CoLab

Deliverable One

CS 3300

Group Seven

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

February 26, 2008

Contents

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). 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 typed 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

Currently, a user can log in using a pre-existing username and password and the server authenticates whether the pair is correct.

For Delivery 2, the user will be able to create new accounts.

2.1.2 Communities

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 log 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 communities of which he is already a member.

For Delivery 2, the user will be able to create new communities, and to join communities by providing the community password.

2.2 Chat Channel Protocol

2.2.1 Functionality

Each user must be users to 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.

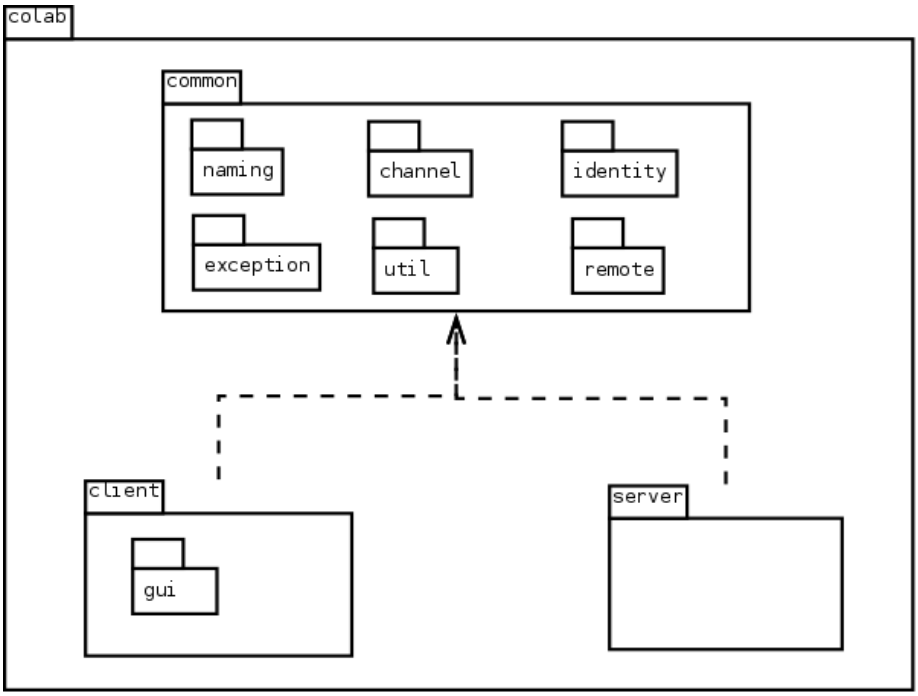
2.2.2 View

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.

3 System Design

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



3.1 colab.client

Purpose of package: Lays the domain for the functions of the application and contains the middle layer between the GUI and the server

Classes

- ClientChannel
- ClientChatChannel
- ColabClient

3.2 colab.client.gui

Purpose of package: Contains all of the UI components. The components use an event-driven system for updating

Classes

- ChannelPanel
- ChatPanel
- ChooseCommunityPanel
- ColabClientGUI
- FixedSizePanel
- LoginPanel

3.3 colab.common.channel

Purpose of package: Holds classes pertaining to Channel and its functions and features

Classes

- Channel
- ChannelData
- ChannelDescriptor
- ChannelType
- ChatChanelData
- ChatDataCollection

3.4 colab.common.exception

Purpose of package: Holds all the exceptions pertaining to network

Classes

- ConnectionDroppedException
- NetworkException
- UnableToConnectException

3.5 colab.common.remote.exception

Purpose of package: Contains exceptions that will be thrown over remote invocations

Classes

- AuthenticationException
- ChannelDoesNotExistException
- CommunityDoesNotExistException
- IncorrectPasswordException
- UserDoesNotExistException

3.6 colab.common.identity

Purpose of package: Contains classes for Identifiers

Classes

- Identifiable
- Identifier
- IdentitySet
- LockIdentifier
- StringIdentifier

3.7 colab.common.naming

Purpose of package: Groups all the wrapper and name rules classes.

Classes

- ChannelName
- ColabNameRules
- CommunityName
- InvalidCommunityNameException
- InvalidNameException
- InvalidUserNameException
- UserName

3.8 colab.common.remote.client

Purpose of package: Provides remote interfaces for remote client objects

Classes

- ChannelInterface
- ColabClientInterface

3.9 colab.common.remote.server

Purpose of package: Provides remote interfaces for remote server objects

Classes

- ColabServerInterface
- ConnectionInterface

3.10 colab.common.util

Purpose of Package: Contains utility classes

Classes

- FileUtils
- StringUtils

3.11 colab.server

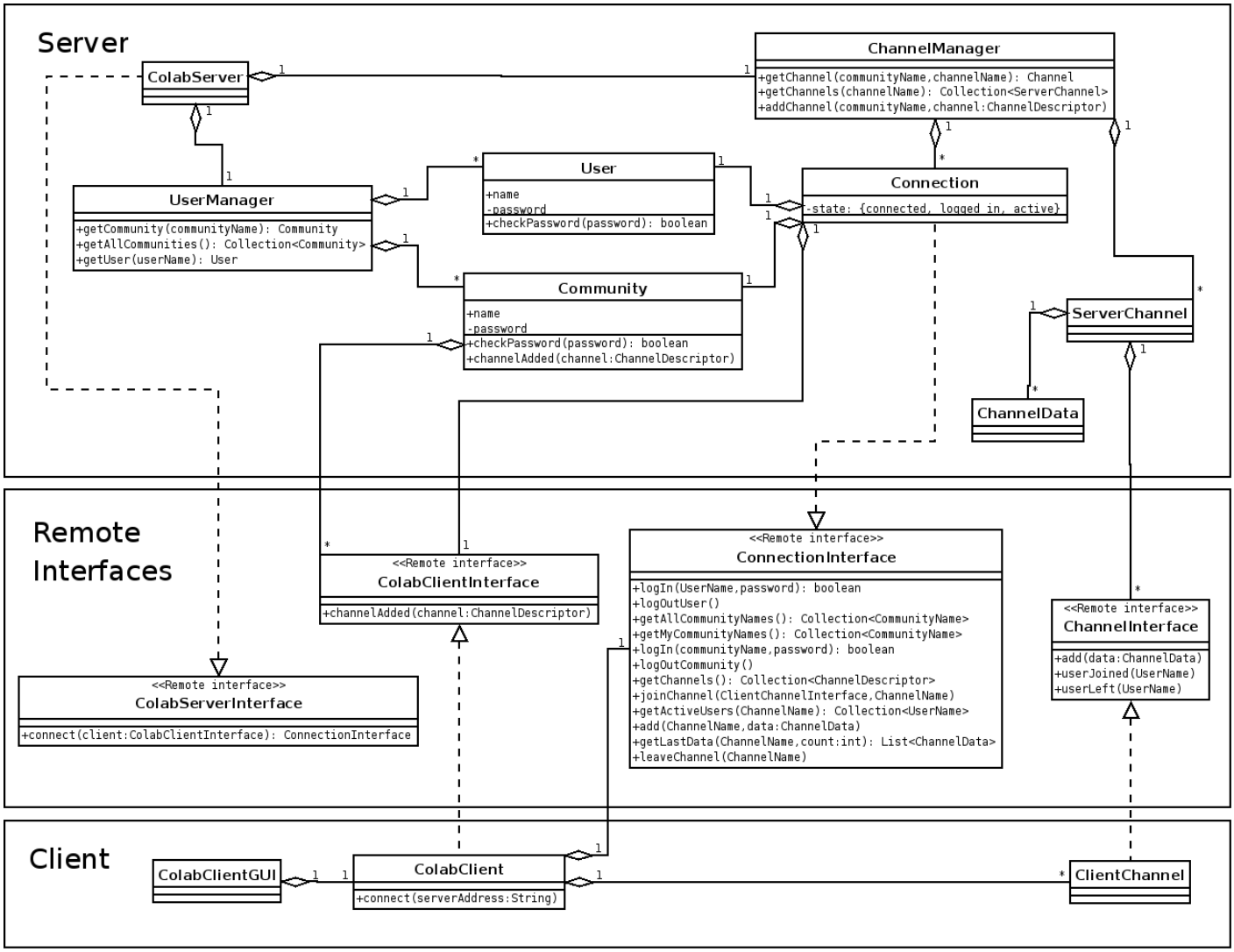
Purpose of package: Contains server objects

Classes

- ChannelManager
- ColabServer
- Community
- Connection
- Password
- ServerChanel
- ServerChatChannel
- User
- UserManager

4 Detailed Design

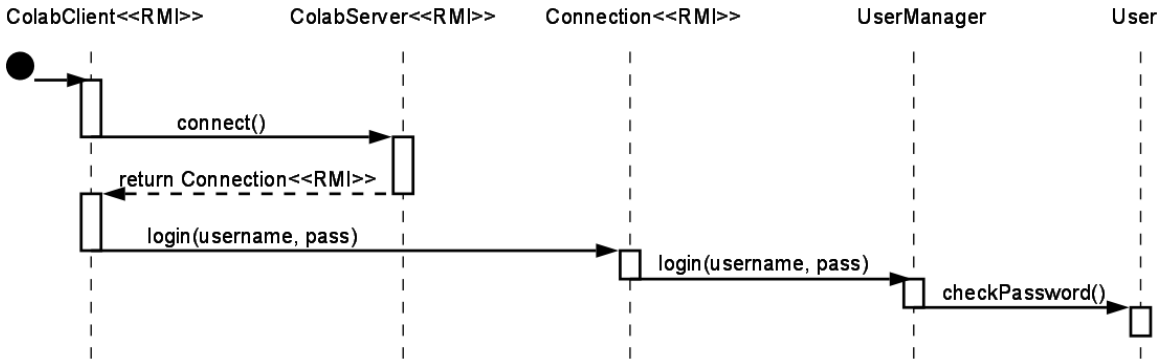
4.1 Class Diagram



4.2 Sequence Diagrams

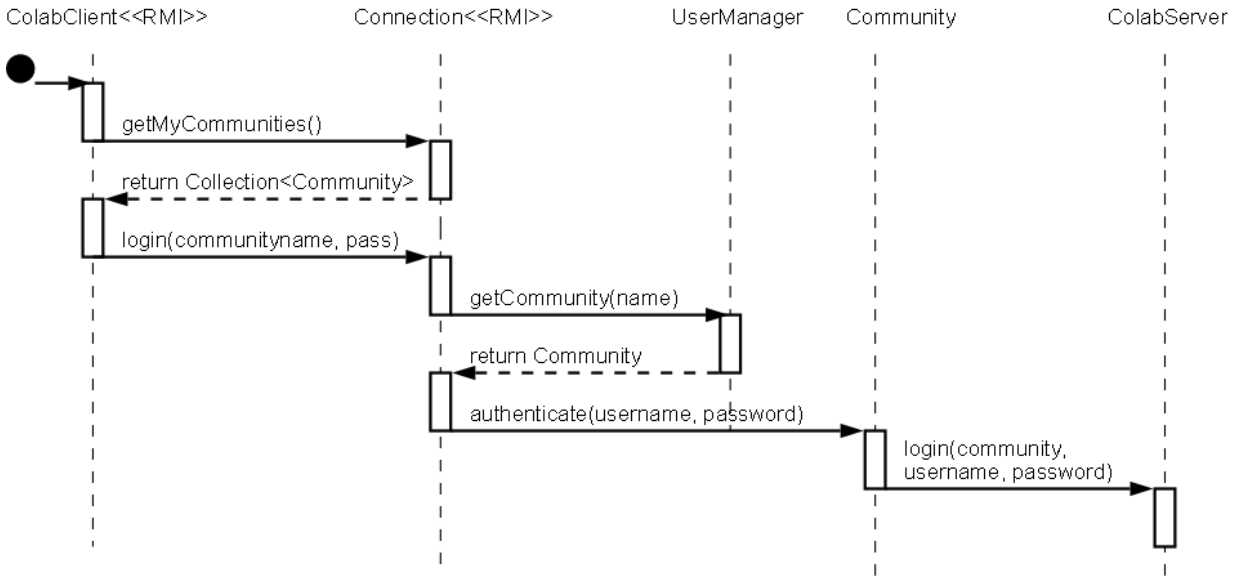
4.2.1 User Login

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



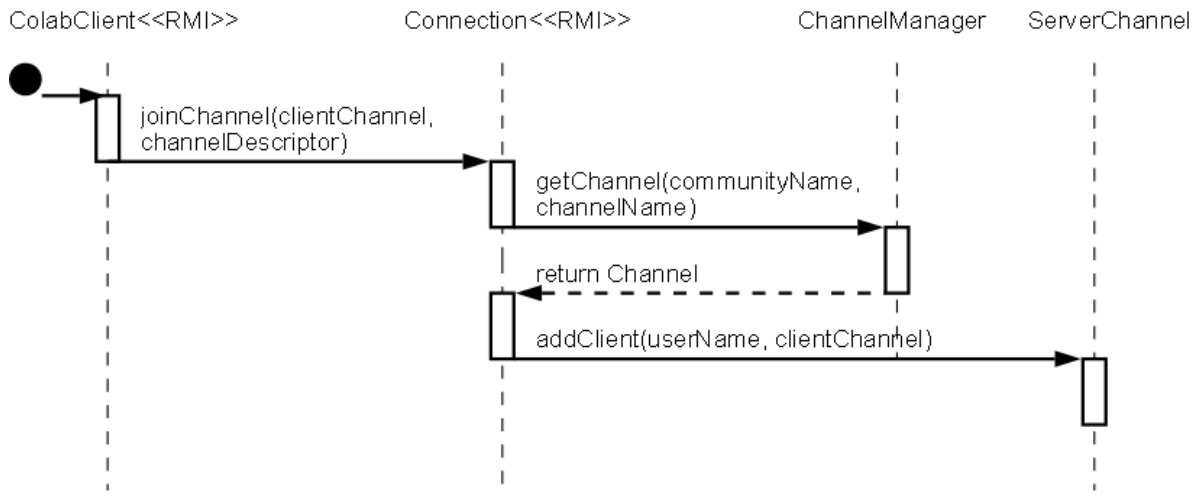
4.2.2 Community Login

After user login, the user logs into a community.



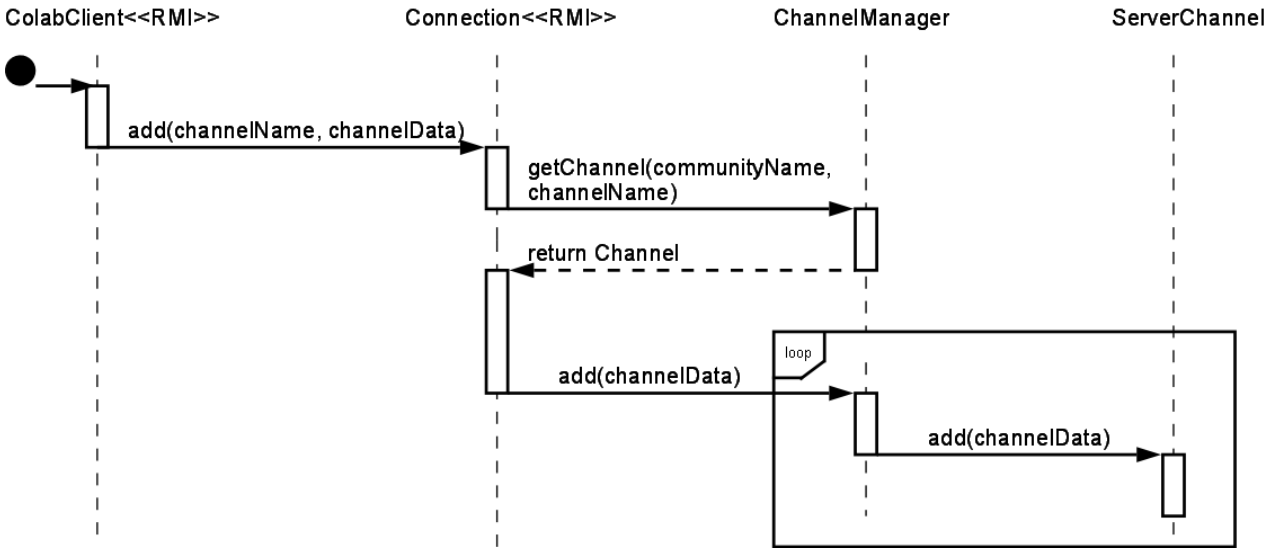
4.2.3 Joining a Channel

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

CoLab uses a class hierarchy where a Channel object represents the concept of a CoLab channel--for example, a chatroom, a document editor, or a whiteboard. Therefore, each Channel object is self-contained and holds all data necessary to re-create the exact state of that channel at a given moment. The CoLab framework also contains a Community class, representing a CoLab community with a list of channels and users who are authorized to view the data. The CoLab framework contains two structuring classes responsible for holding this channel and community information--ChannelManager and UserManager. ChannelManager stores information about Channels, and UserManager holds information about all communities and their users.

In D1, the CoLab server does not support any data persistence. However, later deliverables will include the ability to save and retrieve data for channels and communitites. Data storage will be implemented using XML files so that the data can be edited easily outside of the CoLab application.

6 Interface Specifications

6.1 Client-server Interface

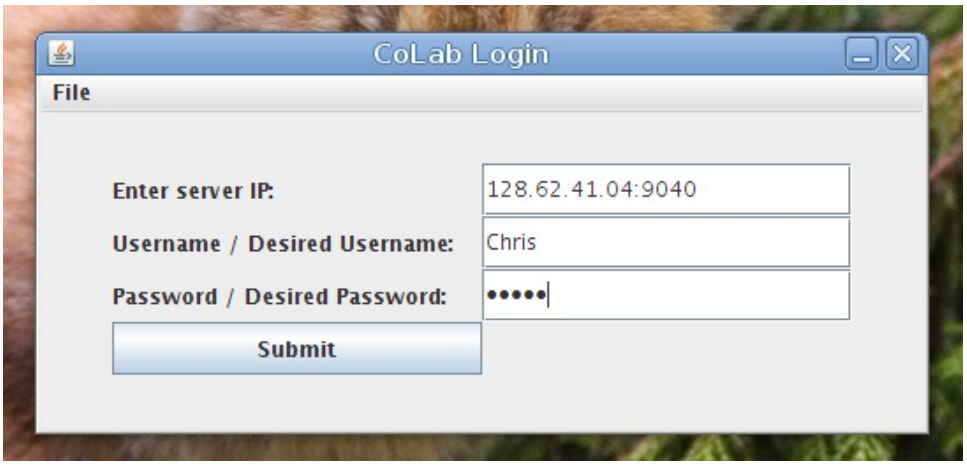
Colab's client-server communication is supported entirely by distributed objects using Remote Method Invocation (RMI).

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. 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 invokations are coming from the user who logged in.

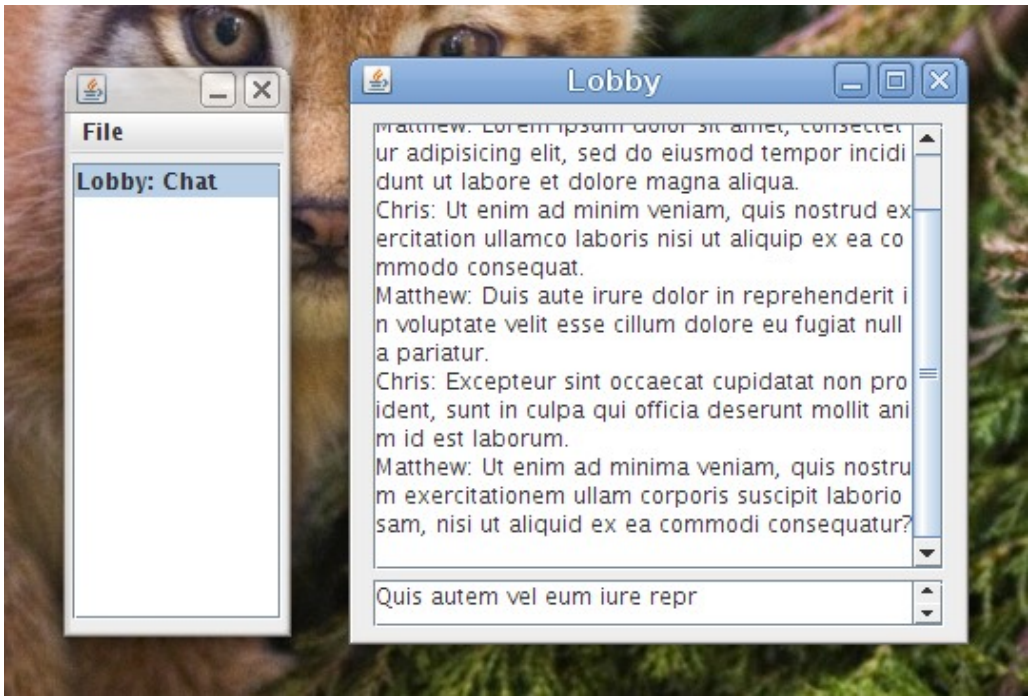
6.2 User Interface

6.2.1 User Login



6.2.2 Community Login

6.2.3 Channel List and Chat



7 Demo Plan

7.1 Demo Overview

D1 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 D1 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, populates it with some test data and waits for incoming connections. The server is designed to run from a command line. There is no GUI, although 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

The CoLab client application presents a GUI to the user prompting for a server IP, username and password. The user logs in using an existing username/password pair (for D1, this is part of the test data pre-populated by the server). The client program connects to the CoLab server using RMI and authenticates whether the username/password combination is correct. If the user is authenticated successfully, the client displays a list of communities of which the user is a member (pre-populated by server) as well as a button to join a community. The user selects a community from the drop-down list and clicks the button to select and join that community. When the button is clicked, the CoLab client presents a new GUI representing a chatroom lobby, 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. Using two machines, we can demonstrate how chat takes place over a network.

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 Results

D1 includes 26 unit tests which cover unit test cases for all major functionality listed in the D1 delivery plan. All unit tests pass when run in JUnit. For integration testing, all developers have used the system and verified that the functionality for D1 is working as expected.

9 Requirements Specification

9.1 Requirements Overview

The D2 plan had the following original goals:

- Document channel protocol is functional
- User can create and use a document channel
- User can create user accounts and communities
- User can create channels, join multiple channels concurrently
- Moderators can modify community preferences (adding passwords, changing blacklist, etc.)

Most of the original D2 plans are still valid. D1 focused on establishing the back-end framework and getting working network connections. Because most of the framework has been written and is working correctly, extending the code will be efficient in D2. However, the D2 plans must be modified to add data storage to the application. D2 will add mechanisms for storing data to disk and retrieving it later. Adding data storage will require significant consumption of developer resources, so the moderators feature will be removed from the D2 plan.

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: March 21

Framework

Users can create new users.

Users can create new communities.

Users can join a community with the community password.

Users can create channels.

Users can participate in multiple channels concurrently.

Server can save and retrieve data on the filesystem to hold persistent data.

Chat Channel Protocol

The chat panel displays a list of the currently participating users.

A user can download and export all data (the entire chat log) for a chat channel.

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.

9.2.3 Delivery 3: April 21

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