**CS560 Project 1 Report**

John Till and Peiyong Chen

**Overview**

The goal of this project is to implement an HTTP server which supports directory listing, static files, and CGI scripts. To motivate the server, we decided to host a collection of poetry. We use Java and its standard library to implement the server. The main testing environment is on Linux (a Ubuntu virtual machine). The server has also been tested on Window 10, which supports all features except for the CGI script because it is implemented by a Linux shell script. In this report, we will show how we run the code and how we implemented the required features. The Java code “FileServer.java” and hosted server files are included in our submission.

To test the server, compile the server code with “javac FileServer.java”, then run the server with “java FileServer 80”. By running the code, a server socket is linked to the port 80, as shown in Figure 1. You can connect to the server in a web browser on the same machine by entering “localhost” in the address bar.

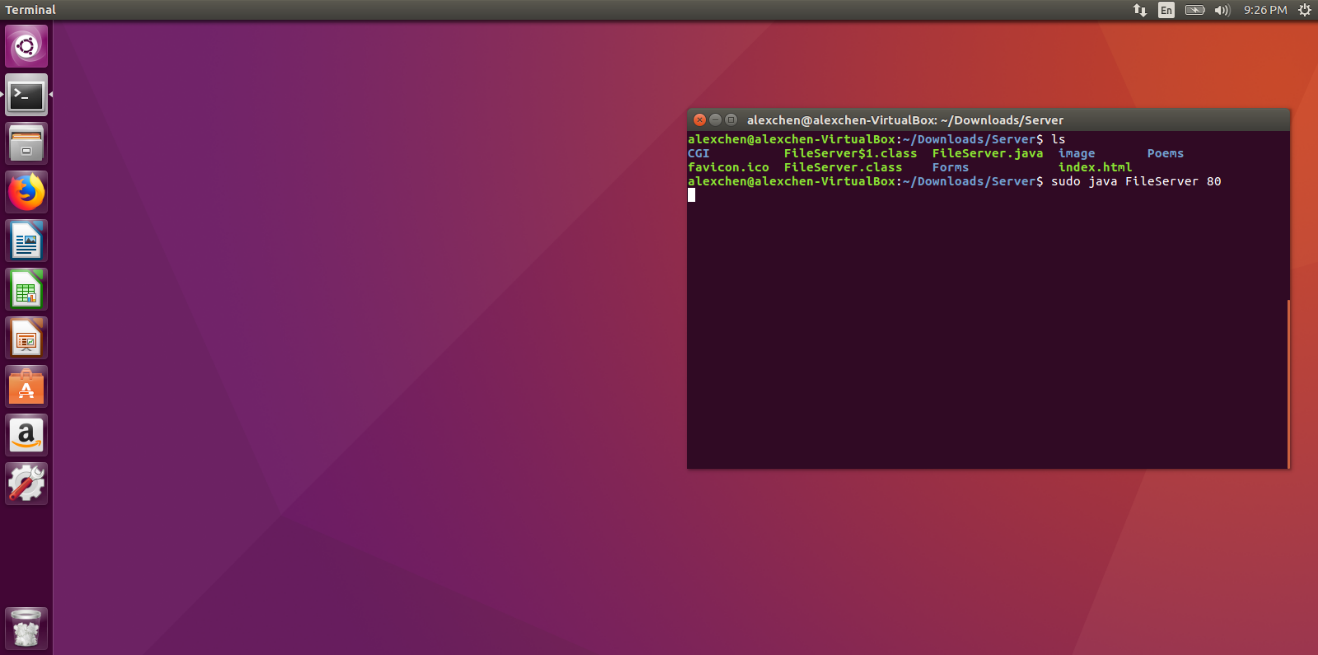


Figure 1

**Design and Results**

**Feature 1: Single connection mode and multiple connection mode (parallel with multiple threads)**

The code to handle connections uses Java ServerSocket and Socket objects as shown in the Java Socket Tutorial found on the course schedule [1]. A ServerSocket object is created once in the lifetime of the server, and each client connect has a Socket object created by calling “.accept()” on the ServerSocket object. Whenever a Socket is created, we call a custom method “resolveClientConnection(Socket clientSocket)” to handle the connection. Thus the multithread feature is trivial- when the server is in multi-thread mode it calls “resolveClientConnection” in a new Java Thread instead of the main one. The steps we describe here are implemented in the following main function:

public static void main**(**String**[]** args**)** **throws** IOException **{**

//Default to single-thread mode if only a port number is provided

boolean multithread\_mode **=** **false;**

**if** **(**args**.**length **==** 2**)** **{**

//Allow a second command line argument for multithread mode

multithread\_mode **=** Boolean**.**parseBoolean**(**args**[**1**]);**

**}else** **if(**args**.**length **!=** 1**){**

System**.**err**.**println**(**"Usage: java FileServer <port> <multithread mode>"**);**

System**.**exit**(**1**);**

**}**

int portNumber **=** Integer**.**parseInt**(**args**[**0**]);**

ServerSocket serverSocket **=** **new** ServerSocket**(**portNumber**);**

//Build directories of static files in html once on startup

generateHtmlDirectories**(**"/Poems"**,** ""**);**

boolean keep\_running **=** **true;**

**while(**keep\_running**){**

Socket clientSocket **=** serverSocket**.**accept**();**

**if(** **!**multithread\_mode **){**

//Resolve the connection in the main thread

resolveClientConnection**(**clientSocket**);**

**}else{**

//Fork a new thread and continue to wait for connections

Thread t1 **=** **new** Thread**(new** Runnable**()** **{**

public void run**()** **{**

resolveClientConnection**(**clientSocket**);**

**}**

**});**

t1**.**start**();**

**}**

**}**

**}**

We have run the server in both modes without incident.

**Feature 2: HTTP GET requests with query and header parsing**

We relied on basic String manipulations to implement GET requests and response messages. The server uses a Scanner to check that the HTTP message starts with the word “GET,” and sends a “Method Not Allowed” error response for any other types of client messages. If the type of message is “GET”, the server scans the next word to find which resource is being requested. The server handles several cases, which include:

* The resource requested is the server homepage, “/”
* The resource requested is a form submission, “/Forms/request\_form? … ”
* The resource requested is a CGI script, “/CGI/guest\_addition? … ”
* The resource requested is a static file which exists in the server subdirectories, and has a whitelisted extension type

Based on the case the server may perform some intermediate work, and eventually will send a response.

As shown in Figure 2, after a client sends a request, we can see the HTTP requests and headers in the console.

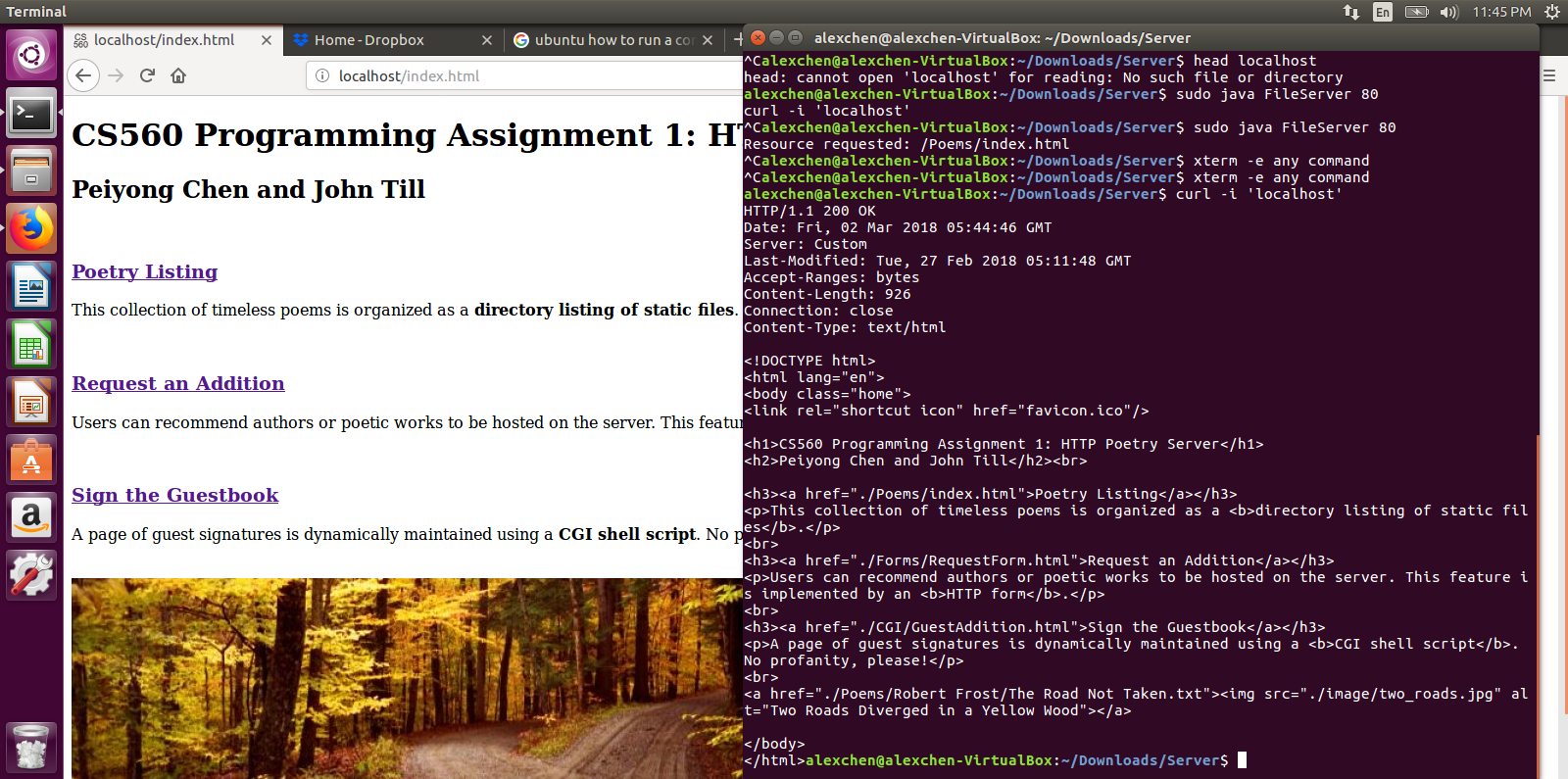


Figure 2

The files in our local server are listed in Figure 3. Then we go to the Firefox browser and type in “localhost” to send out a request to connect the server. In our browser we can view the content “/index.html” which is sent back from the server after the request from client (us). The browser also fetches any images referenced in the HTML file, as well as the “favicon.ico” to display in the tab. The browser view is shown in Figure 4.

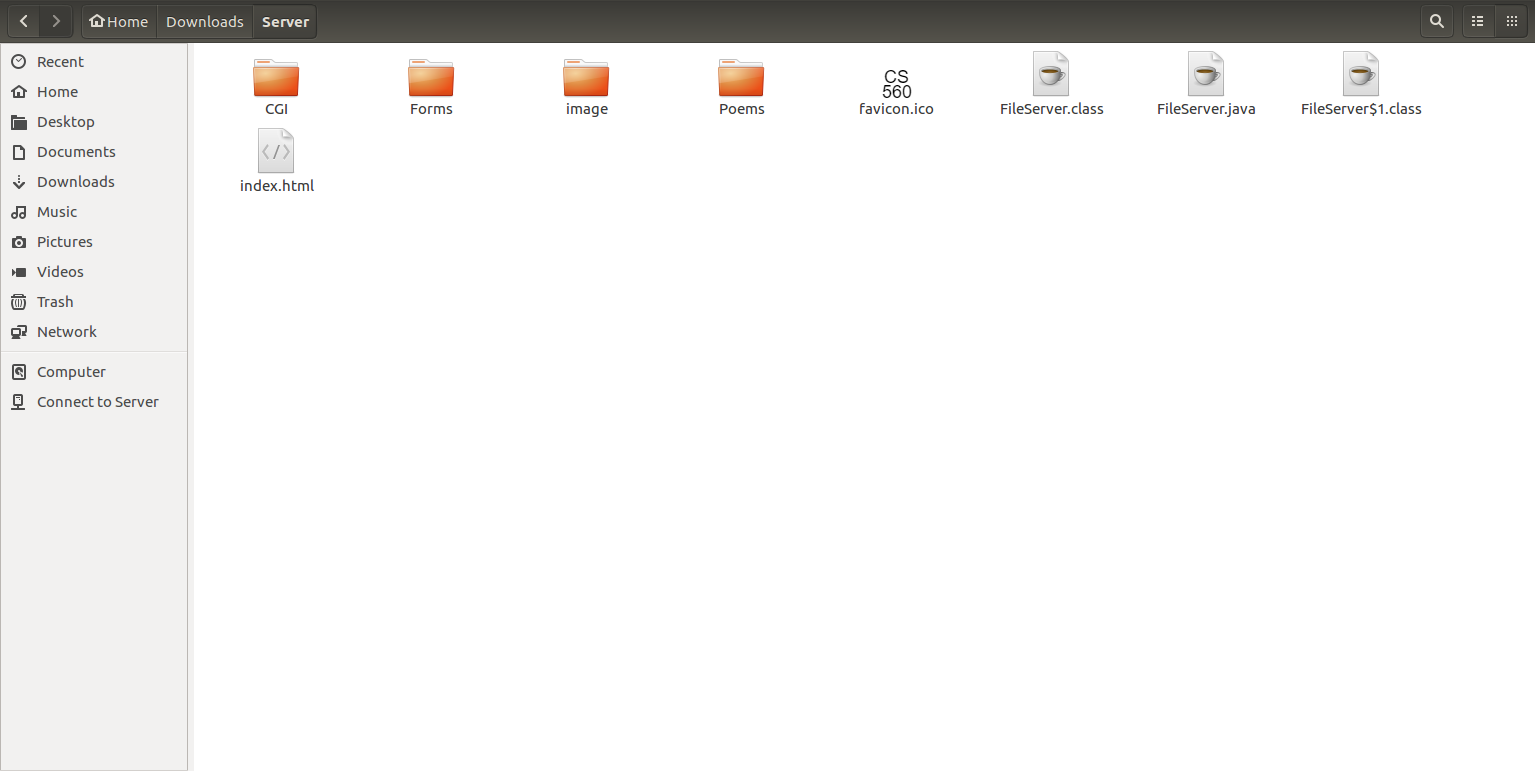


Figure 3

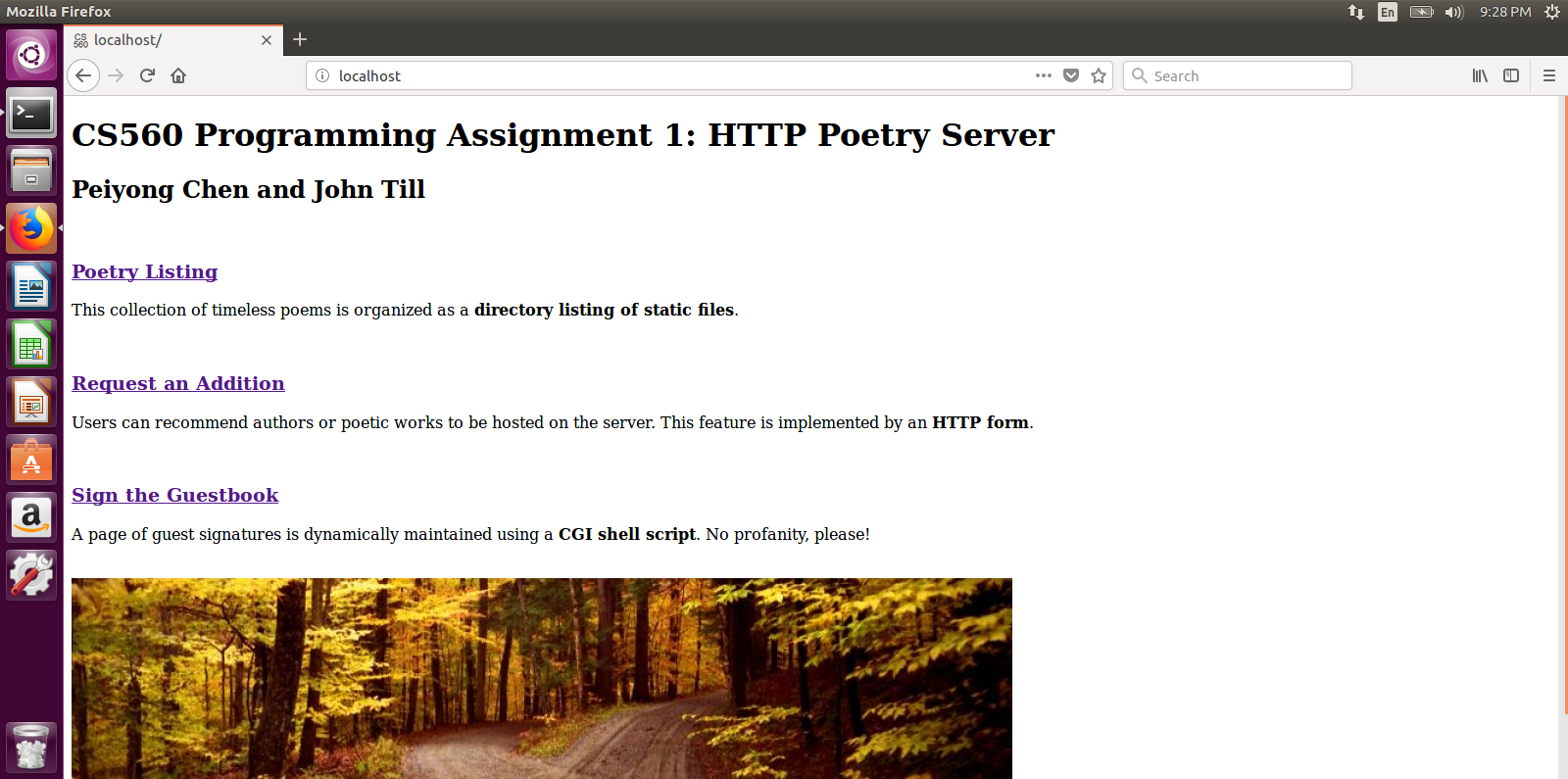


Figure 4

**Feature 3: Automatic directory listing**

On server startup, the contents of “/Poems” are recursively scanned and recorded in “index.html” files which can be sent to clients. This allows clients to browse through our directory structure.

As the client, if we left click the link “Poetry Listing” from the main page, we can go to Directory Contents, which will list all the poetry documents of the local server, as shown in Figure 5.

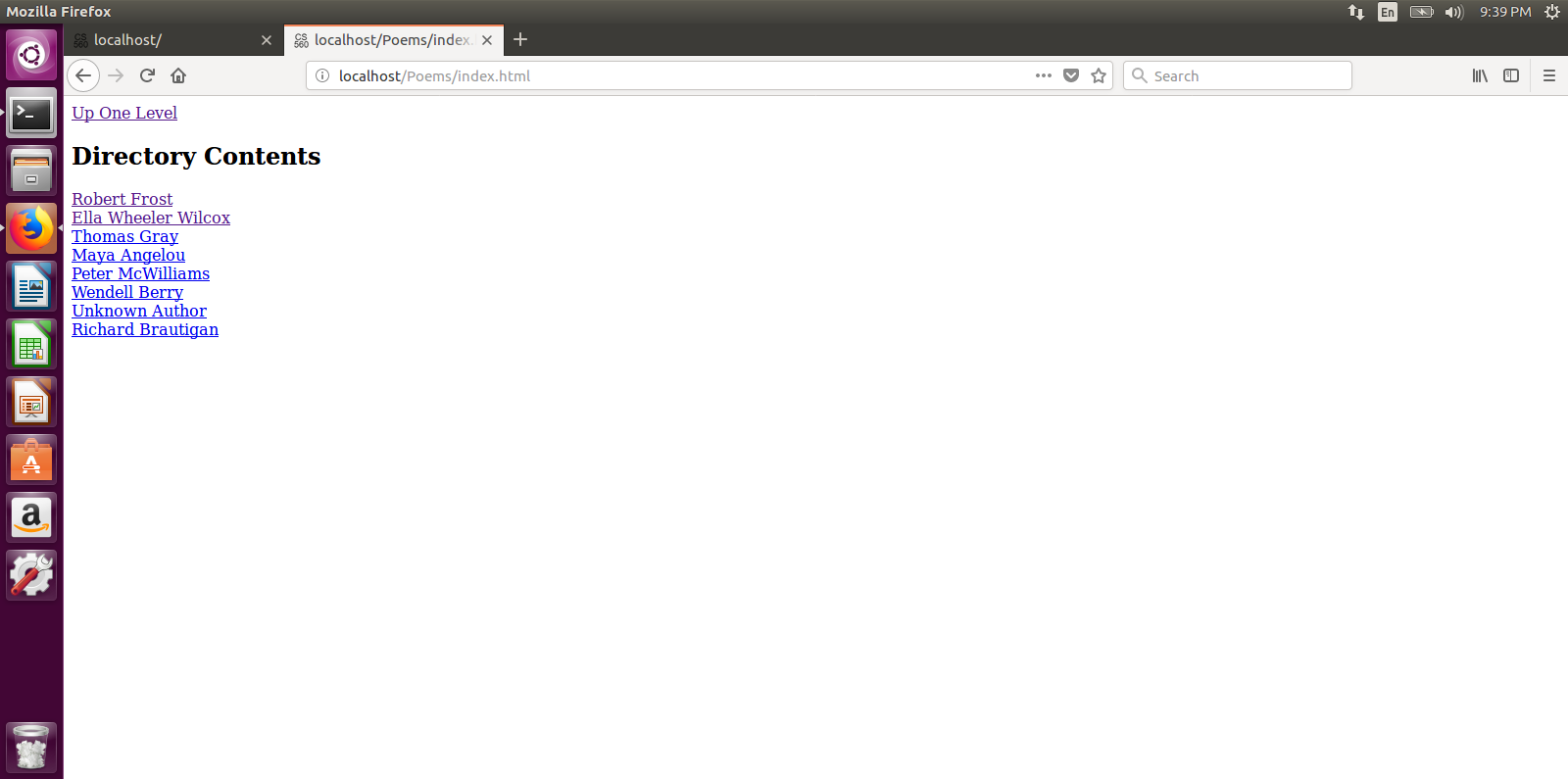


Figure 5

**Feature 4: Static file transport**

We implement a method

“convertResourceToResponse(String resource, PrintWriter print\_out, OutputStream byte\_out)”

to convert the specified resource into an HTTP response. Resource types must be in a whitelist of “txt”, “jpg”, “ico”, and “html” files. This means a client cannot request “FileServer.java” and see our source code. If the request resource exists and is in the whitelist, an HTTP response header and the resource are sent to the client using the output streams.

**Feature 5: Basic CGI support by running a sample CGI script on the server side**

The CGI script implements a guest book feature for the site. A single name is received by an HTTP form. This name is the input for a shell script which appends to name to a local file “guestbook.txt” then shows the whole guestbook in a dynamically generated HTML response.

Screenshots are shown in Figure 6-9. If we type in the “User Name”, such as “Alexchen”, the name “Alexchen” is attached to the tail of the “Guest Book” list. If we further type in the “Alexchen1”, “Alexchen1” will be attached to the tail of the list. The “Guest Book” file is located in the Server. By this we write our input to a local file.

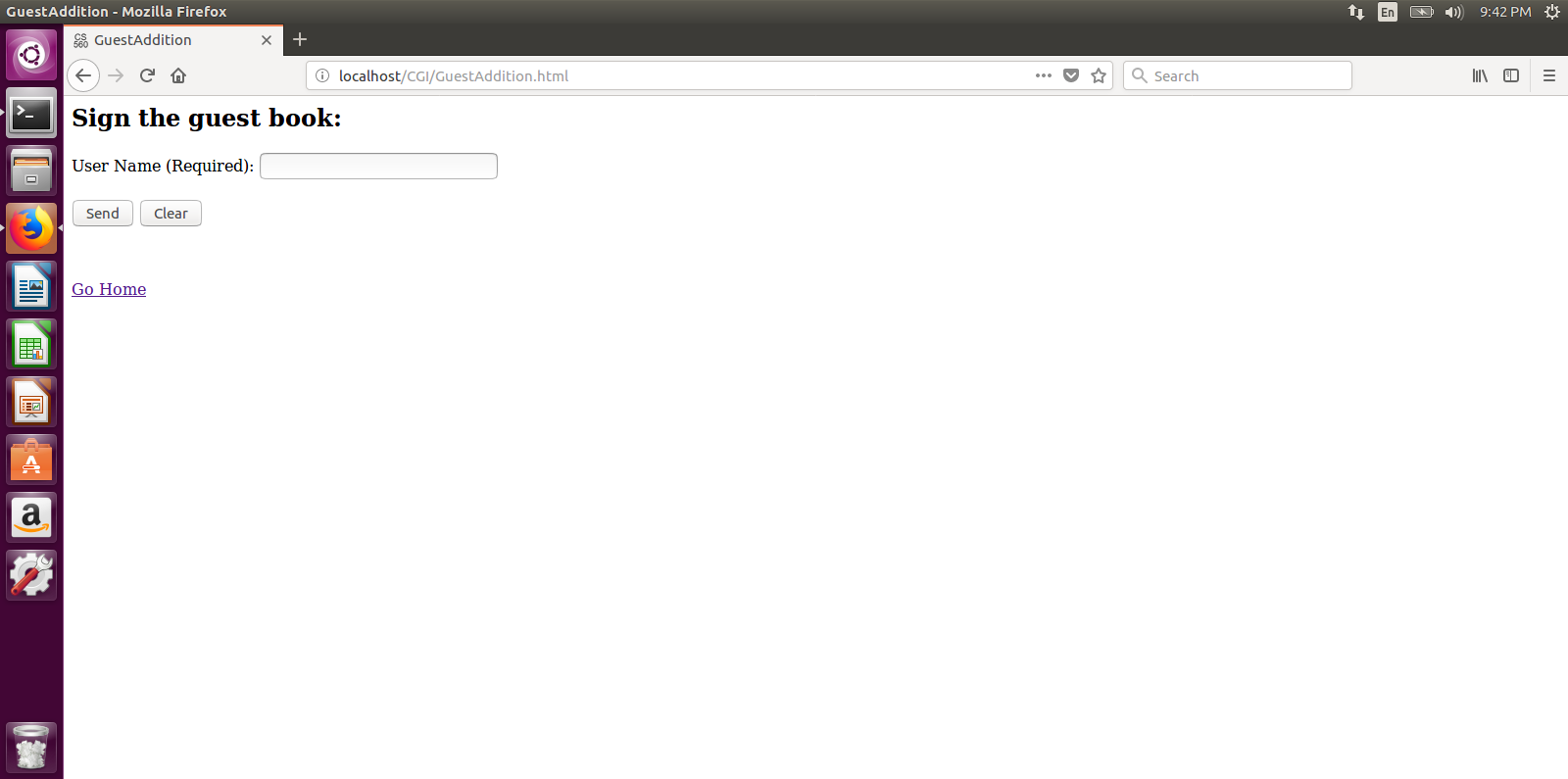


Figure 6

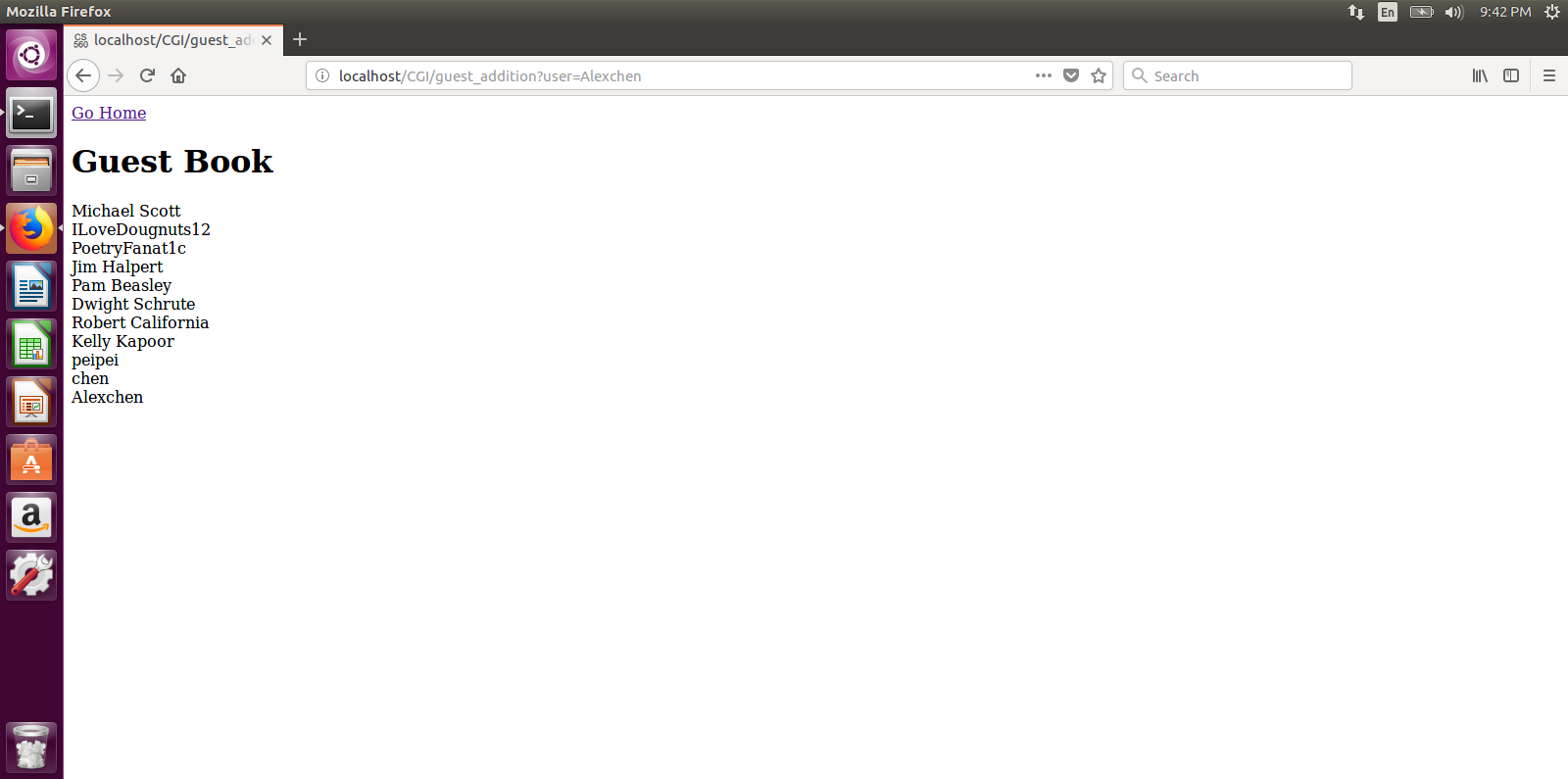


Figure 7

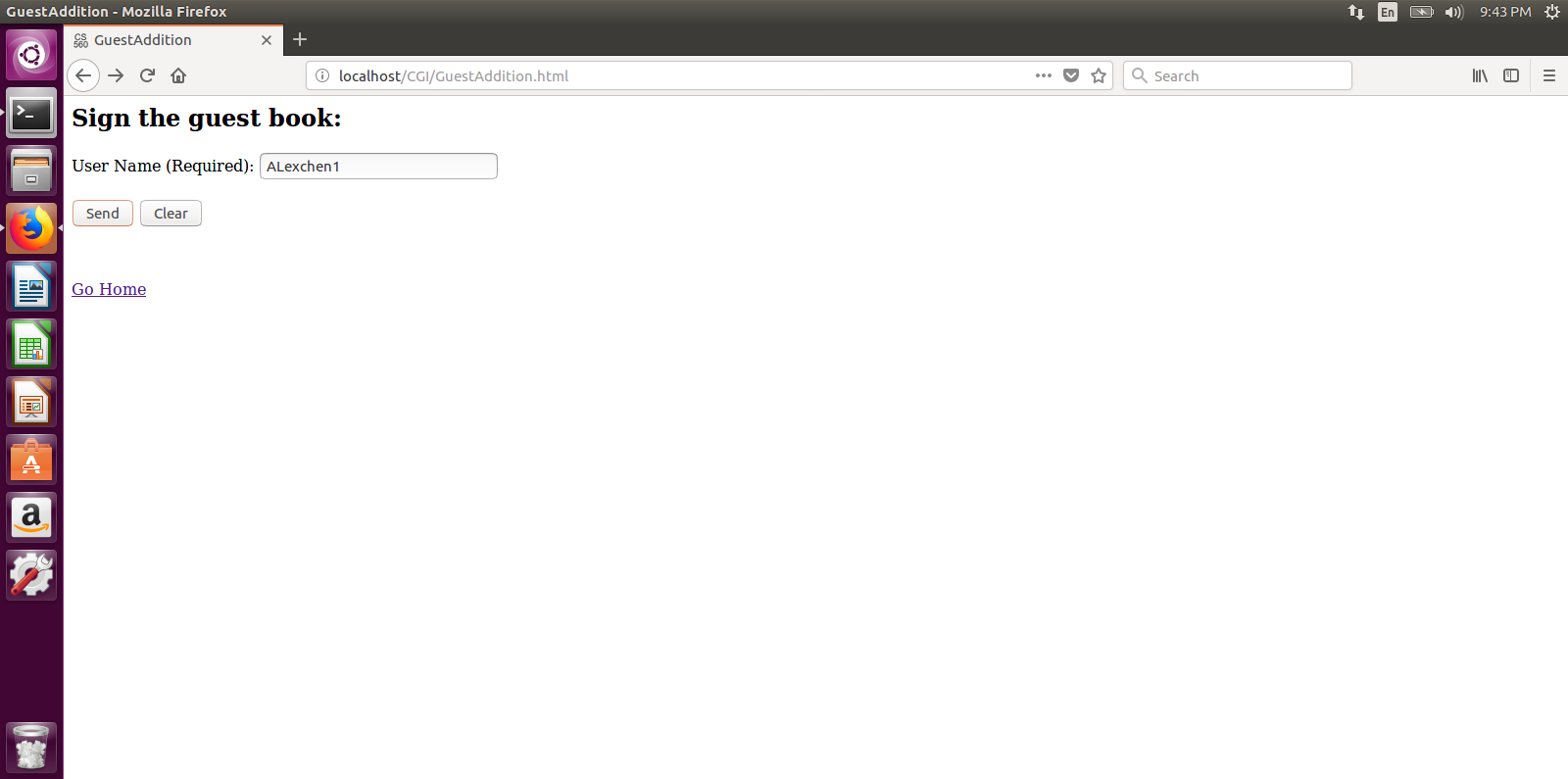


Figure 8

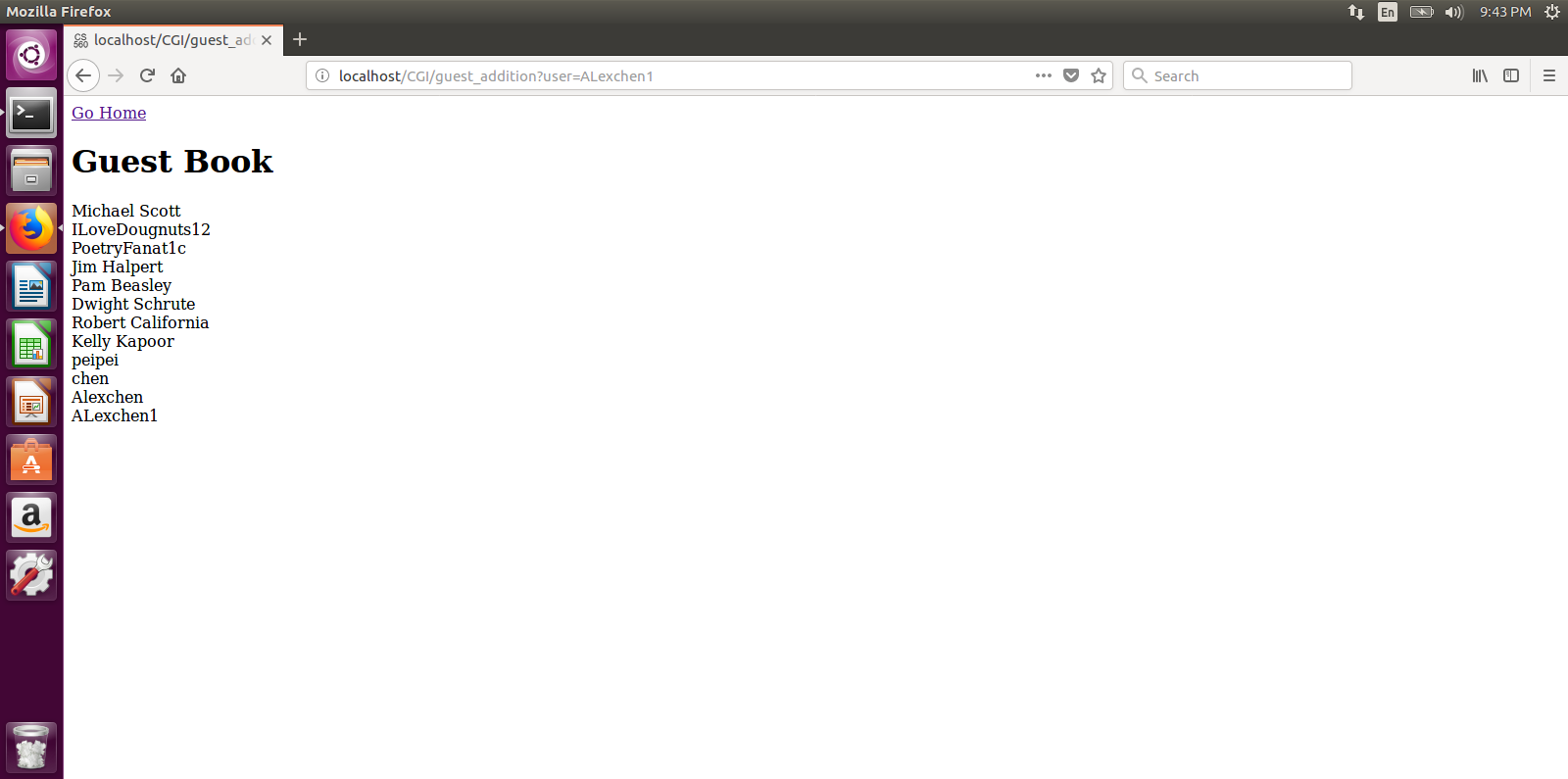


Figure 9

**Extra credit: Form submission with results written into a local file**

We implemented another form so that users may request additional poetic works to be added to our collection. This is implemented by an HTML form, and the results are appended to a local file “Suggestions.txt.” The client receives a response of a static confirmation page.

If we click the “Request an addition”, we will go the page which is implemented by a HTML form. We can input the author’s name and get a confirmation response. For example, if we type in “Robert Frost” and enter, the server will send a response “Thanks for your input”. Both are shown in Figure 10 & 11.

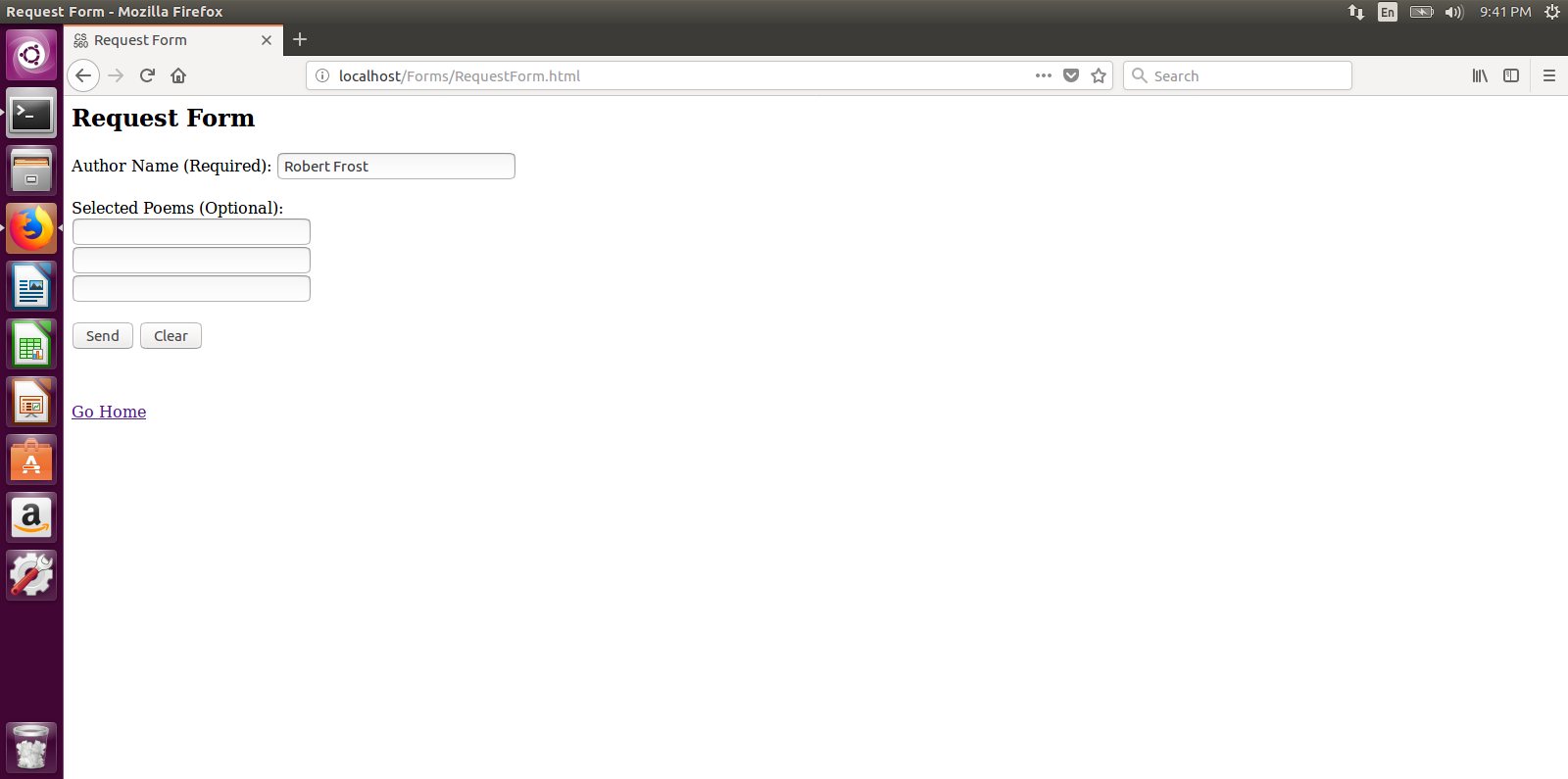


Figure 10

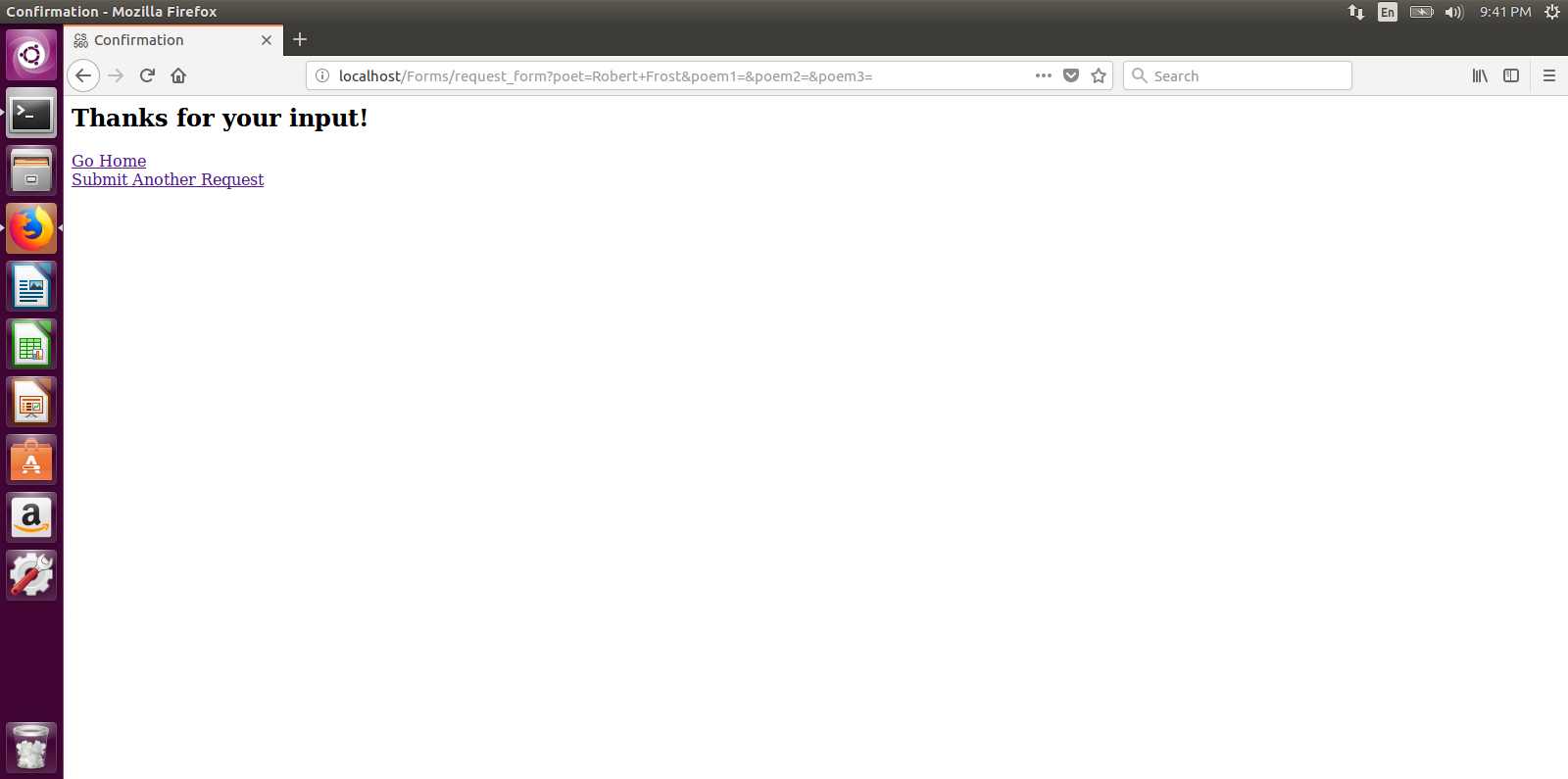


Figure 11

**Further Testing: Network connection**

We also connected to the server from a different machine on the “ut-open” network by specifying the server’s ip-address. This test was performed in Google’s Chrome web browser, and the result was successful. The test result is shown in Figure 12.

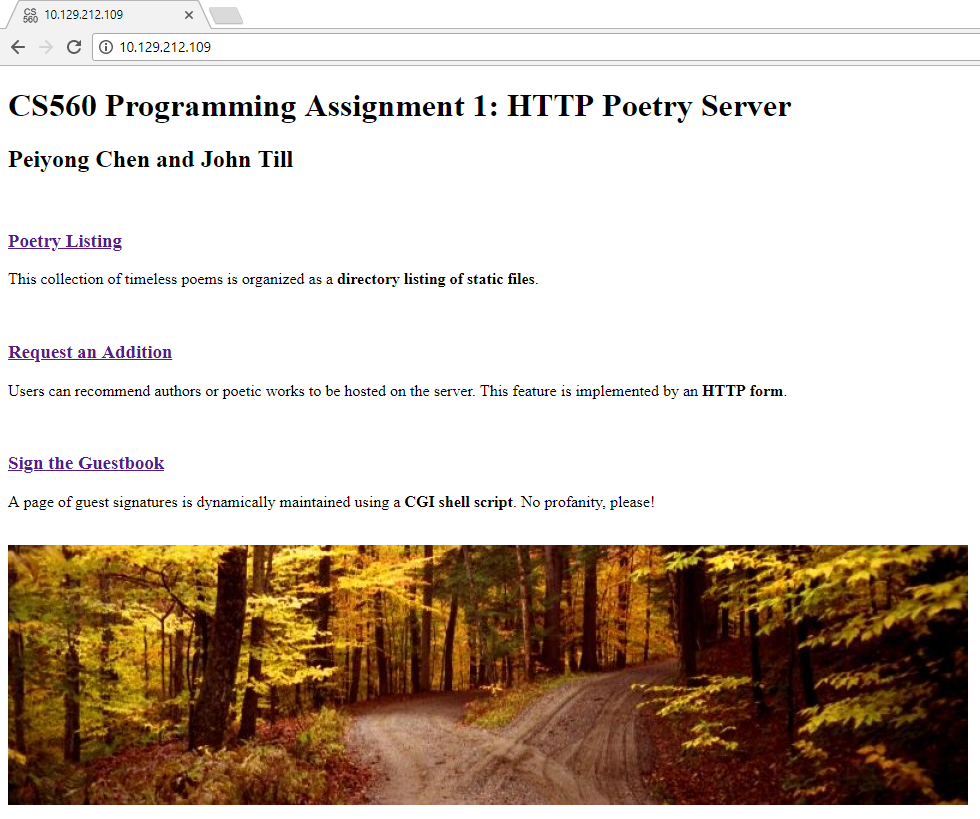


Figure 12

The server prints the client messages to a console (shown in Figure 13):

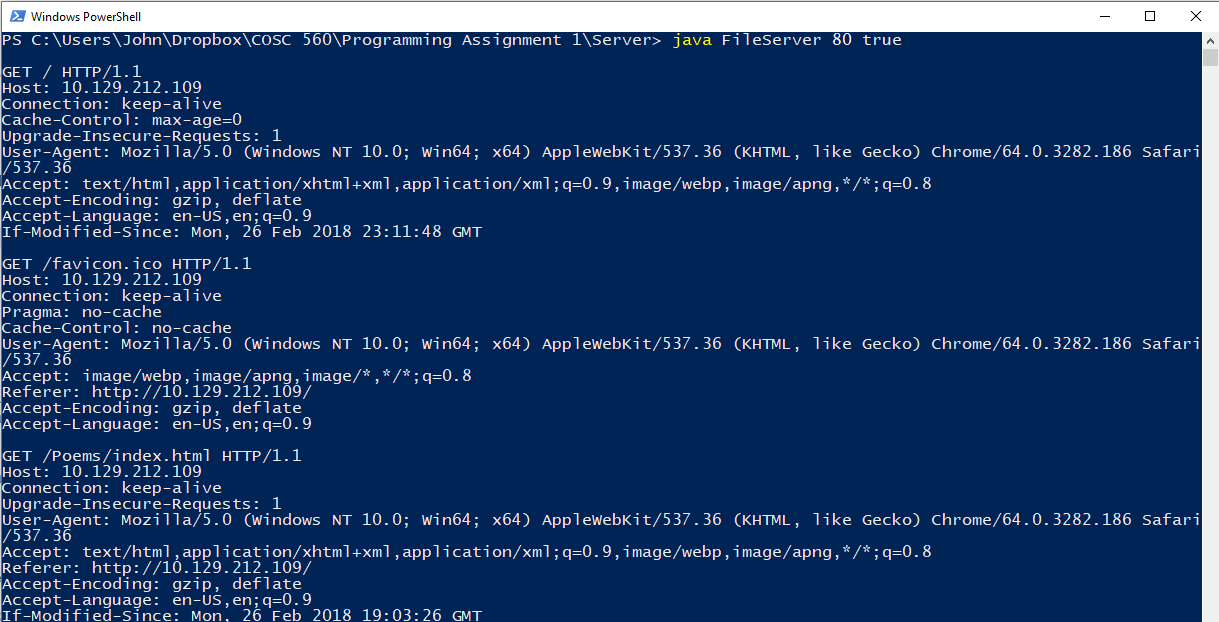


Figure 13

**Works Cited**

[1] “Lesson: All About Sockets.” Web address: docs.oracle.com/javase/tutorial/networking/sockets.