                if (getAfk()) {
                                        println("You return to the keyboard.");
                                        if (getAfkTells()) {
                                                println("There were .tells "
                                                +"while you were away:");
                                                setAfkTells(false);
```

```
                                                        showTell();
                                                }

                                                setAfk(false);
                                                t.area[getArea()].printArea(this, name
                                                +" is back.", false);
                                        }


                                        // Next, we check for NUTS talker shortcuts
                                        // and convert them when necessary.
if (cmd.startsWith(";;")) {
        rest = (cmd.substring(2,cmd.length()).concat(" "+rest)).trim();
        cmd=".remote";
}
if (cmd.startsWith(";")) {
        rest = (cmd.substring(1,cmd.length()).concat(" "+rest)).trim();
        cmd=".emote";
}

// All commands start with a . so if this does not, we
// don't even bother to parse it through the commands.
if (cmd.startsWith(".")==false) { t.say(index,str); }

// Finally, we compare cmd to the command list for matches.
// We use startsWith() instead of equals() to permit us to
// use command shortcuts, for example, .w instead of .who.
// Additionally, we implement checklevel to make sure
// commands are not executed unless users are past a
// certain level.

if (cmd_who.startsWith(cmd)) { t.who(index); continue; }
if (cmd_tell.startsWith(cmd)) { t.tell(index, rest); continue; }
if (cmd_go.startsWith(cmd)) { t.goArea(index, rest); continue; }
if (cmd_read.startsWith(cmd)) { t.area[area].read(this); continue; }
if (cmd_write.startsWith(cmd)) { t.area[area].write(this, rest); continue; }
if (cmd_wipe.startsWith(cmd)) { if (checklevel(1)) { t.area[area].wipe(this, rest); } continue; }
if (cmd_topic.startsWith(cmd)) { t.area[area].setTopic(this, rest); continue; }
if (cmd_quit.startsWith(cmd)) { saveUserData(); quit=true; break; }
if (cmd_shutdown.startsWith(cmd)) { if (checklevel(4)) { t.shutdown(); break; } continue; }
if (cmd_look.startsWith(cmd)) { t.area[area].look(this); continue; }
if (cmd_emote.startsWith(cmd)) { t.emote(index, rest); continue; }
if (cmd_remote.startsWith(cmd)) { t.remote(index, rest); continue; }
if (cmd_review.startsWith(cmd)) { t.area[area].showRev(this); continue; }
if (cmd_cbuffer.startsWith(cmd)) { t.area[area].clearRev(this); continue; }
if (cmd_rmail.startsWith(cmd)) { t.rmail(this); continue;}
if (cmd_smail.startsWith(cmd)) { t.smail(this, rest); continue; }
if (cmd_dmail.startsWith(cmd)) { t.dmail(this, rest); continue; }
if (cmd_description.startsWith(cmd)) {
        rest = new String("desc ").concat(rest);
        t.setUserProperty(this, rest);
        saveUserData();
        continue;
}
if (cmd_set.startsWith(cmd)) { t.setUserProperty(this, rest); saveUserData(); continue; }
if (cmd_idle.startsWith(cmd)) { t.idle(this); continue; }
if (cmd_examine.startsWith(cmd)) { t.examine(this, rest); continue; }
if (cmd_promote.startsWith(cmd)) { if (checklevel(3)) { t.promote(this, rest); } continue; }
if (cmd_demote.startsWith(cmd)) { if (checklevel(3)) { t.demote(this, rest); } continue; }
if (cmd_afk.startsWith(cmd)) {  t.setAfk(this,rest); continue; }
if (cmd_remove.startsWith(cmd)) { if (checklevel(3)) { t.remove(this,rest); } continue; }
if (cmd_revtells.startsWith(cmd)) { showTell(); continue; }
if (cmd_ctells.startsWith(cmd)) { clearTell(); continue; }
if (cmd_delete.startsWith(cmd)) { if (checklevel(3)) { t.deleteUser(this,rest); } continue; }
if (cmd_nuke.startsWith(cmd)) { if (checklevel(3)) { t.deleteUser(this,rest); } continue; }
if (cmd_version.startsWith(cmd)) { t.version(this); continue; }
if (cmd_help.startsWith(cmd)) { t.help(this, rest); continue; }
if (cmd_colors.startsWith(cmd)) { t.displayColors(this); continue; }
```

```java
        // If nothing matches, it is treated as a say().
        t.say(index, str);

                                        }
                                }
                                catch (NullPointerException npe) {
                                        t.printSyslog("UserConnection Run: Null Pointer Exception...");
                                        // I found that sometimes, all the socket setup
                                        // that takes place in Talker takes longer than
                                        // it takes for the Connection class to start its
                                        // thread.  This results in the in.readLine() up
                                        // there still not being initialized.  We just
                                        // try again if we get a N.P.E.
                                        continue;
                                }
                                catch (Exception e) {
                                        //Socket Exception and Array Out of Bounds
                                        t.printSyslog("Time out or Connection drop");
                                        e.printStackTrace();
                                        break;
                                }
                        }
                }
                catch (Exception e) {
                        t.printSyslog("UserConnection Run Exception");
                        e.printStackTrace();
                }
                finally {
                        try {
                                // First, announce a user is leaving (if the user wasn't replaced)
                                if (replaced==false) {
                                        t.printSystemMessage("~FYLEAVING US: "+name
                                        +" "+getUserDesc()+"~N", index);
                                }
                                // Next, t.quitUser() resizes the UserConnection array.
                                t.quitUser(index);

                                // Close Input and Output Streams and the Socket.
                                in.close();
                                out.close();
                                socket.close();
                                t.printSyslog(""+getUserName()+" is disconnected.");
                        }
                        catch (Exception e) {
                                t.printSyslog("Exception in trying to close a connection");
                                e.printStackTrace();
                        }
                }
        }

}  // End of Connection Class
//***************************************************************************
/** Talker is the Main JavaTalker class that links everything else together.
        <p>The Talker Class performs the connection handling portion of the Java Talker.
        The Main method of Talker opens a ServerSocket and then sits in a loop, accepting
        connections and then creating UserConnection Objects and spinning the connections
        off on their own thread. Most of the user commands reside in Talker, mostly
        because they need to communicate messages among other users on the system.
        Talker also creates the connections to the database, builds the Area array and
        manages the UserConnection array. */

public class Talker {
        /** the Talker version */
        String version = "JT 0.9.31";
        /** the date last updated */
        String updated = "11/23/2003";
        /** the person last doing the updates */
        String updater = "Jkb";
        /** This TalkerTimer instance links to the Talker and starts performing
        its time and connection management functions */
        TalkerTimer timer = new TalkerTimer(this);

        /** this is used to specify on which port we want Talker to run. */
        static final int PORT = 2122;

        /** this ServerSocket is used to accept connections on the port listed above. */
        ServerSocket s;

        /** this Socket is a temporary spot for a connection until it is
        spun off onto its own UserConnection. */
        Socket socket;

        /**  the UserConnection array, conn stores the UserConnections. */
        public UserConnection[] conn;

        /** the UserConnection array, ctemp is used to temporarily hold
        the conn array data during resizing. */
        public UserConnection[] ctemp;

        /** This is used to create new UserConnections and link them into the UserConnection Array. */
        UserConnection oneconn;

        /** This is a connection to the Database. */
        Connection DBConnection = null;

        /** This statement is used to execute SQL commands in the database and return data. */
        Statement stmt = null;

        /** This is used to store the current System Date. */
        StringBuffer datestamp = new StringBuffer();

        /** This is used to store the current System Time. */
        StringBuffer timestamp = new StringBuffer();

        /** This is used to describe how we want Dates formatted when we print them. */
        SimpleDateFormat sdf = new SimpleDateFormat("EEE MM/dd/yyyy");

        /** this is used to describe how we want Times formatted when we print them. */
        SimpleDateFormat stf = new SimpleDateFormat("kk:mm");

        /** This Array is used to store all of the Areas available to the users on the Java Talker. */
        public Area[] area;

        /** This boolean tells the system whether or not to encrypt the passwords
        when reading and writing to the User table in the database. */
        private boolean encryptpasswords=true;

        /** main is the static method of Talker, which simply creates a new Talker instance. */
        public static void main (String[] args) {
                Talker c = new Talker();
        }

        /** connectDB() tests the connection to the Database.  If the connection is not made,
                connectDB reopens it using the MySQL Connector-J libraries.
                @return boolean that describes whether or not the database connection was successful.*/
        public boolean connectDB() {
                if (DBConnection != null) {
                        try {
                                if (DBConnection.isClosed()==false) {
                                        //printSyslog("Database Connection is already Open.");
                                        return true;
                                }
                        }
                        catch (Exception e) {
                                printSyslog("Exception caught while checking to see if database was open");
                                e.printStackTrace();
                        }
                }

                // Initiate Connection to the Database using the Connector-J JDBC drivers.
                try {
                        Class.forName("com.mysql.jdbc.Driver").newInstance();
                        DBConnection =
                                DriverManager.getConnection("jdbc:mysql://localhost:3306/jkbtalker",
                                "jkb", "6puppy");
                        stmt = DBConnection.createStatement();
                        stmt.setEscapeProcessing(true);
                }
                catch (Exception e) {
                        printSyslog("ConnectDB: Error Opening Database");
                        e.printStackTrace();
                        return false;
                }
                return true;
        }

        /** getEncryptPasswords() simply returns the boolean value of whether or not the
                database supports encrypted passwords.
                @return boolean that says whether or not the database supports the encrypt() function.*/
        public boolean getEncryptPasswords() {
                return encryptpasswords;
        }

        /** setEncryptPasswords() runs a test on the database to determine whether or not
        the database will support encrypted passwords.  If the test passes an encrypted
        string back to the Java Talker, encryptpasswords is set to true and the system
        will use encrypted passwords in the future.  Otherwise, the system will not use
        encrypted passwords in the database. */
        public void setEncryptPasswords() {

                // Connect to Database.
                connectDB();
                printSyslog("Entered setEncryptPasswords()");

                // We're going to ask for an encrypted version of the word 'PASSWORD'.
                String sql = "SELECT encrypt('PASSWORD', 'PW')";
                String answer="";
                try {
                        // Run the Query...
                        ResultSet rs = stmt.executeQuery(sql);

                        // See if we get an answer...
                        if (rs.next()) {

                                answer = rs.getString(1);
                                printSyslog("setEncryptPasswords(): "+answer);

                                // If we get an answer, the encrypt method is supported.
                                if (answer!=null) {
                                        printSyslog("Answer is probably an encrypted string. True");
                                        encryptpasswords = true;
                                }

                                // If we don't, there is no encrypt method, so the system assumes we're
                                // asking for a user-defined method that doesn't exist, so we get
                                // a null answer.  In which case, we turn off encrypting of passwords.
                                else {
                                        printSyslog("Answer is probably NULL.");
                                        encryptpasswords = false;
                                }
                        }
                        else {
                                // Additionally, if we are unable to get even a null answer
                                // we don't encrypt the passwords.
                                printSyslog("Answer is probably NULL.");
                                encryptpasswords = false;
                        }
                }
                catch (Exception e) {
                        printSyslog("Exception in setEncryptPasswords()");
                        e.printStackTrace();
                        encryptpasswords=false;
                }
                if (encryptpasswords) answer = "True";
                else answer = "False";

                // Display to the Syslog the encryption status
                printSyslog("Exited setEncryptPasswords() with answer of: "+answer);
        }

        /** initDates() Reads the date and time from the system and stores it in some StringBuffers
                for later use in TalkerTimer, Syslog printout and the offline messaging. */
        public void initDates() {
                datestamp = new StringBuffer();
                timestamp = new StringBuffer();
                sdf.format(new java.util.Date(), datestamp, new FieldPosition(0));
                stf.format(new java.util.Date(), timestamp, new FieldPosition(0));
        }

        /** printSyslog() Prints a message along with the date and time stamp.  At present,
        this prints to System.out since the syslog is simply captured to a text file along
        with the stacktraces of any Exceptions that may be thrown.  In the future, we may
        want to write this data to a more formal file or database table for reading so
        certain users on the system can review the logs while online.
        @param message this is the message that gets printed along with the date and time. */
        public void printSyslog(String message) {
                System.out.println(datestamp+"; "+timestamp+":  "+message);
        }

        /** the Talker() constructor does most of the work in the Java Talker.  It
        initializes the dates and times.  It connects the database. It performs tests
        to see if the database will support encryption.  Finally, it listens for and
        manages the connections and the resizing of the UserConnection array. */
        public Talker() {
                // First, we initialize the time and date stamps.
                initDates();

                // Connect to the database.
                if (connectDB()==false) {
                        printSyslog("Unable to Open the Database, System Exiting...");
                        System.exit(1);
                }

                // Test whether or not we can encrypt passwords, and set variables accordingly.
                setEncryptPasswords();

                // Create UserConnection Array and Temporary UserConnection Array.
                conn = new UserConnection[0];
                ctemp = new UserConnection[1];

                // Create a temporary Area and initialize the area array.
                //Area ax = new Area(this);
                //area = ax.initArea(this);
                area = Area.initArea(this);

                int i=0;
                try {
                        // Start ServerSocket
                        s = new ServerSocket(PORT);
                        printSyslog("Server Socket Started");
                }
                catch (Exception e) {
                        printSyslog("Unable to Start Server Socket");
                        e.printStackTrace();
                        System.exit(1);
                }

                try {

                        while (true) {
                                // Sleep the thread for a little bit so we're not
                                // constantly hitting the CPU.
                                try { Thread.sleep(100); }
                                catch (InterruptedException ie) { continue; }

                                try {
                                        // Listen to accept a socket, then block until we do.
                                        socket = s.accept();
                                }

                                // If we get an exception here, it's because the ServerSocket is closed
                                // since the system is shutting down.  Because the accept() method blocks,
                                // the only way to shut down the system is to manually kill the
                                // ServerSocket.
                                catch (Exception e) {
                                        printSyslog("System Shutting down,"
                                                +" No Socket to Accept Connection");
                                        //e.printStackTrace();
                                        break;
                                }

                                // At this point, the system has stopped blocking and has a
                                // connection in socket.

                                try {
                                        // Assign the socket to a UserConnection.
                                        oneconn = new UserConnection(this);
                                        oneconn.addConnection(socket);

                                        // Resize the Array.
                                        ctemp = new UserConnection[conn.length+1];
                                        for (i=0; i<conn.length; ++i) {
                                                ctemp[i] = conn[i];
                                        }

                                        // Add the new UserConnection to the end of it.
                                        ctemp[conn.length] = oneconn;
                                        conn = ctemp;

                                        // Delete the temporary array.
                                        ctemp = null;

                                        // set the index for the new UserConnection.
                                        conn[conn.length-1].setIndex(conn.length-1);

                                }
                                catch (Exception e) {
                                        printSyslog("Caught Exception while adding user to conn array");
                                        e.printStackTrace();
                                        try {
                                                socket.close();
                                        }
                                        catch (Exception ee) {
                                                printSyslog("Error Closing Connection Socket");
                                                ee.printStackTrace();
                                        }
                                }
                        }
                }
                finally {
                        try {
                                s.close();
                        }
                        catch (Exception e) {
                                printSyslog("Error Closing ServerSocket");
                                e.printStackTrace();
                        }
                }
        }

        /** printSystemMessage() is used to print messages to all authenticated users across the system
                regardless of which room they are in.
                @return boolean indicating whether the command failed.
                @param message String containing the message we want to print to the users
                @param ignoreuser integer index of the any user we may want to skip. */
        boolean printSystemMessage(String message, int ignoreuser) {
                for (int i=0; i<conn.length; ++i) {
                        if (ignoreuser==i) continue;

                        try {
                                // If a user is not authenticated, they don't get to see the message.
                                if (!conn[i].isAuthenticated()) continue;
                        }
                        catch (Exception e) {
                                printSyslog("Error Checking if conn is authenticated");
                                quitUser(i);
                                --i;
                                continue;
                        }
                        try {
                        conn[i].println(message);
                        } catch (Exception ee) { printSyslog("Error Printing message to conn"); }
                }
                return false;
        }

        /** tell() is the means of sending online private messages from one user to another.
                in addition to getting the message, the private message gets added to the UserConnections
                personal tell buffer.
                @return boolean indicating whether or not an error occurred.
                @param from integer indicating the index of the user sending the .tell
                @param messagein is the String containing the recipient's name and the message. */
        boolean tell (int from, String messagein) {
                StringTokenizer st = new StringTokenizer(messagein);
                String to;
                String message;
                int toindex=-1;

                // First, parse off the recipient's name.
                if (st.hasMoreTokens()) to = st.nextToken();
                else {
                        conn[from].println("~FR Usage:  .tell username message~N");
                        return true;
                }

                // Next, pull off at least one more token to be the message.
                if (st.hasMoreTokens()) message = st.nextToken();
                else {
                        conn[from].println("~FR Usage:  .tell username message~N");
                        return true;
                }

                // If there are any more tokens, concatenate them onto the message.
                while (st.hasMoreTokens()) {
                        message = message.concat(" ");
                        message = message.concat(st.nextToken());
                }

                // get the index of the recipient in the UserConnection array.
                toindex = getUserIndex(to);

                // If the user is not signed on, report this.
                if (toindex==-1) {
                        conn[from].println("\""+to+"\" is not signed on.");
                        return true;
                }

                // If the user is AFK, print the AFK messages.
                if (conn[toindex].getAfk()) {
                        conn[from].println(conn[toindex].getUserName()+" is AFK: "
                                +conn[toindex].getAfkMsg());
                        conn[toindex].setAfkTells(true);
                }

                // Print the message to each user's socket and tell buffer.
                conn[from].println("~FP-> You say to "+conn[toindex].getUserName()+": "+message+"~N");
                conn[from].addTell("-> You say to "+conn[toindex].getUserName()+": "+message+"~N");

                conn[toindex].println("-> "+conn[from].getUserName()+" tells you: "+message+"~N");
                conn[toindex].addTell("-> "+conn[from].getUserName()+" tells you: "+message+"~N");

                return false;
        }

        /** remote() sends a private online message in the form of an emote to another
        user.  In addition to just printing the message, this will also put the message
        in the UserConnection tell buffer.
        @return boolean indicating whether or not an error has taken place.
        @param from is the index in the UserConnection Array of the user sending the remote message.
        @param messagein is the String containing the username of the recipient and the
        actual message to send. */
        boolean remote (int from, String messagein) {
                StringTokenizer st = new StringTokenizer(messagein);
                String to;
                String message;
                int toindex=-1;

                // Pull off the first token to be the username
                if (st.hasMoreTokens()) to = st.nextToken();
                else {
                        conn[from].println("~FR Usage:  .remote username message~N");
                        return true;
                }

                // Pull off the second token to be the message.
                if (st.hasMoreTokens()) message = st.nextToken();
                else {
                        conn[from].println("~FR Usage:  .remote username message~N");
                        return true;
                }

                // Concatenate all the rest of the tokens to the message.
                while (st.hasMoreTokens()) {
                        message = message.concat(" ");
                        message = message.concat(st.nextToken());
                }

                // Get the Index of the Recipient
                toindex = getUserIndex(to);

                // If the recipient is not signed on, report this and exit.
                if (toindex==-1) {
                        conn[from].println("\""+to+"\" is not signed on.");
                        return true;
                }

                // If the recipient is AFK, report this fact, but proceed with the remote.
                if (conn[toindex].getAfk()) {
                        conn[from].println(conn[toindex].getUserName()+" is AFK: "
                                +conn[toindex].getAfkMsg());
                        conn[toindex].setAfkTells(true);
                }
```

```java
                // Print the message to both sockets and add to the tell buffers
                conn[from].println("~FP-> You remote to "+conn[toindex].getUserName()+": "
                        +conn[from].getUserName()+" "+message+"~N");
                conn[from].addTell("-> You remote to "+conn[toindex].getUserName()+": "
                        +conn[from].getUserName()+" "+message+"~N");

                conn[toindex].println("-> "+conn[from].getUserName()+" "+message+"~N");
                conn[toindex].addTell("-> "+conn[from].getUserName()+" "+message+"~N");
                return false;
        }

        /** promote() increments the user level of the recipient user to allow access to more commands.
                @return boolean indicating whether the promote was successful
                @param conn UserConnection of the user calling the .promote command
                @param other String containing the name of the user to be promoted.*/
        boolean promote (UserConnection conn, String other) {
                if (other.equals("")) {
                        conn.println("~FR Usage:  .promote username~N");
                        return false;
                }

                // Get the Index of the other user.
                int i = getUserIndex(other);

                if (i!=-1) {
                        int level = this.conn[i].getLevel();

                        // Check to see if target user is less than level 5.
                        if (level < 5) {
                                level++;
                        }
                        else {
                                conn.println("~FRCan't promote past level 5~N");
                                return false;
                        }


                        // The user can't be promoted to an equal or higher level than the person
                        // doing the promoting.
                        if (level>=conn.getLevel()) {
                                conn.println("~FRCan't promote to an equal or higher level than yours.~N");
                                return false;
                        }

                        // Set the level of the target user and then print the messages informing
                        // them of the level change.
                        this.conn[i].setLevel(level);
                        conn.println("You promote "+this.conn[i].getUserName()+" to "
                                +this.conn[i].getLevelName());
                        this.conn[i].println(conn.getUserName()+" promotes you to "
                                +this.conn[i].getLevelName());
                        printSyslog(""+conn.getUserName()+" promotes "+this.conn[i].getUserName()
                                +" to "+this.conn[i].getLevelName());
                }
                else {
                        conn.println("That user is not signed on.");
                        return false;
                }

                // Save the UserData of the user promoted.
                this.conn[i].saveUserData();
                return true;
        }

        /** demote() decrements the user level of the recipient user to restrict access to commands.
                @return boolean indicating whether or not the demote was successful.
                @param conn UserConnection of the user calling the .demote command
                @param other String containing the name of the user to be demoted.*/
        boolean demote (UserConnection conn, String other) {
                if (other.equals("")) {
                        conn.println("~FR Usage:  .demote username~N");
                        return false;
                }

                // Get User Index.
                int i = getUserIndex(other);

                if (i!=-1) {
                        int level = this.conn[i].getLevel();

                        // Check to make sure we're not demoting past level 0.
                        if (level >= 1) {
                                level--;
                        }
                        else {
                                conn.println("~FRCan't demote past level 0~N");
                                return false;
                        }

                        // Can't demote users on equal or higher level than you.
                        if (level>=conn.getLevel()) {
                                conn.out.println("~FRCan't demote an equal or higher level than yours.~N");
                                return false;
                        }

                        // Change the level and report to the other users.
                        this.conn[i].setLevel(level);
                        conn.println("You demote "+this.conn[i].getUserName()+" to "
                                +this.conn[i].getLevelName());
                        this.conn[i].println(conn.getUserName()+" demotes you to "
                                +this.conn[i].getLevelName());
                        printSyslog(""+conn.getUserName()+" demotes "+this.conn[i].getUserName()
                                +" to "+this.conn[i].getLevelName());
                }
                else {
                        conn.println("That user is not signed on.");
                        return false;
                }

                // Save the user settings.
                this.conn[i].saveUserData();
                return true;
        }

        /** version() displays version information about the JavaTalker.
                @return boolean indicating whether or not errors have occurred.
                @param conn UserConnection to which the version information gets printed. */
        boolean version (UserConnection conn) {
                conn.println("");
                conn.println("~FG+===================================================+~N");
                conn.println("~FG+  Jkb JavaTalker    Version: ~FY"+version+"~N");
                conn.println("~FG+  Last Updated: ~FY"+updated+"~FG  By: ~FY"+updater+"~N");
                conn.println("~FG+===================================================+~N");
                conn.println("~FG+ The Jkb JavaTalker was designed and developed by  +~N");
                conn.println("~FG+ Jeffrey K. Brown for a graduate project at the    +~N");
                conn.println("~FG+ University of New Haven, CT as part of his pursuit +~N");
                conn.println("~FG+ of a Master's Degree in Computer Science. Enjoy!  +~N");
                conn.println("~FG+===================================================+~N");
                conn.println(" ");
                return false;
        }

        /** examine() looks up information about another user and provides this to the inquirer.
                @return boolean indicating whether or not an error has occurred.
                @param conn UserConnection of the user that the examine information gets printed to.
                @param otheruser Name of user inquiring about. */
        boolean examine (UserConnection conn, String otheruser) {
                int i=0;
                boolean online=true;

                // Get index of user.
                if (otheruser.equals("")) {
                        i = conn.getIndex();
                }
                else {
                        i = getUserIndex(otheruser);
                        if (i==-1) {
                                online = false;
                                if (isUser(otheruser)==false) {
                                        conn.println("There is no such user "+otheruser);
                                        return true;
                                }
                                else {
                                        // Check to get the proper name format from the database.
                                        otheruser = getUserDBName(otheruser);
                                }
                        }
                        else online = true;
                }

                // If the user is already online, we can simply pull the data from the
                // UserConnection object.
                if (online==true) {

conn.println("~BB~FY+======================================================================+~N");
conn.println("~FY "+this.conn[i].getUserName()+" "+this.conn[i].getUserDesc()+"~N");
conn.println("~BB~FY+======================================================================+~N");
conn.println("~FY+ Level:    "+this.conn[i].getLevelName()+"~N");
conn.println("~FY+ Age:      "+this.conn[i].getAge()+"~N");
conn.println("~FY+ Photo:    "+this.conn[i].getPhoto()+"~N");
conn.println("~FY+ Homepage: "+this.conn[i].getHomepage()+"~N");
conn.println("~FY+ Email:    "+this.conn[i].getEmail()+"~N");
conn.println("~BB~FY+======================================================================+~N");
conn.println("~FY+ User is in area: "+area[this.conn[i].getArea()].getName()+"~N");
conn.println("~FY+ Online: "+this.conn[i].getTime()+" Min;  Idle: "
        +this.conn[i].getIdle()+" Min."+"~N");
conn.println("~FY+ Total Time: "+this.conn[i].getTotalTime()+"~N");

// If the user is at a high enough level, we can view the user's site.
if (conn.getLevel()>1) {
        conn.println("~FY+ Connected from: "+this.conn[i].getSite()+"~N");
}
conn.println("~BB~FY+======================================================================+~N");
conn.println(""+this.conn[i].getProfile());
conn.println("~BB~FY+======================================================================+~N");
conn.println(" ");
return false;

                }

                // If the user is not online, we must connect to the Database and pull the user record.
                else {
                        connectDB();
                        String sql="";
                        String desc="";
                        String profile="";
                        String gender="";
                        String email="";
                        String homepage="";
                        String photo="";
                        String lastsite="";
                        String laston="";
                        int level=0;
                        int totaltime=0;
                        int age=0;
                        try {
                                // Create SQL statement.
                                sql = "SELECT description,level,profile,totaltime,gender,age,email,"
                                        +"homepage,photo,colorpref,lastsite,laston FROM user WHERE "
                                        +"username='"+otheruser+"'";

                                ResultSet rs = stmt.executeQuery(sql);
                                if (rs.next()) {
                                        desc = rs.getString(1);
                                        if (desc==null) { desc = "A Generic Desc"; }
                                        level = rs.getInt(2);
                                        profile = rs.getString(3);
                                        if (profile==null) {
                                                profile = "This user has not yet set a profile";
                                        }
                                        totaltime = rs.getInt(4);
                                        gender = rs.getString(5);
                                        if (gender==null) { gender = "Unknown"; }
                                        age = rs.getInt(6);
                                        email = rs.getString(7);
                                        if (email==null) { email = "me@noneofyourbusiness.com"; }
                                        homepage = rs.getString(8);
                                        if (homepage==null) { homepage = "darkages.darkfantastic.net/"; }
                                        photo = rs.getString(9);
                                        if (photo==null) { photo = "darkages.darkfantastic.net/none.jpg"; }
                                        lastsite = rs.getString(11);
                                        if (lastsite==null) { lastsite = "Who knows..."; }
                                        laston = rs.getString(12);
                                        if (laston==null) { laston = "The other day"; }
                                }
                                else {
                                        printSyslog("Unable to load UserData in .examine");
                                        return false;
                                }
                        }
                        catch (Exception e) {
                                printSyslog("examine(): Exception caught trying to Load User Data");
                                e.printStackTrace();
                                return false;
                        }

// Print the data from the database.
conn.println(
"~BB~FY+======================================================================+~N");
conn.println("~FY "+otheruser+" "+desc+"~N");
conn.println(
"~BB~FY+======================================================================+~N");
conn.println("~FY+ Level:    "+conn.getLevelName(level,gender)+"~N");
conn.println("~FY+ Age:      "+age+"~N");
conn.println("~FY+ Photo:    "+photo+"~N");
conn.println("~FY+ Homepage: "+homepage+"~N");
conn.println("~FY+ Email:    "+email+"~N");
conn.println(
"~BB~FY+======================================================================+~N");
conn.println("~FY+ Last Signed on: "+laston+"~N");
conn.println("~FY+ Total Time: "+totaltime+"~N");
if (conn.getLevel()>1) {
        conn.println("~FY+ Last Connected from: "+lastsite+"~N");
}
conn.println(
"~BB~FY+======================================================================+~N");
conn.println("~FY"+profile+"~N");
conn.println(
"~BB~FY+======================================================================+~N");
conn.println(" ");

                        return false;
```

```
                }
        }

        /** help() pulls the help information from the database.  If a user specifies a topic,
                a database access is made to find the help topic.  If found, it returns the particular
                help entry.  If there is no topic specified, it will produce a list of all available
                help topics.
                @return boolean reporting whether or not the call was successful
                @param conn UserConnection of the user calling the .help command
                @param topic String with the topic of the help.*/
        boolean help (UserConnection conn, String topic) {
                String sql = "";
                ResultSet rs;

                // These Strings and column are used to
                // display the Help in three neat columns.
                String col1="";
                String col2="";
                String col3="";
                int column=0;

                // If no topic is specified, just print the list.
                if (topic.equals("")) {
                        sql = "SELECT topic FROM help ORDER BY topic";

                        try {
                                rs = stmt.executeQuery(sql);

                                conn.println("~FYAvailable Help Topics:~N");

                                // Print the Results in columns.
                                while(rs.next()) {

                                        if (column==0) {
                                                col1 = rs.getString(1);
                                                while (col1.length() < 20) {
                                                        col1 = col1.concat(" ");
                                                }

                                                column=1;
                                                continue;
                                        }
                                        if (column==1) {
                                                col2 = rs.getString(1);
                                                while (col2.length() < 20) {
                                                        col2 = col2.concat(" ");
                                                }
                                                column=2;
                                                continue;
                                        }
                                        if (column==2) {
                                                col3 = rs.getString(1);
                                                while (col3.length() < 20) {
                                                        col3 = col3.concat(" ");
                                                }

                                                conn.println("" + col1 + col2 + col3);
                                                col1 = col2 = col3 = "";
                                                column=0;
                                                continue;
                                        }
                                }
                                if (column!=2) conn.println("" + col1 + col2 + col3);
                                conn.println(" ");
                                return true;
                        }
                        catch (Exception e) {
                                printSyslog("help() exception! SQL:"+sql);
                                return false;
                        }
                }

                // If there is a topic specified, search the database for the
                // topic and print the results.
                else {

                        try {
                                sql = "SELECT topic,description FROM help WHERE "
                                        +"LCASE(topic)='"+topic.toLowerCase()+"'";
                                rs = stmt.executeQuery(sql);

                                if (rs.next()) {
                                        String top = rs.getString(1);
                                        String desc = rs.getString(2);
                                        conn.println("\n~FYTopic: "+top+"~N\n\n"+desc+"\n\n");
                                        return true;
                                }
                                else {
                                        conn.println("No Results Found for "+topic);
                                        return false;
                                }
                        }
                        catch (Exception e) {
                                printSyslog("help() exception! SQL:"+sql);
                                return false;
                        }
                }
        }

        /** sendMessage() is used to display a message to users on the system
                regardless of the area they are in.  This is used to print the Joining/Leaving
                messages.
                @param from integer index of user sending the message
                @param message String message to be printed.
                @param toself integer determining whether or not to print the message to "from" user */
        void sendMessage(int from, String message, int toself) {
                int i;

                for (i=0; i<conn.length; ++i) {

                        if ((toself==0) && (from==i)) {
                                try {
                                        if (conn[i].isAuthenticated()) {
                                                conn[i].println("You say: "+ message);
                                        }
                                        continue;
                                }
                                catch (Exception e) {
                                        printSyslog("SendMessage(): Exception Printing "
                                                +"Back to User From:"+from+" i:"+i+"toself:"+toself);
                                        e.printStackTrace();
                                        continue;
                                }
                        }
                        else {
                                try {
                                        i=i;
                                        //System.out.println("Msg From "+from+", To "+i+":"+message);
                                }
                                catch (Exception e) {
                                        printSyslog("SendMessage(): Exception Printing"
                                                +" to System.out From:"
                                                +from+" i:"+i+"toself:"+toself);
                                        e.printStackTrace();
                                        continue;
                                }
                        }
                        try {
                                if (conn[i].isAuthenticated()) {
                                        conn[i].println(""+from+" says: "+message);
                                }
                        }
                        catch (Exception e) {
                                printSyslog("SendMessage(): Exception Printing to User:"
                                        +i+" From:"+from+" toself:"+toself);
                                e.printStackTrace();
                                continue;
                        }
                }
        }

        /** who() produces a list of users on the Java Talker.
                @param index integer index of the user making the call to .who */
        public void who (int index) {
                int i;

                String location="";
                String timestr="";
                String levelstr="";
                int locsize=0;
                int timesize=0;
                int levsize=0;

conn[index].println("~BB~FY"
+"+----------------------------------------------------------------------------+~N");
conn[index].println("                     ~FYUsers Currently Online~N                                   ");
conn[index].println("~BB~FY"
+"+----------------------------------------------------------------------------+~N\n");

                // To make the columns neat, we get a count of the longest String in each column and
                // size the entire column to match.  We pad the smaller strings with spaces to do this.
                for (i=0; i<conn.length; ++i) {

                        if (conn[i].isAuthenticated()==false) continue;

                        location = area[conn[i].getArea()].getName();
                        if (conn[i].getAfk()) location = "<AFK>";
                        if (location.length() > locsize) {
                                locsize = location.length();
                        }

                        timestr = new Integer(conn[i].getTime()).toString();
                        if (timestr.length() > timesize) {
                                timesize = timestr.length();
                        }

                        levelstr = conn[i].getLevelName();
                        if (levelstr.length() > levsize) {
                                levsize = levelstr.length();
                        }
                }

                // Print the lines of the users connected, padding the shorter names with spaces.
                for (i=0; i<conn.length; ++i) {
                        String stat = "  ";
                        // Don't list unauthenticated users.
                        if (conn[i].isAuthenticated()==false) continue;

                        location = area[conn[i].getArea()].getName();
                        if (conn[i].getAfk()) location = "<AFK>";
                        while (location.length() < locsize) {
                                location = location.concat(" ");
                        }

                        timestr = new Integer(conn[i].getTime()).toString();
                        while (timestr.length() < timesize) {
                                timestr = timestr.concat(" ");
                        }

                        levelstr = conn[i].getLevelName();
                        while (levelstr.length() < levsize) {
                                levelstr = levelstr.concat(" ");
                        }

                        conn[index].println("~FY"+stat+
                                location+"~N | "+
                                timestr+" mins. | "+
                                "~FG"+levelstr+"~N | "+
                                "~FY"+conn[i].getUserName()+"~N "+
                                conn[i].getUserDesc()+"~N");

                }
conn[index].println("\n~BB~FY"
+"+----------------------------------------------------------------------------+~N");
if (conn.length==1) conn[index].println("~FY   Total of "+(conn.length)+" User Online~N");
else conn[index].println("~FY   Total of "+(conn.length)+" Users Online~N");
conn[index].println("~BB~FY"
+"+----------------------------------------------------------------------------+~N");
conn[index].println(" ");
        }

        /** idle() prints a list of the idle times for each of the users on the system.
                @param conn UserConnection of the user making the call to .idle */
        public void idle (UserConnection conn) {
                int i;

                String nameblock="";
                int nameblocksize=0;

                conn.println("~FYUsers Idle Times:~N");
                conn.println("~FY+===============+~N");

                for (i=0; i<this.conn.length; ++i) {
                        if (this.conn[i].isAuthenticated()==false) continue;
                        nameblock = this.conn[i].getUserName();

                        if (nameblock.length() > nameblocksize) {
                                nameblocksize=nameblock.length();
                        }
                }

                for (i=0; i<this.conn.length; ++i) {
                        if (this.conn[i].isAuthenticated()==false) continue;

                        nameblock = this.conn[i].getUserName();
                        while (nameblock.length() < nameblocksize) {
                                nameblock = nameblock.concat(" ");
                        }

                        conn.println(" "+nameblock+"    "+this.conn[i].getIdle()+" minutes.");
                }
                conn.println(" ");
        }

        /** displayColors() displays the supported Color Codes and an example
                of what the output looks like.
                @param conn UserConnection of the user making the call to .colors.*/
        void displayColors(UserConnection conn) {

                conn.println("~FGThis is the supported list of colors~N");
                conn.println("You can embed the color tags in a regular line of text to add color to it");

                conn.out.print("~FR ");
                conn.println("~FR- prints text in Red~N");
                conn.out.print("~FG ");
```

```java
                conn.println("~FG- prints text in Green~N");
                conn.out.print("~FY ");
                conn.println("~FY- prints text in Yellow~N");
                conn.out.print("~FB ");
                conn.println("~FB- prints text in Blue~N");
                conn.out.print("~FP or ~FM");
                conn.println("~FP- prints text in Purple~N");
                conn.out.print("~FS ");
                conn.println("~FS- prints text in Sky Blue~N");
                conn.out.print("~FW ");
                conn.println("~FW- prints text in White~N");
                conn.out.print("~BR ");
                conn.println("~BR- prints background in Red~N");
                conn.out.print("~BG ");
                conn.println("~BG- prints background in Green~N");
                conn.out.print("~BY ");
                conn.println("~BY- prints background in Yellow~N");
                conn.out.print("~BB ");
                conn.println("~BB- prints background in Blue~N");
                conn.out.print("~BP or ~BM");
                conn.println("~BP- prints background in Purple~N");
                conn.out.print("~BS ");
                conn.println("~BS- prints background in Sky Blue~N");
                conn.out.print("~BW ");
                conn.println("~BW- prints background in White~N");
                conn.out.print("~UL ");
                conn.println("~UL- prints text in Underline~N");
                conn.out.print("~LI ");
                conn.println("~LI- prints text in Blinky~N");
                conn.out.print("~N ");
                conn.println("~N- prints text in Normal, or Removes Color~N");
                conn.println(" ");
        }

        /** say() simply displays text received from the user to the users in the area.  Text is
                logged into the area's conversation buffer.
                @param from integer index of the user calling the say() method.
                @param message String containing the message the user is saying. */
        void say(int from, String message) {
                int i;

                for (i=0; i<conn.length; ++i) {
                        if (from==i) {
                                try {
                                        // Don't print to the unauthenticated users.
                                        if (conn[i].isAuthenticated()) {
                                                conn[i].println("You say: "+ message+"~N");
                                        }
                                        continue;
                                }
                                catch (Exception e) {
                                        printSyslog("Say(): Exception Printing Back to User From:"
                                                +from+" i:"+i);
                                        e.printStackTrace();
                                        continue;
                                }
                        }
                        else {
                                try {
                                        // Don't print to the unauthenticated users.
                                        if (conn[i].isAuthenticated()==false) {
                                                continue;
                                        }
                                        else if (conn[i].getArea()!=conn[from].getArea()) {
                                                continue;
                                        }
                                        else {
                                                conn[i].println(""+conn[from].getUserName()
                                                        +" says: "+message+"~N");
                                        }
                                }
                                catch (Exception e) {
                                        printSyslog("Say(): Exception Printing to User:"+i+" From:"+from);
                                        e.printStackTrace();
                                        continue;
                                }
                        }
                }

                // Add this say to the area's conversation buffer.
                area[conn[from].getArea()].addRev(""+conn[from].getUserName()+" says: "+message);
        }


        /** emote() displays the username followed by a string of text with the intent of a third-person
                view of the conversation so emotion like "Jkb Smiles" can be printed. Emotes are
                printed to the area conversation buffer.
                @param from integer index of user performing the .emote
                @param message String containing the message */
        void emote(int from, String message) {
                int i;

                for (i=0; i<conn.length; ++i) {
                        if (from==i) {
                                try {
                                        if (conn[i].isAuthenticated()) {
                                                conn[i].println(conn[i].getUserName()
                                                        +" "+ message+"~N");
                                        }
                                        continue;
                                }
                                catch (Exception e) {
                                        printSyslog("Emote(): Exception Printing Back to"
                                                +" User From:"+from+" i:"+i);
                                        e.printStackTrace();
                                        continue;
                                }
                        }
                        else {
                                try {
                                        if (conn[i].isAuthenticated()==false) {
                                                ;
                                        }
                                        else if (conn[i].getArea()!=conn[from].getArea()) {
                                                continue;
                                        }
                                        else {
                                                conn[i].println(conn[from].getUserName()+" "+message+"~N");
                                        }
                                }
                                catch (Exception e) {
                                        printSyslog("Emote(): Exception Printing to User:"+i
                                                +" From:"+from);
                                        e.printStackTrace();
                                        continue;
                                }
                        }
                }

                // Write this emote to the area's conversation buffer.
                area[conn[from].getArea()].addRev(""+conn[from].getUserName()+" "+message);
        }

        /** quitUser() deletes a UserConnection from the UserConnection array.  In addition to deleting the
                UserConnection, it also resizes the array to attempt to conserve resources.
                @param index integer index of the user being deleted.*/
        void quitUser (int index) {
                int i=0;
                int j=0;
                int size = conn.length - 1;

                if (size<0) size=0;

                conn[index].println("\n~FYDisconnecting you from the Java Talker system.\n\nGoodbye!~N\n");
                printSyslog(""+conn[i].getUserName()+" signed off.");

                // First, create a temporary array to hold the new contents of the UserConnection array.
                try {
                        ctemp = new UserConnection[size];
                }
                catch (Exception e) {
                        printSyslog("QuitUser() Exception in creating a new "
                                +"temporary UserConnection array.");
                        e.printStackTrace();

                }

                try {
                        // If this is the last user online signing off, just reinitialize the whole thing.
                        if (size==0) {
                                conn = new UserConnection[0];
                                ctemp = new UserConnection[1];
                        }

                        // Otherwise, start copying the array from the old to the new,
                        // skipping the one to be deleted.
                        else {
                                for (i=0; i<=size; ++i,++j) {

                                        if (i==index) {
                                                ++i;
                                        }
                                        if (i<=size) {
                                                ctemp[j] = conn[i];
                                                ctemp[j].setIndex(j);
                                        }
                                        else {
                                                ;
                                        }
                                }

                                // Next, move the conn array to point at the memory and delete the ctemp.
                                conn = ctemp;
                                ctemp = null;
                        }
                }
                catch (Exception ee) {
                        printSyslog("QuitUser() Exception Caught while shuffling arrays around");
                        //printSyslog("Details:  i:"+i+" j:"+j+" size:"+size+" index:"+index);
                        ee.printStackTrace();

                }

                // print a note to the system log.
                printSyslog("Deleted "+index+" from the array. Conn:"+conn.length);
        }


        /** shutdown() is called to close all UserConnections and the ServerSocket. The user sockets are
                manually closed.*/
        void shutdown () {
                int i=0;
                int num = 0;
                num = conn.length;

                while (i < num) {
                        try {
                                conn[i].println("\n~FRSystem Shutting Down, Disconnecting You!~N\n");
                                conn[i].saveUserData();
                                (conn[i].socket).close();

                        }
                        catch (Exception e) {
                                printSyslog("Shutdown() Exception caught while closing individual "
                                        +"UserConnections' sockets.");
                                e.printStackTrace();
                        }
                        i++;
                }

                try {
                        (oneconn.socket).close();
                        oneconn=null;
                        s.close();
                }
                catch (Exception e) {
                        printSyslog("Shutdown() Exception caught while closing the ServerSocket "
                                +"and Temporary socket");
                        e.printStackTrace();
                }

                conn = null;
                System.exit(0);
        }

        /** getUserIndex() is used to take a name or partial name and return the index of
                the UserConnection in the array associated with the name.
                @return integer index of the UserConnection
                @param name String containing the name of the user. */
        int getUserIndex(String name) {
                name = name.toLowerCase();
                int i;
                // read through conn structure looking for exact matches.
                for (i=0; i<conn.length; ++i) {
                        String compare = conn[i].getUserName();
                        compare = compare.toLowerCase();
                        //System.out.println("Comparing "+name+" to "+compare);
                        if (name.equals(compare)) return i;
                }

                // read through conn structure looking for partial matches.
                for (i=0; i<conn.length; ++i) {
                        String compare = conn[i].getUserName();
                        compare = compare.toLowerCase();
                        //System.out.println("Comparing "+name+" to "+compare);
                        if (compare.startsWith(name)) return i;
                }

                // If no matches, return -1.
                return -1;
        }

        /** getAreaIndex is used to take a name or partial name and return the array index of the
                Area object associated with the name.
                @return integer index of the Area object.
                @param name String name containing the name of the area. */
        int getAreaIndex(String name) {
                name = name.toLowerCase();
                int i;

                // read through area structure looking for exact matches.
                for (i=0; i<area.length; ++i) {
                        //System.out.println("Comparing "+name+" to "+area[i].getName());
                        if (name.equals(area[i].getName().toLowerCase())) return i;
                }
                // read through area structure looking for partial matches.
                for (i=0; i<area.length; ++i) {
                        //System.out.println("Comparing "+name+" to "+area[i].getName());
                        if ((area[i].getName().toLowerCase()).startsWith(name)) return i;
                }
                return -1;
        }

        /** isUser() is used to check through the list of usernames to find out if the
                user actually exists or not.  This is used in the .smail and user creation
                methods.
                @param name String containing the name we are querying for.
                @return boolean containing the value of whether or not the user exists. */
        boolean isUser (String name) {
                name = name.toLowerCase();
                String sql = "SELECT username FROM user WHERE username=LCASE(\""+name+"\")";
                try {
                        ResultSet rs = stmt.executeQuery(sql);

                        if (rs.next()) return true;
                        else return false;
                }
                catch (Exception e) {
                        printSyslog("Error in isUser("+name+")");
                        e.printStackTrace();
                }
                return false;
        }

        /** getUserDBName() returns the correctly spelled name (in the event of a partial search) in the
                database so that when we create a link in the usermail table, we are correctly linking to
                the right key.
                @return String containing correctly spelled and capitalized name.
                @param name String containing the test value. */
        String getUserDBName (String name) {
                name = name.toLowerCase();
                String sql = "SELECT username FROM user WHERE username=LCASE(\""+name+"\")";
                try {
                        ResultSet rs = stmt.executeQuery(sql);

                        if (rs.next()) return rs.getString(1);
                        else return "";
                }
                catch (Exception e) {
                        printSyslog("Error in getUserDBName("+name+")");
                        e.printStackTrace();
                }
                return "";
        }

        /** goArea() is called by the .go method, which changes the area a user is currently in.
                If no parameters are specified, the list of areas is displayed.
                @return boolean determining wheter or not an error has occurred
                @param index integer index of the user making the goArea() call.
                @param str String containing the area name. */
        public boolean goArea (int index, String str) {
                String sql="";
                if (str.equals("")) {
                        conn[index].println("~FR Usage: .go areaname~N");
                        try {
                                sql="SELECT name FROM area ORDER BY name";
                                ResultSet rs = stmt.executeQuery(sql);

                                conn[index].println("~FY  Areas available:~N");
                                while (rs.next()) {
                                        conn[index].out.println("   "+rs.getString(1));
                                }
                                conn[index].println(" ");
                        }
                        catch (Exception e) {
                                printSyslog("goArea() Exception: SQL: "+sql);
                                e.printStackTrace();
                        }
                        return false;
                }

                // Retrieve the Index of the Destination Area
                int room = getAreaIndex(str);

                if (room==-1) {
                        conn[index].println("~FYNo Such Area \""+str+"\" Exists~N");
                }
                else {

                        // Change the area and display the room information.
                        conn[index].setArea(room);
                        area[room].look(conn[index]);
                }

                return false;
        }

        /** rmail() is used to read all of a user's offline private messages, or smail.
                @param conn UserConnection of the user making the call to the .rmail command. */
        public void rmail (UserConnection conn) {
                try {
                        conn.println("~FYHere is your mail:~N\n");

                        connectDB();
                        String sql = "SELECT mail.id,sender,mdate,mtime,message "
                                +"FROM mail,usermail WHERE mail.id=usermail.mailid AND "
                                +"usermail.username=\""+conn.getUserName()
                                +"\" ORDER BY mail.id";

                        ResultSet rs = stmt.executeQuery(sql);
                        int number = 0;

                        while (rs.next()) {
                                number++;
                                try {
                                        // Pull the Sender, Date, Time and Message from the Database
                                        int id = rs.getInt(1);
                                        String sender = rs.getString(2);
                                        String mdate = rs.getString(3);
                                        String mtime = rs.getString(4);
                                        String message = rs.getString(5);

                                        // Display them to the User.
                                        conn.println(" "+mdate+": "+mtime+": From "+sender+":  "+message);
                                }
                                catch (Exception ex) {
                                        printSyslog("Exception Caught in Rmail");
                                        ex.printStackTrace();
                                }
                        }
                        conn.println("\n~FYTotal of "+number+" Mail Messages~N");
                        conn.println(" ");
                }
                catch (Exception e) {
                        printSyslog("Caught Exception in Rmail()");
                        e.printStackTrace();
                }
        }

        /** smail() is used to create offline private messages.  Using the time and date
                stamps, smail() provides information on when the message was sent, the actual
                message and the user sending it.  smail() creates the row in the mail table
                and the link record in the usermail table.
                @return boolean designating whether or not an error occurred
                @param conn UserConnection of user sending the message
                @param message String containing both the intended recipient and the message. */
        public boolean smail (UserConnection conn, String message) {
                StringTokenizer st = new StringTokenizer(message);
                String to;
                int toindex=-1;
                try {
                        // First, pull off the recipient's name.
                        if (st.hasMoreTokens()) to = st.nextToken();
                        else {
                                conn.println("~FR Usage: .smail recipient message~N");
                                return true;
                        }

                        // Check to see if that user is a real existing user.
                        if (isUser(to)==false) {
                                conn.println("~FR\""+to+"\" is not an existing username.~N");
                                return true;
                        }
                        else {
                                to = getUserDBName(to);
                                toindex = getUserIndex(to);
                        }

                        // Pull off the first token in the message, and concatenate the rest onto it.
                        if (st.hasMoreTokens()) message = st.nextToken();
                        else {
                                conn.println("~FR Usage: .smail recipient message~N");
                                return true;
                        }

                        while (st.hasMoreTokens()) {
                                message = message.concat(" ");
                                message = message.concat(st.nextToken());
                        }


                        connectDB();
                        ResultSet rs;
                        int mailid;
                        String sql;
                        int sqlrows=0;

                        // Since this message goes into the database, escape it.
                        message = escapeUserInput(message);

                        // Write the message to the mail table.
                        try {
                                sql = "INSERT INTO mail (sender,mdate,mtime,message)"
                                        +" VALUES (\""+conn.getUserName()+"\",\""+datestamp
                                +"\", \""+timestamp+"\",\""+message+"\")";
                                sqlrows = stmt.executeUpdate(sql);

                                // Removed due to incompatibilities with the Windows Java virtual machines
                                //rs = stmt.getGeneratedKeys();
                                //rs.next();
                                //mailid = rs.getInt(1);

                                //System.out.println("Executed: "+sql+"; "+sqlrows+" Rows Effected.");

                                sql = "SELECT id FROM mail WHERE sender='"+conn.getUserName()
                                        +"' AND mdate='"+datestamp+"' AND mtime='"+timestamp
                                        +"' AND message='"+message+"'";
                                rs = stmt.executeQuery(sql);
                                rs.next();
                                mailid = rs.getInt(1);
                        }
                        catch (Exception e) {
                                printSyslog("Write(): Exception Inserting Mail into Mail Table");
                                e.printStackTrace();
                                return true;
                        }

                        // Write the User to Message link in the UserMail table.
                        try {
                                sql = "INSERT INTO usermail (username, mailid) VALUES (\""
                                        +to+"\","+mailid+")";
                                sqlrows = stmt.executeUpdate(sql);
                                //System.out.println("Executed: "+sql+"; "+sqlrows+" Rows Effected.");
                        }
                        catch (Exception e) {
                                printSyslog("Write(): Exception Inserting Mail and User into UserMail");
                                e.printStackTrace();
                                return true;
                        }
                }
                catch (Exception e) {
                        printSyslog("Caught Exception in Smail()");
                        e.printStackTrace();
                }

                // Provide a little feedback.
                if (toindex!=-1) this.conn[toindex].println("You Receive a Mail Message");
                else if (toindex==conn.getIndex()) conn.println("You send yourself a mail message.");
                else conn.println("Your message has been sent.");

                return false;
        }

        /** dmail() provides a means of deleting messages from a user's Offline Private Message "box".
                If a user specifies a number, that number of messages is deleted.  If a user specifies
                the word all, all of the messages are deleted.
                @return boolean indicating whether or not an error has occurred.
                @param conn UserConnection of user deleting messages
                @param in String containing number of messages to delete or word all. */
        public boolean dmail (UserConnection conn, String in) {
                if (in.equals("")) {
                        conn.println("~FR Usage:  .dmail all/number of msgs~N");
                        return true;
                }

                int todelete = 0;
                try {

                        // First, connect to the database and get a list of all the user mail.
                        connectDB();
                        String sql = "SELECT mail.id FROM mail, usermail WHERE usermail.username=\""
                                +conn.getUserName()+"\" AND usermail.mailid=mail.id ORDER BY mail.id";
                        ResultSet rs = stmt.executeQuery(sql);
                        int number = 0;

                        if (in.equals("all")) {
                                todelete = -1;
                                conn.println("Erasing all of your Mail...");
                        }
                        else {
                                try {
                                        todelete = new Integer(in).intValue();
                                }
                                catch (Exception e) {
                                        return true;
                                }
                                if (todelete <= 0) return true;
                                conn.println("Erasing "+todelete+" of your Mail Messages...");
                        }
                        while (rs.next()) {
                                number++;
                                if (todelete != -1) {
                                        if (number > todelete) {
                                                --number;
                                                break;
                                        }
                                }

                                try {
                                        // Next, continually read the message index of each message.
                                        int id = rs.getInt(1);
                                        int sqlrows = 0;

                                        // Form the SQL statement to delete the message and execute it.
                                        String sql2 = "DELETE FROM mail WHERE mail.id="+id;
                                        sqlrows = stmt.executeUpdate(sql2);

                                        // Form the SQL statement to delete the link from
                                        // usermail and execute it.
                                        String sql3 = "DELETE FROM usermail WHERE mailid="+id;
                                        sqlrows = stmt.executeUpdate(sql3);
                                }
                                catch (Exception ex) {
                                        printSyslog("Exception Caught in Dmail()");
                                        ex.printStackTrace();
                                }
                        }
                        conn.println("Total of "+number+" Mail Messages Deleted.");
                }
                catch (Exception e) {
                        printSyslog("Caught Exception in Dmail()");
                        e.printStackTrace();
                }
                return false;
        }

        /** setUserProperty() is used by a user to set different attributes of the UserConnection.
                Some of the attributes are description, gender, password, age, email, homepage,
                photo and profile.
                @param conn UserConnection of the user calling .set command
                @param value String containing the attribute and the new value
                @return boolean indicating whether or not the .set was successful. */
        public boolean setUserProperty(UserConnection conn, String value) {

                // If no parameters, print the usage information.
                if (value.equals("")) {
                        conn.println("~FR Usage:  .set property value~N");
                        conn.println("   Valid Properties:");
                        conn.println("   desc - User Description");
                        conn.println("   gender - (male/female/unknown)");
                        conn.println("   password - oldpass newpass");
                        conn.println("   age - Number of years old");
                        conn.println("   email - Email Address");
                        conn.println("   homepage - URL of Homepage");
                        conn.println("   photo - URL of Photo");
                        conn.println("   profile - User Profile");
                        conn.println(" ");
                        return false;
                }

                String value2 = "";
                String property="";

                // Separate the property from the values.
                try {
                        StringTokenizer st = new StringTokenizer(value);
                        property = st.nextToken();
                        while (st.hasMoreTokens()) {
                                value2 = value2.concat(" ");
                                value2 = value2.concat(st.nextToken());
                        }
                        value = value2;
                }
                catch (Exception e) {
                        conn.println("~FRUnable to Set That!~N");
                        return false;
                }

                // Handle the desc property.
                if (property.equals("desc")) {
                        conn.setDesc(value.trim());
                        conn.println("Your description is now: "+value);
                        return true;
                }

                // Handle the gender property.
                if (property.equals("gender")) {
                        String gender="";
                        value = (value.toLowerCase()).trim();
                        if (new String("male").startsWith(value)) gender="Male";
                        else if (new String("female").startsWith(value)) gender="Female";
                        else gender="Unknown";

                        conn.setGender(gender);
                        conn.println("You change your gender to: "+gender);
                        return true;
                }

                // Handle the age property.
                if (property.equals("age")) {
                        int oldage = conn.getAge();
                        int newage = -1;
                        try {
                                newage =new Integer(value.trim()).intValue();
                        }
                        catch (Exception e) {
                                printSyslog("Error Setting Age to: "+value+" by "+conn.getUserName());
                                conn.setAge(oldage);
                                conn.println("~FRIllegal Age, Try again!~N");
                                return false;
                        }
                        conn.setAge(newage);
                        conn.println("You set your age to: "+newage);

                        return true;
                }

                // Handle the password property.
                if (property.equals("password")) {
                        String oldpass="";
                        String newpass="";
                        try {
                                StringTokenizer st = new StringTokenizer(value);
                                oldpass = st.nextToken();
                                newpass = st.nextToken();
                        }
                        catch (Exception e) {
                                conn.println("~FRUsage: .set password oldpassword newpassword~N");
                                return false;
                        }

                        if (oldpass.equals(conn.getPassword())) {
                                conn.setPassword(newpass);
                                conn.println("Your password has been changed.");
                                return true;
                        }
                        else {
                                conn.println("Your old password did not match.  "
                                        +"Your password has not been changed.");
                                return false;
                        }
                }

                // Handle the email property.
                if (property.equals("email")) {
                        conn.setEmail(value.trim());
                        conn.println("Your email address is now set to: "+value);
                        return true;
                }

                // Handle the photo property.
                if (property.equals("photo")) {
                        conn.setPhoto(value.trim());
                        conn.println("Your photo URL is now set to: "+value);
                        return true;
                }

                // Handling for the homepage attribute.
                if (property.equals("homepage")) {
                        conn.setHomepage(value.trim());
                        conn.println("Your homepage URL is now set to: "+value);
                        return true;
                }

                // Handling for the profile attribute.
                if (property.equals("profile")) {
                        conn.setProfile(value.trim());
                        conn.println("Your profile is now set to: "+value);
                        return true;
                }

                // If we haven't found a way to handle the property, print an error.
                conn.println("~FRYou cannot set "+property+"!~n");
                return false;
        }

        /** escapeUserInput() replaces characters that would confuse or potentially
                create security vulnerabilities in the database access portions of the program
                with their escaped equivalents.
                @return String containing the escaped string.
                @param input String containing the unescaped input. */
        public String escapeUserInput (String input) {
                int i=0;
                int length=0;
                StringBuffer in = new StringBuffer(input);
                length = in.length();
                for (i=0; i<length; ++i) {

                        // replace double quotes.
                        if (in.charAt(i)=='"') {
                                if (i==0) { in.insert(0,'\\'); length = in.length(); ++i; continue;}
                                if (in.charAt(i-1)=='\\') continue;
                                else {in.insert(i, '\\'); length = in.length(); ++i; continue; }
                        }

                        // replace single quotes.
                        if (in.charAt(i)=='\'') {
                                if (i==0) { in.insert(0,'\\'); length = in.length(); ++i; continue;}
                                if (in.charAt(i-1)=='\\') continue;
                                else {in.insert(i, '\\'); length = in.length(); ++i; continue; }
                        }

                        // replace escape characters.
                        if (in.charAt(i)=='\\') {
```

```
                if (i==0) { in.insert(0,'\\'); length = in.length(); ++i; continue;}
                if (in.charAt(i-1)=='\\') continue;
                else {in.insert(i, '\\'); length = in.length(); ++i; continue; }
            }
        }
        return in.toString();
    }


    /** setAfk() marks a user as being away from keyboard.
        @param conn UserConnection of user going away.
        @param afkmsg String containing the reason why user is going afk. */
    void setAfk(UserConnection conn, String afkmsg) {
        if (afkmsg.equals("")) afkmsg = "ZZz... Zz... z...";

        conn.println("You go AFK: "+afkmsg);
        area[conn.getArea()].printArea(conn, conn.getUserName()+" goes AFK: "+afkmsg, false);

        conn.setAfk(true);
        conn.setAfkMsg(afkmsg);
    }

    /** remove() is a way for a high-leveled user to manually disconnect users from the Java Talker.
        @return boolean indicating whether or not the .remove was successful
        @param conn UserConnection of the user performing the .remove.
        @param other String containing the name of the user to be .removed. */
    boolean remove (UserConnection conn, String other) {
        if (other.equals("")) {
            conn.println("~FR Usage:  .remove username~N");
            return false;
        }


        int i = getUserIndex(other);
        if (i!=-1) {
            // Can't remove a higher level user.
            if (conn.getLevel() < this.conn[i].getLevel()) {
                conn.println("~FR"+other
                    +" is at a higher level than you. Unable to remove.~N");
                return false;
            }

            try {
                this.conn[i].socket.close();
            }
            catch (Exception e) {
                conn.println("~FRFailed to Remove "+other+"~N");
                return false;
            }
            printSystemMessage("~FYLEAVING US: "+this.conn[i].getUserName()
                +" "+this.conn[i].getUserDesc()+"~N", i);
            quitUser(i);
            return true;
        }
        else {
            conn.println("That user is not signed on.");
            return false;
        }
    }


    /** deleteUser() deletes a user account from the database.
        @return boolean indicating whether or not the .delete was successful
        @param conn UserConnection of the user calling the .delete command.
        @param other String containing name of user to delete. */
    boolean deleteUser (UserConnection conn, String other) {
        if (other.equals("")) {
            conn.println("~FR Usage:  .delete username~N");
            return false;
```

```
                }

                // First, find out if the user is signed on or not and remove them...
                int i = getUserIndex(other);
                if (i!=-1) {
                        if (conn.getLevel() < this.conn[i].getLevel()) {
                                conn.println("~FR"+other+" is at a higher than you. Unable to remove.~N");
                                return false;
                        }

                        try {
                                this.conn[i].socket.close();
                        }
                        catch (Exception e) {
                                conn.println("~FRFailed to Remove "+other+"~N");
                                return false;
                        }
                        printSystemMessage("~FYLEAVING US: "+this.conn[i].getUserName()
                                +" "+this.conn[i].getUserDesc()+"~N", i);
                        quitUser(i);

                }

                // Otherwise, check the user file and see what level the user is at.
                else {
                                UserConnection delete = new UserConnection(this);
                                delete.setUserName(getUserDBName(other));
                                delete.loadUserData();
                                delete.loadUserData();
                                if (conn.getLevel() < delete.getLevel()) {
                                        conn.println("~FR"+other
                                                +" is at a higher than you. Unable to remove.~N");
                                        return false;
                                }
                                delete = null;
                }

                // If at any point prior, if there is not enough reason to delete
                // the user, the flow stops.
                String deletename = getUserDBName(other);

                String sql="";
                int sqlrows=0;

                try {

                        // Delete all references to the user in the database.

                        sql = "SELECT id FROM usermail WHERE username='"+deletename+"'";
                        ResultSet rs = stmt.executeQuery(sql);
                        //System.out.println("deleteUser(): Executed: "+sql);

                        while (rs.next()) {

                                sql = "DELETE FROM mail WHERE id="+rs.getInt(1);
                                sqlrows = stmt.executeUpdate(sql);

                        }
                        sql = "DELETE FROM usermail WHERE username='"+deletename+"'";
                        sqlrows = stmt.executeUpdate(sql);

                        sql = "DELETE FROM user WHERE username='"+deletename+"'";
                        sqlrows = stmt.executeUpdate(sql);

                }
                catch (Exception e) {
                        printSyslog("deleteUser(): Exception, probably SQL:"+sql);
                        e.printStackTrace();
                }
```

```java
                // Report the deletion.
                conn.println("~FY"+deletename+" has been deleted.~N");
                printSyslog(conn.getUserName()+" deleted "+deletename);
                return true;
        }


}  // End of Chat Class
//*****************************************************************************

/** Area object
        @author Jeffrey K. Brown
        <p>Area is the object that represents a virtual area or room.  Each Area
        has a message board, a topic and a room description.  Additionally, each
        Area has a conversation buffer.  Users in a virtual area may carry on
        public online conversation between each other.  Offline public communication
        takes place on a room to room basis as well.</p> */
class Area {
        /** t is a link back to the Talker object so we can reference methods in Talker. */
        Talker t;

        /** This is the Area's index in the Talker Area array. */
        int index;

        /** This is the name of the Area. */
        String name;

        /** This stores the area's topic.  Sometimes, it is necessary to set a topic to direct
                the flow of conversation in a room. */
        String topic;

        /** This is the Area description.  This is used to provide atmosphere and environment
                to this made up virtual room. */
        String description;

        /** This is the Area's conversation buffer.  All online public communication taking
                place in a particular Area is stored here until cleared. */
        String[] rev = new String[20];


        /** Area() is the Area Object Constructor.  We simply link back to the Talker
                object here and initialize some of the variables.
                @param t This is a link back to the Talker Object. */
        public Area (Talker t) {
                this.t = t;
                topic = "none";

                for (int i=0; i<20; ++i) {
                        rev[i] = "";
                }
        }

        /** addRev adds a line of online public communication to the area's conversation buffer.
                @param line is the line of text to be added to the conversation buffer.
                @return boolean representing whether or not an error has occurred. */
        public boolean addRev (String line) {
                int i=0;

                for (i=0; i<20; ++i) {

                        // Either put the line in the first empty line we find...
                        if (rev[i].equals("")) {
                                rev[i] = line;
                                return false;
                        }
                }

                // ... or move all the lines up, and add the new line to the last object.
                for (i=0; i<19; ++i) {
                        rev[i] = rev[i+1];
                }
                rev[19] = line;
                return false;
        }

        /** showRev() displays the Conversation Buffer to the user.
                @param conn is the link to the UserConnection of the user calling the .rev command.
                @return boolean indicating whether or not an error took place. */
        public boolean showRev (UserConnection conn) {
                int i=0;
                conn.println("~FYThe "+name+" Conversation Buffer:~N\n");

                for (i=0; i<20; ++i) {
                        if (rev[i].equals("")) break;
                        conn.println("- "+rev[i]);
                }
                conn.println(" ");

                return false;
        }

        /** clearRev() clears the Conversation Buffer.  This is done by overwriting the
                conversation buffer by empty strings.
                @param conn is a UserConnection of the user calling .cbuffer
                @return boolean indicating whether an error took place in the method. */
        public boolean clearRev (UserConnection conn) {
                int i=0;
                conn.println("Conversation Buffer cleared.");

                for (i=0; i<20; ++i) {
                        rev[i] = "";
                }

                return false;
        }

        /** setDescription() simply sets the Area description of the room.
                @param description is the new description that we want to set.
                @return boolean indicating whether or not an error took place. */
        public boolean setDescription(String description) {
                this.description = description;

                return false;
        }

        /** initArea() builds the Area Array for Talker.  This is accomplished by reading the Area
                table and creating Area objects with the data.  The Area objects are then linked in
                an array and passed back to Talker.
                @param t is a link back to the Talker object.
                @return Area[] which is an array of Area Objects.*/
        public static Area[] initArea (Talker t) {
                Area[] areas = new Area[0];
                Area[] atemp = new Area[1];
                Area onearea;

                // Connect to the Database and run the Query.
                String sql = "SELECT name,description FROM area";
                t.connectDB();

                try {
                        ResultSet rs = t.stmt.executeQuery(sql);
                        int i=0;
                        int j=0;

                        // For each Area record, create a new Area Object.
                        while (rs.next()) {
                                String aname = "";
                                String adesc ="";

                                atemp = new Area[areas.length+1];

                                try {
                                        aname = rs.getString(1);
                                        adesc = rs.getString(2);
                                }
                                catch (Exception e) {
                                        t.printSyslog("Exception Getting Area Info");
                                        e.printStackTrace();
                                }

                                // Set the data from the Area record to the onearea
                                // temporary Area object.
                                onearea = null;
                                onearea = new Area(t);
                                onearea.setName(aname);
                                onearea.setDescription(adesc);
                                onearea.setIndex(i);

                                // Resize the Area array to fit the new Area object.
                                for (j=0; j<areas.length; ++j) {
                                        try {
                                                atemp[j] = areas[j];
                                        }
                                        catch (Exception e) {
                                                t.printSyslog("Area Exception");
                                                e.printStackTrace();
                                        }
                                }

                                // Add it.
                                atemp[j] = onearea;

                                // Reset.
                                areas = atemp;
                                atemp = null;
                                ++i;
                        }
                }
                catch (Exception e) {
                        t.printSyslog("Exception Reading Data from Area Table");
                        e.printStackTrace();
                }
                return areas;
        }

        /** getName() returns the name of the Area.
                @return String name of the area. */
        public String getName () {
                return name;
        }

        /** setName() simply sets the name of the Area to the value being passed.
                @param name the new name to set the Area. */
        public void setName (String name) {
                this.name = name;
        }

        /** setIndex() sets the index of the Area in the Area array
                @param index is the new index of the Array. */
        public void setIndex (int index) {
                this.index = index;
        }

        /** setTopic sets the topic for the Area to the parameter.
                @param conn is a UserConnection to the user calling the .topic command.
                @param topic is a String containing the new topic for the room. */
        public void setTopic (UserConnection conn, String topic) {
                if (topic.equals("")) {
                        topic = "none";
                }

                int length = t.conn.length;

                printArea(conn,""+conn.getUserName()+" sets the topic to: "+topic+"~N",false);
                conn.println("~FYYou set the topic to: "+topic+"~N");

                // Add this to the Conversation buffer
                addRev(""+conn.getUserName()+" sets the topic to: "+topic);
                this.topic = topic;
        }

        /** getIndex() returns the index of the room.
                @return integer index of the room in the Area array. */
        public int getIndex () {
                return index;
        }

        /** look() displays information about the current Area to a user.  It is called when the user
                executes the .look command or changes areas.
                @param conn is the UserConnection calling the .look command or changing areas. */
        public void look (UserConnection conn) {
                String sql="empty";
                int users=0;
                String name="";
                try {
                        // Display the name of the Room and Room Description.
                        conn.println("~FYWelcome to the "+name+" Room!~N");
                        conn.println("\n"+description+"\n");

                        // Display a list of users in the room.
                        conn.println("~FY~ULUsers in the Room:~N\n  ");
                        int length = t.conn.length;
                        name = conn.getUserName();

                        for (int i=0; i<length; ++i) {
                                if (t.conn[i].isAuthenticated()==false) continue;
                                if (t.conn[i].getArea()==index) {
                                        if (name.equals(t.conn[i].getUserName())) continue;
                                        conn.out.println("\t"+t.conn[i].getUserName()+" "
                                                +t.conn[i].getUserDesc());
                                        users++;
                                }

                        }
                        if (users==0) conn.println("\tYou are alone here.");
                        conn.out.println("   ");

                        // Connect to the Database and get the number of messages on the board.
                        t.connectDB();
                        sql = "SELECT count(*) FROM messages, messarea WHERE "
                                +"messarea.areaid=\""+t.area[conn.getArea()].getName()
                                +"\" AND messarea.messid=messages.id";
                        ResultSet rs = t.stmt.executeQuery(sql);
                        rs.next();
                        int msgs = rs.getInt(1);

                        // Print the number of messages on the board.
                        conn.println("~FPThere are ~FY"+msgs+"~FP Messages on the board.~N");

                        // Display the Topic.
                        conn.println("~FPThe current topic is: "+topic+"~N");
                        conn.println(" ");

                }
                catch (Exception e) {
                        t.printSyslog("Exception in Look()\n"+sql);
                        e.printStackTrace();
                }
        }

        /** read() reads the Area's message board and displays it to a user.
                @param conn is the UserConnection making the .read request. */
        public void read (UserConnection conn) {
                try {
                        // Display the name of the board.
                        conn.println("~FY~ULThe "+name+" Message Board:~N\n");

                        // Connect to the Database and read the messages information about this room.
                        t.connectDB();
                        String sql = "SELECT messages.id, messdate, messtime, name, message FROM "
                                +"messages, messarea WHERE messarea.areaid=\""
                                +t.area[conn.getArea()].getName()+"\" AND messarea.messid="
                                +"messages.id ORDER BY messages.id";
                        ResultSet rs = t.stmt.executeQuery(sql);
                        int number = 0;

                        while (rs.next()) {
                                number++;
                                try {
                                        int id = rs.getInt(1);
                                        String messdate = rs.getString(2);
                                        String messtime = rs.getString(3);
                                        String messname = rs.getString(4);
                                        String message = rs.getString(5);

                                        // Print each message.
                                        conn.println(""+messdate+": "+messtime+": From "
                                                +messname+":  "+message+"~N");
                                }
                                catch (Exception ex) {
                                        t.printSyslog("Exception Caught in Message");
                                        ex.printStackTrace();
                                }
                        }

                        // Print the total number of messages.
                        conn.println("\n~FYTotal of "+number+" Messages~N");
                        conn.println(" ");
                }
                catch (Exception e) {
                        t.printSyslog("Caught Exception in Read()");
                        e.printStackTrace();
                }

                // Tell everyone that someone is reading the board.
                printArea(conn, ""+conn.getUserName()+" reads the message board", false);
        }

        /** wipe() erases messages from the Area message board.  Wipe can either erase a
                number of messages or all of the messages, depending on the parameter "in"
                @param conn is the UserConnection making the request.
                @param in is a String containing the arguments that determine the type of wipe. */
        public boolean wipe (UserConnection conn, String in) {

                if (in.equals("")) {
                        conn.println("~FR Usage:  .wipe all/numberofmessages~N");
                        return true;
                }

                int todelete = 0;
                try {
                        // Get the number of messages on the board that can be deleted.
                        t.connectDB();
                        String sql = "SELECT messages.id FROM messages, messarea "
```

```
                                +"WHERE messarea.areaid=\""+t.area[conn.getArea()].getName()
                                +"\" AND messarea.messid=messages.id ORDER BY messages.id";
                        ResultSet rs = t.stmt.executeQuery(sql);
                        int number = 0;

                        // If the string is "all", all messages will be erased, otherwise, read the number.
                        if (in.equals("all")) {
                                todelete = -1;
                                conn.println("Erasing all Messages from the "+name+" Message Board...");
                        }
                        else {
                                try {
                                        todelete = new Integer(in).intValue();
                                }
                                catch (Exception e) {
                                        return true;
                                }
                                if (todelete <= 0) return true;
                                conn.println("Erasing "+todelete+" Messages from the "
                                        +name+" Message Board...");
                        }

                        // Run through the list of messages on the message board, reading the message
                        // ID number of each message.  Then, delete the Message ID from both the Messages
                        // table and the mess-area table.

                        while (rs.next()) {
                                number++;
                                if (todelete != -1) {
                                        if (number > todelete) {
                                                --number;
                                                break;
                                        }
                                }

                                try {
                                        int id = rs.getInt(1);
                                        int sqlrows = 0;

                                        String sql2 = "DELETE FROM messages WHERE messages.id="+id;
                                        sqlrows = t.stmt.executeUpdate(sql2);

                                        String sql3 = "DELETE FROM messarea WHERE messarea.messid="+id;
                                        sqlrows = t.stmt.executeUpdate(sql3);


                                }
                                catch (Exception ex) {
                                        t.printSyslog("Exception Caught in Message");
                                        ex.printStackTrace();
                                }
                        }
                        conn.println("~FYTotal of "+number+" Messages Deleted.~N");
                }
                catch (Exception e) {
                        t.printSyslog("Caught Exception in Wipe()");
                        e.printStackTrace();
                }

                // Display to the room that someone has erased messages from the board.
                printArea(conn, ""+conn.getUserName()+" wipes some messsages from the board.", true);
                return false;
        }

        /** write() allows users to write Offline Public messages in an Area.
                @return boolean indicating whether or not an error has occurred.
                @param conn is a UserConnection linking back to the user calling the .write command.
                @param message is the message the User would like to write on the board. */
        public boolean write (UserConnection conn, String message) {
                try {
                        if (message.equals("")) {
                                conn.println("~FR Usage:  .write message~N");
                                return true;
                        }

                        t.connectDB();
                        ResultSet rs;
                        int messid;
                        int areaid;
                        String sql;
                        String areaname;
                        int sqlrows=0;

                        // Because this message is going into the database, we need to escape it.
                        message = t.escapeUserInput(message);

                        try {
                                // Insert the Message into the Messages table.
                                sql = "INSERT INTO messages (messdate,messtime,name,message) VALUES (\""
                                        +t.datestamp+"\", \""+t.timestamp+"\",\""+conn.getUserName()
                                        +"\",\""+message+"\")";
                                sqlrows = t.stmt.executeUpdate(sql);

                                // Removed Due to Windows Java Incompatibility
                                //rs = t.stmt.getGeneratedKeys();
                                //rs.next();
                                //messid = rs.getInt(1);
                                //System.out.println("Executed: "+sql+"; "+sqlrows+" Rows Effected.");

                                // Select the message ID so we can insert this into the Message-Area table.
                                sql = "SELECT id FROM messages WHERE name='"+conn.getUserName()
                                        +"' AND messdate='"+t.datestamp+"' AND messtime='"+t.timestamp
                                        +"' AND message='"+message+"'";
                                rs = t.stmt.executeQuery(sql);
                                rs.next();
                                messid = rs.getInt(1);
                        }
                        catch (Exception e) {
                                t.printSyslog("Write(): Exception Inserting Message into Messages Table");
                                e.printStackTrace();
                                return true;
                        }

                        try {
                                areaname = t.area[conn.getArea()].getName();
                        }
                        catch (Exception e) {
                                t.printSyslog("Write(): Exception Selecting Area ID"
                                        +" From Area Table\n"+sql);
                                e.printStackTrace();
                                return true;
                        }

                        try {
                                // Insert Message ID and Area ID into messarea table.
                                sql = "INSERT INTO messarea (areaid, messid) VALUES (\""+areaname+"\","
                                        +messid+")";
                                sqlrows = t.stmt.executeUpdate(sql);

                        }
                        catch (Exception e) {
                                t.printSyslog("Write(): Exception Inserting Message and ID into MessArea");
                                e.printStackTrace();
                                return true;
                        }
                }
                catch (Exception e) {
                        t.printSyslog("Caught Exception in Write()");
                        e.printStackTrace();
                }

                // Displays to the room a message informing them that someone wrote a new message.
                printArea(conn, ""+conn.getUserName()+" writes a message on the board.", false);
                conn.println("You write a message on the board.");

                return false;
        }


        /** printArea() displays a line of text to all of the users in a particular room.
                @return boolean indicating whether an error has taken place.
                @param conn is a link back to the UserConnection to which we compare
                the other users to when we determine which area to print to.
                @param message is a String with the message that we want to display.
                @param touser is a boolean indicating whether or not we print
                the message to the user "conn". */
        public boolean printArea(UserConnection conn, String message, boolean touser) {
                int areanum = conn.getArea();
                int usernum = conn.getIndex();
                int i = 0;

                for (i=0; i<t.conn.length; ++i) {
                        if ((i==usernum) && (touser==false)) continue;

                        if (t.conn[i].getArea()==areanum) {
                                if (t.conn[i].isAuthenticated()==false) continue;
                                t.conn[i].println(message+"~N");
                        }
                }
                return false;
        }

}

//********************************************************************************************88
/** TalkerTimer System Timer Object
        @author Jeffrey K. Brown
        <p>The TalkerTimer is a Threaded object that continually montiors the UserConnection
        array every 60 seconds.  At each 60 second mark, TalkerTimer gets the time and date
        from the system (to make sure we are reading each minute).  Next, it increments the idle
        and time counters for each UserConnection.  Finally, it disconnects users who have been
        idle past the thresholds for authenticated and unauthenticated users.  The purpose of
        TalkerTimer was to save system resources by not keeping UserConnections open and reading.
        </p>
*/

class TalkerTimer extends Thread {

        /** This is a link back to the main Talker object.  We use it to refer back to
                the array of UserConnections and to call InitDate() and other methods. */
        Talker t;

        /** TalkerTimer() Constructor simply links back to the calling Talker and starts the thread.
                @param t is a link back to the calling Talker object. */
        public TalkerTimer (Talker t) {
                this.t = t;
                start();
        }

        /** run() does all the work of the TalkerTimer.
                <p>First, the Thread is told to sleep for 60 seconds.  After waking, the
                date and time are read from the System.  Next, the TalkerTimer reads through
                the entire UserConnection array in Talker and increments the online and idle
                times by 1 minute.  Finally, the TalkerTimer run() compares the idle times
                the thresholds for disconnection and if the UserConnection is beyond the
                threshold, the user is disconnected from the system. */
        public void run () {
                int length=0;
                int idle=0;
                while (true) {

                        try {
                                // First, sleep.
                                Thread.sleep(60000);  // 60 seconds

                                try {
                                        // Read the System Date and Time
                                        t.initDates();
                                }
                                catch (Exception e) {
                                        t.printSyslog("TalkerTimer: Error with the time stuff...");
                                        e.printStackTrace();
                                        continue;
                                }

                                length = t.conn.length;

                                if (length > 0) {

                                        // Run through each UserConnection in the array.
                                        for (int i=0; i<length; ++i) {

                                                // Increment the Idle and Online Times
                                                t.conn[i].addMinute();

                                                // Read the number of minutes idle.
                                                idle = t.conn[i].getIdle();

                                                // Check the Authenticated User Thresholds
                                                if (t.conn[i].isAuthenticated()==true) {

                                                        // Give a Warning
                                                        if (idle == 55) {
                                                        t.conn[i].println("~FPYou have been idle 55"
                                                        +" minutes."
                                                        +"\nYou will be autoremoved in 5 "
                                                        +"\nminutes unless you respond.~N");
                                                                continue;
                                                        }

                                                        // Actually Disconnect the user.
                                                        // 60 minutes when Thread Sleep is 60000
                                                        if (idle >= 60) {
                                                t.conn[i].println("~FRAssuming you are timed out."
                                                +"  Disconnecting you.~N");
                                                t.printSyslog("Disconnected "
                                                +t.conn[i].getUserName()
                                                +" after 60 minutes of inactivity");
                                                                t.conn[i].quit();
                                                                continue;
                                                        }
                                                }

                                                // Check the Unauthenticated User Thresholds
                                                if (t.conn[i].isAuthenticated()==false) {
                                                        // 5 minutes when Thread Sleep is 60000
                                                        if (idle >= 5) {
                                                t.conn[i].println("\n~FRAssuming you are timed out."
                                                +"\nDisconnecting you.~N");
                                                                t.conn[i].socket.close();
                                                                t.quitUser(i);
                                                                continue;
                                                        }
                                                }
                                        }
                                }
                        }
                        catch (Exception e) {
                                t.printSyslog("TalkerTimer::run() Exception caught...");
                                e.printStackTrace();
                                continue;
                        }
                }
        }
}
```

Appendix B:

Chatlet Applet Source Code

This Appendix contains the source code used to build the Chatlet Java Applet.

```java
import java.awt.*;
import javax.swing.*;
import java.net.*;
import java.io.*;
import java.awt.event.*;

/** SocketPrinter Object
        @author Jeffrey K. Brown
        <p>The SocketPrinter simply reads from the InputStream and displays the
        output to the ScrollPane.</p>
*/
class SocketPrinter extends Thread {
        /** This is the BufferedReader that reads data from the Socket */
        BufferedReader in;

        /** This Chatlet object is used to connect back to the calling Chatlet. */
        Chatlet c;

        /** In the constructor, we start the BufferedReader reading on the socket
        and start the thread.  We also read the first few lines of output out of
        the socket so we can send data like special environment command codes
        without confusing less technical users.
        @param s Socket we are reading from and writing to.
        @param c Chatlet link back to the main Applet
        */
        SocketPrinter(Socket s, Chatlet c) {
                this.c = c;

                // Create a means of reading from the Socket
                try {
                        in = new BufferedReader(new InputStreamReader (s.getInputStream()));;
                }
                catch (IOException ioe) {
                         ioe.printStackTrace();
                }

                // We read the first 12 lines and discard them.
                // This allows us time to send our control codes
                // and log in without confusing the user.
                int i=12;
                while (i>0) {
                        try {
                                if ((in.ready())) {
                                        in.readLine();
                                        --i;
                                }
                        }
                        catch (IOException ie) {
                                --i;
                        }
                }

                // Start the Thread
                start();
        }

        /** This is the method started by thread.  We set the scrolling pane
        to the Maximum, check the Socket Input Stream for input (printing if
        there) and then sleep within the while loop of this method. */
        public void run () {
                int count=0;
                String str="";;
                try {
                        while (true) {

                                c.jsp.setMaximum();

                                try {
                                        if ((in.ready())) {
                                                str = in.readLine();
                                                if (!str.equals("")) {
                                                        c.printText("\n"+str);

                                                }
                                        }
                                        Thread.sleep(100);
                                        count ++;

                                }
                                catch (Exception e) { continue; }

                        }
                }
                catch (Exception e) { e.printStackTrace(); }
        }

}

/** JkbScrollPane is an extension of a JScrollPane that allows us to call a
method, setMaximum(), which basically scrolls the scrollpane to the bottom
of the pane. */
class JkbScrollPane extends JScrollPane {

        /** Jkbscrollpane Constructor.  This defines the parameter jta to
        be a scrollable pane.  In this constructor, we link jta to the
        ScrollPane and specify to never display the scrollbars.
        @param jta JTextArea where the data is displayed. */
        public JkbScrollPane(JTextArea jta) {
                super(jta,JScrollPane.VERTICAL_SCROLLBAR_NEVER,
                        JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

                // I wanted to make the Chatlet scroll, but it
                // was never really stable enough.
                //super(jta,JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                //      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        }

        /** This method sets the vertical scrollbar to the maximum value,
        or the bottom of the pane.  By doing this, the scrollpane scrolls
        down automatically. This, however, was never really as stable as
        I felt it should have been, so we must continually call it to make
        sure the User Interface displays that needs to be displayed.*/
        public void setMaximum() {
                verticalScrollBar.setValue(verticalScrollBar.getMaximum()+10);
        }
}


/** The Chatlet is the main Applet of the Chatlet.  Using the JkbScrollPane,
the JTextField and the SocketPrinter, we are able to read input from the user
and pass it along to the Java Talker by the socket.  The SocketPrinter reads
the Incoming Data from the socket and displays it on the JkbScrollPane.  We
also implement the ActionListener so we can send data when the user hits Enter. */
public class Chatlet extends JApplet implements ActionListener {

        /** This JTextField is used to read input from the user */
        JTextField jtf;

        /** The JTextArea is used to display data read from the Socket */
        public JTextArea jta;

        /** The JkbScrollPane allows us to make the JTextArea appear to scroll */
        JkbScrollPane jsp;

        /** Communication to and From the Java talker is done using this Socket. */
        Socket sock;
        /** The SocketPrinter continuously reads the socket and displays the data
        on the JTextArea. */
        SocketPrinter sp;

        /** The PrintWriter is used to print data received from the JTextField
        to the Socket. */
        PrintWriter out;

        /** This method connects the socket to the Java Talker and performs some
        of the setup.  First, we read the parameter specifying the Java Talker's
        hostname and connects to it.  Next, we assign a PrintWriter for writing
        to the Java Talker.  Then, we send the code to turn the Colors Off and
        the username and password to the system.  Finally, we connect the
        SocketPrinter which starts reading from the Java Talker. */
        public void runChatlet() {
                try {

                        String javahost = getParameter("javahost");
                        sock = new Socket (javahost, 2122);

                        out = new PrintWriter(sock.getOutputStream(), true);

                        out.println("CHATLET_COLOROFF");
                        out.println(getParameter("username").trim());
                        out.println(getParameter("password").trim());



                        sp = new SocketPrinter(sock, this);
                }
                catch (Exception e) {
                        printText(e.getMessage());
                }
        }


        /** This method does the setup of the Applet's User Interface.
        Once the Interface is set up, we being the listening by calling
        the runChatlet() method. We also add the ActionListener so we
        can react to User Input. */
        public void init () {
                Container c = getContentPane();
                c.setLayout(new FlowLayout());
                jtf = new JTextField(80);
                jta = new JTextArea(20,80);
                jta.setLineWrap(true);
                jta.setWrapStyleWord(true);
                jta.setEditable(false);
                Font f = new Font ("Courier",Font.PLAIN,12);
                jta.setFont (f);
                jtf.setFont (f);
                jsp = new JkbScrollPane(jta);
                //jta,JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                //      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

                jtf.addActionListener(this);
                c.setBackground(new Color(0,0,140));

                c.add(jsp);
                c.add(jtf);

                runChatlet();
        }

        /** This method appends text to the end of the jta JTextArea.
        @param s String containing the text to be appended to the JTextArea
        */
        public void printText (String s) {
                jta.append(s);
```

```java
        }

        /** This method is called to react to User Input.  The listener
        listens to the JTextField, jtf and when input is found, it is
        sent to the Java Talker through the out PrintWriter.
        @param ae ActionEvent thrown by the JTextField when a user sends input.
        */
        public void actionPerformed (ActionEvent ae) {
                String message = jtf.getText();
                jtf.setText("");
                out.println(message);
        }
}
```

Appendix C:

Chatlet Web Interface Source Code

This Appendix contains the source code used to build the Chatlet Web Interface.

```
<html>
<!--
Chatlet.php
This is the Main File for the Web Interface portion of the Java Talker Chatlet
In this document, we connect the database and create the Interface.  This page
is divided by function into two sections.  The state of the page is maintained
by the variable $page.
-->

<body background="background.jpg" text="#FFFFFF" alink="#FFFFFF" vlink="#FFFFFF" link="#FFFFFF">

<?
        $version = "0.3.3";

        // First, we will include settings.php, which contains data like the
        // Database authentication, Java Talker, and System settings.  By
        // putting this data into a separate file, we allow a little more
        // customizability.  The DB.php is the PEAR library, which talks to
        // the database at a higher level of abstraction than the MySQL
        // libraries.  This also will allow us to use a different database
        // if we decided to use one.
        require_once('settings.php');
        require_once('DB.php');

        // We connect to the Database using the PEAR API.
        $db = DB::connect("mysql://$dbuser:$dbpass@$dbhost/$dbname");
        if (DB::isError($db)) {
                echo ("<p>Error Connecting to Database</P>");
                die();
        }


        /************************************************************************************/
        // Using a variable to maintain the state allows me to contain elements
        // of my dynamic page system in a single file.  In this file, we have
        // the authentication form and the authenticated interface.  In this
        // section, the $page varaiable has not yet been set, which is what
        // happens prior to authentication.  When the user types the username
        // and password, it is submitted to the database along with the $page
        // variable being set to 1, so the next section is used.

        if (!$page) {

                echo ("<p>");

                // Submit the Form Results back to this page.  $phpself is the
                // PHP variable containing the address of this page
                echo ("<form action=\"$phpself\" method=post>");
                echo ("<table>");

                // This is the text box in which the user enters their name.
                echo ("<tr><td>Username:</td>");
                echo ("<td><input type=text name=username size=25 maxlength=25></td></tr>");

                // This is the text box in which the user puts the password.
                echo ("<tr><td>Password:</td>");
                echo ("<td><input type=password name=password size=25 maxlength=25></td></tr>");
                echo ("</table>");

                // Set the $page state variable to use the next page
                echo ("<input type=hidden name=page value=1>");
                echo ("<input type=submit value=\"Log In\">");
                echo ("</form>");
                echo ("</p>");
        }
        /************************************************************************************/
        // This second page is executed after the authentication information is
        // submitted in the first section.  If the authentication is successful,
        // we provide the Chatlet interface to the user.  The username and
        // passwords are passed directly to the Chatlet Java Applet.  The options
        // to use the Web Based message boards, Examine and Mail are also provided.

        if ($page==1) {

                // First, we perform the query to get the username and password to
                // Authenticate the user.  Notice how we send the password request
                // to the database as encrypted and unencrypted.  This is to handle
                // Windows-based databases that do not support the encrypt function.

                $sql = "SELECT username FROM user WHERE username='$username' AND
(user.password=encrypt('$password','PW') OR user.password='$password')";
                $result = $db->query($sql);

                if ($result->fetchInto($row)) {

                        // If a database Error occurs, we treat it like a bad
                        // username and password combination.  We print out an
                        // "Invalid Password" and then use JavaScript to return
                        // the user to the Username/Password entry page.

                        if (DB::isError($result)) {
                                echo ("<p><b>Invalid Username and Password</b></p>");
                                echo ("<script language=\"JavaScript\">");
                                echo ("setTimeout(\"location='$phpself?page='\",3000);");
                                echo ("</script>");
                                die;
                        }
                        else {
                                // Then we read the username from the database
                                $uname=$row[0];;
                        }
                }

                // If we do not get any results from the Query, the username
                // password combination does not exist.  Therefore, we produce
                // an "Invalid Password" and then use JavaScript to return
                // the user to the Username/Password entry page.

                else {
                        echo ("<p><b>Invalid Username and Password</b></p>");
                        echo ("<script language=\"JavaScript\">");
                        echo ("setTimeout(\"location='$phpself?page='\",3000);");
                        echo ("</script>");
                        die;
                }

                // Now that the User is successfully logged in, we display the user
                // interface.  This interface is displayed in the form of an HTML
                // table with a column containing the message boards from each area,
                // a column containing the link for the User Mail and a listing of
                // the users' names so by clicking them, you can get information
                // about them.

                echo ("<p>");
                echo ("<table cellpadding=2 cellspacing=2>");
                echo ("<tr><th colspan=3 align=center>Java Talker Chatlet</th></tr>");
                echo ("<tr><td valign=top><b>Message Boards</b></td>");

                echo ("<td align=center>");
```

```
// In this section, we print the form that will make the request to
// the Online Mail function.  This function is stored in utility.php
// which contains different scripting based on the type variable.
// The username and password are passed to this script so malicious
// users can't attempt to use cross-site scripting to attempt to
// pull someone else's mail.

echo ("<form method=post action=\"utility.php\" target=\"_blank\">");
echo ("<input type=hidden name=\"type\" value=\"mail\">");
echo ("<input type=hidden name=\"username\" value=\"$username\">");
echo ("<input type=hidden name=\"password\" value=\"$password\">");
echo ("<input type=submit name=\"button\" value=\"Check your Mail\">");
echo ("</form>");
echo ("</td>");


echo ("<td valign=top align=right><b>Users</b></td></tr>");
echo ("<tr>");

// Next, we produce a list of Areas on the system so we can read
// message boards.  Each area name is anchored by a hyperlink to
// the method (also contained in utility.php) that prints the
// message board contents.

echo ("<td valign=top>");

// Get the List and print the names.

$sql = "SELECT name FROM area ORDER BY name";
$result = $db->query($sql);

while ($success = $result->fetchInto($row)) {

        // If we get a Database error, we print an unavailable
        // message and stop printing the Message Board names.
        if (DB::isError($success)) {
                echo ("Message Boards");
                echo ("<br>Unavailable");
                break;
        }
        else {
                // We print the links in the form of a GET request to call
                // the message board function and display the contents of
                // the board in a new window.
                echo ("<br>");
                echo ("<a href=\"utility.php?type=messageboard&board=$row[0]\"");
                echo (" target=\"_blank\">");
                echo ("<small>$row[0]</small></a>");
        }
}

echo ("</td>");

echo ("<td valign=top>");
echo ("<p align=center> ");

// Here, we call the Java Applet containing the Java Chatlet
// Applet.  We pass a few parameters to the Applet.  We pass the
// Username and Password, so the user does not need to retype
// them.  Additionally, we pass the Java Talker Hostname to the
// System so that we don't have to recompile the Chatlet Applet
// if we change the hostname.

echo ("<applet width=600 height=375 code=Chatlet.class name=Chatlet> ");
echo ("<param name=username value=\"$username\">");
echo ("<param name=password value=\"$password\">");
echo ("<param name=javahost value=\"$javahost\">");
echo ("</applet> ");
echo ("</p> ");
```

```
                echo ("</td>");

                // Finally, we display the list of users with the links for the
                // Online Examine.  Because the usernames are displayed in a simple
                // columnar list, the page could be distorted if there were many
                // users.  This display should be modified to a web form that can
                // produce the list of users in some scrollable box that submits
                // to a new window.

                echo ("<td valign=top align=right>");
                $sql = "SELECT username FROM user ORDER BY username";
                $result = $db->query($sql);

                while ($success = $result->fetchInto($row)) {

                        // Database Errors cause the Online Examine
                        // Unavailable error message.

                        if (DB::isError($success)) {
                                echo ("Online Examine");
                                echo ("<br>Unavailable");
                                break;
                        }

                        // Otherwise, the usernames are linked to the utility.php
                        // examine method.

                        else {
                                echo ("<br>");
                                echo ("<a href=\"utility.php?type=examine&user=$row[0]\"");
                                echo (" target=\"_blank\"><small>$row[0]</small></a>");
                        }

                }

                echo ("</td>");

                echo ("</tr>");
                echo ("</table>");

        }
        /*************************************************************************************/

?>

<p>&nbsp</p>
<p>&nbsp</p>
<hr>

<!--
At the very end, we provide a footer to be displayed on all of the pages that
has links to the Java Talker Weblog and the Java Talker Javadocs.  This may be
interesting to users who are curious about how the Java Talker works.
-->

<p>
<small>JavaTalker Chatlet Version <?=$version?><br>
<a href="http://jkb.tdf.ca/weblog/" target=_blank>Java Talker Weblog</a></small>
<a href="http://jkb.tdf.ca/weblog/javadoc" target=_blank>Java Talker Javadocs</a></small>
</p>

</body>
</html>
```

```php
<?
/*
        This file contains settings used by the Chatlet Web Interface.
        It is included by the Chatlet.PHP page and the Utility.PHP
        There are variables for both the Java Applet and the Database.
*/

        // This Variable has the Database Name
        $dbname = "jkbtalker";

        // This is the username for the Database Table.
        $dbuser = "jkb";

        // This is the password for the Database
        $dbpass = "lk139kaks93";

        // This is the Hostname for the Database
        $dbhost = "localhost";

        // This is the Hostname for the Java Applet.
        $javahost = "moose.universe.net";
?>
```

```
<html>
<!--
Utility.php
This file is used as a target for Chatlet.php at which to post form results.
Like Chatlet.php, the page is divided using the $type variable, which defines
the type of form results to produce.
-->
<body background="background.jpg" text="#FFFFFF" alink="#FFFFFF" vlink="#FFFFFF" link="#FFFFFF">

<?

// First, we will include settings.php, which contains data like the
// Database authentication, Java Talker, and System settings.  By
// putting this data into a separate file, we allow a little more
// customizability.  The DB.php is the PEAR library, which talks to
// the database at a higher level of abstraction than the MySQL
// libraries.  This also will allow us to use a different database
// if we decided to use one.
require_once('settings.php');
require_once  ('DB.php');

// We connect to the Database using the PEAR API.
$db = DB::connect("mysql://$dbuser:$dbpass@$dbhost/$dbname");
if (DB::isError($db)) {
        echo ("<p>Error Connecting to Database<br>Command Unavailable</P>");
        die();
}

// If the $type variable is not set, we produce an error message.
if (!$type) {
        echo ("<p>No Command Specified</p>");
        die;
}

// This command is the Online Examine Command.  Using this command, users
// are able to produce information about other users.  In this version,
// the examine command provides access to a user's description, level,
// profile, time, gender, age, email address, homepage and photo.  The
// data is displayed in a neat HTML table.

if ($type=="examine") {
        echo ("<p align=center>");

        // Read the desired information and pass it to the database.

        $sql = "SELECT
description,level,profile,totaltime,gender,age,email,homepage,photo,colorpref,lastsite,laston FROM user
WHERE username='$user'";
        $result = $db->query($sql);

        echo ("<table width=50%>");
        while ($success = $result->fetchInto($row)) {

                // Error if there is a problem.
                if (DB::isError($success)) {
                        die ($success->getMessage());
                }

                // Read the Output from the Results and Produce
                // the HTML table with the data.
                else {
                        $description = $row[0];
                        $level = $row[1];
                        $profile = $row[2];
                        $totaltime = $row[3];
                        $gender = $row[4];
                        $age = $row[5];
                        $email = $row[6];
                        $homepage = $row[7];
                        $photo = $row[8];

                        echo ("<tr><td colspan=2><big>$user $description</td></big></tr>");
                        echo ("<tr><td>Level</td><td>$level</td></tr>");
                        echo ("<tr><td>Gender</td><td>$gender</td></tr>");
                        echo ("<tr><td>Age</td><td>$age</td></tr>");
                        echo ("<tr><td>Photo</td><td>$photo</td></tr>");
                        echo ("<tr><td>Homepage</td><td>$homepage</td></tr>");
                        echo ("<tr><td>Email</td><td>$email</td></tr>");
                        echo ("<tr><td>Total Time</td><td>$totaltime</td></tr>");
                        echo ("<tr><td colspan=2>Profile:<br>$profile</td></tr>");
                }
        }
        echo ("</table>");
        echo ("</p>");
}

// This next section is the Online Message Board command.  This displays
// the Message Board for a particular area defined by the $board variable
// in a neat HTML Table.

if ($type=="messageboard") {
        echo ("<p align=center><big><big><strong>The $board Message Board</strong></big></big></p>");

        // We select message attributes from the database based on the
        // requested message board.
        $sql = "SELECT messages.id, messdate, messtime, name, message FROM messages, messarea WHERE messarea.areaid=\"$board\" AND messarea.messid=messages.id ORDER BY messages.id";
        $result = $db->query($sql);

        $i=0;
        echo ("<table width=100%>");

        // Next, we read and process the results
        while ($success = $result->fetchInto($row)) {

                if (DB::isError($success)) {
                        die ($success->getMessage());
                }

                // And Print the Table.
                else {

                        $from = $row[3];
                        $mdate = $row[1];
                        $mtime = $row[2];

                        echo ("<tr>");
                        echo ("<td bgcolor=\"#666666\" width=0 valign=top>");
                        echo ("$mdate: $mtime:");
                        echo ("<br>From <a href=\"utility.php?type=examine&user=$from\">$from</a>:</td>");
                        echo ("<td bgcolor=\"#FFFFFF\" valign=top width=70%>");
                        echo ("<font color=\"#000000\">$row[4]</font></td>");
                        echo ("</tr>");
                        $i++;
                }
        }
        echo ("</table>");
        echo ("<p align=center><big><strong>Total of $i Messages</strong></big></p>");
}
```

```
// This last section deals with the Online Mail Form Results page.
// The $type variable, along with the username and password are sent
// via HTTP POST to this form.  The username and password are used to
// get the user id and if found, it is used to get the mail messages
// for that user.

if ($type=="mail") {
        echo ("<p align=center><big><big><strong>Your Mail</strong></big></big></p>");

        // Here we select the mail attributes, provided the username and password match.
        $sql = "SELECT sender,mdate,mtime,message,mail.id FROM mail,usermail,user WHERE
mail.id=usermail.mailid AND usermail.username=\"$username\" AND usermail.username=user.username AND
(user.password=encrypt('$password','PW') OR user.password='$password') ORDER BY mail.id";
        $result = $db->query($sql);

        $i=0;
        echo ("<table width=100%>");

        // Read and Process the Results
        while ($success = $result->fetchInto($row)) {
                if (DB::isError($success)) {
                        echo ("<p>Command Unavailable</p>");
                        die;
                }

                // Display the mail messages in a net HTML table.
                else {
                        $from = $row[0];
                        $mdate = $row[1];
                        $mtime = $row[2];
                        $message = $row[3];

                        echo ("<tr>");
                        echo ("<td bgcolor=\"#666666\" width=0 valign=top>");
                        echo ("$mdate: $mtime:");
                        echo ("<br>From <a href=\"utility.php?type=examine&user=$from\">$from</a>:</td>");
                        echo ("<td bgcolor=\"#FFFFFF\" valign=top width=70%>");
                        echo ("<font color=\"#000000\">$message</font></td>");
                        echo ("</tr>");
                        $i++;
                }
        }
        echo ("</table>");
        echo ("<p align=center><big><strong>Total of $i Messages</strong></big></p>");
}

// Finally, we close the database connection.
$db->disconnect();
?>

</body>
</html>
```

Appendix D:

SetupDatabase Source Code

This Appendix contains the source code used to build the SetupDatabase program for building the Java Talker database.

```
import java.io.*;
import java.sql.*;


/** SetupDatabase builds a new Java Talker database.  You will need to change the
        username, password, and database strings to match your database account. */
public class SetupDatabase {

        /** Change this to your Database Username */
        String username="dbuser";
        /** Change this to your Database Password */
        String password="dbpass";
        /** Change this to your Database Name */
        String database="jkbtalker";

        /** This is a connection to the Database. */
        Connection DBConnection = null;

        /** This statement is used to execute SQL commands in the
        database and return data. */
        Statement stmt = null;

        /** This is the main method that basically calls the constructor */
        public static void main (String[] args) {
                SetupDatabase sd = new SetupDatabase();
        }

        /** The Constructor simply calls the buildDatabase() method,
        which does the work of SetupDatabase */
        public SetupDatabase() {
                if (buildDatabase()==false) {
                        System.out.println("Errors have occurred.");
                        System.out.println("Database was NOT successfully populated.");
                }
                else {
                        System.out.println("Database was Successfully created.");
                }
        }

        /** buildDatabase() connects to the database and executes the updates required to
                get the tables in the database created.
                @return boolean value of whether or not things completed successfully */
        public boolean buildDatabase() {
                String connection = "jdbc:mysql://localhost:3306/"+database;
                String sql = "";
                int sqlrows=0;


                //***********************************************************
                System.out.println("Initializing MySQL Drivers");

                try {
                        Class.forName("com.mysql.jdbc.Driver").newInstance();
                        DBConnection = DriverManager.getConnection(connection, username, password);
                        stmt = DBConnection.createStatement();
                        stmt.setEscapeProcessing(true);
                }
                catch (Exception e) {
                        e.printStackTrace();
                        return false;

                }
                //***********************************************************
                System.out.print("Creating User Table...");
                sql = "CREATE TABLE user ("
                        +" username varchar(20) NOT NULL default '',"
                        +" password varchar(20) NOT NULL default '',"
                        +" description varchar(40) default 'a new user',"
                        +" level int(11) NOT NULL default '0',"
                        +" profile text,"
                        +" totaltime int(11) default '0',"
                        +" gender varchar(10) NOT NULL default 'Unknown',"
                        +" age int(11) default '0',"
                        +" email varchar(60) default 'n/a',"
                        +" homepage varchar(60) default 'n/a',"
                        +" photo varchar(60) default 'n/a',"
                        +" colorpref int(11) default '2',"
                        +" lastsite varchar(60) default 'n/a',"
                        +" laston varchar(30) default 'n/a',"
                        +" PRIMARY KEY  (username)"
                        +") TYPE=MyISAM;";

                try {
                        sqlrows = stmt.executeUpdate(sql);
                        System.out.println(" Done");
                }
                catch (Exception e) {
                        System.out.println(" Failed");
                        return false;
                }


                //***********************************************************
                System.out.print("Populating User Table With Default Users...");
                sql = "INSERT INTO user VALUES ('Sysop','PWfklK0xQkC0o','System Operator',5,'This is the
default user on the
system',1,'Unknown',18,'your@emailaddress.here','darkages.darkfantastic.net/','darkages.darkfantastic.net/n
one.jpg',2,'/127.0.0.1, Port: 1045','Wed 10/08/2003, 19:45');";

                try {
                        sqlrows = stmt.executeUpdate(sql);
                        System.out.println(" Done");
                }
                catch (Exception e) {
                        System.out.println(" Failed");
                        return false;
                }



                //***********************************************************
                System.out.print("Creating Area Table...");
                sql = "CREATE TABLE area ("
                        +"  id int(11) NOT NULL auto_increment,"
                        +"  name varchar(50) default NULL,"
                        +"  description text,"
                        +"  idx int(11) NOT NULL default '0',"
                        +"  PRIMARY KEY  (id)"
                        +") TYPE=MyISAM;";

                try {
                        sqlrows = stmt.executeUpdate(sql);
                        System.out.println(" Done");
                }
                catch (Exception e) {
                        System.out.println(" Failed");
                        return false;
                }
                //***********************************************************
                System.out.print("Populating Area Table With Default Areas...");
                sql = "INSERT INTO area VALUES (1,'Main','This is the Main Room of the Java Talker!
Welcome to it',0);";

                try {
                        sqlrows = stmt.executeUpdate(sql);
                        System.out.println(" Done");
                }
                catch (Exception e) {
                        System.out.println(" Failed");
                        return false;
                }


                //***********************************************************
                System.out.print("Creating Messages Table...");
                sql = "CREATE TABLE messages ("
                        +"  id int(11) NOT NULL auto_increment,"
                        +"  name varchar(25) default NULL,"
                        +"  messdate varchar(30) default NULL,"
                        +"  messtime varchar(30) default NULL,"
                        +"  message text,"
                        +"  PRIMARY KEY  (id)"
                        +") TYPE=MyISAM;";

                try {
                        sqlrows = stmt.executeUpdate(sql);
                        System.out.println(" Done");
                }
                catch (Exception e) {
                        System.out.println(" Failed");
                        return false;
                }


                //***********************************************************
                System.out.print("Populating Message Table with Initial Messages...");
                sql = "INSERT INTO messages VALUES (1,'Jkb','Mon 08/11/2003','22:41','Welcome to the Java
Talker!');";

                try {
                        sqlrows = stmt.executeUpdate(sql);
                        System.out.println(" Done");
                }
                catch (Exception e) {
                        System.out.println(" Failed");
                        return false;
                }


                //***********************************************************
                System.out.print("Creating Area-Messages Table...");
                sql = "CREATE TABLE messarea ("
                        +"  id int(11) NOT NULL auto_increment,"
                        +"  messid int(11) NOT NULL default '0',"
                        +"  areaid varchar(25) default NULL,"
                        +"  PRIMARY KEY  (id)"
                        +") TYPE=MyISAM;";

                try {
                        sqlrows = stmt.executeUpdate(sql);
                        System.out.println(" Done");
                }
                catch (Exception e) {
                        System.out.println(" Failed");
                        return false;
                }
```

```java
//***********************************************************
System.out.print("Populating Area-Messages Table with Initial Messages...");
sql = "INSERT INTO messarea VALUES (1,1,'Main');";

try {
        sqlrows = stmt.executeUpdate(sql);
        System.out.println(" Done");
}
catch (Exception e) {
        System.out.println(" Failed");
        return false;
}

//***********************************************************
System.out.print("Creating Mail Table...");
sql = "CREATE TABLE mail ("
        +"  id int(11) NOT NULL auto_increment,"
        +"  sender varchar(25) default NULL,"
        +"  mdate varchar(50) default NULL,"
        +"  mtime varchar(50) default NULL,"
        +"  message text,"
        +"  PRIMARY KEY  (id)"
        +") TYPE=MyISAM;";

try {
        sqlrows = stmt.executeUpdate(sql);
        System.out.println(" Done");
}
catch (Exception e) {
        System.out.println(" Failed");
        return false;
}


//***********************************************************
System.out.print("Creating User-Mail Table...");
sql = "CREATE TABLE usermail ("
        +"  id int(11) NOT NULL auto_increment,"
        +"  username varchar(25) NOT NULL default '',"
        +"  mailid int(11) NOT NULL default '0',"
        +"  PRIMARY KEY  (id)"
        +") TYPE=MyISAM;";

try {
        sqlrows = stmt.executeUpdate(sql);
        System.out.println(" Done");
}
catch (Exception e) {
        System.out.println(" Failed");
        return false;
}

//***********************************************************
System.out.print("Creating Help Table...");
sql = "CREATE TABLE help ("
        +"  topic varchar(50) NOT NULL default '',"
        +"  description text,"
        +"  PRIMARY KEY  (topic)"
        +") TYPE=MyISAM;";

try {
        sqlrows = stmt.executeUpdate(sql);
        System.out.println(" Done");
}
catch (Exception e) {
        System.out.println(" Failed");
        return false;
}
```

```
                //************************************************************
                System.out.println("Populating Help Table with Default Help Topics");

                sql = "INSERT INTO help VALUES ('.emote','The .emote command lets you show actions and
emotions from a third person point of view.  For instance, typing \\\".emote grins\\\" prints the message
\\\"Jkb grins\\\" and shows happiness.  This command is also shortcut by the \\';\\' character.');";
executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.remote','The .remote command lets you show actions and
emotions from a third person point of view, like the .emote command.  Like the .tell command, the .remote
command sends a private online message to another user which is unseen by other users on the system.
Typing \\\".remote summoner grins\\\" prints the message \\\"Jkb grins\\\" privately to Summoner.  The
.remote command is shortcut by the \\\';;\\\' keyboard shortcut.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.who','The .who command displays the users currently on
the system.  In addition to the users names, the locations of the user, the length of time for the
user\\\'s current session, their level and their description is also displayed.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.tell','The .tell command lets you send online private
messages to a second user on the system which are unseen by other users on the system.  Typing \\\".tell
summoner hello there!\\\" prints the message \"Jkb tells you: hello there!\" privately to Summoner.');";
executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.go','The .go command changes you to a different area, or
\\\"room\\\" on the Talker.  A list of rooms is available by typing .go with no parameters.  If you wanted
to go to a room called \\\"Other\\\", you would type .go other.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.read','The .read command displays the contents of the
message board in the current room.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.write','The .write command prints a new message on the
message board in the current room.  Typing \".write Hello Everyone\" prints an offline public message on
the board that everyone on the system can read at their leisure.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.wipe','The .wipe command takes a parameter instructing
it to delete a number of messages or all of the offline public messages on the board in the current room.
Typing \".wipe 13\" will erase 13 messages from the board, while \".wipe all\" will totally erase the
board.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.topic','The .topic command can be used to set the topic
of conversation in a room.  Suppose the users wanted to discuss Egyptian History.  Someone in the room
could type \".topic Egyptian History\", so when new users enter the room, they will see what the topic of
conversation is and decide whether or not they should stay and participate or leave.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.quit','The .quit command safely exits the user from the
system.  This is important if a user would like their time and settings saved for next time they sign
on.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.shutdown','The .shutdown command safely disconnects all
of the users from the system, closes all the open sockets and database connections and closes the
ServerSocket.  Finally, this terminates the Java Talker process and exits.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.look','The .look command displays information about the
room that you are currently in.  This information includes the room name and description, the number of
messages on the message board, the topic of the room, and the users currently in the room.');";
executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.review','The .review command lets the user read the
conversation buffer in a particular room.  .Review holds the last 20 online public messages, just in case
someone was not paying close enough attention and needs to re-read the conversation just to get back up to
speed.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.cbuffer','The .cbuffer command allows a user to clear
the conversation buffer for the sake of privacy.  Users are able to sign on and perform a .review and read
the last 20 lines of conversation even after the originators sign off, so if you had something private or
secret to say, you can use .cbuffer to ensure your public messages are only for users currently signed
on.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.rmail','The .rmail command lets users read their mail
records, which are for offline private messages.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.smail','The .smail command lets users send offline
private messages to other users.  Typeing \\\".smail summoner hello there\\\" will send Summoner a mail
message saying \\\"hello there\\\".');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.dmail','The .dmail command lets you delete offline
private messages from other users from your mailbox.  Typing a numeric parameter like \\\".dmail 12\\\"
will delete that number of messages from your mailbox, in this case 12.  Typing the word \\\"all\\\" as in
\\\".dmail all\\\" will delete all of your messages.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.description','The .description command lets you specify
a short description of yourself, your views, your mood, or something to spark interest and generate
conversation.  You can put whatever you like as your .description provided it is within the rules of the
system and good taste.');";  executeSQL(sql);
```

```
                sql = "INSERT INTO help VALUES ('.set','The .set command lets you specify different
properties about yourself like gender, description, password, age, email, homepage, photo and profile.
These are used to customize your UserConnection object to yourself.  Gender tells everyone which sex you
are, Description is a short description that tells a little about yourself, Password is used when you log
onto the Java Talker system, Age is the number of years experience living you may have, Email is your
email address, Homepage is where one can see your website.  Photo is where one can see your photograph.
Profile is a field where you can print your life story if you like.  The .set command is passed two
parameters, the property you are setting and the value.  For instance, if I wanted to set my age to 26, I
would type \\\".set age 26\\\"');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.idle','The .idle command displays the amount of time all
of the users on the system have been idle.  This gives a user an idea of whether or not a person in a
conversation who has not responded in a while is still connected to the system.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.examine','The .examine command displays information
about different users on the system.  Some information displayed are Name, Age, Description, Photo, Email,
Homepage, and connection specific information like time online and idle time and the user\\\'s current
room.  If a user types \\\".examine\\\" with no parameters, the users own information will be displayed.
To display information about another user, you would type \\\".examine summoner\\\" to view Summoner\\\'s
information.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.promote','The .promote command increases a user\\\'s
level, which enables them to use higher level commands.  You would use the .promote command by typing
\\\".promote summoner\\\".  You can\\\'t promote people to levels higher than your own.');";
executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.demote','The .demote command decreases a user\\\'s
level, which prevents them from using higher level commands.  You would use the .deomote command by typing
\\\".demote summoner\\\".  You can\\\'t demote people who start at levels higher than your own.');";
executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.afk','The .afk command stands for \\\"Away From
Keyboard\\\".  This command lets other users in the room and on the system know that you are away from
your keyboard and not paying attention at the moment.  When online private messages arrive at an AFK user,
the sender gets a message informing them of your AFK status.  Additionally, you can set up a message
informing everyone of where you are going or when you will be back.  For instance to tell people you are
getting up for a snack, you could type \\\".afk getting snacks\\\" and the message \\\"Summoner is AFK,
getting snacks\\\" will be printed for everyone interested.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.remove','The .remove command allows higher level users
to manually disconnect users from the system.  This command is used by typing \\\".remove summoner\\\"
where Summoner is an example user to be removed.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.revtells','The .revtells command allows a user to review
their personal \\\"Tell Buffer\\\" of the last 20 online private messages.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.ctells','The .ctells command allows a user to clear
their personal \\\"Tell Buffer\\\" of the last 20 online private messages.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.delete','The .delete command is used by higher level
users to delete users from the User database.  The .delete command will not work on users of higher level
than the user executing the .delete command.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.nuke','The .nuke command is used by higher level users
to delete users from the User database.  The .nuke command will not work on users of higher level than the
user executing the .nuke command.');";  executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.help','This is the help command.  Type \\\".help
topic\\\" where topic is one of the listed topics to get more information about that command.');";
executeSQL(sql);
                sql = "INSERT INTO help VALUES ('.version','The .version command displays to a user the
current version, updated date and updater\\\'s name.  It also provides a little bit of information about
the Java Talker.');";  executeSQL(sql);
                return true;
        }

        /** executeSQL executes the SQL statement passed as a parameter.
                @param sql String value of the SQL statement to be executed.
                @return boolean indicating whether or not the operation was successful. */
        public boolean executeSQL (String sql) {
                try {   stmt.executeUpdate(sql);
                        System.out.println(" Done");
                        return true;  }
                catch (Exception e) {
                        System.out.println(" Failed");
                        e.printStackTrace();
                        return false;
                }
        }
}
```

Appendix E:

Talker Startup Script Source Code

This Appendix contains the source code for the Talker Startup Script.

```
#!/usr/bin/perl

#####################################################################################
# Java Talker Startup Script
# CS690 Project
# Jeffrey K. Brown
#
# This Java Talker Startup Script is for Unix and Linux systems. It uses two
# different utilities, the AT scheduler and the Perl Interpreter to pull the system
# time and then use it to schedule the Java Talker to start.  It is necessary to
# start Java servers on Linux in this manner because Java does not allow processes to
# fork, or spawn new processes.  If you start the Java Talker server and then attempt
# to disconnect from the server, you will either disconnect the Java Talker Server,
# or your server connection will not terminate.  We get around this by scheduling the
# server process to start $delay minutes in the future with the AT scheduler.  The AT
# scheduler creates a new shell and launches the Java Talker Server in it.  Shutting
# down the Talker Server is done by the Talker command .shutdown.  If necessary, the
# Java Talker Server can also be forcibly killed by the Unix command kill.
#
#####################################################################################
# These variables can be customized to your Java Talker installation and server.
#
$delay = 1;                             # Number of Minutes until JT Starts
$jtpath = "/home/jkb/jolt/";            # Path to Java Talker directory
$jtfile = "Chat";                       # Name of Java Talker Server class
$javapath= "/usr/lib/java/bin/java";    # Path to Java Interpreter
$atpath = "/usr/bin/at";                # Path to AT scheduler
$logfile = "jolt.log";                  # Path to Java Talker Logfile
#
#####################################################################################
# Java Talker Startup Script Methodology:
#
# To start the Java Talker, we need to make a call looking like this:
# java Talker
#
# Because AT emails all the output of the call to you upon process termination,
# I have all of the output going into a log file:
# java Talker > JavaTalker.log
#
# The AT schedule, however, will only take one parameter for the file to be executed.
# Additionally, what if the user is calling the script from a different directory
# than the Java Talker directory.  For this reason, we generate a script file to
# connect to the Java Talker directory, execute the Talker Server process and then
# capture the output to a log file.
#
# Using the $delay variable, we add the number of minutes to the system time and
# schedule the new script to run at that time.  Finally, we delete the script.
#####################################################################################

# Assign a name to a temporary file based on a random number.  We also put the
# temporary file in the Java Talker Path.

$random = rand 100000000;
$tempfile = "$jtpath$random";

# Next, we open the temporary file and then add commands to change to the Java Talker
# Path and to run the Java Talker Server, capturing all output to the logfile.  Last,
# we close the file.

open (FILE, ">$tempfile") or die "Unable to Create and Open Temp File: $!\n";
print FILE "\ncd $jtpath";
print FILE "\n$javapath $jtfile > $jtpath$logfile";
print FILE "\n";
close (FILE);

# Next, we make the new script executable, so AT will be able to run the commands
# stored inside and start the Java Talker.

chmod 0755, "$jtpath/$random";
```

```perl
# At this stage, we get the system time and separate the individual attributes
# using Perl's localtime function.

($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);

# Here, we add the number of seconds to wait ($delay) to the system minutes and
# check the new time, making changes if needed.  These checks work best if $delay
# is 1 minute.  If you wanted to increase the delay, you might want to verify that
# these checks still provide acceptable performance.

$min += $delay;
if ($min<10) {
        $min = "0$min";
}

if ($min==60) {
        $min="00";
        $hour=$hour+1;
        if ($hour==24) {
                $hour="00";
        }
}

# Now we generate the system call to the AT Scheduler.  $atpath contains the path
# to the AT scheduler executable.  The "-f" tells AT to execute the file whose name
# is the next token.  The $tempfile is the name of our temporary file.  $hour$min is
# the new time we generated at which time the Java Talker will be started.  Finally,
# we run the command.

@startup = ("$atpath", "-f", "$tempfile", "$hour$min");
system(@startup)==0 or die "Error Running At! $!";

# Last, we delete the temporary file.

unlink $tempfile;

# The End.  The Java Talker executable should be scheduled and will start in 1 minute

#####################################################################################
```

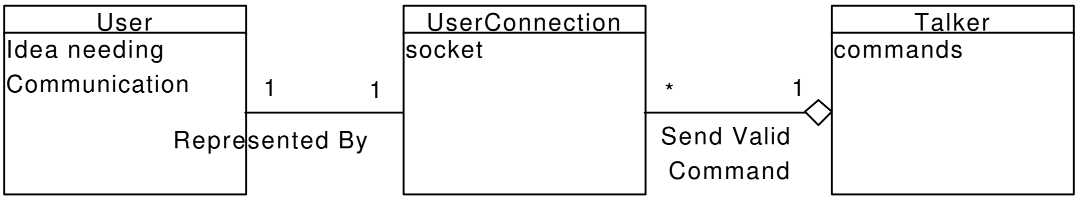
Appendix F:

UML Diagrams

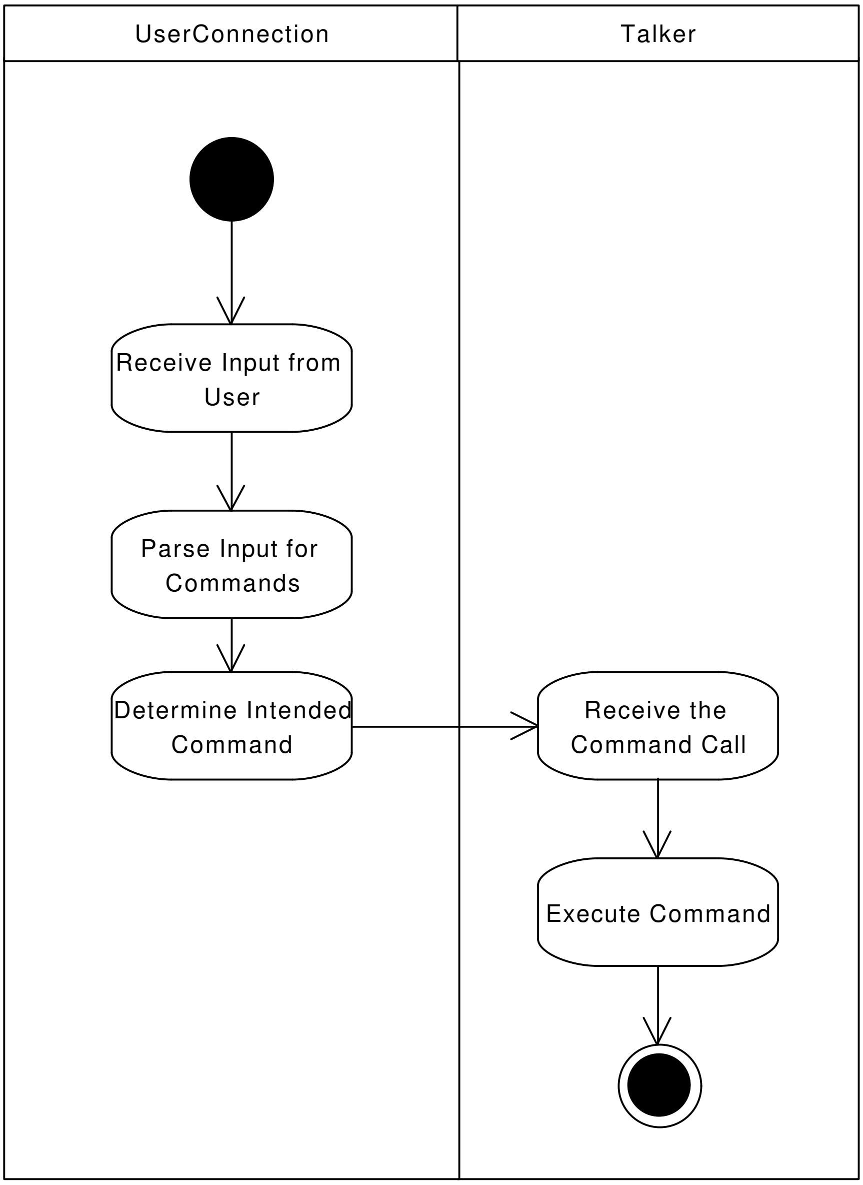
This Appendix contains UML Diagrams of various aspects of the Java Talker Project.

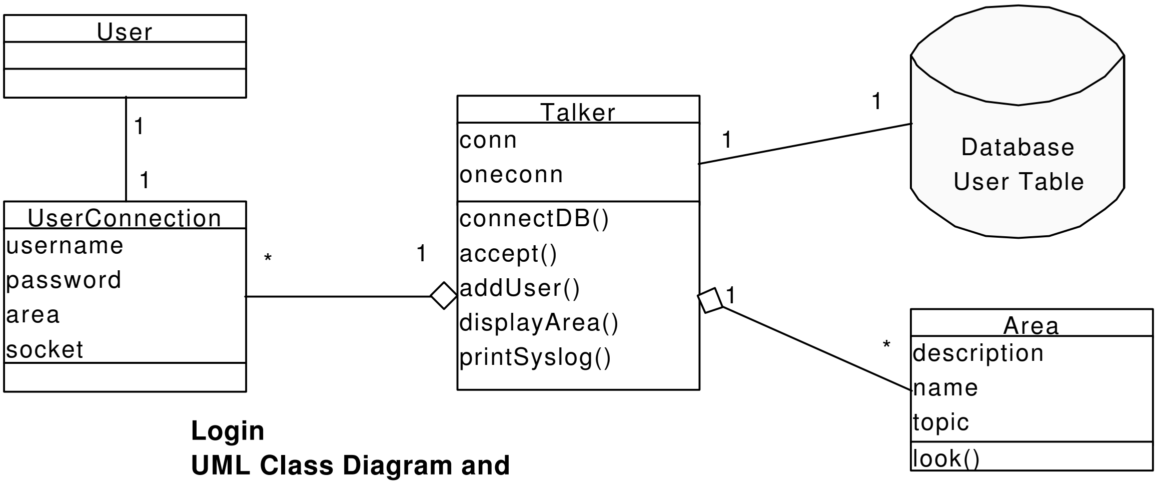
Appendix F UML Diagrams

Table of Contents

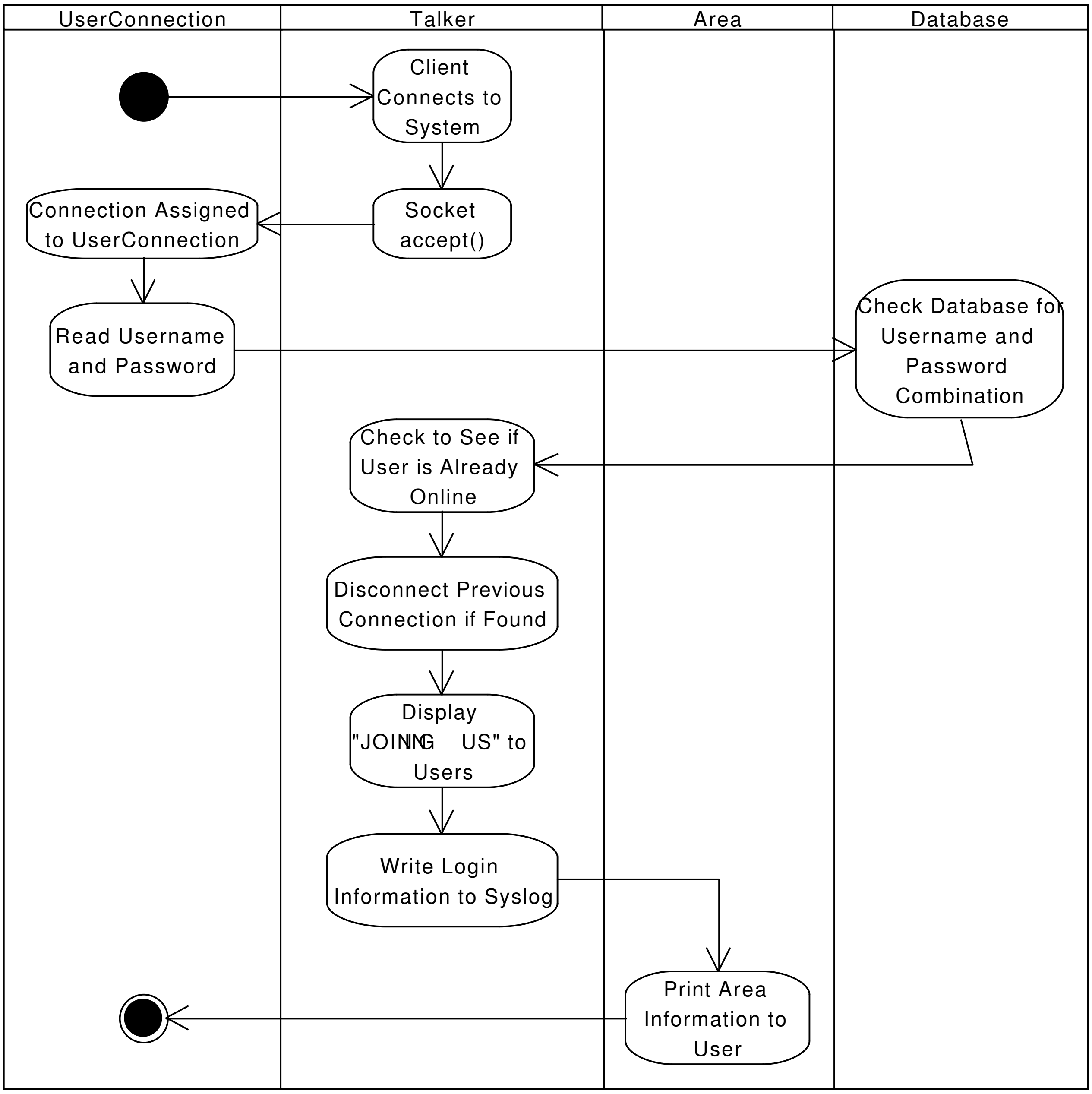


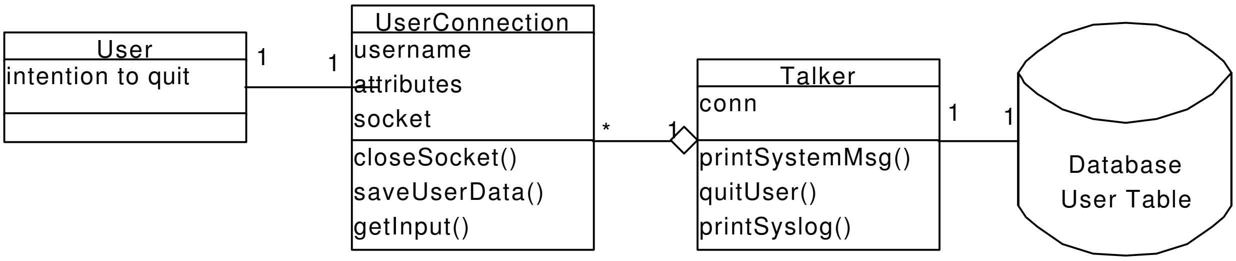
Command Parsing
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown



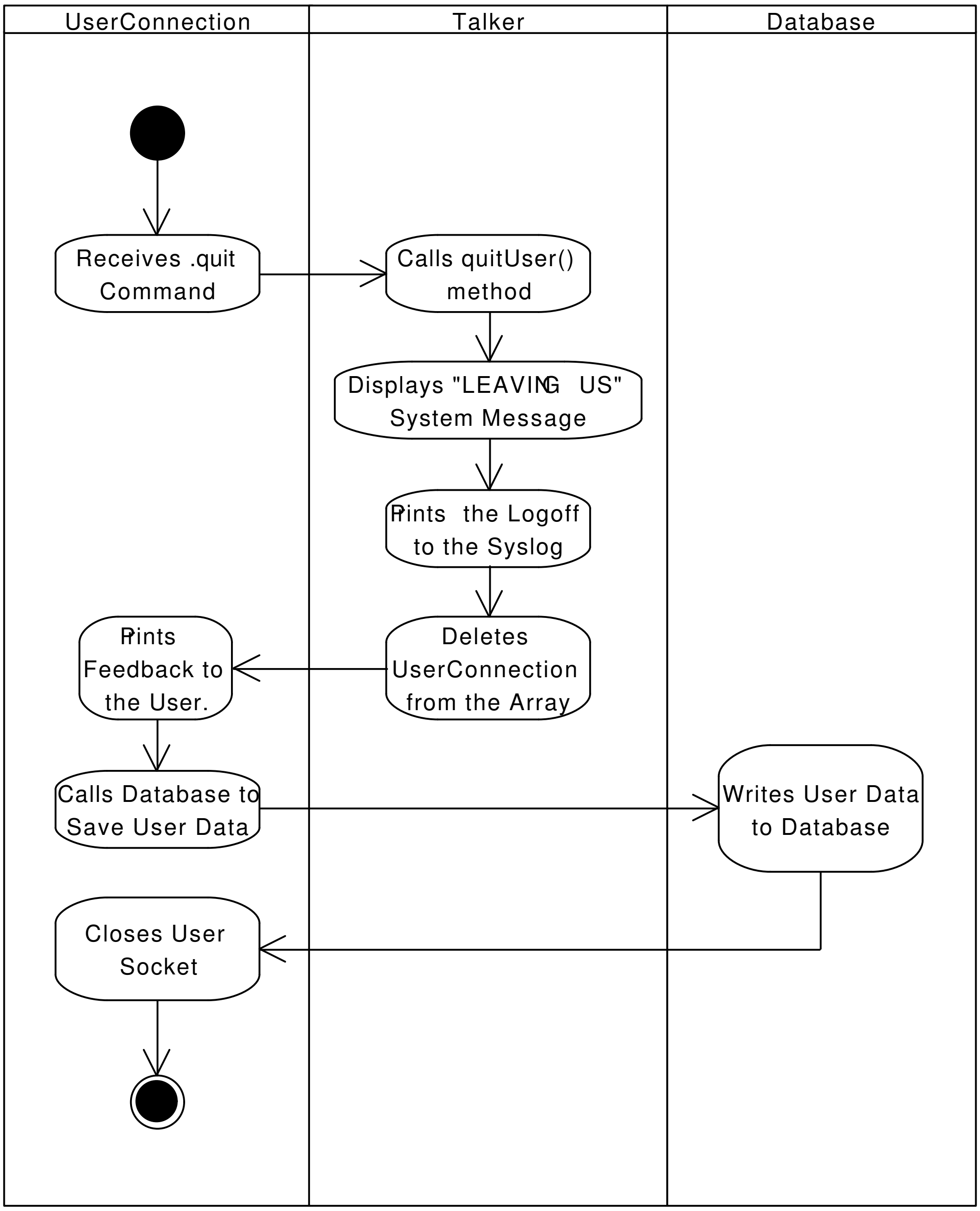


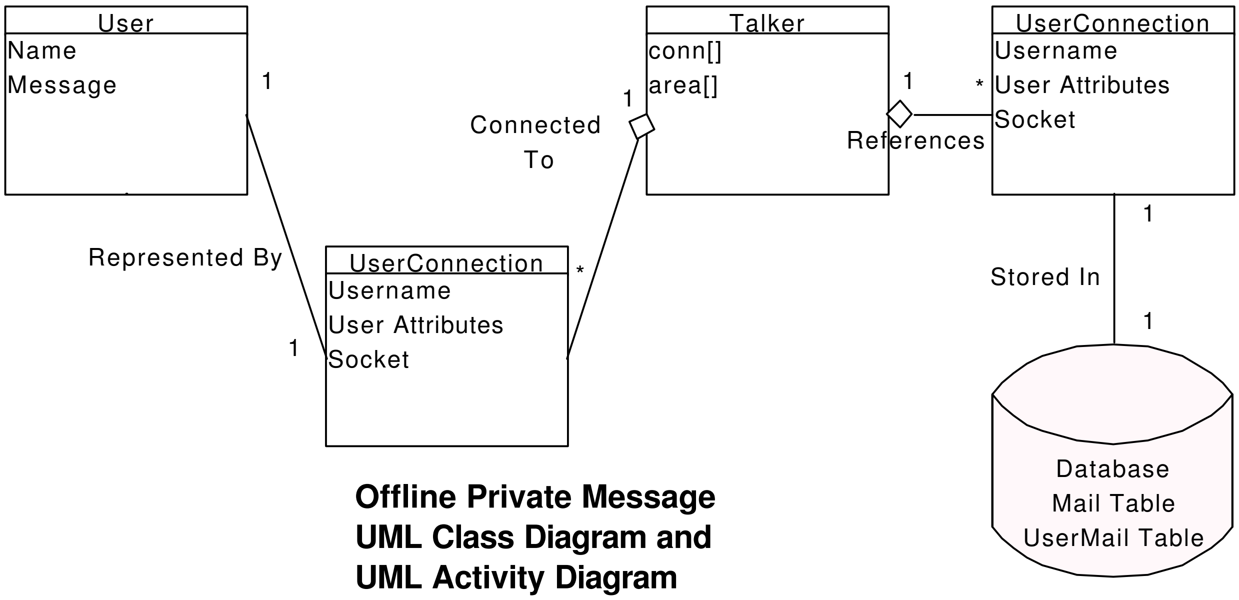
Login
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown


UserConnection
Talker
Area
Database
Client Connects to System
Socket accept()
Connection Assigned to UserConnection
Read Username and Password
Check Database for Username and Password Combination
Check to See if User is Already Online
Disconnect Previous Connection if Found
Display "JOINING US" to Users
Write Login Information to Syslog
Print Area Information to User

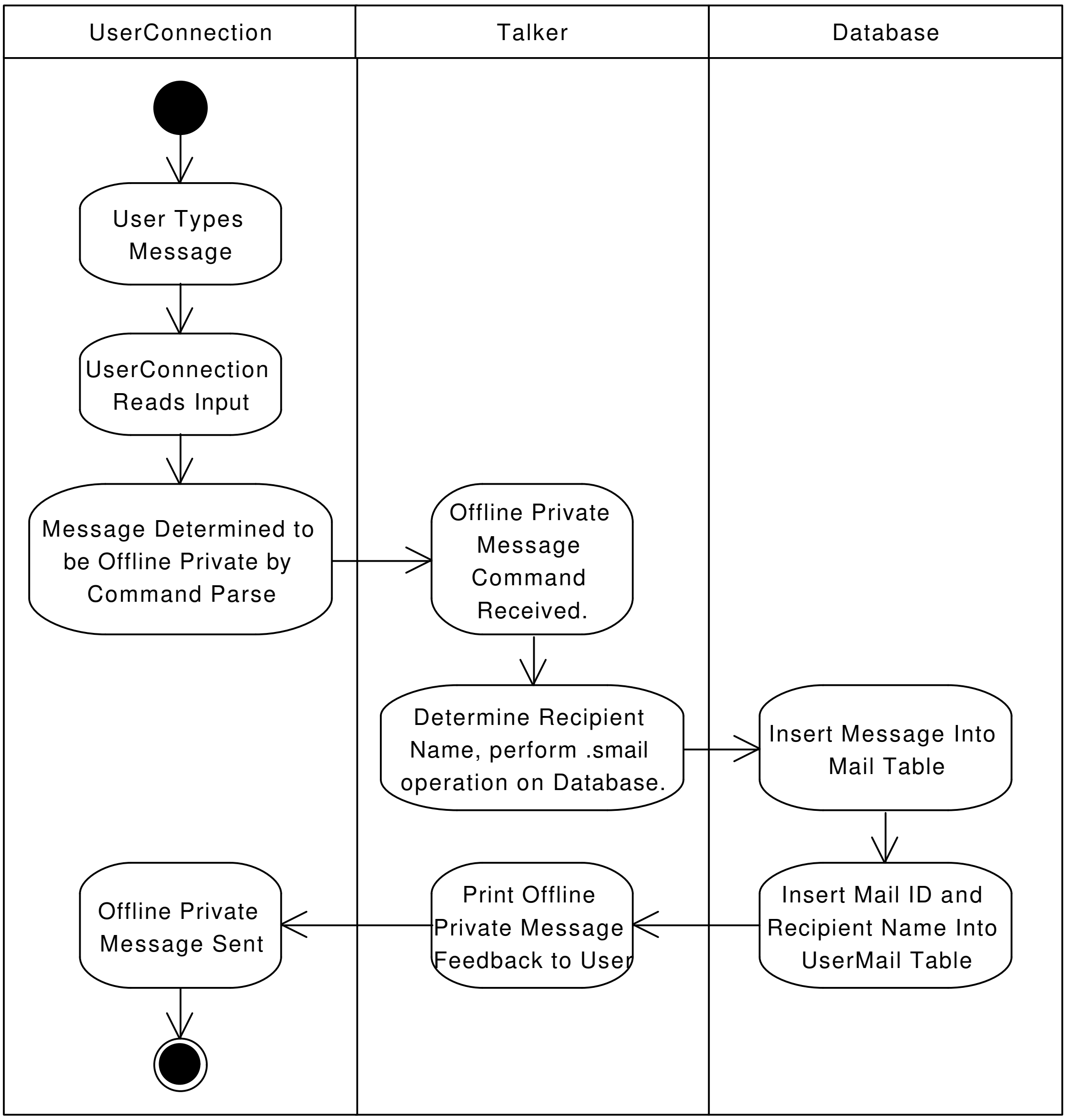


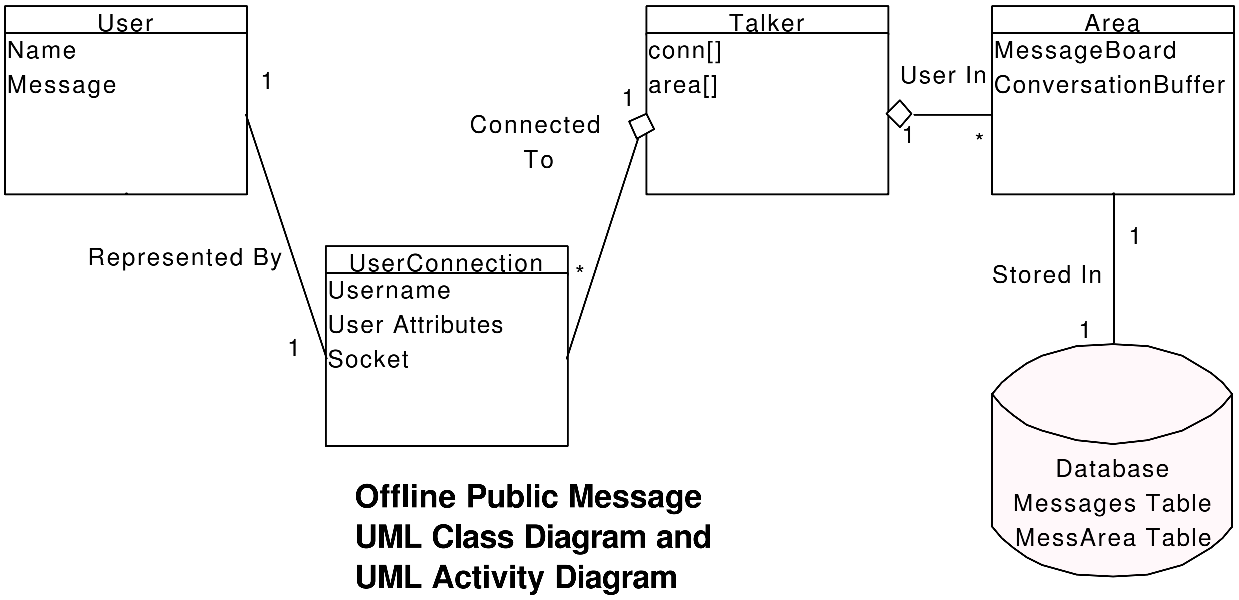
Logoff
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown


UserConnection
Talker
Database
Receives .quit Command
Calls quitUser() method
Displays "LEAVING US" System Message
Prints the Logoff to the Syslog
Deletes UserConnection from the Array
Prints Feedback to the User.
Calls Database to Save User Data
Writes User Data to Database
Closes User Socket

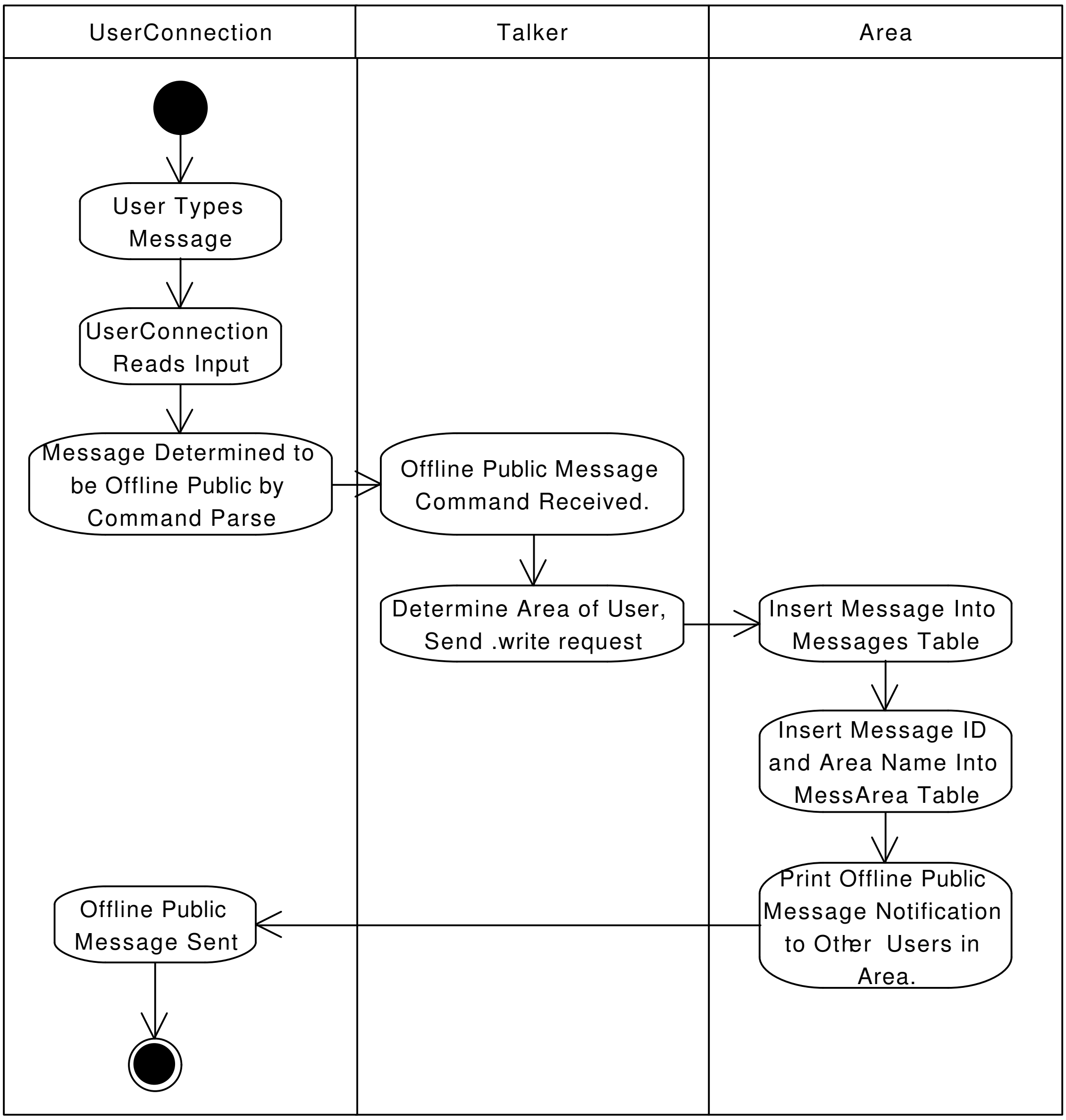


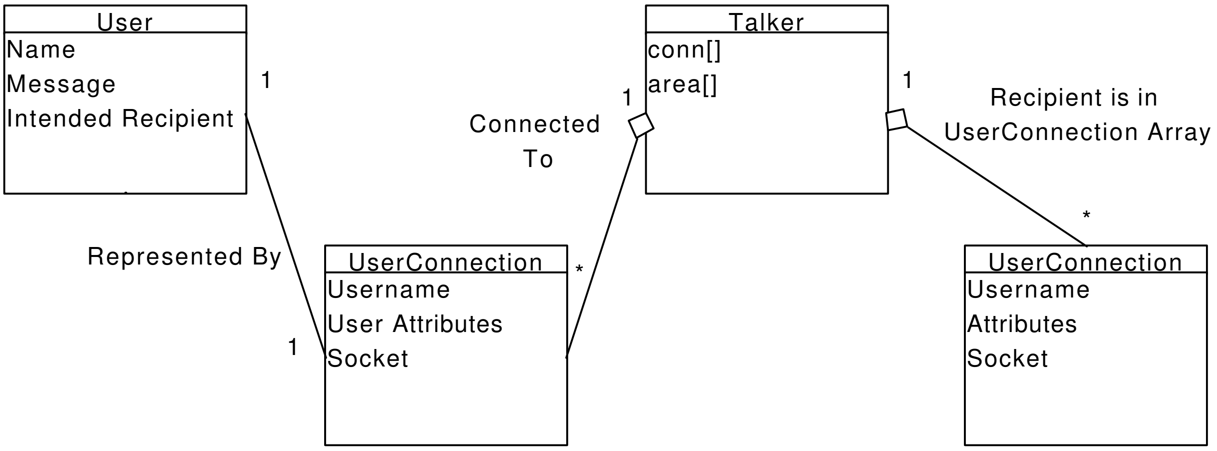
Offline Private Message
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown


UserConnection
Talker
Database
User Types Message
UserConnection Reads Input
Message Determined to be Offline Private by Command Parse
Offline Private Message Command Received.
Determine Recipient Name, perform .smail operation on Database.
Insert Message Into Mail Table
Insert Mail ID and Recipient Name Into UserMail Table
Print Offline Private Message Feedback to User
Offline Private Message Sent

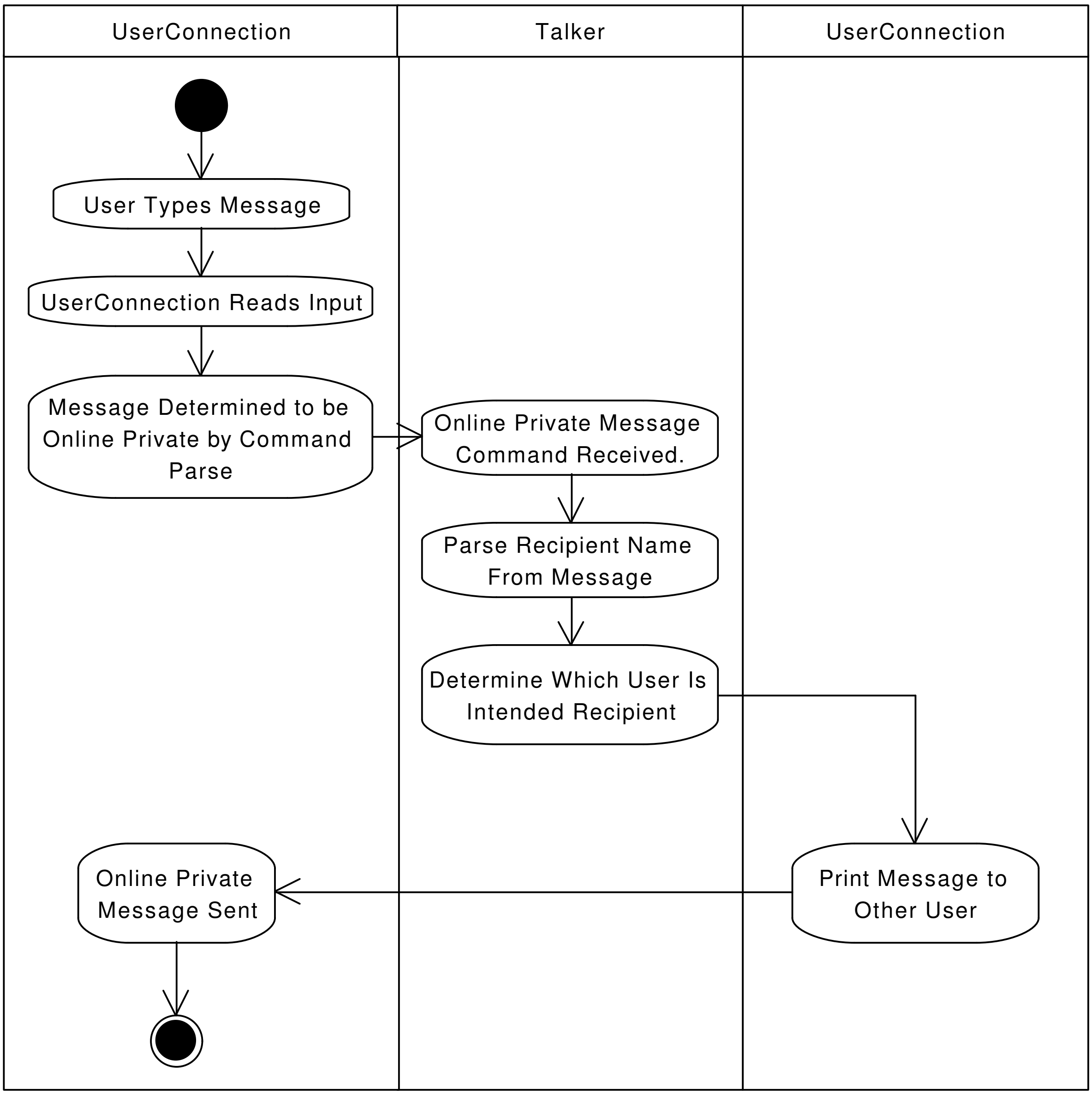


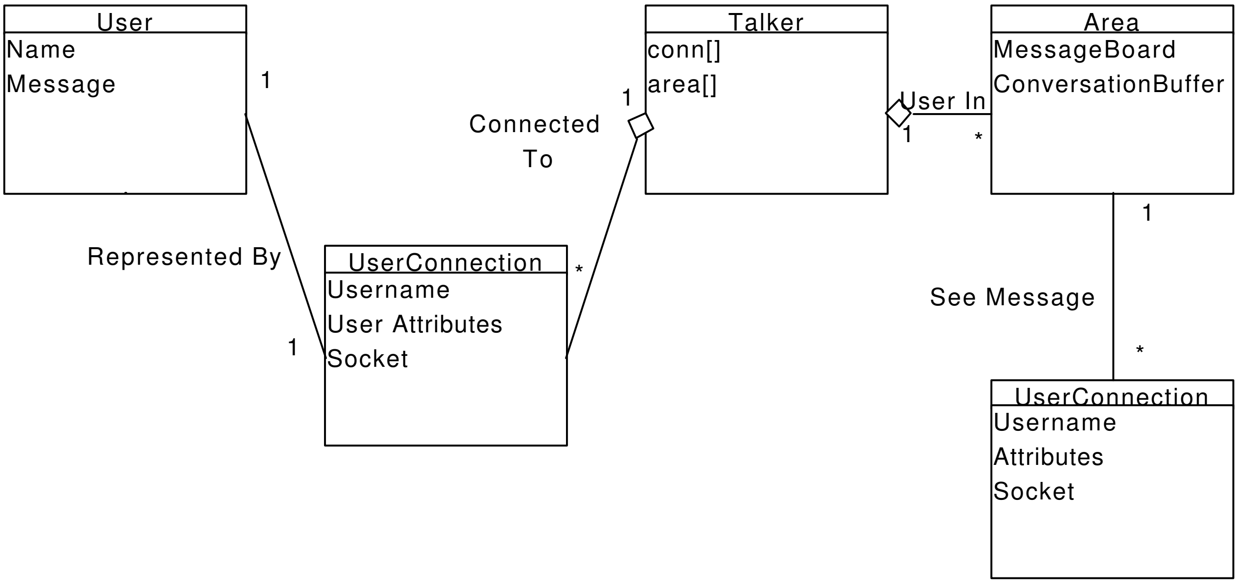
Offline Public Message
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown

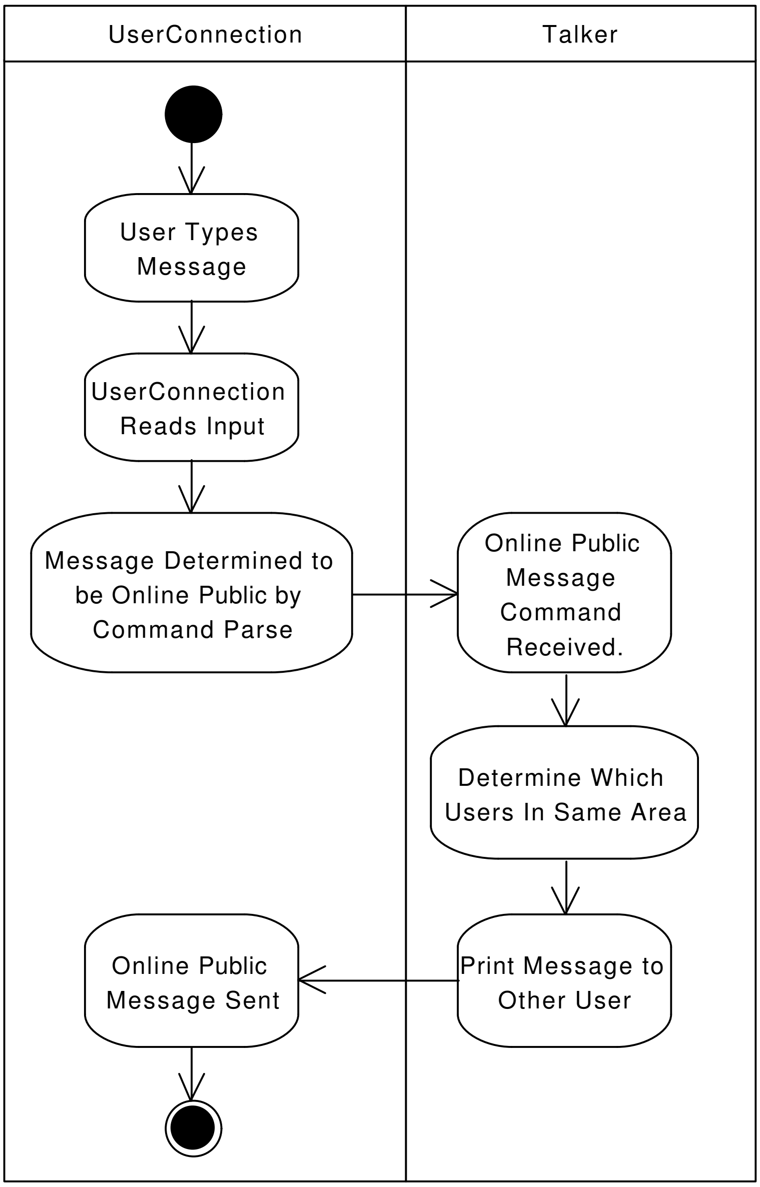

UserConnection
Talker
Area
User Types Message
UserConnection Reads Input
Message Determined to be Offline Public by Command Parse
Offline Public Message Command Received.
Determine Area of User, Send .write request
Insert Message Into Messages Table
Insert Message ID and Area Name Into MessArea Table
Print Offline Public Message Notification to Other Users in Area.
Offline Public Message Sent



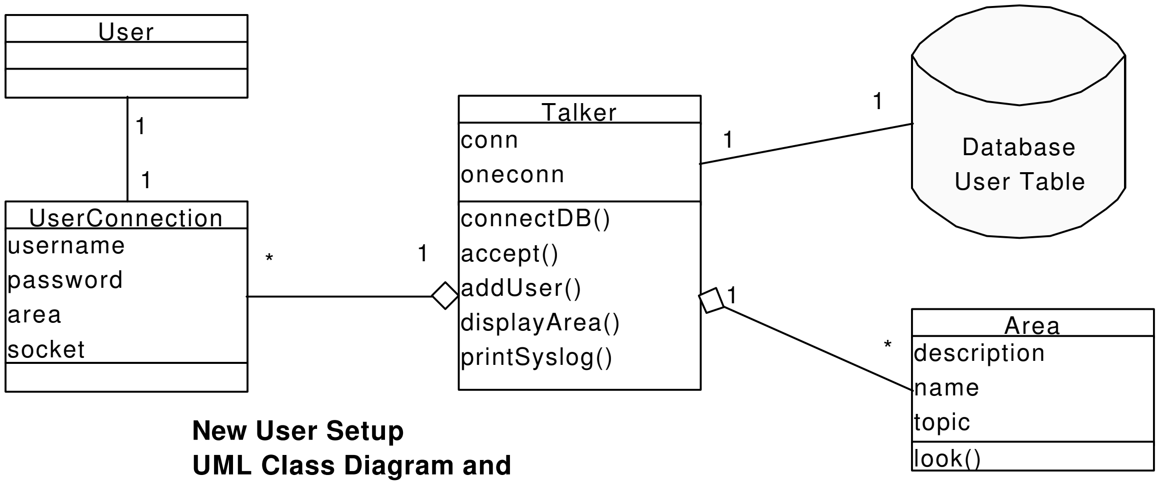
Online Private Message
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown


UserConnection
Talker
UserConnection
User Types Message
UserConnection Reads Input
Message Determined to be Online Private by Command Parse
Online Private Message Command Received.
Parse Recipient Name From Message
Determine Which User Is Intended Recipient
Print Message to Other User
Online Private Message Sent

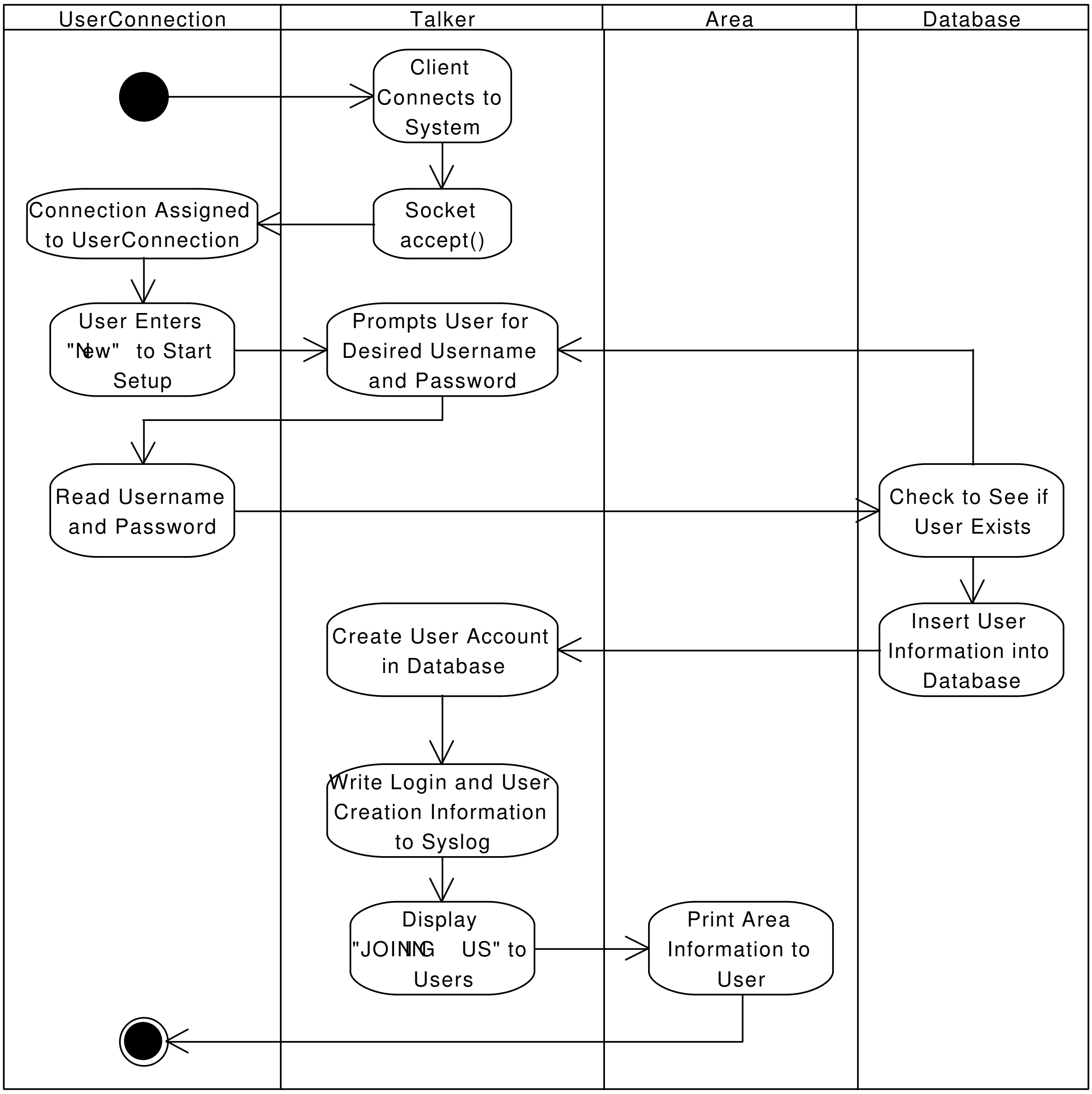


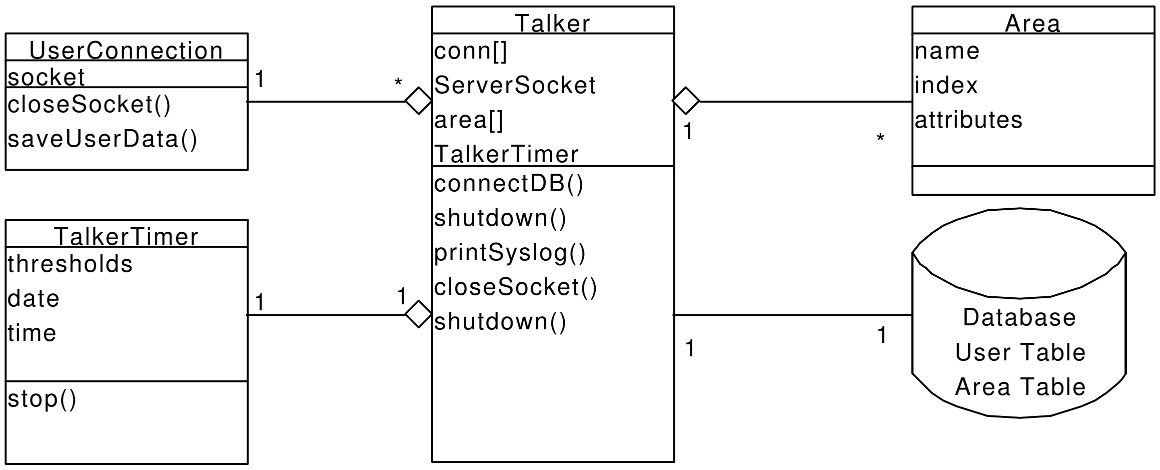


Online Public Message
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown

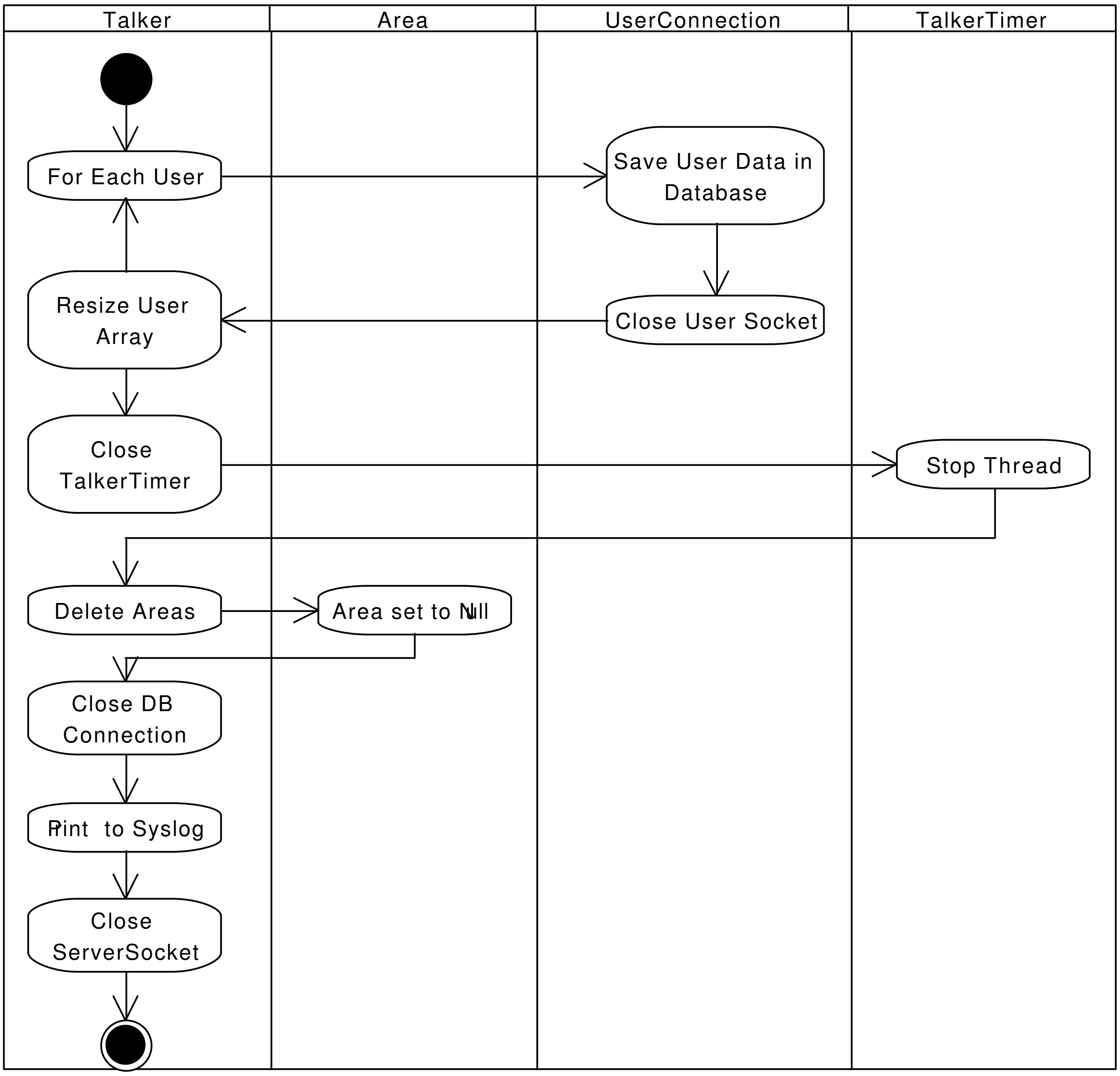


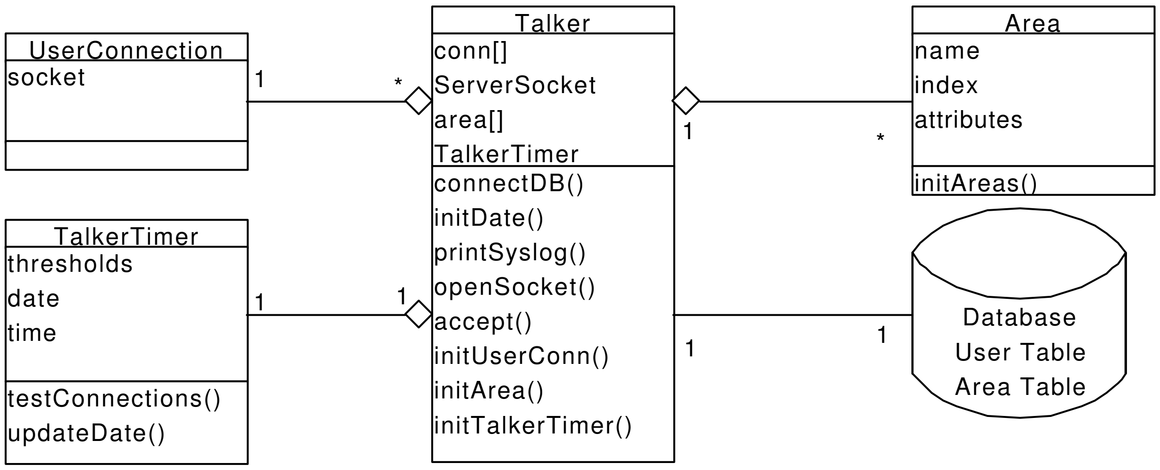
New User Setup
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown


UserConnection
Talker
Area
Database
Client Connects to System
Socket accept()
Connection Assigned to UserConnection
User Enters "New" to Start Setup
Prompts User for Desired Username and Password
Read Username and Password
Check to See if User Exists
Insert User Information into Database
Create User Account in Database
Write Login and User Creation Information to Syslog
Display "JOINING US" to Users
Print Area Information to User

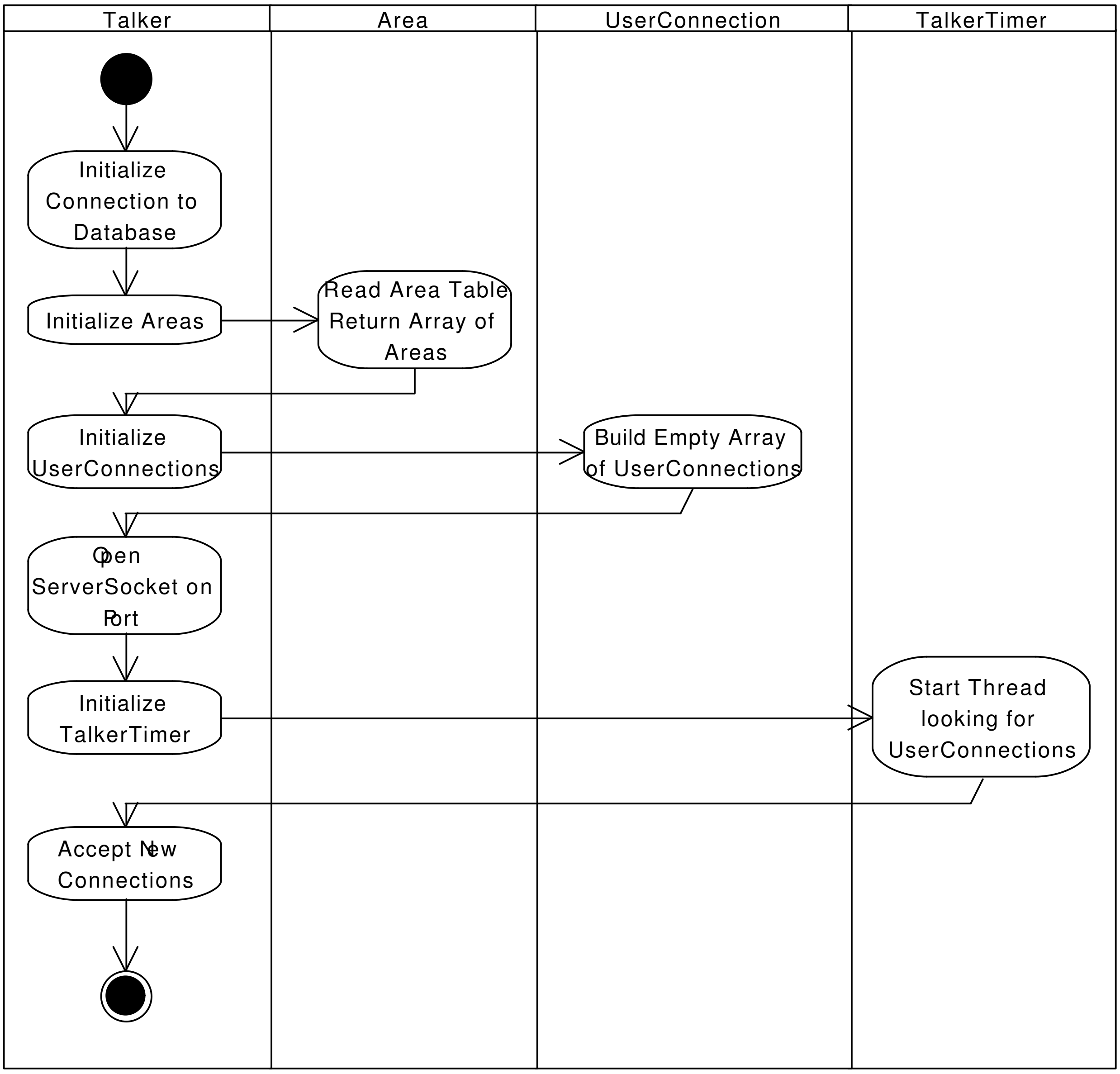


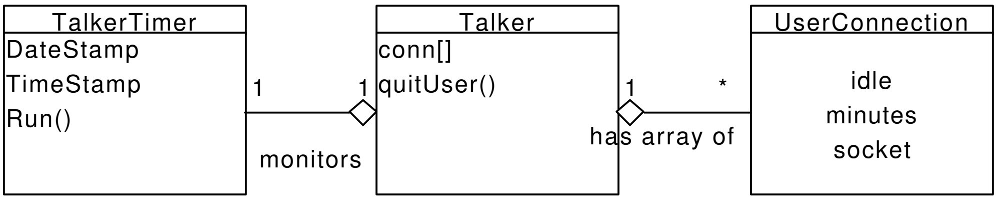
Shutdown
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown



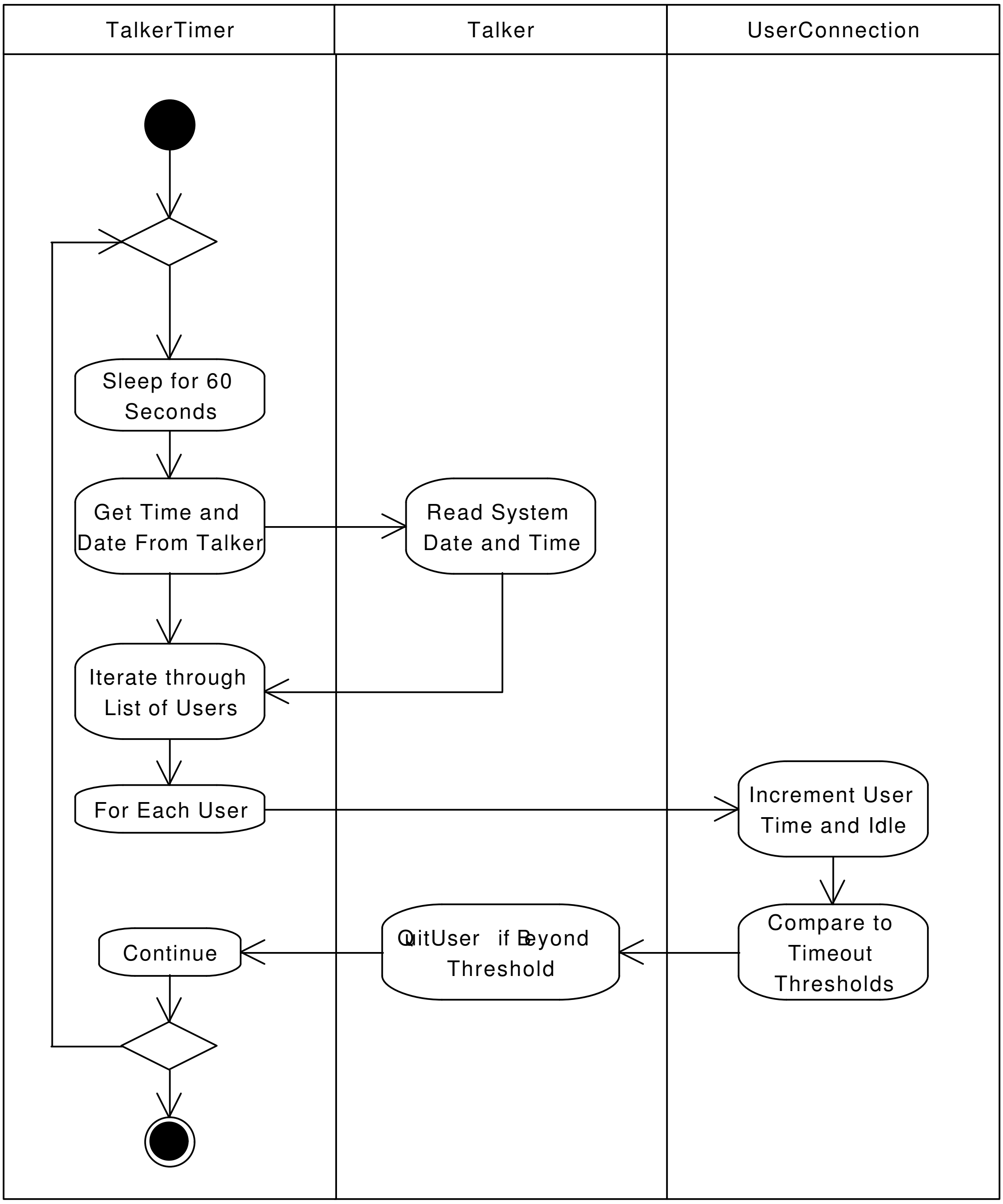


Startup
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown


Talker
Area
UserConnection
TalkerTimer
Initialize Connection to Database
Initialize Areas
Read Area Table Return Array of Areas
Initialize UserConnections
Build Empty Array of UserConnections
Open ServerSocket on Port
Initialize TalkerTimer
Start Thread looking for UserConnections
Accept New Connections



TalkerTimer
UML Class Diagram and
UML Activity Diagram
Jeffrey K. Brown


TalkerTimer
Talker
UserConnection
Sleep for 60 Seconds
Get Time and Date From Talker
Read System Date and Time
Iterate through List of Users
For Each User
Increment User Time and Idle
Compare to Timeout Thresholds
QuitUser if Beyond Threshold
Continue

Appendix G:

Command Analysis

This Appendix contains an Analysis of the Commands to be Implemented into the Java Talker Project based on User Input and the Command Report

Command Analysis
CS690 Project: Java Talker Project
Jeffrey K. Brown
6/22/03

This report runs down the most important commands on the Dark Ages Talker that will be implemented. Additionally, the Command Report will be taken into account, as I will implement some of the most popular commands, too.
For the examples of the command's output, I will use two example names, Summoner and Penguin.

Public Communication Commands:

.say

The .say command is the most often used command. On the command report, it is not listed, because it is the default command. It is used to communicate with all users in the same room as the user with something to say.

```
      Called using:  .say Hello Everyone
      Summoner sees: You say: Hello Everyone
      Everyone sees: Summoner says: Hello Everyone
```

.emote

The .emote command is used to show emotion that humans in the real world would ordinarily communicate nonverbally with expressions. Evolution of the talker has made it used to show action as well.

```
      Called Using: .emote smiles at everyone
      Summoner sees: Summoner smiles at everyone
      Everyone sees: Summoner smiles at everyone
```

.afk

The .afk command is short for "Away from Keyboard". When this command is called, a message is displayed to the users in the room informing them of the user's intent to go away from the keyboard. Additionally, all private communication to the user is returned with a message to the sender informing them that the user is away from the keyboard. The .afk command takes a parameter that informs the rest of the room of any additional information about where the user may be going.

```
      Called using:  .afk feeding the dog
      Summoner sees:  You go .afk, feeding the dog
      Everyone sees:  Summoner goes .afk, feeding the dog
```

.review

.review is used to review the last 20 lines of public conversation that took place in a room. If a conversation is going on and someone missed an important point, they may call this .review command and get that point again.

```
      Called using:  .rev
      Summoner sees:  Penguin says: Java is fun
                      Summoner says: Yes, I agree
                  Penguin nods.
```

.cbuff

Suppose someone was discussing a controversial topic and do not want anyone coming into the room later and doing a .review command and reviewing the conversation. The command, .cbuff will allow a paranoid user to clear the public conversation buffer, and protect the messages from those who have not already seen it.

```
      Called using:  .cbuff
      Summoner sees:  Conversation buffer cleared.
```

Offline Public Communication Commands:

.write

Each room has a message board in it. If a user wanted to post a message that all users who enter the room may read, the user would use the .write command. The .write command takes the message to be displayed as a parameter.

```
      Called using:  .write Today is Penguin's birthday
      Summoner sees:  You write the message on the board.
      Everyone sees:  Summoner writes a message on the board.
```

.read

To read messages on the board, one would use the .read command.

```
      Called using:  .read
      Summoner sees:  From Summoner (Jun 22, 2003: 12:45):  Today is
Penguin's birthday
      Everyone sees:  Summoner reads the message board.
```

.wipe

If a message board in a room gets cluttered with too many messages, an enabled user may wipe the message board using the .wipe command. The .wipe command takes one parameter, either all or a number. If the parameter is all, all of the messages are deleted. If the parameter is a number, the line number of the message is deleted.

```
      Called using:  .wipe all
      Summoner sees:  You wipe all messages from the board.
      Everyone sees:  Summoner wipes all messages from the board.
```

Private Communication Commands:

.tell

The .tell command is the private equivalent of .say. Private commands require an additional parameter saying to whom the message is sent. These private messages are invisible to other users on the talker.

```
Called using:  .tell penguin Hi Penguin
Summoner sees:  You tell Penguin: Hi Penguin
Penguin sees: -> Summoner tells you: Hi Penguin
Everyone sees:
```

.remote

Like the .tell command, .remote is the private equivalent of .emote. This command sends an .emote privately.

```
Called using:  .remote penguin gives a high five
Summoner sees:  -> You emote to Penguin: Summoner gives a high five
Penguin sees:  -> Summoner gives a high five
Everyone sees:
```

Offline Private Communication Commands:

.smail

The .smail command is short for "send mail". This command stores a message in the mailbox of individual users, allowing for communication or sending of a message to a user whether they are logged on or not. Because it is private, users in the room do not see anything either.

```
Called by:  .smail penguin Hello Penguin Friend
Summoner sees:  Your mail is delivered to Penguin
Penguin sees by .rmail:  From Summoner (date and time):  Hello Penguin Friend
```

.rmail

The .rmail command is short for "read mail". This command lists all the messages stored in a user's mailbox.

```
Called by:  .rmail
Summoner sees:  From Penguin (Jun 06 2002; 12:39): What's new buddy?
                From Penguin (Jun 14 2002; 15:02): Nice Joke...
```

.dmail

Mail is deleted using the .dmail command. Dmail takes one argument. Depending on whether the argument is all or a number, the .dmail command deletes either all of the mail, or the specific line number of the mail listing.

Called by: .dmail all
Summoner sees: All mail deleted.

World Commands

.help

The command, .help, is used to display the online help documentation.

Called by: .help
Summoner sees the help file

.quit

The .quit command is used to disconnect the user from the system safely, saving settings and displaying a message to inform the other users that the quitting user has left.

Called by: .quit
Summoner sees: You leave the realm of Dark Ages, Goodbye!
Everyone sees: Leaving Us: Summoner

.examine

.Examine is used by a user to check another user's or one's own settings, profile or user information.

Called by: .examine pengin
Summoner sees Penguin's user information and profile.

.who

The .who command displays a list of users currently signed on the system. This list includes user's name, short description, their current room, their level, and the elapsed time logged on.

Called by: .who
Summoner sees the list of users logged onto the system.

.look

The .look command displays information about the room, including the topic, the number of messages on the message board, the users in the current room, and the description of the room.

Called by: .look
Summoner sees the room description, users in the room, number of messages on the board and the users in the room.

.idle

The .idle command displays the idle time of users on the system. This gives the user an indication of whether or not a user is paying attention or not.

```
Called by:  .idle
Summoner sees the list of users on the system along with their idle time.
```

.topic

The .topic command allows a user to specify or change the topic of conversation in a room.

```
Called by:  .topic Socket Programming
Summoner sees:  You have changed the topic to "Socket Programming"
Everyone sees:  Summoner changed the topic to "Socket Programming"
```

.go

The .go command is used to change from one room of the talker to another. When a user changes rooms, users in the starting room see a message notifying them that the user has left, and the users in the fininshing room see a message notifying them that the user has arrived.

```
Called by:  .go Penguin
Summoner sees the description of the Penguin room
Penguin sees: Summoner enters the room
Everyone sees:  Summoner leaves the room.
```

Appendix H:

Command Report

This Appendix contains a count of the number of times certain commands were executed in a given uptime period. The data was used to decide which commands needed to be implemented into the Java Talker Project.

Command Report
CS690 Project
Jeffrey K. Brown
5/1/03

This report shows the number of times each command has been executed since the talker's booting on March 31, 2003.

```
.numcom 7
0. .quit = 370                             48. .remove = 1
1. .who = 753                              49. .semote = 0
2. .shout = 14                             50. .bansite = 0
3. .tell = 3060                            51. .unbansite = 0
4. .ack = 1                                52. .listbans = 0
5. .ignore = 1                             53. .addackdoor = 1
6. .look = 63                              54. .cls = 25
7. .go = 49                                55. .time = 5
8. .lock = 0                               56. .dsay = 0
9. .unlock = 0                             57. .think = 43
10. .invite = 0                            58. .afk = 113
11. .emote = 2960                          59. .idle = 328
12. .areas = 1                             60. .bbcast = 0
13. .knock = 2                             61. .colors = 2
14. .write = 176                           62. .xwipe = 9
15. .read = 729                            63. .cbuff = 47
16. .wipe = 18                             64. .set = 51
17. .site = 2                              65. .frog = 5
18. .to = 0                                66. .duty = 0
19. .badname = 0                           67. .pemote = 0
20. .remote = 52                           68. .cloak = 0
21. .kill = 0                              69. .decloak = 0
22. .shutdown = 0                          70. .greet = 57
23. .search = 0                            71. .arrest = 0
24. .review = 161                          72. .macros = 1
25. .help = 19                             73. .send = 2
26. .bcast = 3                             74. .bring = 0
27. .news = 2                              75. .punt = 0
28. .system = 2                            76. .ranks = 0
29. .move = 1                              77. .follow = 0
30. .access = 6                            78. .home = 3
31. .log = 0                               79. .wtell = 4
32. .atmos = 0                             80. .show = 7
33. .echo = 9                              81. .lastsys = 76
34. .desc = 105                            82. .revtell = 37
35. .newusers = 0                          83. .ctellbuf = 1
36. .version = 0                           84. .barge = 0
37. .entpro = 2                            85. .syslog = 0
38. .examine = 114                         86. .sing = 21
39. .people = 29                           87. .motd = 0
40. .dmail = 14                            88. .wmotd = 0
41. .rmail = 170                           89. .warn = 0
42. .smail = 141                           90. .nuke = 23
43. .wake = 22                             91. .hide = 3
44. .promote = 0                           92. .wemote = 0
45. .demote = 0                            93. .wnote = 3
46. .map = 16                              94. .wwipe = 0
47. .passwd = 2                            95. .restrict = 0
```

```
96. .unrestrict = 0                        131. .wrev = 0
97. .lrestr = 0                            132. .cwizbuf = 1
98. .wizards = 1                           133. .disable = 2
99. .ctopics = 0                           134. .allowcom = 1
100. .commands = 39                        135. .rescom = 0
101. .reinit = 7                           136. .snews = 0
102. .rename = 0                           137. .fmail = 2
103. .whisper = 0                          138. .social = 6
104. .xcomm = 0                            139. .refill = 1
105. .assist = 0                           140. .shield = 3
106. .mroom = 0                            141. .soak = 1
107. .decorate = 0                         142. .target = 1
108. .edit = 0                             143. .rules = 33
109. .mv = 0                               144. .mnote = 9
110. .tpromote = 0                         145. .mwipe = 0
111. .entermsg = 0                         146. .gripe = 14
112. .addtalker = 0                        147. .gwipe = 0
113. .talkers = 1                          148. .suggest = 0
114. .addfriend = 1                        149. .swipe = 0
115. .delfriend = 0                        150. .agripe = 0
116. .friends = 1                          151. .rdesc = 0
117. .lfriends = 1                         152. .xmail = 1
118. .ftell = 0                            153. .syssearch = 55
119. .femote = 0                           154. .shhhhhhhhh = 2
120. .bounce = 0                           155. .nslookup = 0
121. .delete = 0                           156. .transport = 0
122. .botact = 2                           157. .destroy = 0
123. .exitmsg = 1                          158. .whois = 2
124. .addtime = 0                          159. .csbuf = 1
125. .frmail = 0                           160. .revshout = 1
126. .ustat = 1                            161. .topic = 30
127. .numcom = 5                           162. .accreq = 36
128. .getemail = 0                         163. .agree = 33
129. .anote = 0                            164. .dictionary = 0
130. .awipe = 0                            165. .dictadd = 0

<Jkb - 08:08,00:35>
```

Appendix I:

Analysis of Design Choices

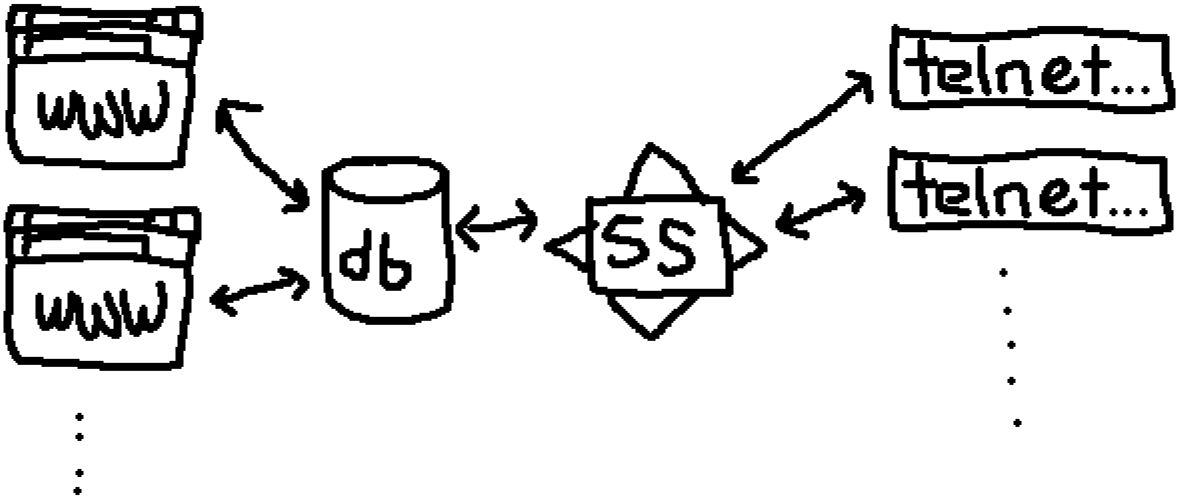
This Appendix contains an analysis of the three main design alternatives used to build the Java Talker Server portion of the Java Talker Project.

Analysis of Design Choices
CS690 Project
Jeffrey K. Brown
5/4/03

I was faced with a dilemna. I knew I needed a socket server, a database, a telnet interface and a web page interface. The question was one of how they fit together. I came up with three possibilities and thought about them at length. I decided to diagram them and discuss them with Professor Sonderegger to hear her opinion on them. In the end, I decided upon Design 3.

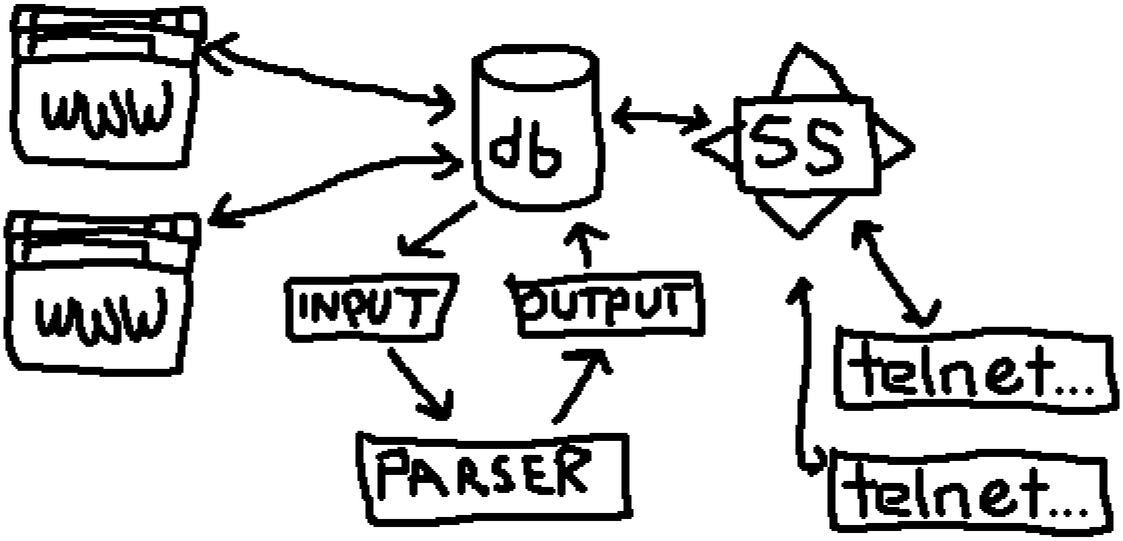
Lets take a closer look at the three Designs.

Design 1:



In this approach, the web interface would directly communicate with the database. The telnet interfaces would talk to the Socket Server, which would maintain the connections and also talk to the database on behalf of each connection. This was my original idea, but I decided against this choice for two main reasons. First, because the web interface talked to the system one way, and the socket server interface talked another way, I would have to implement each command twice. This double coding to satisfy both interfaces proved to be the biggest turnoff. Secondly, the tons of hits to the database may impact performance.

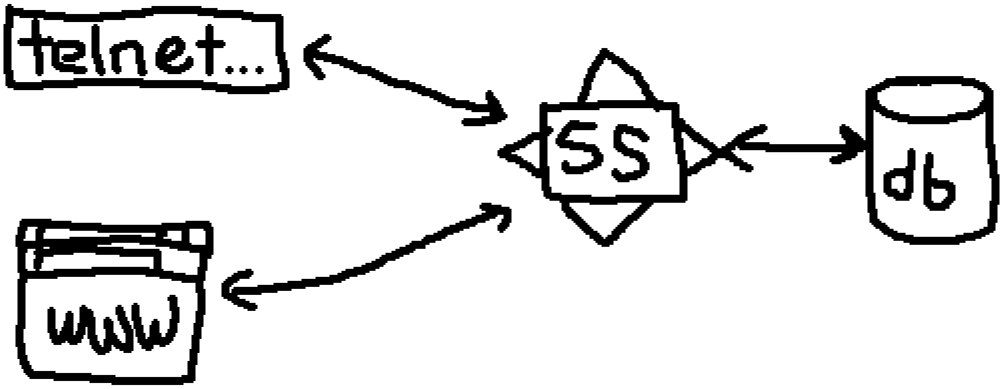
Design 2:



Design 2 attempted to overcome the double coding shortfalls of Design 1. The web interface and the telnet interface would both communicate directly to the database, like before. The difference would be that there is a third "server-type" element called the Parser. The Parser would read the Input Table, which is populated by the user input coming from each interface. The Parser would then translate the input into the appropriate output table and then the Web interface and Socket interfaces would query the output table for some output and produce it for the users when it appears.

I liked this approach very much, because it provided a standard approach for either (and other) interfaces. In the event where someone would like to run a talker without a web interface, or a web interface without a socket interface, this would be great. Of course, the shortcoming to this would be the database hits. There would still be lots of them.

Design 3:



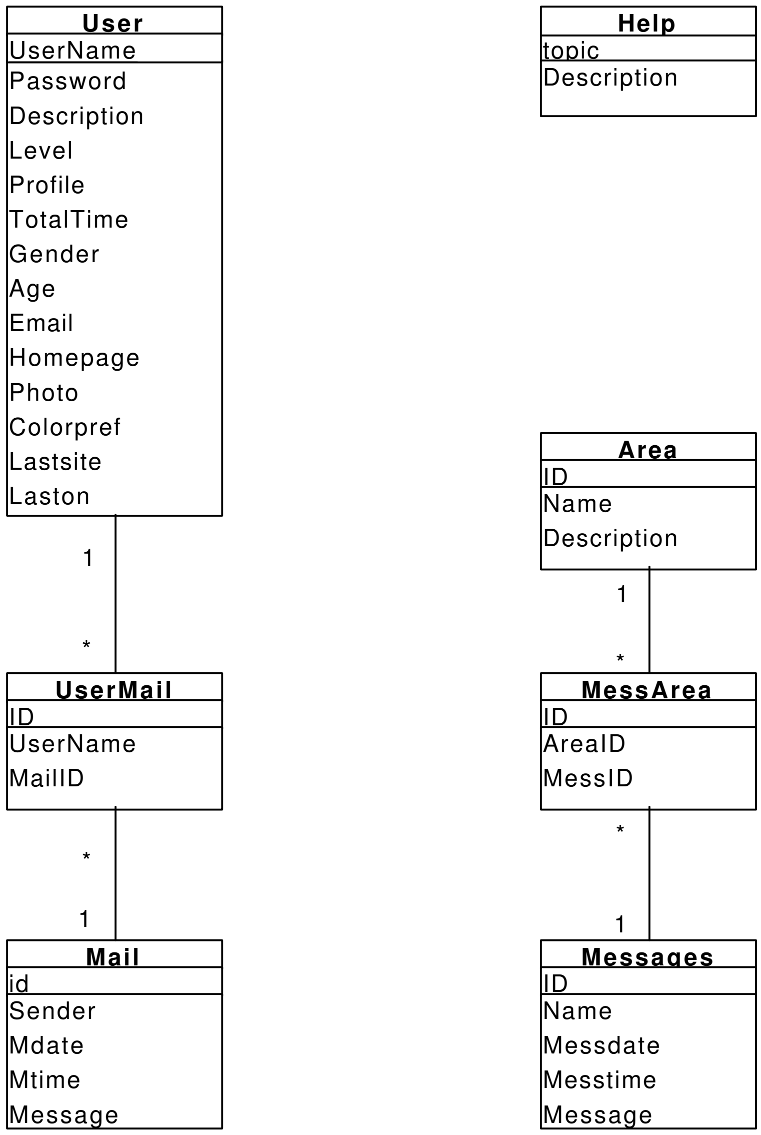
Design 3 is the simplest one. The web interface would be achieved using a combination of Java Applets and socket communication, while the Telnet interface would work the way it usually does. The web interface would have to send special input codes so that the socket server realizes that it is a web interface talking rather than a socket interface, so it would send data parsed in that particular way. Because the communication is not handled by the database, our database acecsses are no longer a problem. We will still use the database for user and environmental data.

Now that I have decided upon a design strategy, I can begin implementing the design.

Appendix J:

Database Entity Relationship Diagram

This Appendix contains an entity relationship diagram of the Java Talker database.



CS690: Java Talker
Database Relationship Diagram

Appendix K:

C++ Socket Server Example

This Appendix contains the source code for a socket server written using the C socket API. This can be used to compare against the Java Socket API employed by the Java Talker Server.

```
/*
C/C++ Example of a Multiuser Socket Server
Jeffrey K. Brown
CS690

This example was put together using examples from the Peterson & Davie,
Walton and Gay textbooks and my previous knowledge of Socket Programming
from the NUTS server.

This was an experiment to see whether I should use C++ to build the new
Talker Server or not.  You will probably find that this program is not as
refined as the Java Talker.  This is because I was looking to see what was
involved with creating it, rather than actually designing the finished
product.

With this example, I hope to show some means of comparing this Socket API
to the Java Socket API.
*/

/*
First, we include some of the necessary headers...
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <netdb.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <fcntl.h>
#include <dirent.h>
#include <signal.h>
#include <time.h>
#include <errno.h>

#include <sys/uio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/select.h>

/*
Next, we set up some constants.
*/
#define SERVER_PORT 2222
#define MAX_PENDING 5
#define MAX_LINE 256

/*
I've defined the color codes here.  You'll see in this example
that they are used the same way as in the Java Talker.
*/
#define RED     "\033[31m\033[1m"
#define GREEN   "\033[32m\033[1m"
#define YELLOW  "\033[33m\033[1m"
#define BLUE    "\033[34m\033[1m"
#define PURPLE  "\033[35m\033[1m"
#define SKY     "\033[36m\033[1m"
#define WHITE   "\033[37m\033[1m"
#define NORMAL  "\033[0m"

/*
I created a USER object which holds the user's socket and some basic
user information.  The UserConnection Object in the Java Talker stores
the User data in the same manner.
*/
class USER {
        public:
                char name[30];
                int socket;
                unsigned int len;
                USER();
};

USER::USER() {
        strcpy(name,"");
        socket=-1;
        len=0;
}



int main () {


        struct sockaddr_in sin;

        /* These are some variables that are used */
        char buffer[256];
        char outbuf[256];
        unsigned int len;
        int s, new_s;
        int buflen;
        int flag = 1;
        int user=0;
        int u=0;
        int retval;
        int tshutdown=0;
        int i=0;
        int erase=0;

        /*
        In this example, the array of Users with sockets is manually
        set to maximum of 3.  In the Java Talker, we grow and shrink
        the Array based on the number of connections needed.
        */
        USER ustr[3];

        /*
        Readmask is the set of File Descriptors that we check for any
        available socket input.
        */
        fd_set readmask;

        sprintf(buffer, "\n\n%sStarting the Server%s\n\n", GREEN,NORMAL);
        printf(buffer);
        strcpy(buffer,"");

        // Initialize (bzero) and Build Address Data Structure
        bzero((char*) &sin, sizeof(sin));
        sin.sin_family = PF_INET;
        sin.sin_addr.s_addr = INADDR_ANY;
        // Network Protocols like to talk with big-endian numbers
        // htons() is short for host to network short, so it changes
        // SERVER_PORT from its representation on the system to
        // big-endian representation.
        sin.sin_port = htons(SERVER_PORT);

        /* Initializes the Server Socket and sets the Queues for Reading
        and Writing the Server Socket. */
        if ((s = socket (PF_INET, SOCK_STREAM, 0)) < 0) {
                printf("\nError Opening the Server Socket!\n");
                exit(1);
        }

        /*
        During testing of this simple server, if I made a modification,
        recompiled and restarted the server, sometimes I would get a message
        claiming that the port was already in use.  This sets the socket
        options at the Socket Level to permit the reuse of the port even
        if other resources weren't totally done with it.  Different layers
        like SOL_IP (or SOL_IPV6) and SOL_TCP can have options set, too.
        SOL_SOCKET is the socket layer.
        */
        setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)&flag, sizeof(flag));

        /* Connects the Queues represented by the Server Socket to the Network */
        if ((bind (s, (struct sockaddr*) &sin, sizeof(sin))) < 0) {
                printf("\nError Binding the Server Socket to the Socket!\n");
                exit(1);
        }

        /*
        This is only allowed for SOCK_STREAM sockets.  It tells the socket s
        to listen for and create a queue of pending connections.
        */
        listen (s, MAX_PENDING);

        /*
        This is used to tell the File Descriptors related to the socket to
        be nonblocking.  If Input is seen, it will be held until the system
        gets around to it, but processing will continue.
        */
        fcntl(s, F_SETFL, O_NDELAY);

        while (1) {
                /*
                Initializes all of the File Descriptors by setting them to 0.
                */
                FD_ZERO(&readmask);

                /*
                Check Each available socket and set the File Descriptor of
                that Socket to the File Descriptor Set.  This creates a flag
                of sorts that will indicate that there is input waiting to
                be read on the corresponding File Descriptor.
                */
                for (user=0; user<3; user++) {
                        if (ustr[user].socket != -1) {
                                FD_SET(ustr[user].socket,&readmask);
                        }
                }
                /*
                Set the Server Socket's File Descriptor in the File Descriptor
                Set.  This will allow us to see any new activity from the Server
                Socket by using a flag.
                */
                FD_SET(s, &readmask);

                /*
                Select() runs through all of the File Descriptors in the File
                Descriptor set checking to see if the corresponding socket is
                ready to be read.  The "Ready for Reading" flags in readmask
                are set if the socket is indeed ready for reading.  We could
                also check the "Ready for Writing" or the "Exception" flags,
                but we are only interested in reading at the moment.  The
                error value is -1, so we start over if we encounter one.
                */
                if (select(FD_SETSIZE,&readmask,0,0,0)== -1) continue;
                printf(".");

                for (user=0; user<3; user++) {

                        printf("\n%d.) %d %d %d", user, ustr[0].socket, ustr[1].socket, ustr[2].socket);

                        strcpy(buffer, "");
                        strcpy(outbuf, "");
                        len = sizeof(sin);

                        /*
                        Accept reads the Server Socket for a new connection.  If one
                        exists, it will be assigned to new_s.  Sockets are represented
                        by integers greater than 3, so we test for it.  If we find it,
                        we assign the new socket to the UserConnection array in the
                        next available spot.  Otherwise, this returns 0, meaning no
                        new sockets have been accepted.
                        */
                        new_s = accept (s, (struct sockaddr *)&sin, &len);

                        if (new_s > 3) {
                                for (u=0; u<4; ++u) {
                                        if (ustr[u].socket == new_s) continue;
                                        if (ustr[u].socket == -1) {
                                                ustr[u].socket = new_s;
                                                break;
                                        }
                                        if (u==4) break;
                                }
                                if (u==4) {
                                        sprintf(buffer, "\nSorry, no room to log in!\n");
                                        write (new_s, buffer, strlen(buffer));
                                        close(new_s);
                                        continue;
                                }
                        }

                        /* If we are looking at an empty User object, continue */
                        if (ustr[user].socket == -1) continue;

                        /*
                        The select() section above sets the File Descriptor flags if any
                        input is detected on a socket.  This macro checks the flags,
                        continuing if there is no input detected.
                        */
                        if (!FD_ISSET(ustr[user].socket, &readmask)) continue;

                        /*
                        If the flag for this socket is set, the flow would reach this
                        point, and we read the data from the socket, storing it in
                        buffer.  The length of the data is returned and stored in len.
                        */
                        if ((len = read (ustr[user].socket, buffer, sizeof(buffer)) && (ustr[user].socket
!= -1))) {

                                /*
                                This reads a ".q" command as a Quit Command.  It
                                prints a message to the user's socket using the
                                write() function and then disconnects that user's
                                socket.
                                */
                                if ((buffer[0]=='.') && (buffer[1]=='q')) {
                                        sprintf(buffer, "\nDisconnecting you...\n\n");
                                        len = write (ustr[user].socket, buffer, strlen(buffer));
                                        strcpy(buffer,"");
                                        sprintf(buffer, "\nUser %d quits...\n", user);
                                        printf(buffer);
                                        retval = close(ustr[user].socket);
                                        ustr[user].socket = -1;
                                }

                                /*
                                The ".y" command works as a System Shutdown command.
                                We print a message back to the user with write() and
                                then falsify the loop condition.  Once out of the loop
                                we will close all the sockets.
                                */
                                else if ((buffer[0]=='.') && (buffer[1]=='y')) {
                                        sprintf(buffer, "\nShutting down server...\n\n");
                                        len = write (ustr[user].socket, buffer, strlen(buffer));
                                        strcpy(buffer,"");
                                        tshutdown=1;
                                }

                                /*
                                This is where normal communication would be processed.
                                In its present state, the system simply echoes the data
                                read back out to the user via the write() function and
                                prints the data to the server's stdout.
                                */
                                else {
                                        erase=0;
                                        for (i=0; i<strlen(buffer); ++i) {
                                                if (buffer[i] < 32) {
                                                        buflen= i+1;
                                                        erase=1;
                                                        continue;
                                                }
                                                if (erase==1) buffer[i]=0;
                                        }

                                        printf("\nReceived from %d: %s", user,buffer);

                                        sprintf(outbuf, "%sReceived: %s%s", RED, buffer, NORMAL);
                                        len = write(ustr[user].socket, outbuf, strlen(outbuf));

                                        strcpy (buffer, "");
                                        strcpy (outbuf, "");
                                }
                        }

                        /*
                        If the Talker Server is shutting down, we close all of
                        the User Sockets and then close the Server Socket.
                        */
                        if (tshutdown == 1) {
                                printf("\n\nShutting Down the System\n\n");
                                for (i=0; i<3; ++i) retval = close(ustr[i].socket);
                                retval = close(s);
                                break;
                        }

                }
                if (tshutdown==1) break;
        }

        sprintf(buffer, "\n\n%sServer Closed!%s\n\n",GREEN,NORMAL);
        printf(buffer);

        exit(0);

}
```

Appendix L:

Talker Packet Sniffing

This Appendix contains and experiment and results of running a packet sniffer to capture communication data from talkers. With this data, I demonstrate the vulnerability of Talker messaging due to lack of encryption of its server-client communication.

Talker Packet Sniffing using TCPDUMP
CS690 Java Talker
Jeffrey K. Brown

This is an analysis of the output of a program called TCPDUMP. TCPDUMP's purpose is to capture the packets passing a network interface that meet a certain criteria. It is often used to analyze network traffic, troubleshoot network problems and in some cases, used by malicious users and attackers to capture information about a computer system. I asked the TCPDUMP program to capture packets intended for port 2121 on my local network where I have a development version of NUTS-based Dark Ages Talker running. Through this experiment, I am backing up my assertion that many talkers send unencrypted conversation data across the Internet.

First, let me explain the environment in which I am performing this experiment. I have two computers participating, llama.universe.net running Windows 2000 and chimaera.universe.net running SuSE Linux. Chimaera will host the actual Talker server on port 2121 and permit connections. Additionally, I will connect to the Talker from Chimaera. Then, the user, Summoner, will connect to the Talker server from Llama using a popular Talker client, GMud. TCPDUMP will monitor the communication taking place from Chimaera. On the uncontrolled Internet, an attacker probably will not have direct access to one of the systems participating in a data exchange. However, once an attacker positions the monitoring equipment along the path between client and server, communication is monitored in exactly the same manner.

Next, let me explain how to use the TCPDUMP program. This is the syntax I used for this TCPDUMP command:

```
# tcpdump -i eth0 -s 0 -tlX port 2121 > tcpdump.log
```

Parts of the Command:

```
#                Root Shell
tcpdump          Call to the TCPDUMP program
-i eth0          We are asking TCPDUMP to listen on my Ethernet card.
-s 0             This is the "snaplen" value for TCPDUMP.  In other words, this
                 is used for the maximum number of bytes to capture for each
                 packet.  By default, TCPDUMP captures 68 bytes.  To capture
                 the entire conversation, I needed to get all the bytes in the
                 packet, and using 0 for the length means "grab the whole thing"
-t               Do not print a timestamp.  I wanted more room for the ASCII
l                Buffer stdout
X                Print the ASCII text in addition to the Hexadecimal numbers.
port 2121        Capture packets with the source or destination port of 2121
> tcpdump.log    Redirect all output to the log file, which I used to create
                 this document.
```

Next, we're going to view a packet and dissect the parts, so the reader will understand what we are looking at when the experiment gets interesting. Below you will see a typical TCPDUMP packet. In this particular packet, nothing exciting is going on, so it will be a good packet to dissect.

```
llama.universe.net.1158 > chimaera.universe.net.2121: . ack 1400 win 64136 (DF)
0x0000   4500 0028 0f5d 4000 8006 67b2 c0a8 0139        E..(.]@...g....9
0x0010   c0a8 0137 0486 0849 62fc 0a7a 850e 6156        ...7...Ib..z..aV
0x0020   5010 fa88 d0e0 0000 0000 0000 0000             P.............
```

First, let's look at the header line below. We are going to pay attention to the host name and port, the direction of communication, and the destination host name and port. Other data is printed, but it is beyond the scope of this document.

```
llama.universe.net.1158 > chimaera.universe.net.2121: . ack 1400 win 64136 (DF)
```

Parts of the Header:

```
llama.universe.net       This is the source address or hostname.  Since the
                         computer doing the logging has the capability to look
                         up the name, it prints the name instead of the address
```

```
.1158                   This is the source port of the connection.  This source
                        port is usually defined by the computer's operating
                        system.
>                       This is a symbol showing the "direction" of the packet
                        i.e. from source to destination.
chimaera.universe.net   This is the destination adddress or hostname.
.2121                   This is the destination port of the connection.  Since
                        the Talker service runs on port 2121, connections to it
                        must be destined for that service port.
everything else         While useful, this data is beyond the scope of this
                        experiment, so disregard it.
```

Next, we will take a quick look at the payload section of the packet. There are three portions of the payload section that we will be interested in. The first column is the sequence of bytes in the payload. The Hex number shows the sequence of the first byte on that row. Next, we have the Hexadecimal representation of the packet payload. Finally, in the last column, we have the corresponding ASCII representation of the packet payload for the readers who are not as skilled at reading in Hex. The packet below does not send any data that is readable, but we will see some later.

```
Sequence         Hexadecimal Representation            ASCII Representation
 Number             of the Packet Payload               of Packet Payload
|-----|-|---------------------------------------|------|-----------------|
0x0000   4500 0028 0f5d 4000 8006 67b2 c0a8 0139        E..(.]@...g....9
0x0010   c0a8 0137 0486 0849 62fc 0a7a 850e 6156        ...7...Ib..z..aV
0x0020   5010 fa88 d0e0 0000 0000 0000 0000             P.............
```

Now we will get into the actual communication taking place. Prior to the interesting packet, I will leave my comments and explain what is taking place and why I consider it interesting.

We pick up the conversation after the Talker system has prompted the user for the username. The user types the username and transmits it from Llama to Chimaera. As you can see, the username is clearly displayed.

```
llama.universe.net.1158 > chimaera.universe.net.2121: P 1:10(9) ack 1400 win 64136
0x0000   4500 0031 0f5e 4000 8006 67a8 c0a8 0139        E..1.^@...g....9
0x0010   c0a8 0137 0486 0849 62fc 0a7a 850e 6156        ...7...Ib..z..aV
0x0020   5018 fa88 110c 0000 7375 6d6d 6f6e 6572        P.......summoner
0x0030   0a                                             .
chimaera.universe.net.2121 > llama.universe.net.1158: . ack 10 win 5840 (DF)
0x0000   4500 0028 7f0f 4000 4006 3800 c0a8 0137        E..(..@.@.8....7
0x0010   c0a8 0139 0849 0486 850e 6156 62fc 0a83        ...9.I....aVb...
0x0020   5010 16d0 b490 0000                            P.......
```

Next, the Talker asks the user to authenticate by providing a password.

```
chimaera.universe.net.2121 > llama.universe.net.1158: P 1400:1443(43) ack 10 win 5840
0x0000   4500 0053 7f10 4000 4006 37d4 c0a8 0137        E..S..@.@.7....7
0x0010   c0a8 0139 0849 0486 850e 6156 62fc 0a83        ...9.I....aVb...
0x0020   5018 16d0 9f10 0000 1b5b 3332 6d1b 5b31        P........[32m.[1
0x0030   6d4f 6b2c 2070 726f 7665 2074 6861 7420        mOk,.prove.that.
0x0040   6974 2069 7320 796f 753a 1b5b 306d 201b        it.is.you:.[0m..
0x0050   5b30 6d                                        [0m
llama.universe.net.1158 > chimaera.universe.net.2121: . ack 1443 win 64093 (DF)
0x0000   4500 0028 0f5f 4000 8006 67b0 c0a8 0139        E..(._@...g....9
0x0010   c0a8 0137 0486 0849 62fc 0a83 850e 6181        ...7...Ib.....a.
0x0020   5010 fa5d d0d7 0000 0000 0000 0000             P..]..........
```

The User, Summoner, types his Password.

```
llama.universe.net.1158 > chimaera.universe.net.2121: P 10:19(9) ack 1443 win 64093
0x0000   4500 0031 0f60 4000 8006 67a6 c0a8 0139        E..1.`@...g....9
0x0010   c0a8 0137 0486 0849 62fc 0a83 850e 6181        ...7...Ib.....a.
0x0020   5018 fa5d 252b 0000 6d65 6368 6d65 6368        P..]%+..mechmech
0x0030   0a                                             .
chimaera.universe.net.2121 > llama.universe.net.1158: P 1443:1475(32) ack 19 win 5840
0x0000   4500 0048 7f11 4000 4006 37de c0a8 0137        E..H..@.@.7....7
```

```
0x0010   c0a8 0139 0849 0486 850e 6181 62fc 0a8c        ...9.I....a.b...
0x0020   5018 16d0 6b1b 0000 1b5b 3333 6d1b 5b31        P...k....[33m.[1
0x0030   6d41 7574 6f20 5772 6170 2073 6574 1b5b        mAuto.Wrap.set.[
0x0040   306d 0a0d 1b5b 306d                            0m...[0m
llama.universe.net.1158 > chimaera.universe.net.2121: . ack 1475 win 65535 (DF)
0x0000   4500 0028 0f61 4000 8006 67ae c0a8 0139        E..(.a@...g....9
0x0010   c0a8 0137 0486 0849 62fc 0a8c 850e 61a1        ...7...Ib.....a.
0x0020   5010 ffff cb0c 0000 0000 0000 0000             P.............
```

Once authenticated, the system provides a welcome message to the user.

```
chimaera.universe.net.2121 > llama.universe.net.1158: P 1475:2761(1286) ack 19 win 5840
0x0000   4500 052e 7f12 4000 4006 32f7 c0a8 0137        E.....@.@.2....7
0x0010   c0a8 0139 0849 0486 850e 61a1 62fc 0a8c        ...9.I....a.b...
0x0020   5018 16d0 aebe 0000 0a0a 0a0d 5765 6c63        P...........Welc
0x0030   6f6d 6520 5072 696e 6365 2053 756d 6d6f        ome.Prince.Summo
0x0040   6e65 720a 0a0d 1b5b 306d 4c61 7374 206c        ner....[0mLast.l
0x0050   6f67 6765 6420 696e 206f 6e20 4672 6920        ogged.in.on.Fri.
0x0060   4f63 7420 3331 2032 313a 3339 3a33 3920        Oct.31.21:39:39.
0x0070   3230 3033 2066 726f 6d20 6c6c 616d 612e        2003.from.llama.
0x0080   756e 6976 6572 7365 2e6e 6574 0a0a 0d1b        universe.net....
```

Next, we will skip a few packets and jump ahead to some actual conversation taking place.

Summoner, from Llama, sends a friendly greeting to the Jkb user on Chimaera.
The Talker system sends Summoner an acknowledgement by repeating what was sent.

```
llama.universe.net.1158 > chimaera.universe.net.2121: P 19:46(27) ack 2761 win 64249
0x0000   4500 0043 0f63 4000 8006 6791 c0a8 0139        E..C.c@...g....9
0x0010   c0a8 0137 0486 0849 62fc 0a8c 850e 66a7        ...7...Ib.....f.
0x0020   5018 faf9 9613 0000 6865 6c6c 6f21 2020        P.......hello!..
0x0030   486f 7720 6172 6520 796f 7520 746f 6461        How.are.you.toda
0x0040   793f 0a                                        y?.
chimaera.universe.net.2121 > llama.universe.net.1158: P 2761:2800(39) ack 46 win 5840
0x0000   4500 004f 7f13 4000 4006 37d5 c0a8 0137        E..O..@.@.7....7
0x0010   c0a8 0139 0849 0486 850e 66a7 62fc 0aa7        ...9.I....f.b...
0x0020   5018 16d0 5538 0000 596f 7520 6173 6b3a        P...U8..You.ask:
0x0030   2068 656c 6c6f 2120 2048 6f77 2061 7265        .hello!..How.are
0x0040   2079 6f75 2074 6f64 6179 3f1b 5b30 6d          .you.today?.[0m
```

Next, the Jkb user responds. The TCPDUMP packet sniffer was placed on the Ethernet interface. Since Jkb was connected from the server machine and does not need to traverse the Ethernet interface, messages sent by Jkb to the Talker are not captured. Jkb's messages sent through the path that TCPDUMP is monitoring, however, are captured and logged, as you can see below. This is a realistic simulation because all users are not necessarily connected over the same path through the Internet.

```
chimaera.universe.net.2121 > llama.universe.net.1158: P 2832:3051(219) ack 46 win 5840
0x0000   4500 0103 7f15 4000 4006 371f c0a8 0137        E.....@.@.7....7
0x0010   c0a8 0139 0849 0486 850e 66ee 62fc 0aa7        ...9.I....f.b...
0x0020   5018 16d0 ac55 0000 4a6b 6220 7361 7973        P....U..Jkb.says
0x0030   3a20 4669 6e65 2e20 2049 2061 6d20 7573        :.Fine...I.am.us
0x0040   696e 6720 5443 5044 554d 5020 746f 2076        ing.TCPDUMP.to.v
0x0050   6965 7720 616c 6c20 6f66 2074 6865 2070        iew.all.of.the.p
0x0060   6163 6b65 7473 2063 6f6d 696e 6720 696e        ackets.coming.in
0x0070   746f 2061 6e64 206f 7574 206f 6620 7468        to.and.out.of.th
0x0080   6520 5461 6c6b 6572 2773 2073 6572 7665        e.Talker's.serve
0x0090   7220 736f 636b 6574 2e20 2042 6563 6175        r.socket...Becau
0x00a0   7365 2074 6865 7920 6172 6520 7472 616e        se.they.are.tran
0x00b0   736d 6974 7465 6420 756e 656e 6372 7970        smitted.unencryp
0x00c0   7465 642c 2069 7420 6973 2065 6173 7920        ted,.it.is.easy.
0x00d0   746f 2070 756c 6c20 7468 6520 6461 7461        to.pull.the.data
0x00e0   2063 6f6e 7465 6e74 7320 6f75 7420 6f66        .contents.out.of
0x00f0   2074 6865 2070 6163 6b65 7473 2e0a 0d1b        .the.packets....
0x0100   5b30 6d                                        [0m
```

Summoner responds to this message and gets another acknowledgement.

```
llama.universe.net.1158 > chimaera.universe.net.2121: P 46:64(18) ack 3051 win 65535
```

```
0x0000   4500 003a 0f6b 4000 8006 6792 c0a8 0139        E..:.k@...g....9
0x0010   c0a8 0137 0486 0849 62fc 0aa7 850e 67c9        ...7...Ib.....g.
0x0020   5018 ffff 9e32 0000 5665 7279 2069 6e74        P....2..Very.int
0x0030   6572 6573 7469 6e67 210a                       eresting!.
chimaera.universe.net.2121 > llama.universe.net.1158: P 3051:3085(34) ack 64 win 5840
0x0000   4500 004a 7f16 4000 4006 37d7 c0a8 0137        E..J..@.@.7....7
0x0010   c0a8 0139 0849 0486 850e 67c9 62fc 0ab9        ...9.I....g.b...
0x0020   5018 16d0 68b6 0000 596f 7520 6578 636c        P...h...You.excl
0x0030   6169 6d3a 2056 6572 7920 696e 7465 7265        aim:.Very.intere
0x0040   7374 696e 6721 1b5b 306d                       sting!.[0m
llama.universe.net.1158 > chimaera.universe.net.2121: . ack 3085 win 65501 (DF)
0x0000   4500 0028 0f6c 4000 8006 67a3 c0a8 0139        E..(.l@...g....9
0x0010   c0a8 0137 0486 0849 62fc 0ab9 850e 67eb        ...7...Ib.....g.
0x0020   5010 ffdd c4b7 0000 0000 0000 0000             P.............
```

What I have demonstrated using the preceding experiment was the fact that Internet communication, in this case, Talkers, often send data in unencrypted form. Attackers employing tools such as TCPDUMP can capture this data and perform acts of blackmail, industrial or political espionage and disclosure of secrets. Capturing usernames and passwords could allow malicious users to commit identity theft crimes.

In conclusion, I think it would be a good idea to add some level of encryption to Talker communication. One of the goals of reimplementing the Talker in Java was because Java has some platform independent encryption libraries. Hopefully, experience with this project will allow me to provide some ways in which to implement encryption and secure the Talker communication.

Appendix L:

Project CD

On this CD, you will find the following:

1. PDF Version of Java Talker Report
2. Java Talker Source Code
3. Javadoc Documentation