PROJECT DESIGN AND PROTOCOL

ericadu - connieh - kahuang

We will be using a Client/Server Model for our project.

An EDIT has five attributes: **document_name version_number**, **startIndex**, **length**, and **replacement**

document_name ::== the name of the document that is currently being edited

version_number ::== corresponds to the current working version that this edit is operating on.

startIndex ::== where the edit starts.

length ::== how many characters we are replacing.

replacement ::== the string to replace the length # of characters in front of **startIndex** with.

space ::== " "

EDIT ::== **document_name** space **version_number** space **startIndex** space **length** space **replacement**

The Server consists of a set of deltas (changes) to a document state with corresponding version numbers, a set of Threads listening for input from each of the clients, and a queue which the threads update with messages.

The Client consists of the GUI with which the user will be interacting with, the client's version of the document, and two queues, one to send deltas to the server and one to receive valid deltas from the server.

The Protocol for communication between Server/Client is as follows:

Pull Request ::== pull version_number

Push Message ::== push version_number startIndex length replacement

The Client constantly sends pull requests to the server to get the latest deltas using its local version_number. The server sends back **Push Messages** from version_number + 1 and on which are the valid deltas that the server has curated. The client applies these deltas to its local copy of the document (and also updates the version number), and then updates the GUI.

The server is constantly listening for **Push Messages** from the clients and puts them all in a queue and applies the edits one at a time.

If a client pushes to the server and the server's version number is greater, we need to *transform* the client's edit to ensure its validity. We go through the version numbers ahead of this edit in order. If one of the version's edits behind this one's, we shift this edit's startIndex ahead by the appropriate amount. If it's ahead, we do nothing. If an edit replaces over where this edit occurs, we shift the edit back to the startIndex of this overlapping edit.

TESTING STRATEGY

Our testing strategy was very modular in design. At a high level, we wanted to first test server functions and communication, then GUI and client functions, the communication between the two components, and lastly multiple clients.

SERVER FUNCTION

First, we implemented test methods to test how the server responded to client messages. At this point, no client was built, so the messages were written according to our grammar, and then sent over a socket opened via telnet. We were looking to see if:

1. the server added the edit messages to the queue when the client sent a "push" message
2. the server was able to respond with edit messages from the queue when the client sent a "pull" message, in chronological order
3. the server was able to pull the correct amount of edit messages from the queue when the client sent a "pull" message, by comparing version numbers

Automated tests were not written for this part of the testing, because telnet was used instead. We have written automated tests checking adding edit Strings to the serverside edit list.

After confirming the basic server function, we implemented automated test methods that handled the special case of merging. As the design portion specified, a "merge" occurs during a "push" message, when the client version number and the server version number conflict. Our test results confirmed that merging functioned as designed.

GUI FUNCTION

Simultaneously, the GUI was tested component by component. We started with the rtceGUI, the main editing GUI with the JTextArea, and then moved on to the startGUI.

Testing consisted of two parts:
1. The Look and Feel of the GUI
2. Listeners

Part 1 of testing, the look and feel, was the simplest part. No automated tests were written. Things like checking if buttons worked, if adding text was possible, etc.

Part 2 of testing, was more involved. No automated tests were written for these purposes either. We stepped through and made sure the rtceGUI was able to register text changes, before testing whether or not the startGUI had the capability of starting the rtceGUI and connecting to the server.

CLIENT FUNCTION

The ClientModel class is driven with automated tests in ClientModelTest.java. The goal of this testing suite was to confirm the ability to make edits to the client-side document once the messages were received, and parsed from a message string into usable parts. ClientModelTest tested functions that took in information (push, client version number, start index, length of edit, edit content) and applied the changes to the client side document by ensuring the expected output string equaled the output string.

CONNECTION

The final piece of the testing puzzle that we implemented was testing the socket connection between the client and the server. We monitored this by using println statements and observing the console.

CONCURRENCY/THREAD SAFETY ARGUMENT

Our system should not run into concurrency issues for a variety of reasons.
1. The only shared datatype between the clients that is stored on the server is the serverside list of edits.
2. The only other datatype that is used is a ConcurrentLinkedQueue. This is a threadsafe datatype.