Design Document: jcotler-keanu-dmendels
Jane Cotler, Keanu Delgado, Daniel Mendelsohn

DESIGN COMMITMENTS AND CONSIDERATIONS

- At any given point, the documents will be synchronized at a time in the near future
- Users should be able to see their edits immediately, regardless of latency
- The server will maintain persistence even through multiple client connects/disconnects
- Multiple clients and multiple documents are supported
- Response times are fast
- All clients see the same data, though they may be working on different documents
- An HTTP protocol is used to low latency through message transfers
- Rich text formatting is supported in the form of bold, italic, and underline
- Bold, italic, and underline changes are not transmitted by selecting text and changing it, only by typing in a certain format after clicking and unclicking buttons.

ABSTRACT DESIGN

Client display is visualized in an instance of ClientGUI.
Fields in ClientGUI:

- JScrollPane scrollPane for ability to scroll in the text pane
- JTextPane textPane that user types into
- String currentTitle of current document displayed in the text pane
- StyledDocument currentDoc displayed in the text pane
- Integer currentDocID of current document displayed in the text pane
- JButton bButton for toggling bold
- JButton iButton for toggling underline
- JButton uButton for toggling italic
- Boolean isBold whether bold is on
- Boolean isItalic whether italic is on
- Boolean isUnderline whether underline is on
- JComboBox docSelect for switching between documents
- ArrayList<Integer> docIDList of documents by their ID
- JButton newDoc for creating new documents
- ClientDataModel dm associated with this client
- VerifiedClientDataModel vcdm associated with this client
- ClientController controller associated with this view
- String host that client is connected to
- Int port that client is connected to

ClientController manages the client
Fields in ClientController:

- String host that client is connected to
- Int port that client is connected to.

- Gson gson object to use as a Util to convert JSON objects
- VerifiedClientDataModel model that client is connected to
- Int clientID of this client
- Int version of documents that this client last saw
- URL getURL to sent “get” request
- URL postURL to send “post” request
- URL connectURL to sent “connect” request

Server manages all of the clients
Fields in Server:

- ServerDataModel model associated with the server
- Gson gson object to use as a Util to convert JSON objects
- Int clientCounter number of clients currently connected to the server
- Int docCounter number of documents currently being edited

ServerDataModel, ClientDataModel, and ClientController all extend BasicDataModel, which implements the DataModel interface.
Fields in BasicDataModel:

- HashMap<Integer, Pair<String, StyledDocument>> docTable stored documents by document ID
- ArrayList<ChangeRequest> log list of all changes made to the data model

Fields in ServerDataModel:

- Int versioncounter number of most recent data model version

Fields in ClientDataModel:

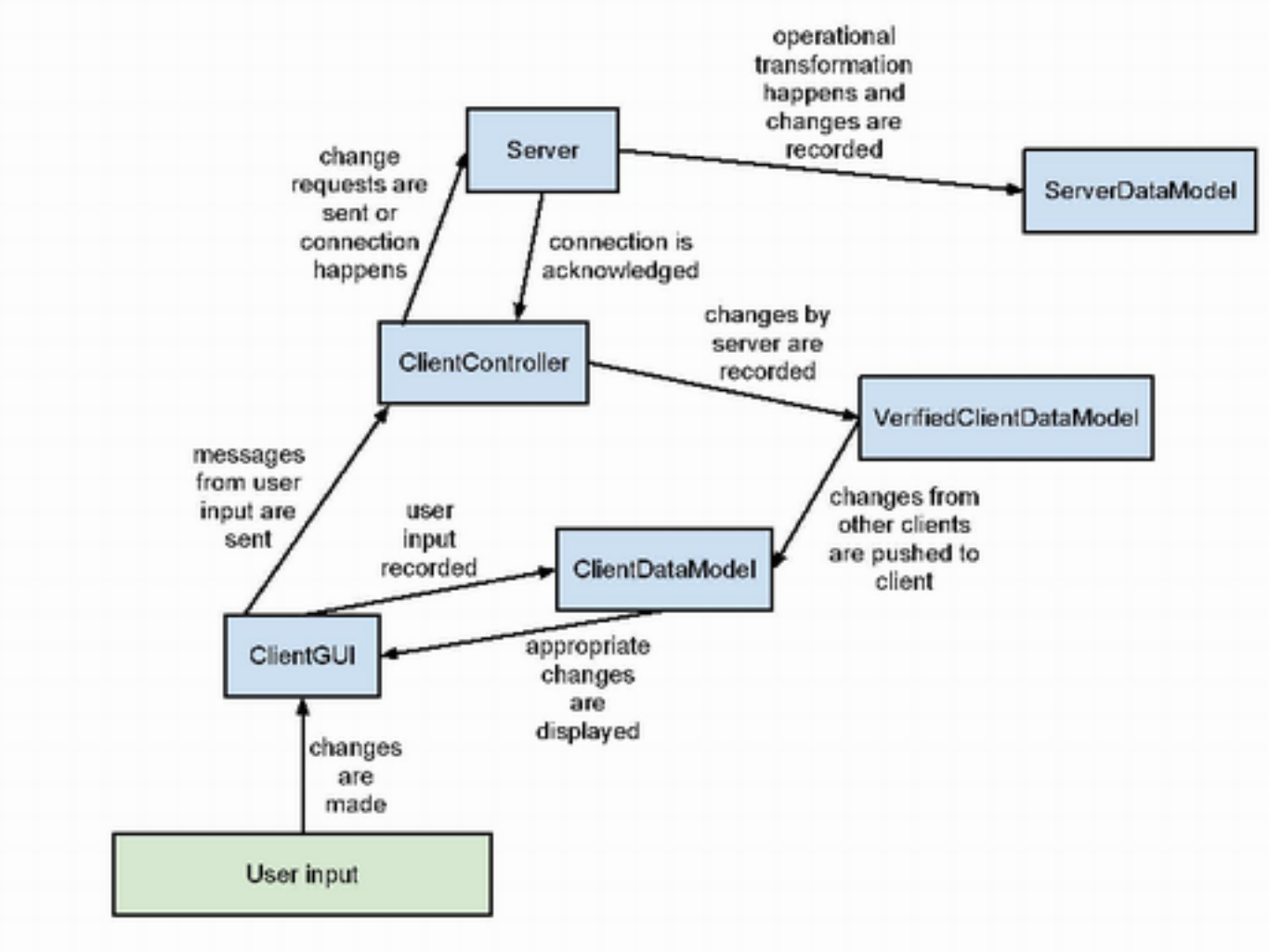
- ClientGUI gui the GUI that this model contains the display data for

Fields in VerifiedClientDataModel:

- ClientDataModel guiModel represents the model that this model pushes changes onto
- Int clientID represents unique identification number of client

System explained on next page

System:



PROTOCOL

Messages between the client and the server are sent over HTTP as JSON objects.

Messages requesting modification to document data models are ChangeRequest objects. A String is converted into a ChangeRequest object by methods in the static Utils class.
Fields in ChangeRequest:

- OperationType operation represents which operation is being made
    - enum in ChangeRequest: NEWDOC, INSERT, DELETE, or STYLECHANGE
- Integer docID, which represents the ID of the document requesting a change.
- Integer clientID, which represents the ID of the client requesting a document change.
- Integer versionID, which represents the last version of the document seen by the client.
- Change change, which represents the change to be made. Change is an interface implemented by the objects NewDoc, Insert, Delete, or StyleChange
    - Fields in NewDoc:
        - String title which the new document should be titled

    - Fields in Insert:
        - Integer position at which to insert text
        - String text to insert
        - Boolean isBold true if text is bold, false otherwise
        - Boolean isItalic true if text is italic, false otherwise
        - Boolean isUnderline true if text is underlined, false otherwise
    - Fields in Delete:
        - Integer position before text which is to be deleted
        - Integer numchars number of characters to delete
    - Fields in StyleChange:
        - StyleType styletype represents the style to change text to
            - enum in StyleChange: BOLD, ITALIC, or UNDERLINE
        - Integer position at which to begin the style change
        - Integer numchars number of characters after position to apply change
        - Boolean isEnabling true if styletype is enabled, false otherwise

Messages sent by the client to the server requesting a periodic update are sent as UpdateRequest JSON objects.
Fields in UpdateRequest:

- Integer clientID represents the unique client ID of the client sending a request
- Integer versionID represents the last seen version of documents by the client

When the server gets a client connected to it, a ConnectResponse JSON object is sent from the server to the client.
Fields in ConnectResponse:

- Integer clientID unique client identification number assigned by the server.

When the server receives a post update from the client, it sends the client back an acknowledgment in the form of a PostResponse JSON object.
Fields in PostReponse:

- String message either “success” or “failure”

ORIGINAL DESIGN

Current design was based off of this design, but it was eventually scrapped because we wanted to use a faster HTTP protocol and JSON object passing was easier to work with and modify than message parsing.

<u>Grammar for Client to Server messages:</u>
Message := (Edit-request | Doc-op | Hello | Bye) ClientID Newline
Edit-request := Edit-op Space DocID Space Position Space Text
Doc-op := Doc-create | Doc-delete | Doc-rename
Doc-create := “newdoc” Space Text
Doc-delete := “deletedoc” Space DocID

Doc-rename := “rename” Space DocID Space Text
Space := “ “
Newline := “\\n”
Edit-op := “insert” | “delete” | “bold” | “unbold” | “italic” | “unitalic” | “underline” | “ununderline”
Position := [0-9]+
DocID := [0-9]+
ClientID := [0-9]+
Text := .+
Hello := “&”
Bye: = “_”

Grammar for messages from Server to Client:
Message := Edit-alert Newline
Newline := “\\n”
Space := “ “
Edit-alert := Edit-op Space DocID Space Position Space Text
Edit-op := “insert” | “delete” | “bold” | “unbold” | “italic” | “unitalic” | “underline” | “ununderline”
Position := [0-9]+
DocID := [0-9]+
Text := .+

States of Server:

- Idle - server is waiting for input
- Receiving - server recieves and parses input
- Updating - server is determining the correct action and updating the data
- Publishing - server ensures clients are displaying real-time data

Data:

- StringBuffer will be used to represent the editor’s text content.
- The server will use operational transformation to handle concurrent requests to modify the data.
- Requests will be stored in a queue until they are processed. This will allow operational transformation to modify requests appropriately

States of Client:

- Idle - waiting for input or server messages
- Processing input from user - moves to this state via listeners
- Processing server updates - moves to this state via network listener
- Sending message -

Operation types:

- insert
- delete
- bold/unbold

- italic/unitalic
- underline/ununderline
- new document, delete document, rename document

State Machine for message-passing:
Two states: Disconnected, Connected

- Disconnected transitions to Connected upon initial network connection
- Connected transitions to Disconnected upon user-requested disconnect
- Connected transitions to Disconnected when the server gets no response from the client and determines that the client has disconnected

OTHER IDEAS CONSIDERED (BUT NOT USED)

- How to ensure edit requests are still “fresh” at processing time?
    - Possible later addition: be careful about confirming that the request is still “fresh” at processing-time, such as having the protocol send a checksum for the line it wants edited. That way, the server won’t edit a different line if some other processing messed things up in between.
    - Decided not to do this because operational transformation is better
- Server handles document internally as List<Line>, where a Line is essentially a character buffer. Each Line object represents a sequence of characters ending in a new line character
    - We decided there was no need to break up the document this way, and that as long as we handle edits cleanly, we can use a single StringBuffer
- Server sends clients updates as a list of “Line changes”, where a line change contains the document ID, the index of the changing line, and the text of the new version of that line
    - Not doing this since we’re scrapping the line-by-line data structure idea

- Representation of document data:
    - Character buffer (we chose this essentially, as we’re doing a String Buffer)
    - Line buffer (might be complicated because when a line gets too long, it’ll overflow)
    - Arbitrary group of characters buffer (say 50 characters)
        - avoids the aforementioned line buffer problem thanks to abstraction
        - benefit is that client side can send over entire group of characters that it is currently editing, and since its small the performance hit will be small
    - We decided that our edits would be very data-light, so groupings of characters don’t save that much effort. That’s why we went with StringBuffer
- Maybe do a line-by-line lock?
    - Due to the rest of our design, this isn’t necessary because the server handles all synchronization issues and alerts the clients as to what they should show

- Idea for request format: Server could keep track of cursor location, then the messages just have to be the keystrokes that the users perform (e.g. cursor at location 5555, press ‘backspace’, press ‘up key’, press ‘k’, press ‘e’)
    - We decided this could get really complicated and messy, especially if there’s a mistake and the server and client disagree about the cursor location