CoLab

Deliverable Two

CS 3300

Group Seven

Liem, Johannes
Luongo, Matt
Martin, Chris
Overman, Pamela

April 1, 2008

Summary of Document Changes From D1

Product Overview	No change
Delivery Description	Covers D2 delivered components
Design Overview	No change
Design Detail	Class list now includes comments, methods and psuedocode
Design UML	No change
Data Design	Persistence support has been added to the server, and the format for saved files has been defined
Interface Specification	User interface for document editor channel specified
Demo Plan	Expanded to include functionality added in D2
Test Plan	System test cases added.
Requirements Specification	Includes delivery plan details for D3

Contents

Product Overview

1.1 Product Overview

CoLab is a network-enabled utility for communication, brainstorming, and collaborative document authoring. It is centered around a basic chat-room style environment, and is enhanced with inclusion of several tools to facilitate idea sharing within small groups. These tools include a collaborative document editor and whiteboard drawing tool.

Users create and join communities which are hosted on a single server. All of a community's related data is stored on a server for this application, including revisions of documents, saved drawings, and chat logs.

Within a community interface, users can open multiple editors in new channels and co-author documents simultaneously. For example, several people may be working on a diagram in the drawing tool while discussing it in a text chat.

1.2 Deliverables

The final product consists of two independent applications: a server and a client. Deliverables will consist of the full source code and compiled application code.

1.3 Definitions

This application introduces several terms to cover collaborative groups and functions. A *community* represents a group of people (*users*) and a project on which they are collaborating, such as a university class or a company project group.

Users with the permission to control membership in the community and access to channels are called *moderators*. In a classroom setting, moderator status would be limited to instructors and teaching assistants (TAs). In a flat group with no distinct authority, all users may be moderators.

Each document or collaborative entity is referred to as a *channel*. A channel represents content being edited, and the workspace in which the group is dealing with it. A *protected channel* can only be modified by community moderators. Each channel is of a particular type called a *channel protocol* which specifies the type of data represented by the channel and the way in which that data is displayed and manipulated by users. Examples of channel protocols are chat, document, and whiteboard.

2 Delivery Description

The networking and authentication framework has been implemented, including all features required to log in, join a chat channel, and use the chat feature.

2.1 Framework

2.1.1 User Accounts

A user can subsequently log into an account by providing a username and the correct password.

If the user enters a username that does not exist, the user is given the option to create a new user account.

The server keeps track of all user accounts, and saves them in a file to maintain this data persistently.

2.1.2 Communities

Every channel exists within a community. A community member can see every channel in the community, and the data is hidden from non-members.

Once a user authenticates with the server, he may then sign in to any community in which he has membership. An instance of the client application can participate in at most one community at a time.

A user can join a new community by choosing it from a list of all communities on the server and providing the community password.

A new community can be created by any authenticated user.

The server keeps track of all communities, and saves them in a file to maintain this data persistently.

2.1.3 Channels

Once logged into a community, a user sees the list of channels in that community.

The user can create new channels, and join already existing channels.

The framework provides a Revision History panel, which can be used by any channel to display a list revisions to the channel.

Users can participate in multiple channels concurrently.

The server keeps track of all data added to channels, and saves them in a file to maintain this data persistently.

2.2 Chat Channel Protocol

A user can post messages to the channel. When a message is posted, it becomes immediately visible to all other users participating in the channel.

Each message includes a timestamp, and all messages are seen chronologically.

The view looks like a typical chat system. Most of the screen space is used to display the list of messages which have been posted; older messages can be viewed by scrolling upwards in this panel. A single-line text entry mechanism below that display is used to post new messages.

2.3 Document Channel Protocol

Documents are divided into newline-delimited entities called *paragraphs*. At most one user can modify a paragraph at one time.

When a user begins editing a paragraph, his client acquires a lock on that paragraph to prevent editing conflicts.

A lock held by another user is designated by a discolored background on the paragraph. Similarly, another color signifies a lock held by one's own client.

A paragraph can be designated as a *header*. Headers come in several levels to enable hierarchically organized documents.

The entire view for a document channel consists of a scrolling panel which displays the document text.

A user can switch his view into *revision history* mode, which provides a navigation mechanism for selecting prior revisions of the document and viewing them in an uneditable panel.

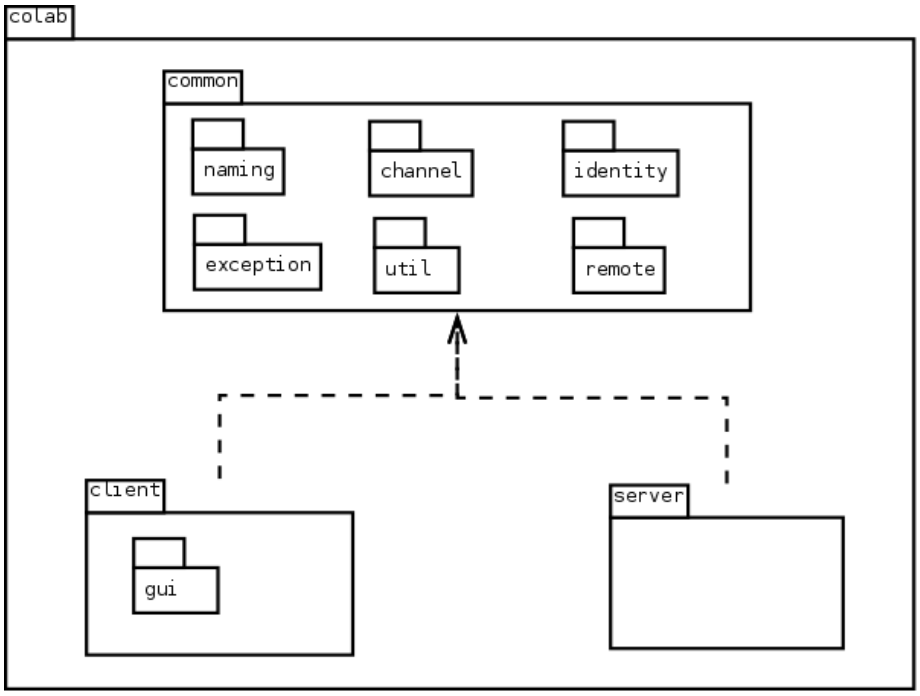
A button allows the user to revert to a given revision. This updates the current contents of the channel to an old state, and puts the view back into editing mode. All locks are removed, and all participants in the channel see the change immediately.

The entire contents of the document can be exported as an html file.

3 Design Overview

The Java code is segmented into three main packages

- The client application
- The server application
- A common package which contains classes used by both applications and specifies interfaces for communication between the client and server



Design Detail

3.1 colab.client

Lays the domain for the functions of the application and contains the middle layer between the GUI and the server.

ClientChannel

A client-side remote Channel object.

Implements: ChannelRemote

void handleUserEvent(UserJoinedEvent)
    Add the new user to my list of channel members
    Notify all listeners (to update the UI)

void handleUserEvent(UserLeftEvent)
    Removes the user from my list of channel members
    Notify all listeners (to update the UI)

ClientChatChannel

A ClientChannel that handles chat-protocol channels.

Extends: ClientChannel

void add(ChannelData)
    Add a piece of chat channel data to my list of messages
    Notify all listeners (to update the UI)

ClientDocumentChannel

A ClientChannel that handles document-protocol channels.

Extends: ClientChannel

void add(ChannelData)
    Add a piece of document channel data to my list of messages
    Notify all listeners (to update the UI)

ColabClient

The CoLab client application which is used to make requests to the server.

void connect(String serverAddress)
    Obtain a connection to the server
    Update my internal state to note that client has connection

void loginUser(String username, char[] password, String serverAddress)
    connect(serverAddress)
    Send the user login request to the server
        Throw AuthenticationException on failure
    Update my internal state to note that the client is logged in to a user account

void createUser(UserName userName, char[] password)
    Send a request to the server to create a new user account

void loginCommunity(CommunityName, char[] password)
    Send the server a request to join a new community with the provided password
        Throw AuthenticationException on failure
    Update my internal state to note that the client is logged in to a community

void loginCommunity(CommunityName)
    Send the community login request to the server
    Update my internal state to note that the client is logged in to a community

Collection<CommunityName> getAllCommunityNames()
    Retrieve from the server: a collection of all community names

Collection<CommunityName> getMyCommunityNames()
    Retrieve from the server: the communities of which the user is a member

ClientChannel joinChannel(ChannelDescriptor)
    Create a ClientChannel for the given channel descriptor
    Send the server a remote reference to the ClientChannel

void leaveChannel(ChannelDescriptor)
    Inform the server that the client has left the channel

void add(ChannelName, ChannelData)
    Send the new piece of channel data to the server
    Set the data's identifier (provided by the server)

void logOutUser()
    Send the user logout message to the server
    Update my internal state to note that the client is connected but not logged in

void logOutCommunity()
    Send the community logout message to the server
    Update my internal state to note that the client is not logged into a community

void createCommunity(CommunityName, char[] password)
    Send the community creation request to the server
        Throw a CommunityAlreadyExistsException if server threw one

void createChannel(ChannelDescriptor)
    Send a request to the server to create a channel matching the given descriptor
        Throw a ChannelAlreadyExistsException if the server threw one

void exitProgram()
    Sends a user logout message to the server, to be polite
    Halts the application

3.2 colab.common

3.2.1 colab.common.channel

Holds classes pertaining to Channel and its functions and features.

ChannelData

Abstract class that represents some piece of data in a Channel.

ChannelDataIdentifier id
: The id of this piece of date (unique to its respective channel).

UserName creator
: The user from whom this piece of data originated.

Date timestamp
: The time at which the data was originally created.

ChannelDataIdentifier

Identifier for channel data to uniquely identify data elements within a channel, and to keep them in sequence.

Extends: IntegerIdentifier.

ChannelDataSet

A set of ChannelData, sorted by their identifiers.

Parameter <T>: The type of channel data in the set

Extends: ChannelData

Implements: ChannelDataStore

private Integer nextDataId = 1
: An integer which is always higher than the highest identifier value in the set
Used to assign an identifier to the next piece of data that is added.

ChannelDescriptor

Descriptive data used to identify a channel.

ChannelName channelName
: The name of the channel.

ChannelType channelType
: The type of channel.

ServerChannel createServerChannel(File file)
: If file is null, data will not be stored persistently
createServerChannel(null) is equivalent to createServerChannel().

ChatChannelData

Represents a message posted to a chat channel.

String getMessageString(boolean timestampEnabled)
Returns the message with additional formatting applied.
The parameter timestampEnabled determines whether to include the timestamp.

3.2.2 colab.common.remote.client

Provides remote interfaces for remote client objects.

ChannelRemote (Interface)

A remote interface for a client-side object which the server application uses when it needs to update the client with some information about a channel in which the client is participating.

void add(ChannelData)
Informs the client that data has been added to the channel.

void handleUserEvent(UserJoinedEvent)
Informs the client that a user has joined (been added to) the channel.

void handleUserEvent(UserLeftEvent event)
Informs the client that a user has left (been removed from) the channel.

ColabClientRemote (Interface)

A remote interface for a client-side object which the server application uses when it needs to update the client with some information that is not specific to any particular channel.

void channelAdded(ChannelDescriptor)
Notifies the client that a channel exists in the community the user has joined.

3.2.3 colab.common.remote.server

Provides remote interfaces for remote server objects.

ColabServerRemote (Interface)

The server interface is the first point of contact that the client has to the server. A single instance should be registered with the rmi registry so that clients can make invocations upon it. The purpose of this remote interface is for the client to gain a remote reference to a {@link ConnectionRemote} for any further activity.

ConnectionRemote connect(ColabClientRemote)
Creates a connection object for the client.

ConnectionRemote (Interface)

A remote object on the server which represents a client's session.

void logIn(UserName, char[] password)
Attempts to authenticate a user, using the name and password.

void logOutUser()
Logs out the user.

void logIn(CommunityName, char[] password)
Attempts to log into a community, using the name and optional password.

void logOutCommunity()
Logs out of the community.

Collection<CommunityName> getAllCommunityNames()
Retrieves the names of every community on the server.

Collection<CommunityName> getMyCommunityNames()
Retrieves the every community of which the current user is a member.

void joinChannel(ChannelRemote, ChannelDescriptor)
Retrieves a channel. The channel will be created if it does not exist.

void leaveChannel(ChannelName)
Indicates that the client has exited from a channel.

boolean isMember(CommunityName)
Checks whether the logged in user is a member of a given community.
Throws CommunityDoesNotExistException if the community does not exist.

List<ChannelData> getLastData(ChannelName, int count)
Gets the last [count] pieces of data from the specified Channel.

ChannelDataIdentifier add(ChannelName, ChannelData)
Adds this channel and data to the connection.

Collection<UserName> getActiveUsers(ChannelName)
Get a list of all users currently joined to this channel.

UserName getUserName()
Returns the user that is logged in.

void createUser(String userName, char[] password)
Creates a new user in this connection.

void createCommunity(String name, char[] password)
Creates a new community in this connection.

void createChannel(ChannelDescriptor)
Creates a new channel in the currently logged-in community.

boolean hasUserLogin()
Checks whether a user has logged into the program.

3.2.4 colab.common.util

Contains generic utility classes.

FileUtils

A utility class containing methods for dealing with files.

static void appendLine(File file, String line)
Writes a string and a line break to the end of a file.

static String getContentAsString(File file)
Reads all of the content from a file and returns it as a single string.

StringUtils

A utility class containing methods for dealing with strings.

static String emptyIfNull(String str)
Returns a given string, or an empty string if null.

static boolean isEmptyOrNull(String str)
Determines whether a string is empty or null.

static boolean containsOnlyCharacters(String str, String validCharacters)
Checks whether every character in a string is present in a valid characters set.

static String toLetters(byte[] bytes)
Converts a byte array, arbitrarily but consistently, to a string.

static String repeat(String str, int times)
Concatenates a string with itself repeatly and returns the result.

3.3 colab.server

Contains server objects.

ColabServer

Server implementation of ColabServerInterface

Implements: ColabServerRemote

UserManager userManager
The manager object that keeps track of users and communities for this server.

ChannelManager channelManager
The manager object that keeps track of channels for this server instance.

String rmiAddress
The address in the rmi registry to which this server is bound; null if not bound.

void publish(int port)
Adds the server to the rmi registry.

void unpublish()
Removes the server from the rmi registry.

void createCommunity(CommunityName, Password)
Creates a new community.
Throws CommunityAlreadyExistsException if community name exists.

void createCommunity(CommunityName, Password, UserName creator)
Creates a new community.

Throws CommunityAlreadyExistsException if community name exists.
Makes the creator a member of the group.

void createChannel(ChannelDescriptor, CommunityName)
Creates a new channel in a given community.
Throws CommunityDoesNotExistException if no such community exists
Throws ChannelAlreadyExistsException if channel name already exists

void addUser(User)
Adds a new user.
Throws UserAlreadyExistsException if user name already exists

boolean isMember(UserName, CommunityName)
Checks whether a user is a member of a given community.
Throws CommunityDoesNotExistException if no such community exists

boolean checkPassword(UserName, char[] password)
Checks that a user's password is correct
Throws AuthenticationException if not correct

Community getCommunity(CommunityName)
Retrieves a community by name
Throws CommunityDoesNotExistException if no such community exists

3.3.1 colab.server.channel

ChannelConnection

ChannelConnection is a wrapper for a remote ChannelRemote which associates the channel interface with the Connection on which it was created.

Connection connection
The connection which created the channel interface.

ChannelRemote channelInterface
The remote client channel object.

ChannelManager

A channel manager provides channels. This implementation does not load any channel data into memory until a channel is joined by one or more users.

public boolean channelExists(CommunityName, ChannelName)
Checks whether a channel exists in a given community.

ServerChannel

A ServerChannel is an abstract server-side object that represents a channel.

Parameter <T extends ChannelData>: The type of data the channel holds.

Extends: ChannelData

Implements: DisconnectListener

abstract List<T> getLastData(int n)
Return the last n data elements from the channel.

public boolean contains(Connection client)
Checks whether a client is part of this channel.

public boolean contains(UserName)
Checks whether a user is part of this channel.

protected void sendToAll(T data)
Sends a data element to every connected client except the data's creator.

ServerChatChannel

ServerChatChannel extends ServerChannel and deals with chat channels.

ChannelDataStore<ChatChannelData> messages
The channel data.

ServerDocumentChannel

ServerChatChannel extends ServerChannel and deals with document channels.

ChannelDataStore<DocumentChannelData> revisions
The channel data.

3.3.2 colab.server.connection

Connection

Server implementation of ConnectionRemote.

Implements: Identifiable<ConnectionIdentifier>, ConnectionRemote

private static Integer nextId
The integer to use as the id number for the next instantiated connection.
Used to ensure that each connection has a unique id.

ConnectionIdentifier connectionId
This connection's arbitrary but unique identifier.

ColabClientRemote client
A remote reference to the connected client.

ConnectionState state
The current state of the connection.

UserName username
The user that has logged in on this connection (if any).

Community community
The community that has been logged into on this connection (if any).

void addDisconnectListener(final DisconnectListener)
Ensures that the listener will be notified when this connection gets disconnected.

void disconnect(final Exception e)

Called when connection must be aborted due to fatal RMI exception.
Cleans up and removes itself from objects it was listening on.

ConnectionIdentifier

An integer identifier for a Connection.

Extends: IntegerIdentifier

3.3.3 colab.server.event

DisconnectEvent

An event which indicates that a Connection has been dropped.

ConnectionIdentifier connectionId

The id of the dropped connection.

Exception cause

An exception which may provide information about why the connection was lost.

DisconnectListener (Interface)

An object which can be notified when a connection is dropped.

void handleDisconnect(DisconnectEvent)

Called when the connection is dropped and event e is fired.

3.3.4 colab.server.file

ChannelFile

A collection of data in a channel, backed by a ChannelDataSet and maintained persistently by a text file. Implements ChannelDataStore<T> where T extends ChannelData.

ChannelDataSet<T> dataCollection

The backing data set.

File file

The file used for persistent storage.

CommunityFile

A collection of files, backed by a CommunitySet and maintained persistently by a text file.

Implements: CommunityStore.

CommunitySet communities

The backing community set.

File file
: The file used for persistent storage.

UserFile

A collection of users, backed by a UserSet and maintained persistently by a text file.

Implements: UserStore

UserSet users
: The backing user set.

File file
: The file used for persistent storage.

3.3.5 colab.server.user

Community

Represents a community which can be joined by users. This class represents the community domain object, and also provides server-related services by keeping track of which clients are connected and providing them with community-wide notifications (such as informing clients of newly-added channels).

Implements: Identifiable<CommunityName>, DisconnectListener

CommunityName name
: A unique string identifying this community.

Collection<UserName> members
: The users which have joined this community and can log in to it.

Password password
: The password to join this community.

IdentitySet<ConnectionIdentifier, Connection> clients
: A list of actively connected clients.

void channelAdded(ChannelDescriptor)
: Tells all clients that a new channel has been added.

void addClient(Connection)
: Informs the community that a client has logged in and connected.

void removeClient(final Connection)
: Informs the community that a user has logged out or disconnected.

boolean checkPassword(char[] attempt)
: Verifies whether a given password string is correct for this community.

boolean isMember(UserName)
: Determines whether a user is a member of the community.

boolean isActive(final UserName)
: Determines whether a user is currently logged into the community.

boolean authenticate(final UserName, final char[] passAttempt)
: Handles a user attempt to log in to this community.
If the user is already a member, the authentication is approved.
If password is correct, the user becomes a member of the community.
If neither, authentication is denied.

CommunitySet

A simple CommunityStore which stores all communities in a set.

Implements: CommunityStore

IdentitySet<CommunityName, Community> communities
: All of the communities that exist.

Password

Represents a string password. Stores only a hashed version of the string.

String hash
: The hashed password.

static MessageDigest digest
: A digest used to run the hash function.

boolean checkPassword(char[] pass)
: Determines whether a password string matches the password.

static String doHash(char[] characters)
: Performs a one-way hash on a string.

static String doHash(byte[] bytes)
: Performs a one-way hash on a byte array.

static void clear(byte[] bytes)
: Clears an array to ensure that the contents do not remain in memory.

User

Represents a user of the system.

UserName name
: A unique string identifying this user.

Password pass
: The password for this user to log in.

UserManager

A simple user manager that holds all users and communities in memory.

UserStore userStore
: The collection of users.

CommunityStore communityStore
: The collection of communities.

Collection<Community> getAllCommunities()
: Retrieves all of the communities on the server.

void addCommunity(Community)
: Adds a new community
: Throws CommunityAlreadyExistsException if the community already exists

boolean checkPassword(UserName, char[] password)
: Checks that a user's password is correct.
: Throws AuthenticationException if not correct

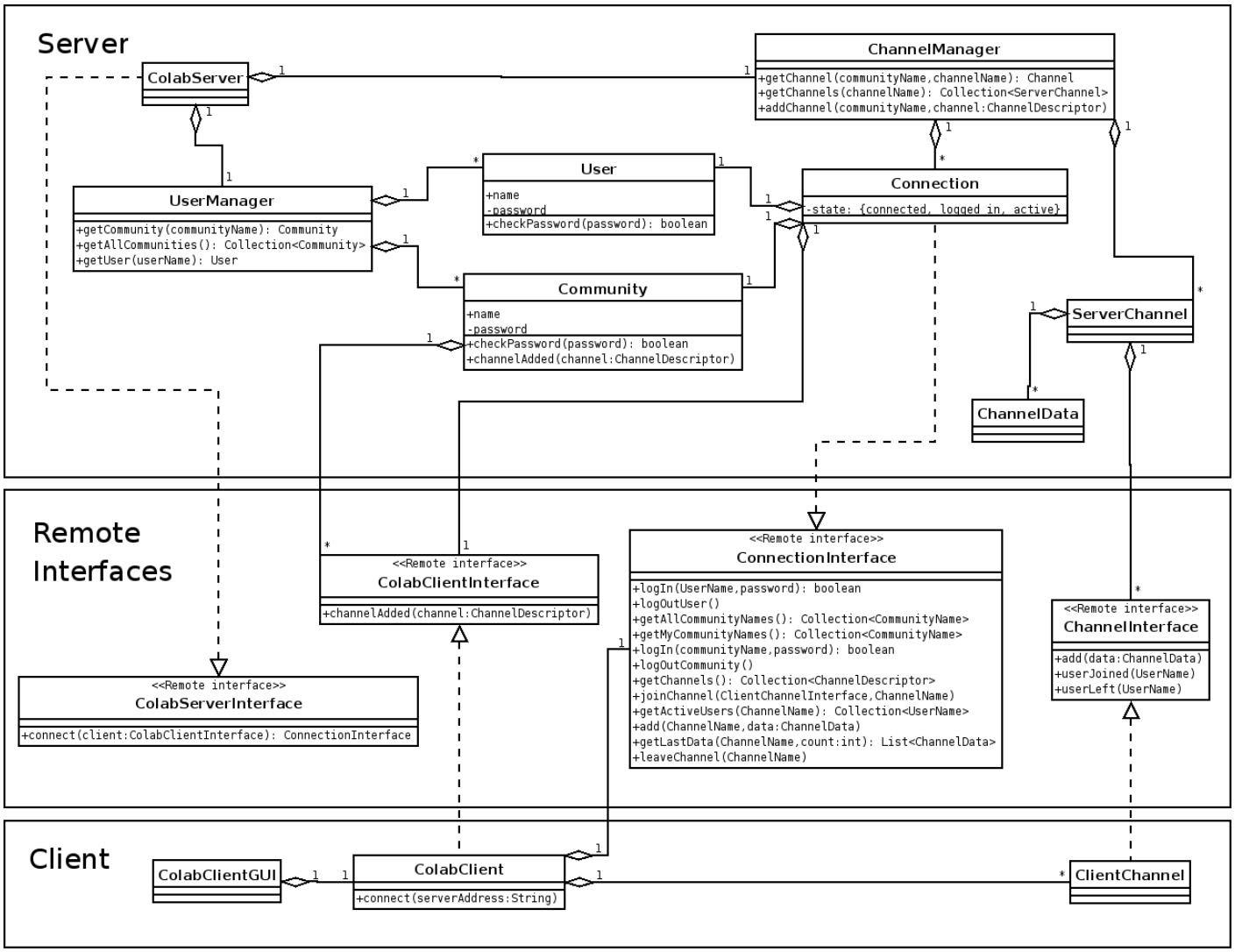
UserSet

A simple UserStore which stores all users in a set.

IdentitySet<UserName, User> users
: All of the users that exist.

4 Design UML

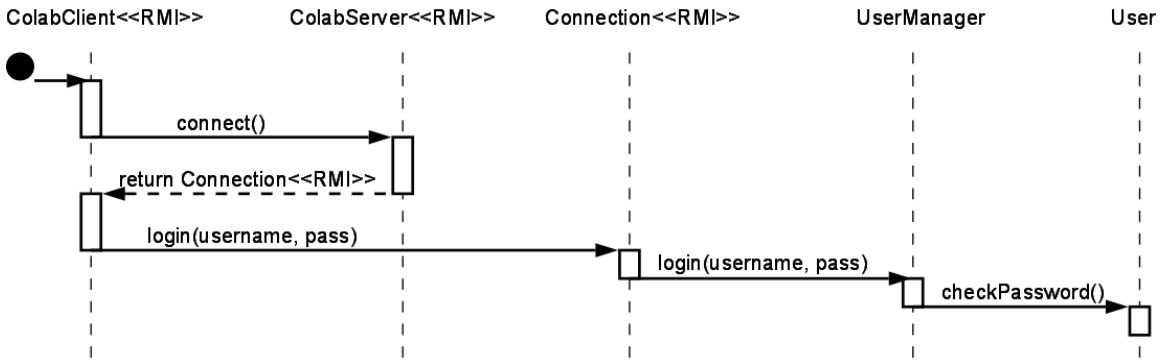
4.1 Class Diagram



4.2 Sequence Diagrams

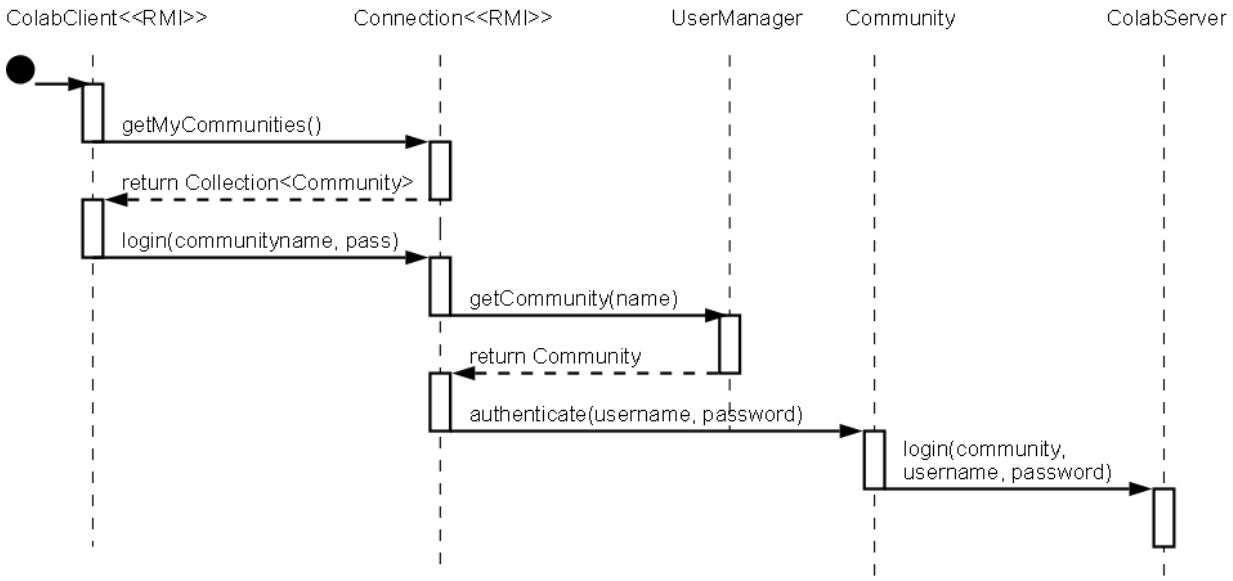
4.2.1 User Login

Client connects to the server and logs in as a user.



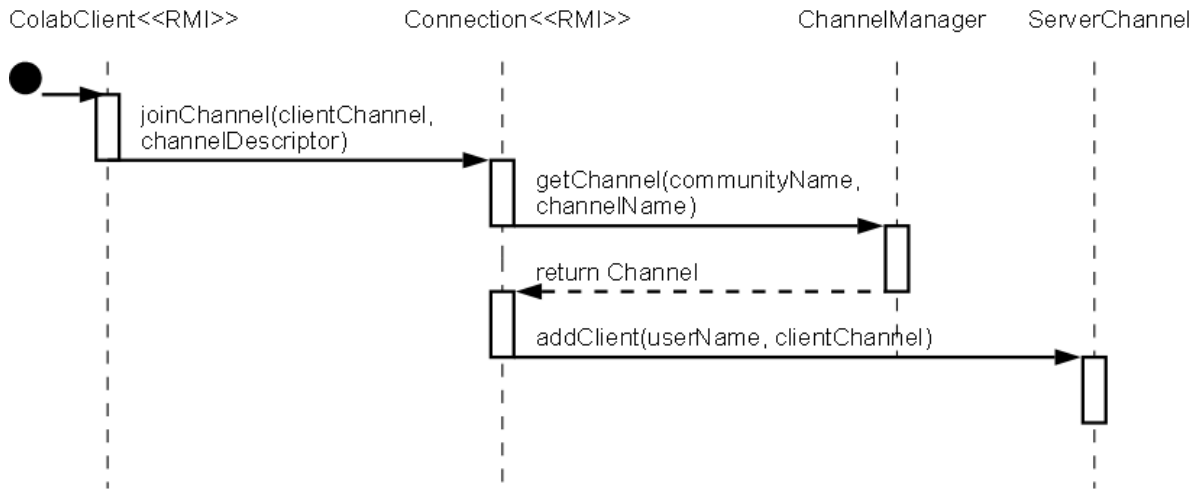
4.2.2 Community Login

After user login, the user logs into a community.



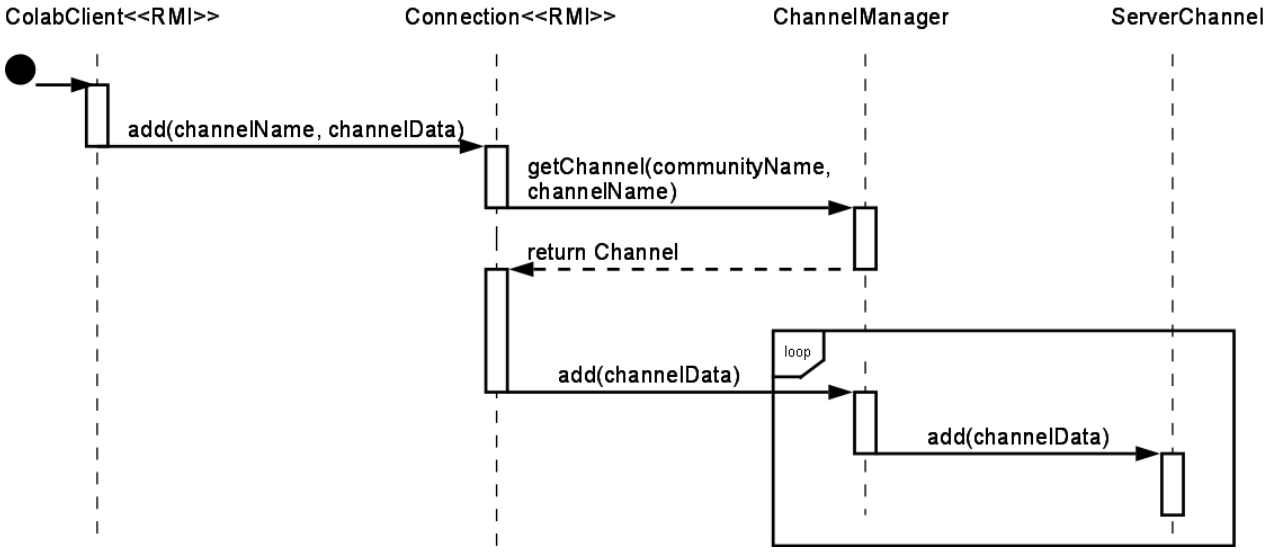
4.2.3 Joining a Channel

The user joins a channel so that it will receive channel data as it is added.



4.2.4 Sending Data

A client sends data to a channel, and the server forwards it to all other channel members.



5 Data Design

To persistently save and restore the state of the server, channel and user data is stored in text files. It is also possible to run the server without saving any data, for testing or demonstration purposes.

5.1 Data format

The data format is a simplified version of XML. This format was chosen so that the data can be edited, easily read and modified outside of the CoLab application if necessary.

A full, true XML implementation is not used because the scope of this project does not require all of the features of XML. One significant way in which the CoLab format differs is that it does not require a document to have a single root node. This was chosen for efficiency so that as data elements are rapidly added to a channel, they may simply be appended to the end of the file.

5.2 User data

The users file contains a list of users and their authentication information.

Example users file:

```
<User password="ephnfsiagugqenls" name="Chris"></User>
<User password="amlmcsqjlupyfdqe" name="Pamela"></User>
<User password="rhfideserbztdfth" name="Matt"></User>
<User password="gdsnkrzhvhrszbgf" name="Johannes"></User>
```

5.3 Community data

The communities file contains a list of communities, with the member list for each.

Example communities file:

```
<Community password="xkxlewzkacloxkwh" name="Group Seven">
   <UserName>Chris</UserName>
   <UserName>Johannes</UserName>
</Community>
<Community password="zbisaohiseoyfhcs" name="Team Awesome">
   <UserName>Johannes</UserName>
   <UserName>Matt</UserName>
   <UserName>Pamela</UserName>
</Community>
```

5.4 Channel data

Each Channel object, representing a chatroom, a document editor, or a whiteboard, is self-contained and holds all data necessary to re-create the exact state of that channel at a given moment. The ChannelManager is responsible for designating where on the filesystem a channel may store its data, and the individual Channels take care of serializing their data to that location.

6 Interface Specifications

6.1 Client-server Interface

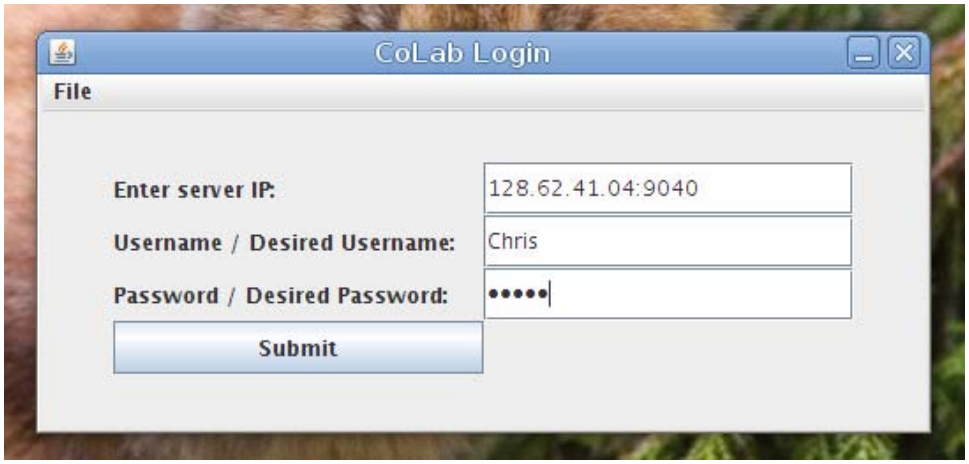
Colab's client-server communication is supported entirely by distributed objects using Remote Method Invocation (RMI).

The server application binds a single instance of ColabServer to the server's RMI registry. This uses a port which can be specified by a parameter when launching the server.

The client application remotely invokes the ColabServer.connect(), which returns a Connection object. The Connection, via its remote interface ConnectionInterface, is used for all subsequent requests to the server. The Connection class keeps track of the state of the connection and knows which user has logged into it. Each client communicates only with its own remote object for security, because an object cannot trust remote invokations unless it knows which user "owns" (has a reference to) it. The ConnectionInterface, after a single authentication, can be sure that any remote invocations are coming from the user who logged in.

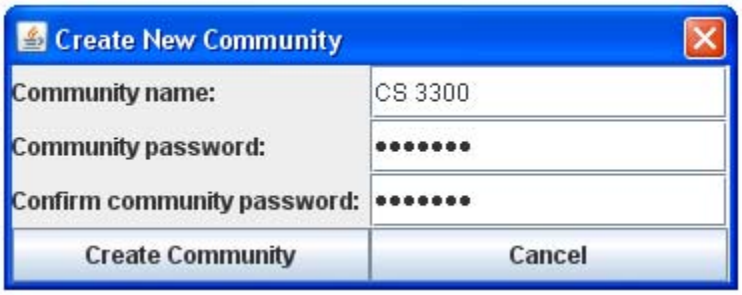
6.2 User Interface

6.2.1 User Login

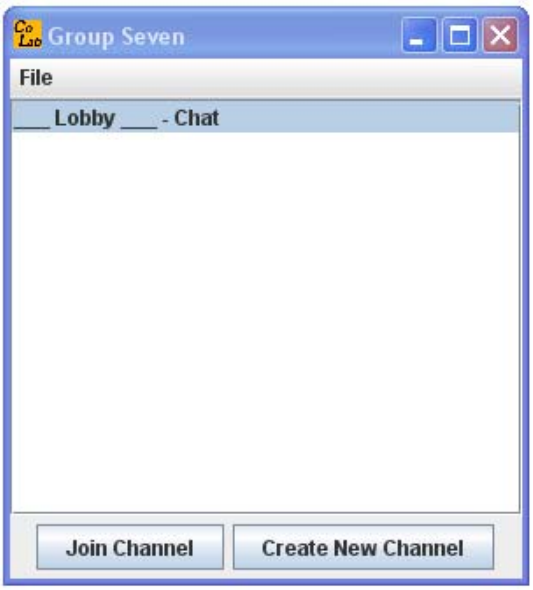


6.2.2 Community Login

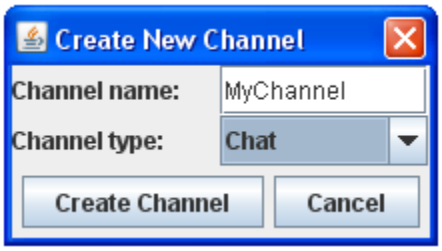
6.2.3 Creating a New Community



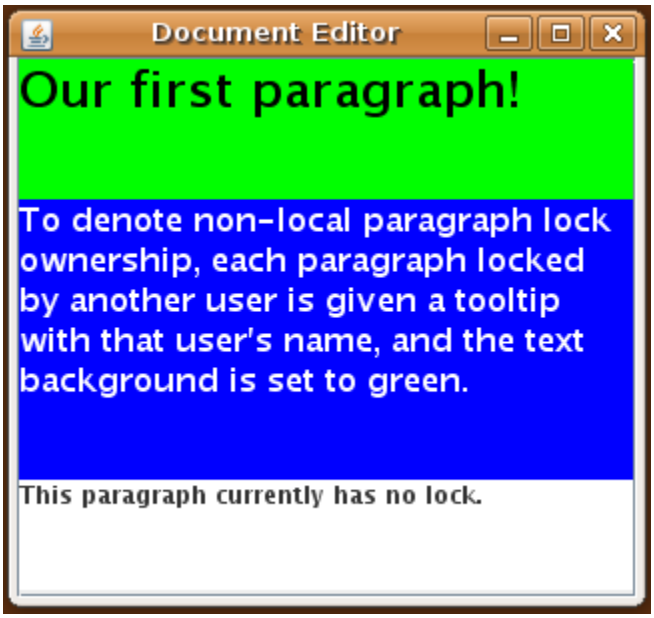
6.2.4 Channel List and Chat Channel



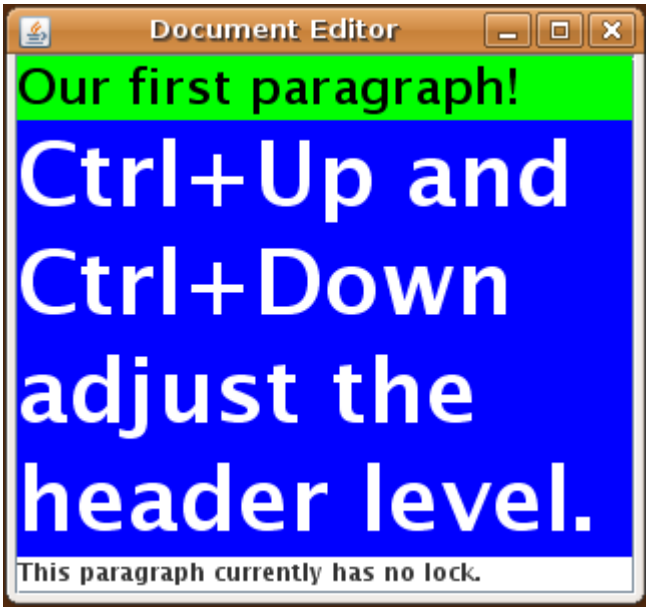
6.2.5 Creating a New Channel



6.2.6 Document Channel



6.2.7 Document Channel with Header Styles



7 Demo Plan

7.1 Demo Overview

D2 will consist of two executable deliverables, the CoLab server and the CoLab client. In addition to the functionality described below that will be displayed during the demo, the deliverables for D2 will also include the channel and community framework used by our client and server. However, the framework has no component that can be demonstrated independently from the server and client.

7.2 CoLab Server

The CoLab server creates a new server using our custom-built application framework and waits for incoming connections. The server is designed to run from a command line. There is no GUI, although the server will print a message when it has finished initializing successfully. The server will run forever until the process is manually terminated.

7.3 CoLab Client

7.3.1 User Creation and Login

The CoLab client application presents a GUI to the user prompting for a server IP, username and password. The client program connects to the CoLab server using RMI and authenticates whether the username/password combination is correct. Since the demo will run without any pre-populated users or communities, the client will prompt the user if he wants to create a new user with the login information.

7.3.2 Community Creation and Login

The client will then display an empty list of communities. Again, the user will need to create a community; the list will then be refreshed with the new community appearing pre-selected in the list. The user selects a community from the drop-down list and clicks the button to select and join that community.

A second user will also connect to the server and create a user name. Since he is not a member of any communities, he will have to select the check box to view all communities. The community that the first user created will then show in the list; if this second user selects it, he will be prompted for a password in order to join. An erroneous password will spawn an error dialog, but a correct password will open the community's channel manager.

7.3.3 Channel Creation and Joining

The CoLab client presents a new GUI representing all the community's channels. Every community has a lobby chat channel by default, so the list will contain the lobby chat

where the user has a window with chat messages and an input box. By typing text in the input box and clicking a button to send, the user's text is sent to the server, then relayed to all CoLab clients connected to this community's lobby.

The first user will create a new channel in this community by clicking the “Create New Channel” button. A GUI will be presented asking for channel information. When the user clicks “Create Channel”, the list of channels will be refreshed with the newly added channel highlighted.

The second user will have his channel list refreshed automatically and will be able to join the new channel.

Using two machines, we can then demonstrate how chat takes place over a network as well as the how the document editor works. Either user can create new chat or document editor channels, and after user logouts, the server saves the users, communities, and channels information.

7.3.4 Document Editor

One user will create a new document channel and both users will join it.

The first user will begin editing a paragraph. The paragraph’s background will turn blue to indicate his client acquires a lock on that paragraph to prevent editing conflicts.

The second user will see that the paragraph currently being edited by the first user has turned green, indicating another client has locked it.

While the first user edits the paragraph, the client will periodically send all changes to other users in the channel, so the second user will see the updated contents appear on his screen.

The first user will switch his view into *revision history* mode, which provides a navigation mechanism for selecting prior revisions of the document and viewing them in an uneditable panel.

The user will export the contents of the document as an html file by using the menu option.

7.3.5 Persistent Data Storage

The server initially runs with no user information, channels or communities. As users create accounts, channels and communities, the server will store data in a custom format similar to XML and described elsewhere in this document. After demonstrating channel and community creation, the server will be restarted and it will reload all saved data from disk. The server will save usernames and passwords, communities, the names and types of channels in each community and the members of a community.

8 Test Plan

The test plan will cover unit testing and integration testing through a combination of whitebox and blackbox techniques. Whitebox methods will be used to ensure the tests have complete statement coverage of the code, and blackbox methods will be used to check that the tests produce the correct values. Both unit tests and integration tests will be conducted by all developers.

8.1 Unit Testing

JUnit will be used to faciliate the running of unit test cases. Unit test cases will be written for functionality that can be tested in such a manner, but some components depend heavily on others and must be checked using integration testing. In addition, the front-end GUI cannot be easily tested using unit tests and will be verified through integrated usability tests by the developers.

8.2 Integration Testing

System integration will be checked in two ways. First, test cases will be written which rely on a variety of components and will require correct values from each separate component in order to pass. Second, developers will manually verify that the application is working by running the GUI and looking for reasonable values at each step.

8.3 Test Details and Results

Test Description	Passed?
Login Window behavior	
Existing username and password pairs succeed.	√
Non-existent username and password pairs generate a prompt to either create a new user account or enter login information again.	√
User can see a menu bar with “File” as title and are given the open to quit.	√
Server connection	√
If a server IP is entered correctly, the client connects.	√
If the server IP is incorrect, an error window suggests entering the IP address again.	√
Community manager behavior	

The current user's community's names are displayed in the drop down menu in the community manager window.	√
The option to view all communities on the server can be selected with any additional communities then showing in the menu.	√
The user can join a community by clicking on a selection in the menu and clicking the "Join Community" button.	√
If the user is not a member of the selected community, he can become one by entering into a window prompt the community's password.	√
A user can create a new community by clicking the "Create New Community" button and entering the community's information in a new community window prompt.	√
If the user enters a name of an already existing community, an error window alerts him to try another name.	√
User can see a menu bar with "File" as title and are given the open to logout and quit.	√
Channel manager behavior	
All channels in the current community are displayed in a list with the appropriate names and channel type.	√
The user can open any of the channels by double clicking on the channel in the list or by selecting the channel in the list and clicking the "Join Channel" button.	√
A new channel can be created by clicking the "Create New Channel" button and entering the appropriate channel information and choosing the channel type in a new window; if the user enters a name of an already existing channel, an error window alerts him to try another name.	√
User can see a menu bar with "File" as title and are given the open to change communities, logout, and quit.	√
Chatting behavior	
All users in chat channel names display in right hand pane.	√
User can type in field at bottom and hit enter to send his text to all users and have it display in the main pane in the form "<Username>: <text>."	√
User can see a menu bar with "File" and "Options" as titles and are given the open to show time stamps, export chats and exit.	√
If the user enables timestamps, new messages in the chat window will appear with a timestamp pre-pended.	√
User can select "Exit" from the File menu to leave the channel.	√

User can select Revision Mode from the File menu to open a new window with the revision history.	√
User can select the export menu option to open a new window where the chat text can be saved to a file.	√
Changing communities behavior: if the user selects “Change Communities” from the menu bar, the client logs the user out of the current community and the UI displays the Choose Community window.	√
Logging out behavior: if the user selects “Log out” from the menu bar, the client logs the user out of the current community, the client logs the user off the connection, and the UI displays to the Login window.	√
Quitting behavior: if the user selects “Quit” from the any menu bar, the client logs the user out of the current community, the client logs the user off the connection, the client terminates the connection to the server, the program exits, and all program windows close.	√
Document behavior	
User can insert a new paragraph into the document.	√
User can delete a paragraph from the document by pressing a specified key combination.	√
The client will periodically send the updated text to all users in this channel as the user edits a paragraph.	√
When a user edits a paragraph locally, the paragraph is locked and will be shown with a blue background.	√
When another user in the channel is editing a paragraph, the paragraph is locked and will be shown with a green background.	√
When a paragraph is not being edited by any user, the paragraph is unlocked and will be shown with a white background.	√
User can add, remove or edit text in any unlocked paragraph.	√
User can select the export menu option to open a new window where the document contents can be saved to a file.	√
User can select Revision Mode from the File menu to open a new window with the document history in read-only form.	√

9 Requirements Specification

9.1 Requirements Overview

The D3 plan had the following original goals:

- ●Whiteboard channel protocol is functional.
- ●User can create and use a whiteboard channel.
- ●Moderators can modify community preferences (adding passwords, changing blacklist, etc.).

Because the D2 requirements were met, the D3 plan remains the same as the one proposed in the D1 document.

9.2 Updated Delivery Plan

9.2.1 Delivery 1: February 26

- ●Client can enter user name and password and login
- ●Server can authenticate incoming clients
- ●Chat channel protocol is functional
- ●Client automatically opens a single chat channel upon login
- ●GUI available for client login, lobby chat

9.2.2 Delivery 2: April 1

- ●User can create and use a document channel
- ●User can create user accounts and communities
- ●User can create channels, join multiple channels concurrently
- ●Server stores all data persistently
- ●Document channel protocol is functional

9.2.3 Delivery 3: April 21

A whiteboard represents a two-dimensional raster image.

Drawing

Users can select a color, and modify an image using several basic drawing tools such. A pen tool allows freeform drawing. A shape tool allows easy addition of shapes such as rectangles, ovals, and straight lines.

A *clear* button is available to wipe the entire canvas, removing all changes. All previous changes will still be available in the revision history.

Revision History

A user can switch into *revision history* mode, similar to that of the document channel protocol. Revision history can be used to view older versions of the drawing.

Exporting

A flattened version of the image can be exported as a png file. From the revision history view, any revision can be exported in the same manner.

Community Moderators

The creator of a community is treated as a *moderator*. Moderators have access to a preferences panel which allows them to administer the community through features such as changing the community password, creating a blacklist (a list of members who are not allowed to join the community) or removing members from the member list.