

# **CoLab**

## **Deliverable Three**

### **CS 3300**

#### **Group Seven**

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

April 21, 2008

## Summary of Document Changes From D2

<b>Product Overview</b>	No change
<b>Delivery Description</b>	Covers D3 delivered components
<b>Design Overview</b>	No change
<b>Design Detail</b>	No change
<b>Design UML</b>	No change
<b>Data Design</b>	No change
<b>Interface Specification</b>	User interface for whiteboard channel specified
<b>Demo Plan</b>	Expanded to include functionality added in D3
<b>Test Plan</b>	System test cases added for new functionality
<b>Requirements Specification</b>	Included future plans for enhancement

# Contents

<b>1 Product Overview.....</b>	<b>7</b>
1.1 Product Overview.....	7
1.2 Deliverables.....	7
1.3 Definitions.....	7
<b>2 Delivery Description.....</b>	<b>8</b>
2.1 Framework.....	8
2.1.1 User Accounts.....	8
2.1.2 Communities.....	8
2.1.3 Channels.....	8
2.2 Chat Channel Protocol.....	9
2.3 Document Channel Protocol.....	9
2.4 Whiteboard Channel Protocol.....	9
<b>3 Design Overview.....</b>	<b>11</b>
<b>4 Design Detail.....</b>	<b>12</b>
4.1 colab.client.....	12
ClientChannel.....	12
ClientChatChannel.....	12
ClientDocumentChannel.....	12
ClientWhiteboardChannel.....	12
ColabClient.....	13
4.2 colab.common.....	14
4.2.1 colab.common.channel.....	14
ChannelData.....	14
ChannelDataIdentifier.....	14
ChannelDataSet.....	14
ChannelDescriptor.....	15
ChatChannelData.....	15
DocumentChannelData.....	15
WhiteboardChannelData.....	15
4.2.2 colab.common.remote.client.....	15
ChannelRemote (Interface).....	15
ColabClientRemote (Interface).....	16

4.2.3 colab.common.remote.server.....	16
ColabServerRemote (Interface).....	16
ConnectionRemote (Interface).....	16
4.2.4 colab.common.util.....	17
FileUtils.....	17
StringUtils.....	17
4.3 colab.server.....	18
ColabServer.....	18
4.3.1 colab.server.channel.....	19
ChannelConnection.....	19
ChannelManager.....	19
ServerChannel.....	19
ServerChatChannel.....	19
ServerDocumentChannel.....	20
ServerWhiteboardChannel.....	20
4.3.2 colab.server.connection.....	20
Connection.....	20
ConnectionIdentifier.....	21
4.3.3 colab.server.event.....	21
DisconnectEvent.....	21
DisconnectListener (Interface).....	21
4.3.4 colab.server.file.....	21
ChannelFile.....	21
CommunityFile.....	21
UserFile.....	22
4.3.5 colab.server.user.....	22
Community.....	22
CommunitySet.....	23
Password.....	23
User.....	23
UserManager.....	23
UserSet.....	24
<b>5 Design UML.....</b>	<b>25</b>
5.1 Class Diagram.....	25
5.2 Sequence Diagrams.....	26
5.2.1 User Login.....	26
5.2.2 Community Login.....	26
5.2.3 Joining a Channel.....	27
5.2.4 Sending Data.....	27
<b>6 Data Design.....</b>	<b>28</b>

6.1	<i>Data format</i>	28
6.2	<i>User data</i>	28
6.3	<i>Community data</i>	29
6.4	<i>Channel data</i>	29
<b>7</b>	<b>Interface Specifications</b>	<b>30</b>
7.1	<i>Client-server Interface</i>	30
7.2	<i>User Interface</i>	30
7.2.1	User Login	30
7.2.2	Community Login	31
7.2.3	Creating a New Community	31
7.2.4	Community Administration Menu	32
7.2.5	Managing Community Members	32
7.2.6	Changing Community Password	33
7.2.7	Channel List and Chat Channel	34
7.2.8	Creating a New Channel	34
7.2.9	Document Channel	35
7.2.10	Document Channel with Header Styles	35
7.2.11	Whiteboard Channel	36
7.2.12	Whiteboard Channel with Color in Image	37
<b>8</b>	<b>Demo Plan</b>	<b>38</b>
8.1	<i>Demo Overview</i>	38
8.2	<i>CoLab Server</i>	38
8.3	<i>CoLab Client</i>	38
8.3.1	User Creation and Login	38
8.3.2	Community Creation and Login	38
8.3.3	Channel Creation and Joining	39
8.3.4	Whiteboard Editor	39
8.3.5	Community Administration	39
8.3.6	Persistent Data Storage	40
<b>9</b>	<b>Test Plan</b>	<b>41</b>
9.1	<i>Unit Testing</i>	41
9.2	<i>Integration Testing</i>	41
9.3	<i>Test Details and Results</i>	41
<b>10</b>	<b>Requirements Specification</b>	<b>45</b>

<i>10.1 Requirements Overview.....</i>	<i>45</i>
<i>10.2 Updated Delivery Plan.....</i>	<i>45</i>
10.2.1 Delivery 1: February 26.....	45
10.2.2 Delivery 2: April 1.....	45
10.2.3 Delivery 3: April 21.....	45
Drawing.....	46
Revision History.....	46
Exporting.....	46
Community Moderators.....	46
10.2.4 Future Goals and Enhancements.....	46

# 1 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).

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. 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 moderator of a community is able to change the password for joining the community and remove members of the community.

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 user who creates the community becomes the moderator of that community.

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 that can be used by any channel to display a list of 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.

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

## 2.4 Whiteboard Channel Protocol

Whiteboards are divided into entities called *layers*. At most one user can modify a layer at one time.

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

A lock held by another user is designated by the discoloration of a layer label in the layer selection panel. Similarly, another color signifies a lock held by one's own client.

The view for a whiteboard consists of several components. On the left, the user can select one of several drawing tools (e.g. pencil, rectangle, and ellipse tools). On the bottom, the user can select the pen color. On the right is the layer selection panel, where the user chooses a layer to modify. Finally, the center of the whiteboard is an editable region in which the user can paint.

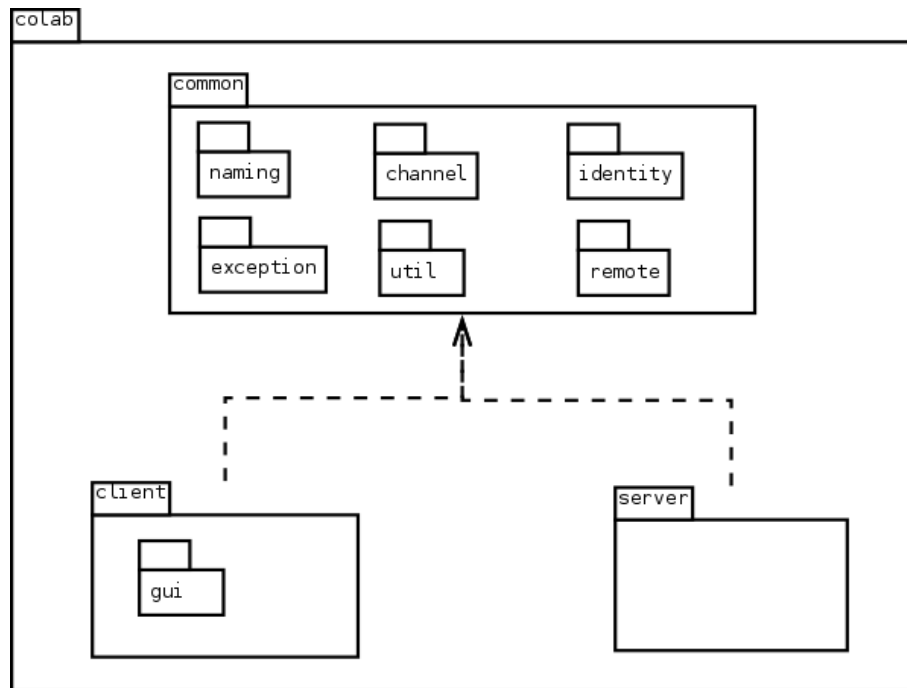
A user can switch his view into *revision history* mode, which provides a navigation mechanism for selecting prior revisions of the whiteboard and viewing them in an uneditable panel.

The entire contents of the whiteboard can be exported as an image 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



## 4 Design Detail

### 4.1 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 the channel's Document

    Notify all listeners (to update the UI)

#### **ClientWhiteboardChannel**

A ClientChannel that handles whiteboard-protocol channels.

Extends: ClientChannel

void add(ChannelData)

    Add a piece of whiteboard channel data to the channel's Whiteboard

    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

## 4.2 colab.common

### 4.2.1 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 data (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.

### **DocumentChannelData**

Represents a message posted to a document channel.

DocumentChannelDataType getType()

Returns the data type of this object.

void apply(final Document doc)

Applies this data to a document.

### **WhiteboardChannelData**

Represents a message posted to a whiteboard channel.

void apply(final Whiteboard board)

Applies this data to a whiteboard.

## **4.2.2 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.

## **4.2.3 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 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 getUserUsername()`

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.

#### **4.2.4 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 repeatedly and returns the result.

## 4.3 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

### 4.3.1 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**

ServerDocumentChannel extends ServerChannel and deals with document channels.

ChannelDataStore<DocumentChannelData> revisions  
The channel data.

### **ServerWhiteboardChannel**

ServerWhiteboardChannel extends ServerChannel and deals with whiteboard channels.

ChannelDataStore<WhiteboardChannelData> revisions  
The channel data.

## **4.3.2 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

### **4.3.3 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.

### **4.3.4 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.

## **4.3.5 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.

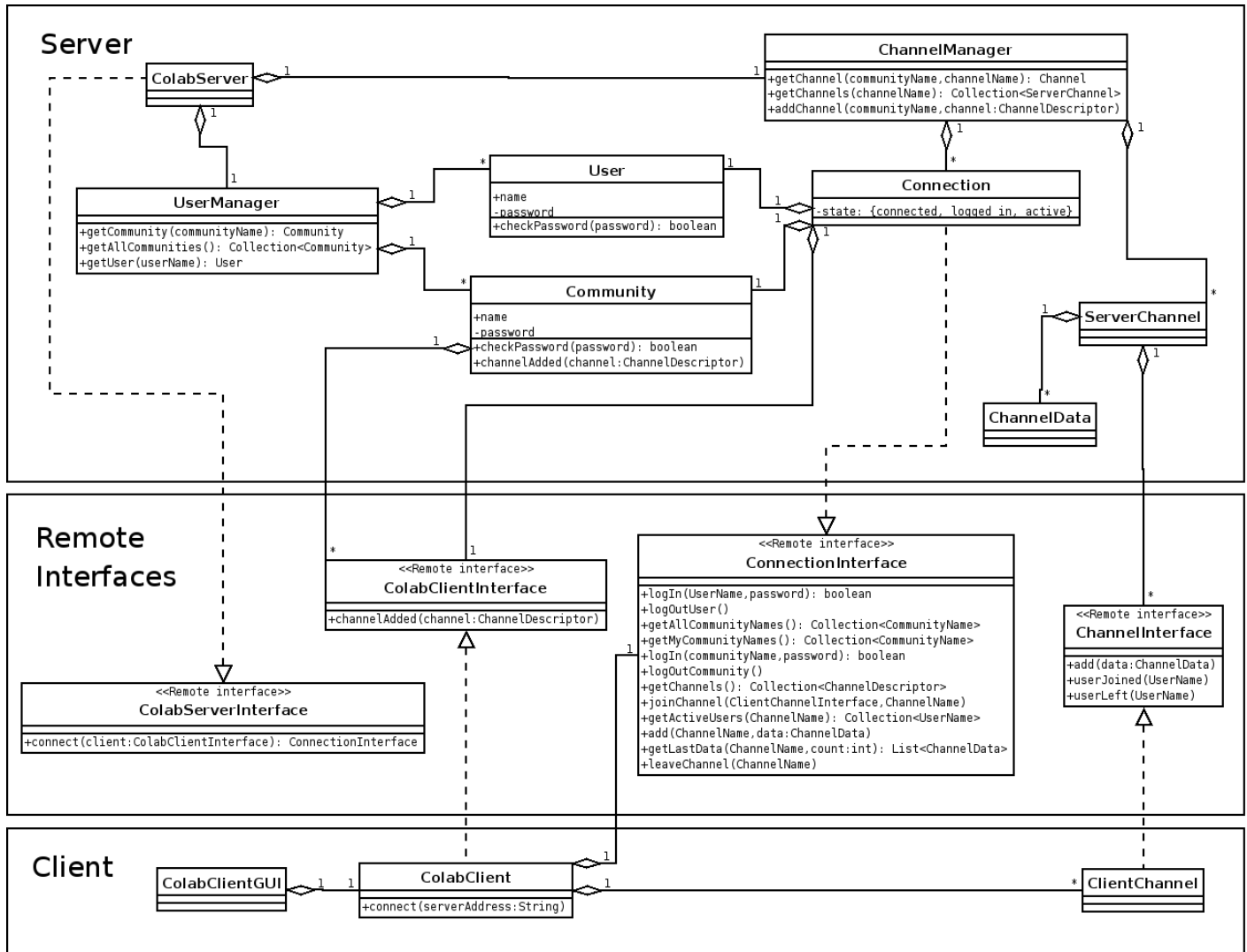
IdentitySet<UserName, User> users

All of the users that exist.



## 5 Design UML

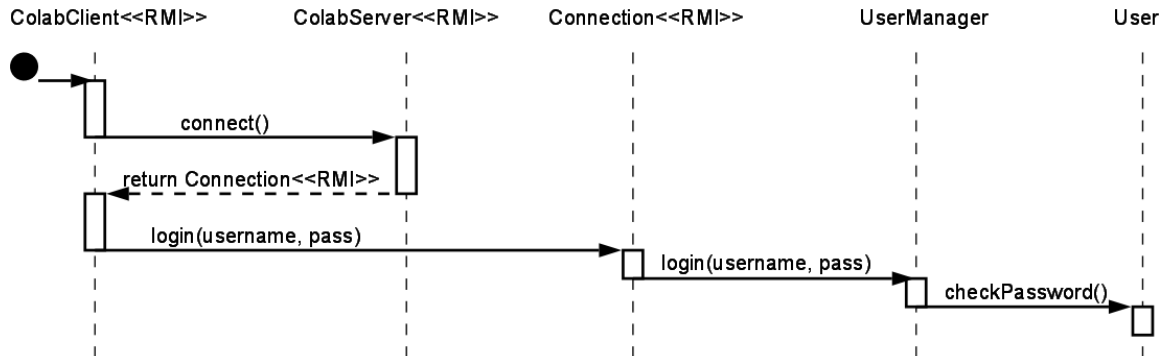
### 5.1 Class Diagram



## 5.2 Sequence Diagrams

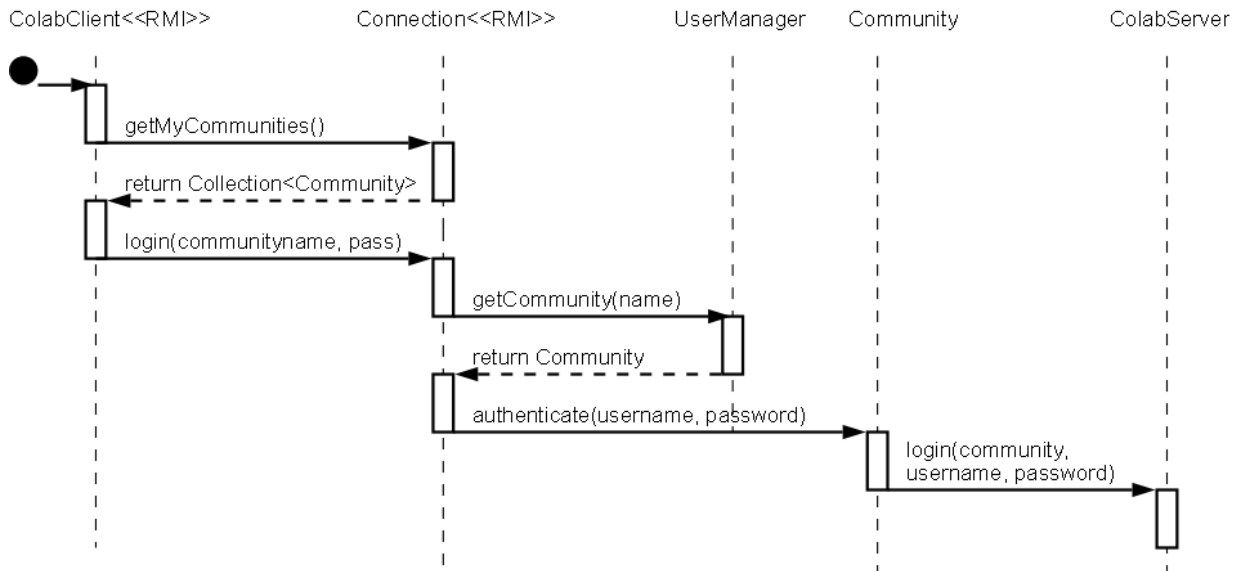
### 5.2.1 User Login

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



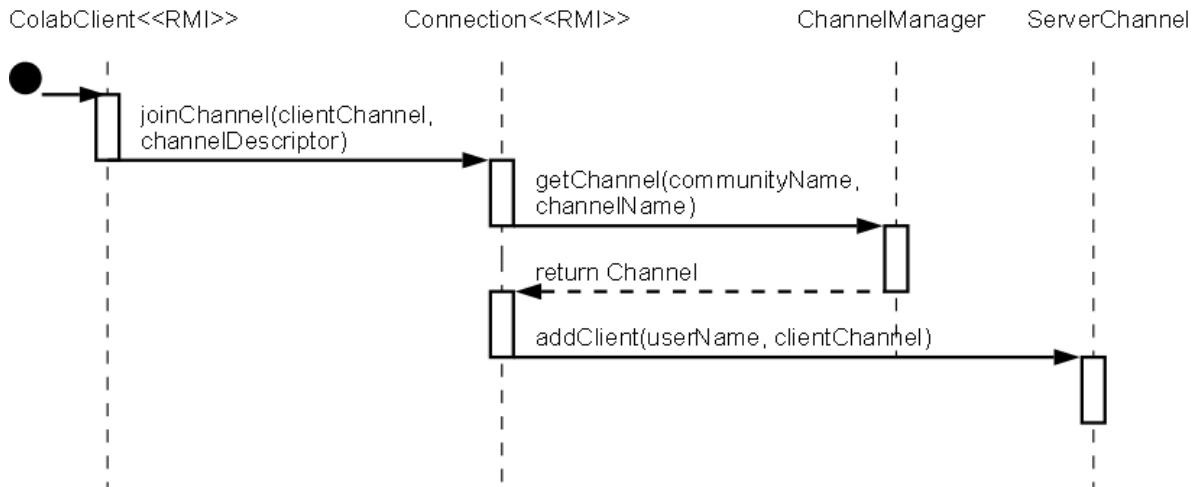
### 5.2.2 Community Login

After user login, the user logs into a community.



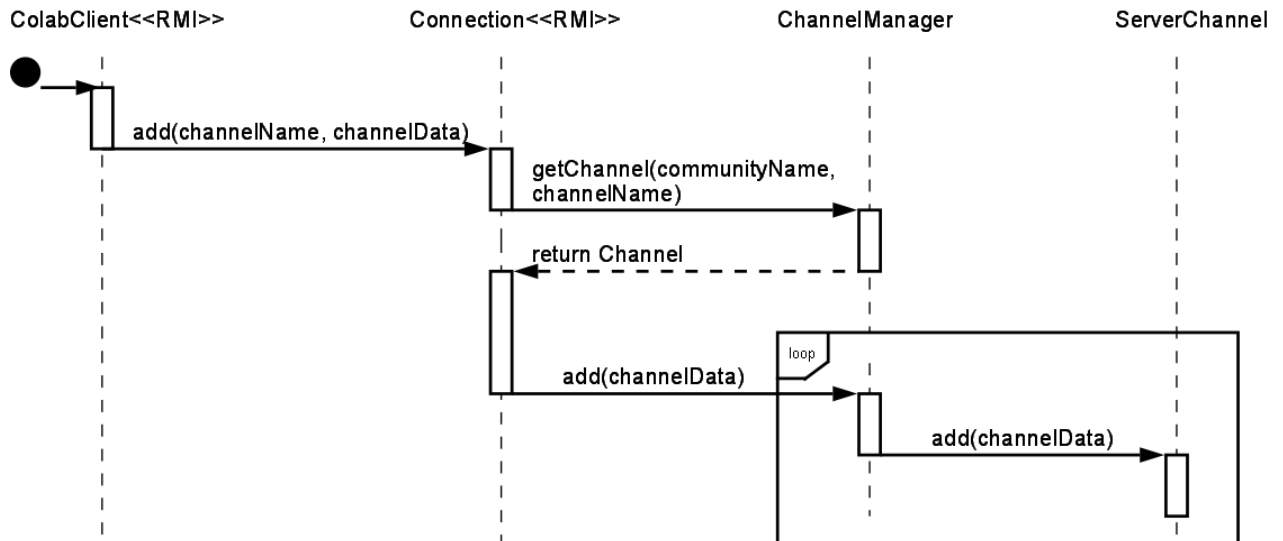
### 5.2.3 Joining a Channel

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



### 5.2.4 Sending Data

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



## 6 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.

### 6.1 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.

### 6.2 User data

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

Example users file:

```
<User password="ephnfsiaguggenls" name="Chris"></User>
<User password="amlmcsqjlupyfdqe" name="Pamela"></User>
<User password="rhfideserbztdfth" name="Matt"></User>
<User password="gdsnkrzhvhrszbfg" name="Johannes"></User>
```

## 6.3 Community data

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

Example communities file:

```
<Community password="xkxlewkacloxkwh" 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>
```

## 6.4 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.

Example channel file:

```
<ChatMessage creator="Johannes" time="1 Apr 2008 14:33:25.373"
  id="1">Hey man</ChatMessage>
<ChatMessage creator="Chris" time="1 Apr 2008 14:33:31.699"
  id="2">This program is awesome!!</ChatMessage>
<ChatMessage creator="Johannes" time="1 Apr 2008 14:33:35.982"
  id="3">I know!</ChatMessage>
```

## 7 Interface Specifications

### 7.1 Client-server Interface

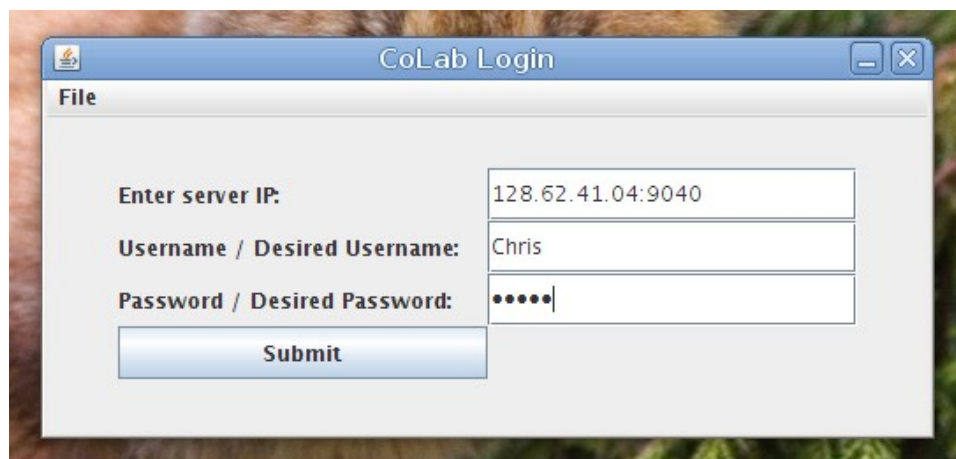
Colab's client-server communication is supported entirely by distributed objects using Remote Method Invocation (RMI).

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.

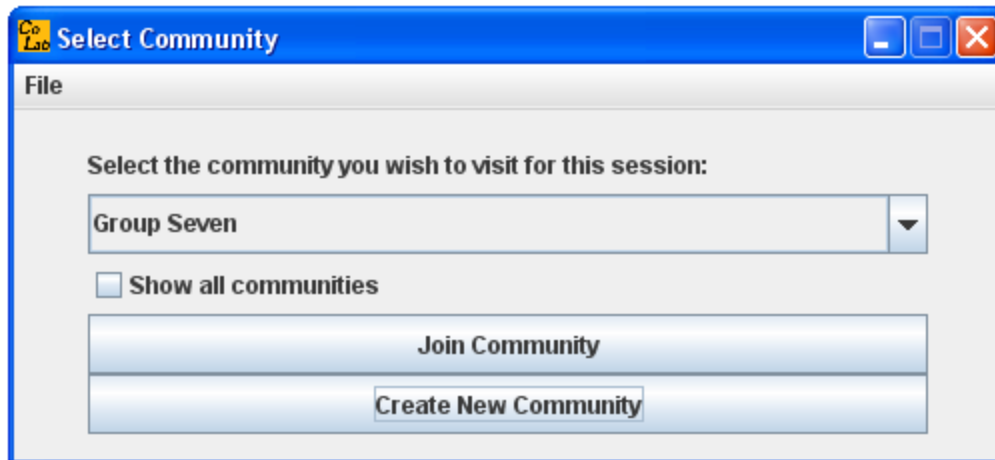
The client application remotely invokes the ColabServer.connect(), which returns a Connection object. The Connection, via its remote interface ConnectionInterface, is used for all subsequent requests to the server. The Connection class keeps track of the state of the connection and knows which user has logged into it. Each client communicates only with its own remote object for security, because an object cannot trust remote invocations 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.

### 7.2 User Interface

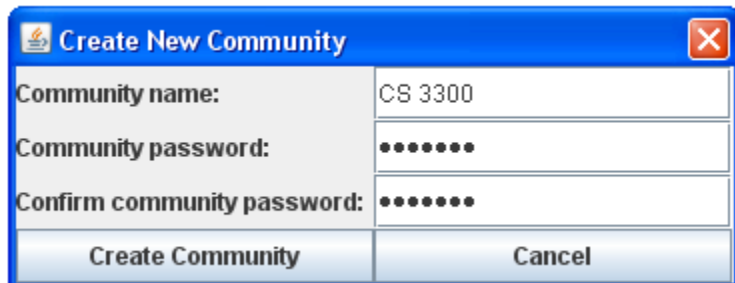
#### 7.2.1 User Login

A screenshot of a Java-style window titled "CoLab Login". The window has a standard title bar with minimize, maximize, and close buttons. Below the title bar is a menu bar with the word "File". The main content area contains three labels with corresponding text input fields: "Enter server IP:" with the value "128.62.41.04:9040", "Username / Desired Username:" with the value "Chris", and "Password / Desired Password:" with five dots. Below these fields is a "Submit" button. The window is set against a background image of a tree.

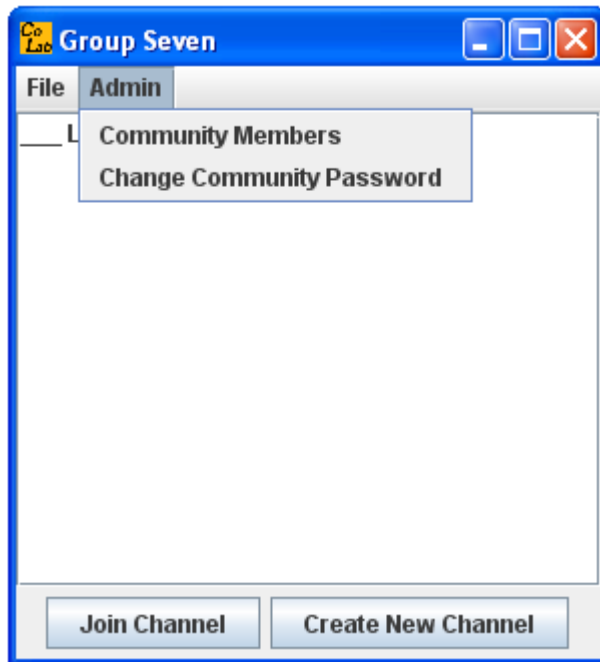
## 7.2.2 Community Login



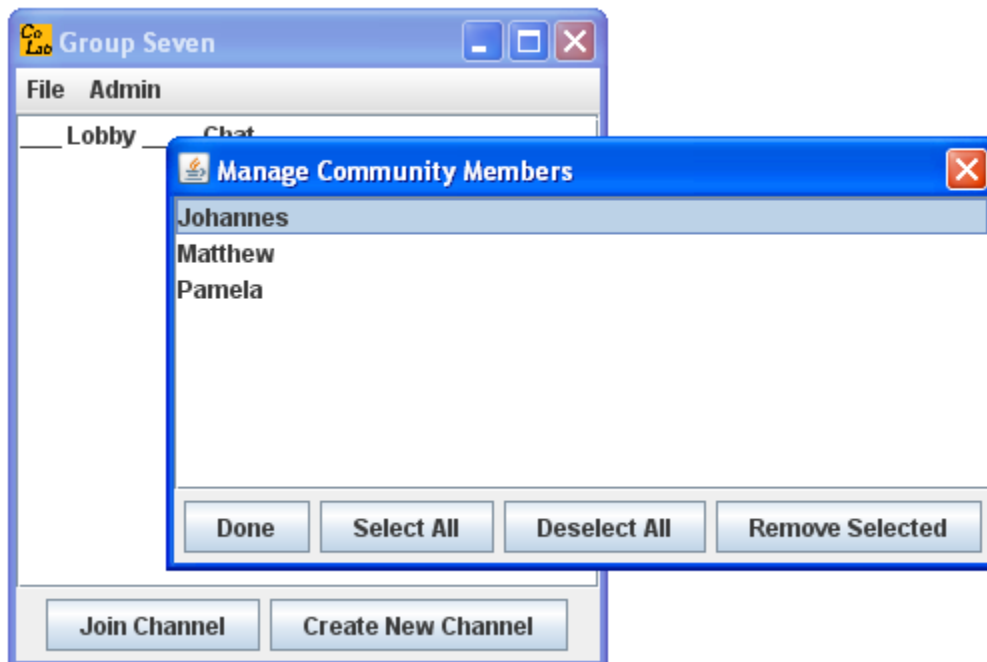
## 7.2.3 Creating a New Community



## 7.2.4 Community Administration Menu

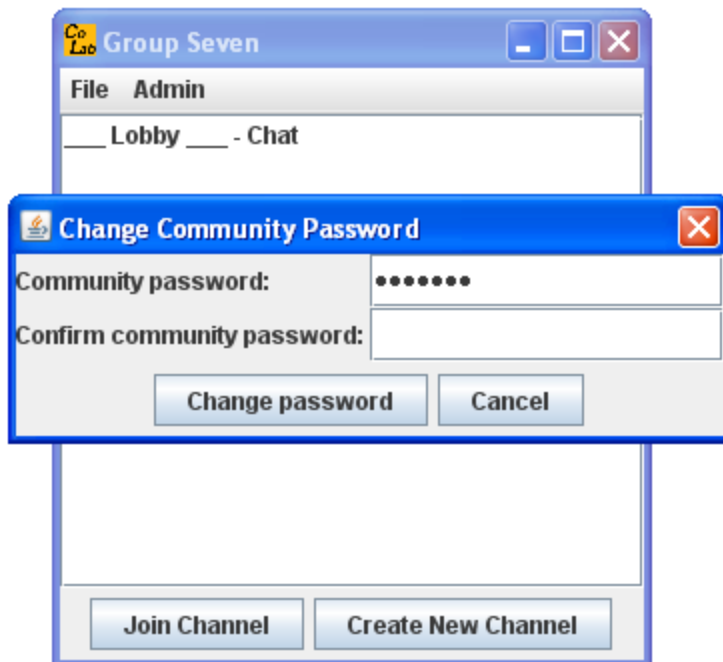


## 7.2.5 Managing Community Members

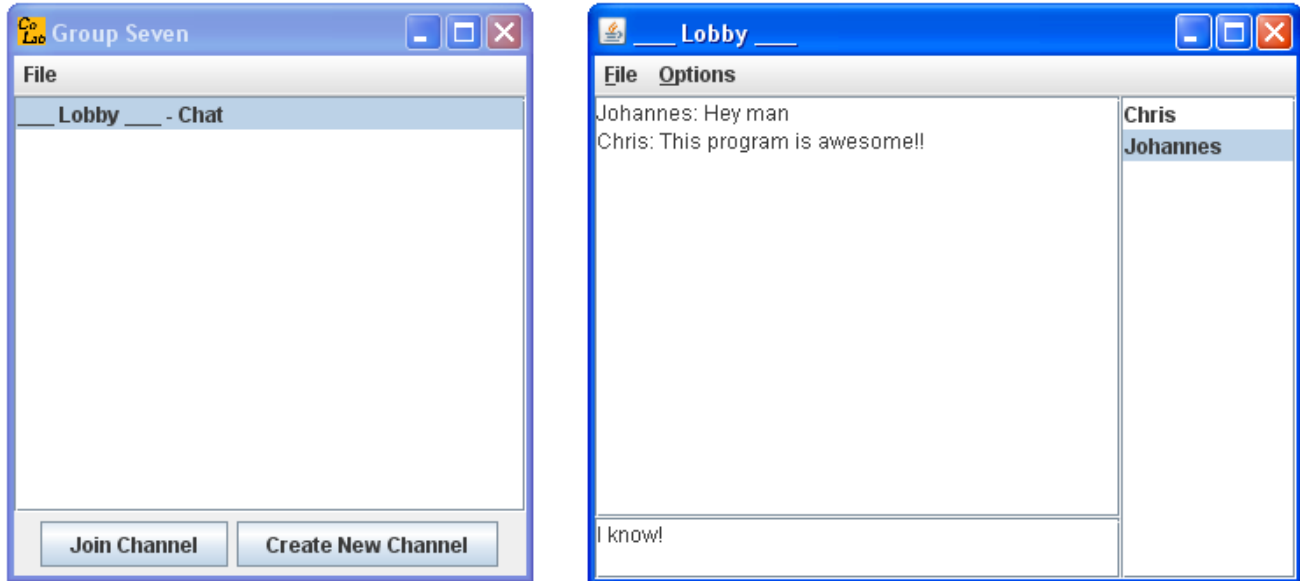




## 7.2.6 Changing Community Password



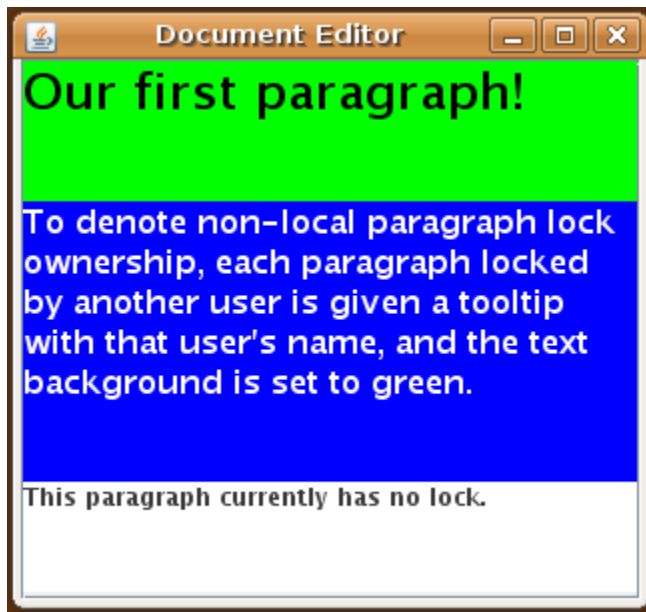
## 7.2.7 Channel List and Chat Channel



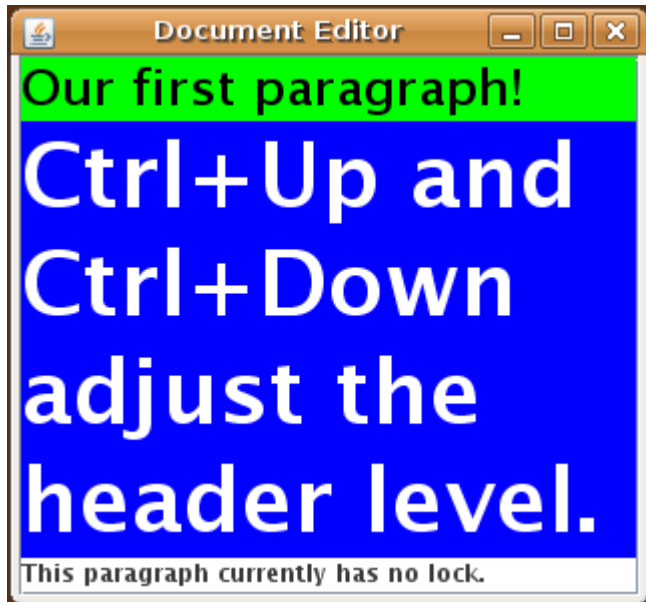
## 7.2.8 Creating a New Channel



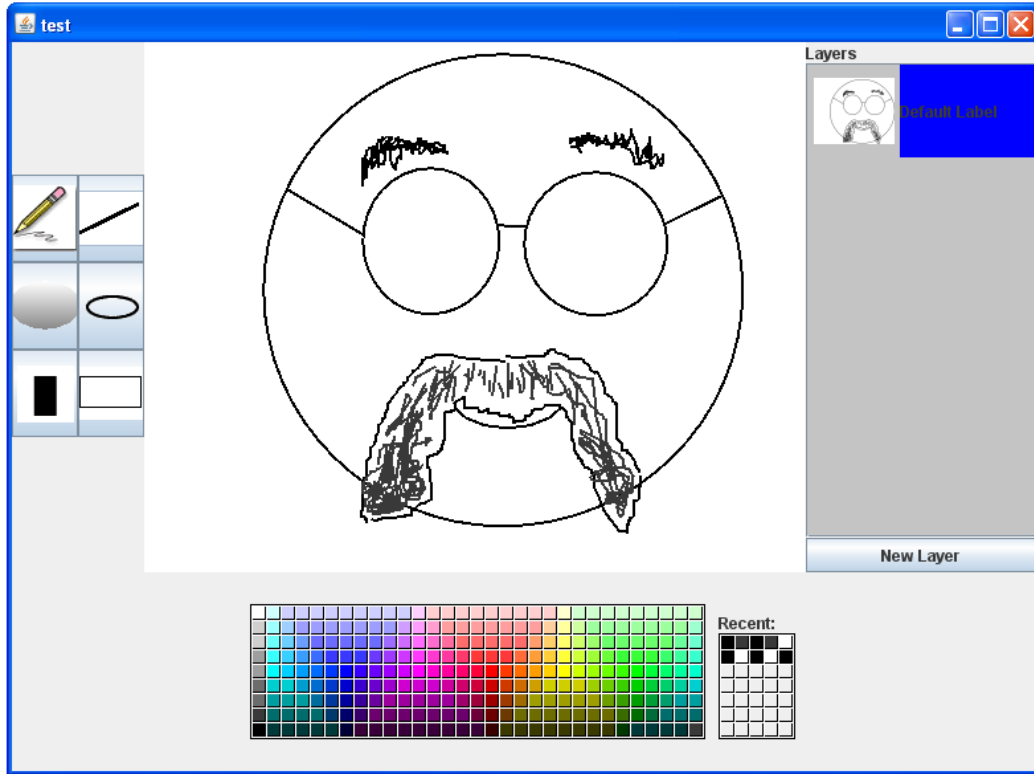
### 7.2.9 Document Channel



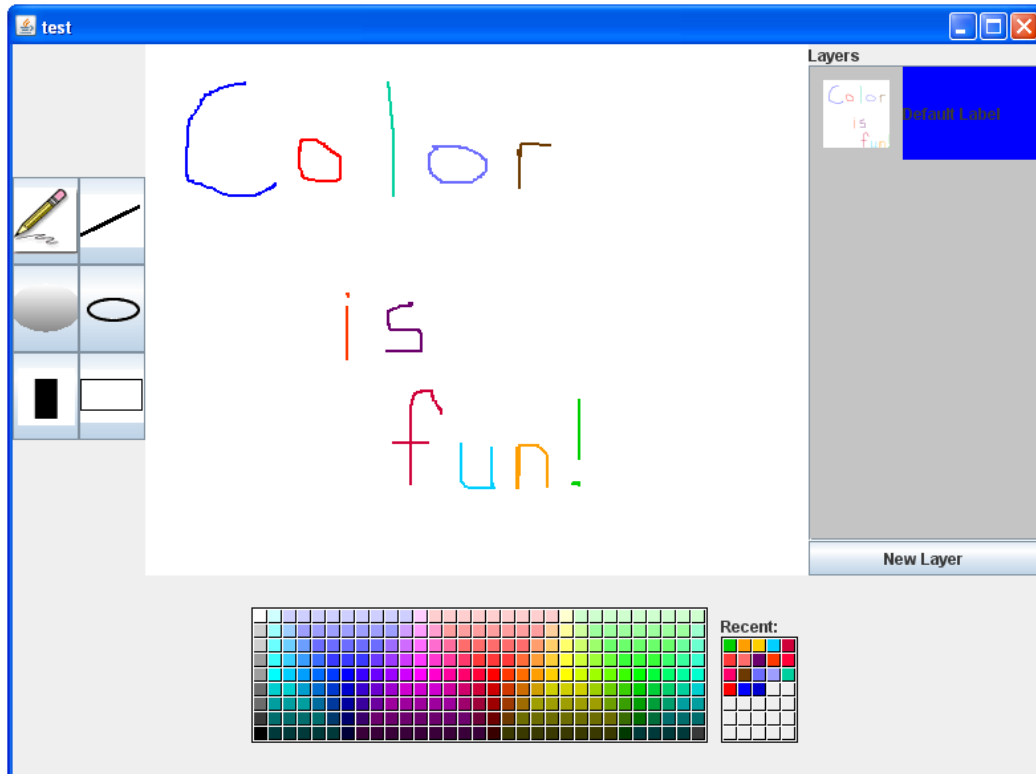
### 7.2.10 Document Channel with Header Styles



## 7.2.11 Whiteboard Channel



## 7.2.12 Whiteboard Channel with Color in Image



## 8 Demo Plan

### 8.1 Demo Overview

D2 consists 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 D3 also include the channel and community framework used by our client and server.

### 8.2 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.

### 8.3 CoLab Client

#### 8.3.1 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 server using RMI and authenticates using the username/password combination. Since the demo will begin 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.

#### 8.3.2 Community Creation and Login

The client will then display an empty list of communities. The user will 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. His joined community list is empty. To join a community, he will select the check box to view all communities of which he is not a member. The community that the first user created will then be shown in the list; the second user will select it, and will be prompted for a password in order to join. He will enter an erroneous password, spawning an error dialog. Using his correct password, he will exit the dialog, enter the password, and be shown his new community's channel window.

### 8.3.3 Channel Creation and Joining

The client presents a new window listing all the community's channels. Every community has a lobby chat channel by default; the list will initially contain the lobby channel.

The first user will create a new channel in this community by clicking the "Create New Channel" button. A window 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 will demonstrate shared use of a channel. D1 demonstrated a chat channel and D2 demonstrated the document editor; this demo will feature the whiteboard.

### 8.3.4 Whiteboard Editor

The first user will create a new whiteboard channel and both users will join it.

The first user will select the Pencil tool and select the default layer. The layer's background will turn blue to indicate his client has acquired a lock on that layer.

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

While the first user edits the drawing, the client will periodically send all changes to other users in the channel. 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 whiteboard and viewing them in an uneditable panel.

The user will export the contents of the whiteboard as an image file by using the menu option. We will open the exported image file to show that the contents saved correctly.

### 8.3.5 Community Administration

The first user will navigate to the community admin menu and access the members list. He will remove the second user from the community. Furthermore, the first user will change the community password. After the second user logs out, he will be unable to join that community unless he enters the new password.

### **8.3.6 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 use of the whiteboard channel, the server will be restarted and it will reload all saved data from disk. The server will save all data from the demonstrated whiteboard, and deliver the data to each newly connected client.



## 9 Test Plan

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

### 9.1 Unit Testing

JUnit is used to facilitate the running of unit test cases. Unit test cases are 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 is verified through integrated usability tests by the developers.

### 9.2 Integration Testing

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

### 9.3 Test Details and Results

Test and Expected Result	Passed?
<b>Login Window behavior</b>	
Existing username and password pairs succeed.	√
Non-existent username and password pairs generate a prompt to create a new user account or an error prompting the user to enter login information again.	√
User can see a menu bar with "File" as title and are given the option to quit.	√
<b>Server connection</b>	
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.	√
If the server cannot be reached an error message shows alerting the user of a	√

problem with the server is displayed and the client returns to the login window.	
If at some point in the CoLab session the connection to the server is lost, the client displays	
<b>Community manager behavior</b>	
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.	√
If a user creates a community, he is made the moderator of the community.	√
User can see a menu bar with "File" as title and is given the option to logout and quit.	√
If the user is the moderator of the community, he can see an additional menu title "Admin" with the options to manage the community's members and to change the community's password.	√
A community's administrator can view the community's current members in a list and choose one or several members to remove from the community.	√
A community's administrator can change the password required to join the community.	√
If a user has been removed from a community, it will no longer show in the Choose Community window unless "Show all" is checked.	√
If a user tries to join a community that he has been removed from, he will be prompted for the new password.	√
<b>Channel manager behavior</b>	
All channels in the current community are displayed in a list with the appropriate names and channel types.	√
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.	√
If the user types in a name with characters not allowed, an error message shows, and the user must retype a name for the channel.	√
User can see a menu bar with “File” as title and is given the open to change communities, logout, and quit.	√
<b>Chatting behavior</b>	
In chat channel, all active user 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.	√
<b>Document behavior</b>	
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.	√
User can select “Exit” from the File menu to leave the channel.	√
<b>Whiteboard behavior</b>	
New whiteboards open with a blank white canvas and already existing whiteboards open with all current layers displayed.	√
The pencil tool is automatically selected with black as the color	√
User can gain a lock on the current, unlocked layer	√
After a period of inactivity, a user's client will yield a paragraph lock	√
User can draw on current layer.	√
Only one user may draw on a locked layer at a time	√
Thumbnail previews for a layer are accurate and correctly updated	√
User can create new layer.	√
User can select and use one tool at a time by clicking on the tool's corresponding button.	√
User can select and draw with one color at a time by clicking on the color chooser.	√
User can select “Exit” from the File menu to leave the channel.	√
User can select the export menu option to open a new window where the whiteboard contents can be saved to a file.	√

## 10 Requirements Specification

### 10.1 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.).

Support for blacklists was deferred for future implementations. All other D3 goals are fully implemented.

### 10.2 Updated Delivery Plan

#### 10.2.1 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

#### 10.2.2 Delivery 2: April 1

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

#### 10.2.3 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.

## **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 or removing members from the member list.

### **10.2.4 Future Goals and Enhancements**

- Whitelists and blacklists for community membership.
- Ability to revert to a previous revision of a document or whiteboard image.
- Support for multiple moderators in a community.
- Video and audio chat.

After validation testing, we would solicit feedback from users to find areas of our program that require improvement.