Pypad

A magical and revolutionary collaborative editing platform

Apr 2010
R. Crosby, M. Huang, J.J. Poon, S. Zhang

0. CODE OVERVIEW

0.1. FILES

- PypadGui.py - user interface for text and drawing editing

  Contains the PypadGui class (hasa PypadGuiText and PypadGuiDrawing object)
  The PypadGui class is the view in the model view controller model of Pypad,
  displaying the text editor and canvas necessary for collaborative editing

- PypadServer.py - creates remote object server where data is stored

  Contains two classes-- PypadData and PypadServer

  PypadData contains all non remote attributes
  Server contains all generic subject methods
  (from Subject-Observer-Modifier template)
  For the purposes of other comments in this code package, PypadServer will be referred to as client

  The PypadServer class stores the centralized data, receives and updates info from clients, and notifies other clients of these changes
  Pypad is the subject of the subject-observer design pattern,
  and the model of the model-view controller design pattern.

- PypadClient.py - sends/receives data between server and client

  The PypadClient class is the controller in the model view controller design pattern
  of Pypad.

- RemoteObject.py - Class wrapper for Pyro

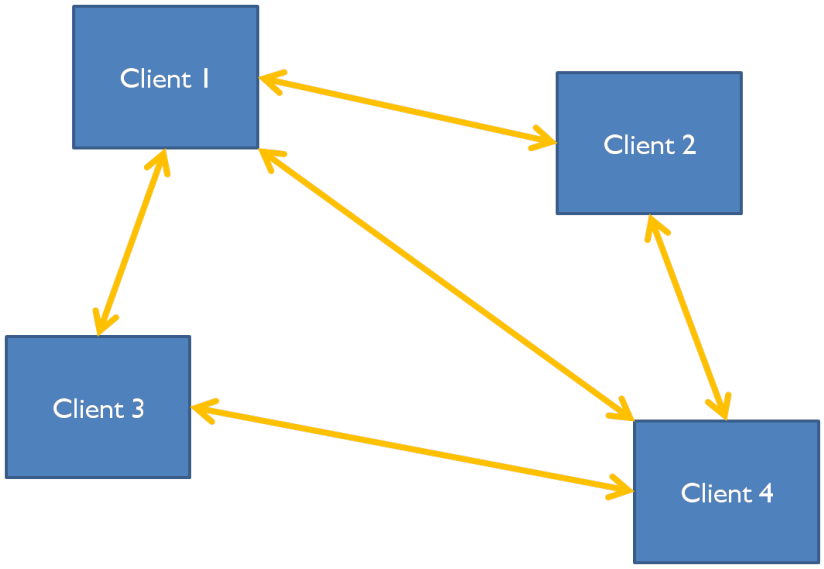
0.2. PROGRAM RUNNING INSTRUCTIONS

1. Run nameserver

2. Get computer's IP address
3. Enter IP address default_ns_host value in RemoteObject.py (line 9)
4. Run PypadServer.py
5. Run PypadClient.py
6. Repeat step 5 as many times as desired on any computer on the local network.

1. PROBLEM STATEMENT

The goal of our Software Design project is to create an easy to use, remote collaborative text editor, inspired by the now-defunct Etherpad, in Python. We want to allow many different users to be able to work on the same document, whether it is text, images, or drawings, and not have to worry about which revision they are working on or if they have the fully compiled document that everyone is working on.



To do this, we needed to identify and use a variety of Python libraries for GUIs and networking and incorporate them with our own code, which will allow for collaborative text and drawing editing. The main challenge in our project was identifying incremental steps for development and defining clear interfaces to divide up the problem.

2. THE PYPAD DESIGN

Our collaborative text editor, named Pypad, allows multiple clients to engage in real-time text document editing on the same page. Clients are able to edit and view changes made by themselves and their peers in real time. All of this text is saved indefinitely on a remote server, which can be accessed at any time by anyone who has the identifier of the server used to host the text document. By simply entering the IP address of the server into a Python script, any person is able to gain access to the remote object, and thus, free to view and make changes to the document. In addition, the remote server keeps track of all changes made by any user to the text document. This allows clients to view previous revisions of the document through the GUI. In these ways, Pypad is an excellent solution for people seeking to work collaboratively with each other in real time.

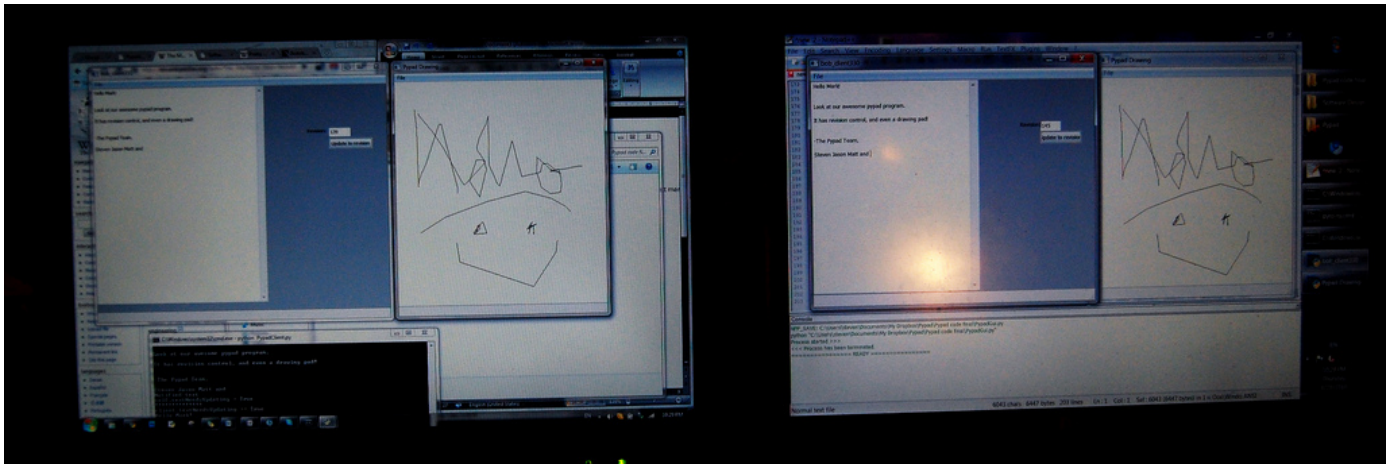


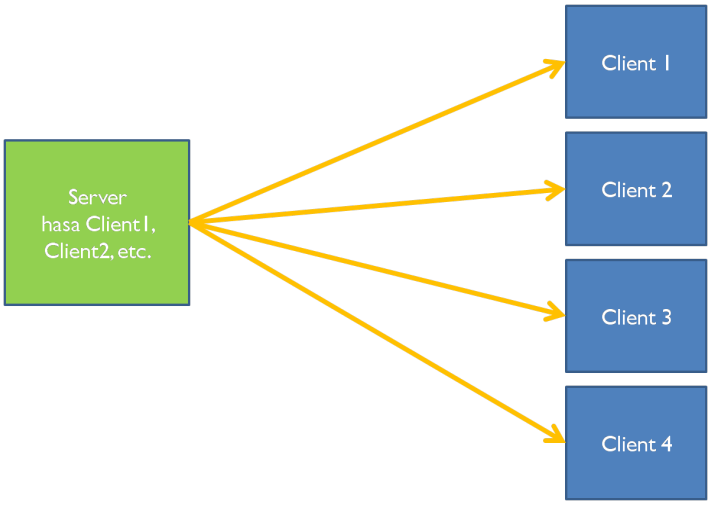
Photo: Pypad text and drawing client running on two separate computers

2.1. DESIGN HIGHLIGHTS

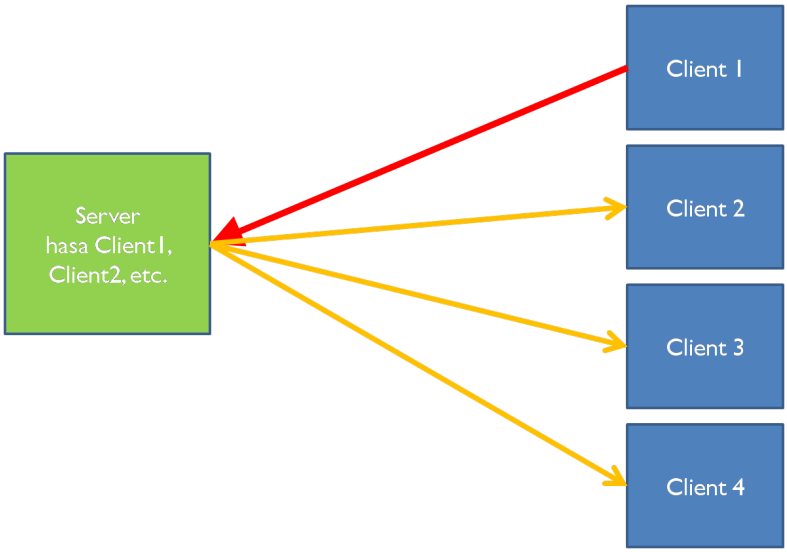
Our design implementation to solve this problem includes several key features:

- A single server that hosts all the data, which is accessible to every client. This data currently includes

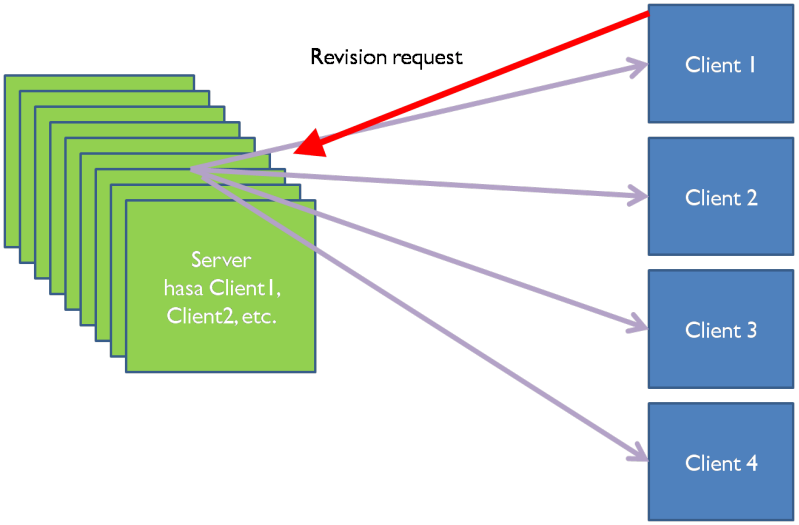
1. plain text
2. hand drawings



- Any user has the ability to write new text or draw a new object, and update the remote object on the server. This update will immediately propagate to every client connected to the server.

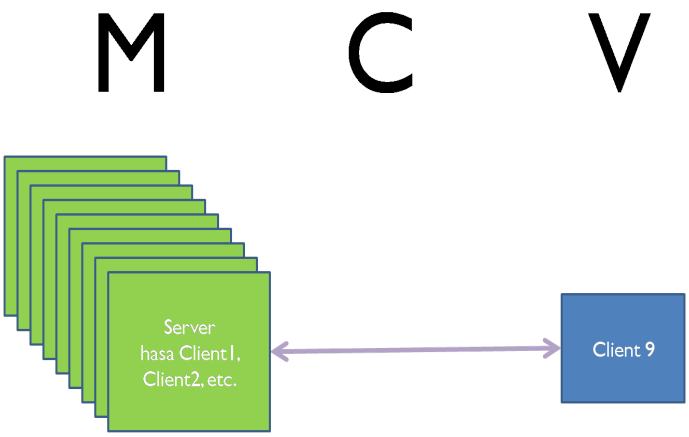


- Each client can view every previous revision made to the text document, and revert the page to that revision.



2.2. DESIGN METHODOLOGY

Our collaborative text editor project is a perfect candidate for a program based on the Model/View/Controller paradigm. This design decision for modularity allowed us to divide tasks among team members.



The model contains all the collaborative data that is shared among users, stored in a central Pyro (Python remote objects) data object. The data objects consist of a list of objects that multiple clients can read and write to. Currently, the object consists of a string (text) and a list of points (a drawing). This object can be expanded to include many new features, such as rich text, images, etc.

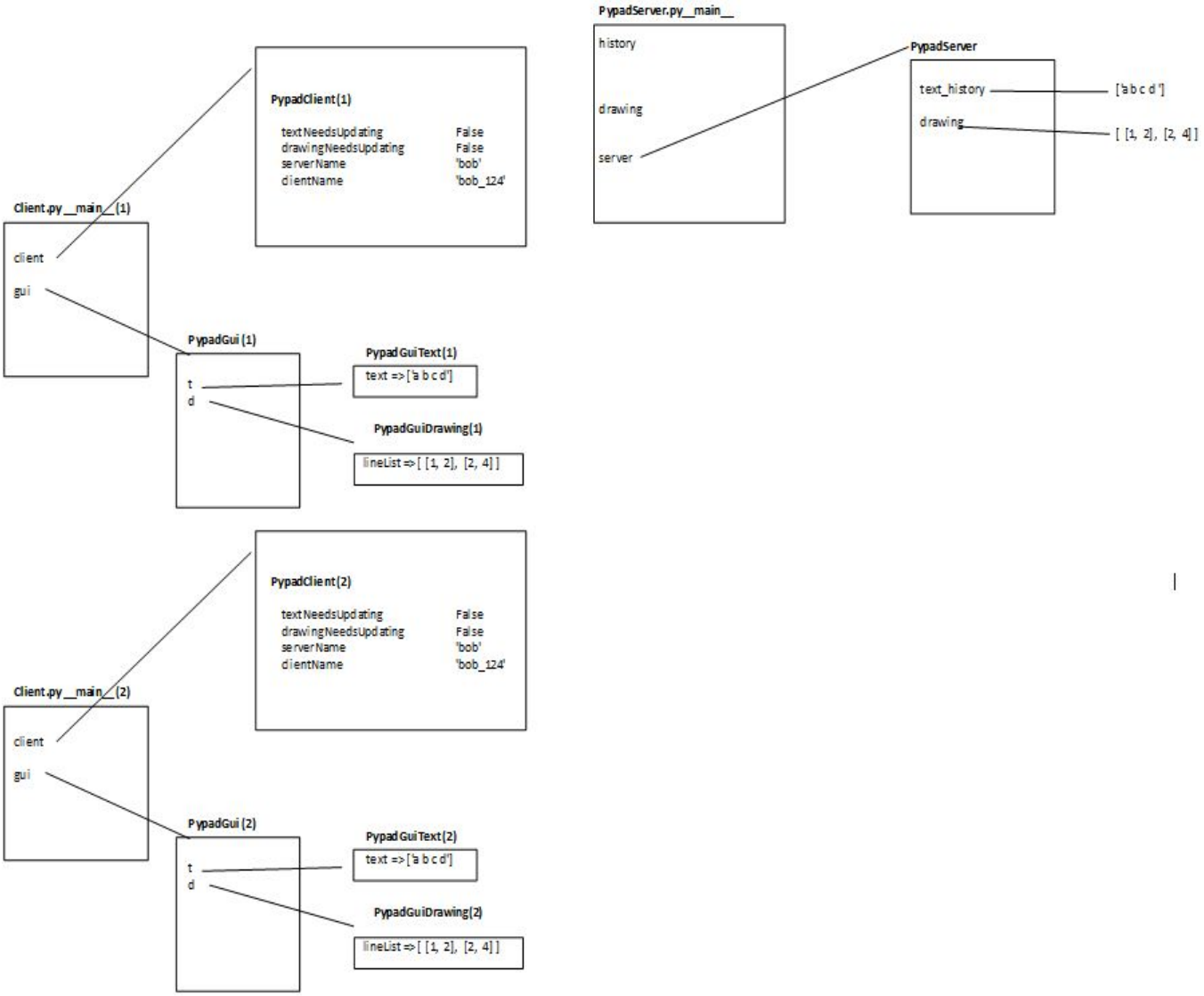
Our view is a WxPython based text and drawing editor that we modified from some Wx tutorials. Currently, users can collaborate in real-time on a text editor, collaborate in real-time on a drawing canvas, and view and revert back to every revision made on the text object. In addition, text files can be opened to the text editor, and saved on each client's local hard drive. Line numbers and a color coding by user may be added in the future.

Our controller code acts as the link between the server (model) and the client (view). It manages communication with Pyro and our GUI implementation and contains the code that keeps everything synchronized.

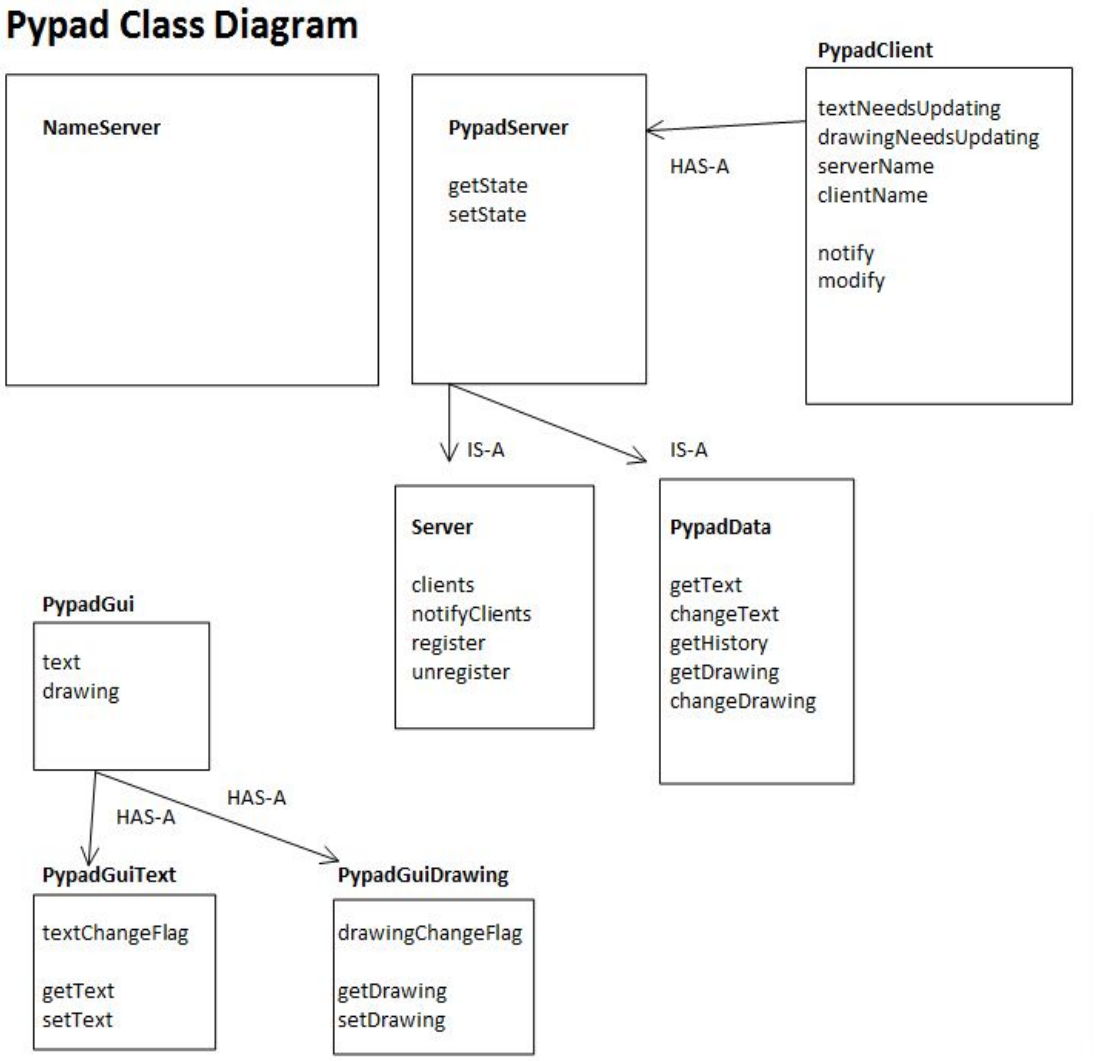
This is a good example of the appeal of Model/View/Controller; since our model and view are largely contained within external libraries, confining the bulk of our project's implementation into a self- contained piece of code. This modularity allows us to, for example, experiment with different GUI libraries without changing the file defining our program's functionality.

In addition, we were able to expand our data object to include many different kinds of data types. Currently, the text and the drawing are stored as completely different attributes of the data object. This allows excellent flexibility in the future for including new features for Pypad.

2.2.1. OBJECT DIAGRAM



2.2.2. CLASS DIAGRAM



2.3. DESIGN CHANGES

- Originally, we planned to store our remote data object as a single string. We planned to do this because we thought that the only kind of data we would be sending is plain text. Eventually, we realized the expandability of our platform and the power of Pyro. Thus, we changed our data object from a single string into a list of objects. The advantages of this implementation is two fold. First, the list allows us to store all the previous revisions of the object, and to access them at any time. This turned out to be one of our key design highlights. Second, since the data type is an object, we were able to store a wide range of data values. Thus, our platform becomes flexible enough to allow expansion for things like rich text, images, and drawing.

- In our original implementation of the controller between the server and the client, we had the controller send data from the server to the client every five seconds (i.e. fixed intervals). Thus, even if no changes were being made, the controller would send the server data to every client. In later iterations, we changed this so that if the client made changes to his/her object, it would raise a flag in the server. When this server flag was raised, it would immediately send the server data to every client. Now, all the clients only receive updated server data when it has been changed. This allows for live, real-time text and drawing updates between clients.

- We removed the PypadControl class from our previous implementation of Pypad. Originally, we planned to use PypadControl as a 'layer' between the client and the server. We wanted this layer to include methods that would commit new text to the server and update next text to the client. Ultimately, we decided to include these methods directly in the PypadClient class. By doing so, we were able to create a much simpler implementation of client-server interaction, and also achieve much more reliable results. This design change is reflected in our updated class diagram, which is subsequently much more simple and clear than our previous class diagram.

- In class, we discussed the advantages of using methods to retrieve attributes as opposed to accessing the attribute directly. Thus, in this revision of Pypad, we protected most of our instance variables (not shown on Object Diagram) and used methods in order to retrieve these attributes. We found this extremely helpful in putting together individual code. It allowed us to understand our teammate's code and apply modularity in order to use the method as originally intended. (4/21)

- We originally had only one thread running which updated the GUI text periodically. This was slow. To make client updates faster, we added several threads to check for new content in current editors and push new content to all connected clients. Adding threads changed our program into a much more seamlessly interactive experience. (4/23)

- Originally, we had an issue in which the cursor jumped to the beginning of the text editor every time a client's text was updated from the server. A design change we made to fix this was to use an instance variable to keep track of the cursor position for each client.

2.4. ABSTRACTION AND LANGUAGE

In the project, the essential abstraction we made was the **server** layer, **controller** layer, and the **client** layer. The server layer is a **remotely accessible data object** that contains the **drawing** attribute, the **text** attribute, and the **text history** attribute, which is a collection of all previous **text** attributes. This layer servers as the **cloud** which serves multiple clients who want to access and change the **text** and **drawing** attribute. Since this layer is very flexible, in the future, we hope to add new attributes to this layer, which will enable the sharing of rich text, images, etc.

The **controller** layer is remote controller that contains methods that can act on the **client** layer and the **server** layer. These methods include retrieving the data object from the server to update outdated data object on the client side, committing new data objects to the server, and changing the highlighting of the text to the server. This last method will be used to identify different clients who are acting on the same server.

Finally, the **client** layer is the GUI that appears on each users' screen. It allows the user to manipulate text and drawing, and to see changes that other users have made to the same data object.

3. DESIGN AND DEVELOPMENT ANALYSIS

3.1. ANALYSIS

By the end of this project, we met our minimum deliverable of having a simple collaborative text editor that multiple users could edit in real time. Our implementation was extremely stable, and multiple users could type

and see updates in real time.

In addition, we were able to meet many aspects of our maximum deliverable. First, we were able to implement the ability to view history of revisions. We allow users to keep track of all revisions ever made to the data object, and revert the text document back to that revision. Second, we were able to implement the ability for users to draw collaboratively. Users are able to draw on a simple drawing canvas and collaboratively draw things with their peers.

Most importantly, our platform is extremely flexible to allow us to achieve all parts of our maximum deliverable with few changes to our object code. Our data remote object implementation has the ability to send many different kinds of data types. This enables us to implement things like highlighting, color, italics, bold-face, and images, if we had more time.

3.2. DESIGN REFLECTION

Our project contains a number of design choices that we made over the course of creating Pypad. The best design decision we made was choosing our fundamental approach to writing code: using the Model/View/ Controller paradigm.

Using this paradigm allowed us to have incredible modularity in how we implemented the remote data object, the client GUI, and the controller between the data object and the client. For example, we were able to easily add the drawing capability in the remote data object without affecting the client GUI. Later, we were free to add a drawing canvas to the GUI.

In addition, it let us split up our work evenly, with different people writing different components of our code. For example, this allowed the GUI to be written completely independently from the RemoteObject connection code and the central controller. After deciding to structure our project like that, team members could make their own smaller design decisions.

If we were to do the project again, we would approach the GUI design differently. We chose to use wxPython for our projects, which produced good looking GUIs, but our unfamiliarity with the library was a stumbling block in our development. We managed to create our individual GUI components smoothly enough, but managing multiple GUI elements in our project was an arduous task. For example, this prevented the drawing support from reaching its full potential. Ideally, we would have been more comfortable with the GUI library we picked.

3.3 . DIVISION OF LABOR

One of the biggest challenges in our project was identifying incremental steps of development so that each member of our team had an evenly distributed amount of code to write. Because our project was divided into three separate components-the server, client, and GUI, we were initially able to divide labor based on the components that had to be made. While we initially found it more convenient to collaborate on making the framework for each component, as it would have been incredibly difficult to write each piece individually and then put them all together.

Once the initial framework of each component was written, we were able to divide tasks more efficiently. Features like revisions and drawing were assigned to specific team members to develop and incorporate into the frameworks. As components, particularly the GUI, got more features added on, another team member was tasked with cleaning up the code and ensuring the features worked smoothly with each other.

Certain components of our project, such as the real-time updating, were more difficult to implement than others.

In hindsight, labor could have been divided more efficiently so that team members working on less challenging tasks had more to do. Nevertheless, we all learned a lot about the challenges of team-based software development.

3.4. BUG REPORT

One of the most difficult bugs we had to deal with was the compatibility between our remote data object and Pyro. Pyro can only support remote objects that are pickleable. Pickling is a standard Python mechanism that allows serialization. Picklable object include booleans, numbers, strings, tuples, lists, sets, dictionaries, functional, class, and certain objects (see http://docs.python.org/library/pickle.html for full documentation). Essentially, if the object was not pickleable, Pyro doesn't work.

When we were trying to expand our remote data object beyond a simple string to include canvas drawings, we continued to get Python errors that our data object was not pickleable. Since this was new territory for us, we thought that the issue was a semantic error written by us. We continued to look for where and why we could not send the object containing the string and the drawing.

Eventually, we realized that we were getting a pickling error from Pyro. We investigated what pickling was, and we realized that each line segment in our canvas included three data attributes: two point objects and a 'Pen' object. The point objects allowed our GUI (wxPython) to draw the canvas. However, the 'pen' object was a unique object that helped wxPython understand the features of the drawing, such as the color, the line width, etc. We realized that the 'Pen' object was not pickleable. Thus, Pyro was unable to make this into a remote object.

To fix this problem, we had to create our own data representation. This took us about a day's worth of work, most of which was figuring out the problem. Our new data representation of the canvas includes only the point objects, which allows us to create a remote data object from the canvas and still convey the drawing. One drawback of this fix was that none of the formatting of the picture is transmitted. Thus, we are currently only able to send drawings in a fixed color and a fixed line width.

In retrospect, we could have found this bug faster if we read the Pyro documentation more carefully, and understood the limitations of what it could and could not make into a remote object. If we had done this earlier, we would have realized that certain unique data objects are not pickleable, and thus, are not compatible with Pyro.