Final Project (Word Searcher) – COP 2805

Background:

An old concept in computer science is to have a more powerful computer perform computations at the request of a slower computer. This is more popular in modern times with web services which use XML or JSON as the communication protocol. We are going to mimic this by creating a service that searches through a large amount of text for occurrences of a word or phrase, though it won't be a true web service since the XML/JSON message formatting is not required and our server will not be connected to the internet.

Objective:

We are going to build an example server/client where the client sends a request to the server and receives a response. The server will represent a word searcher functionality where it loads in a large text file and searches for occurrences of a word or phrase within the array of strings. It responds with a list of integers which represent each line number that the word/phrase appeared in.

You will construct two applications: a client and a server. The client will contain a GUI where a user can type in a word or phrase and press a button to connect to the server. The client will display the list of results into a JList on the GUI. The server will accept connections and process the word search, responding with a list of integers for the client to then process.

Word Searcher (20 points):

I would recommend beginning with a class that runs alongside the server called WordSearcher. This class will take in a file name as its constructor and have a single method.

At construction the WordSearcher should open the file and read all of its contents into a class level List<String> object. This object will be populated by the files.readAllLines() function call.

WordSearcher should **NOT** be case sensitive. This is easily implemented by **making all strings upper case.** For the List<String> this can be done with the following code:

```
lines.replaceAll(String::toUpperCase);
```

where lines is the name of your List<String> object.

The method for the WordSearcher class will search the List<String> object for all occurrences of a particular String. The method's input would be a String parameter and it will return a List<Integer> which would be every index in which the string was found. An example algorithm to do the substring search is presented here in pseudocode:

```
searchString = searchString.toUpperCase()
for(i = 0 to lines.size())
      String str = lines[i]
      if(str.indexOf(searchString) >= 0)
            returnList.add(i)
```

where searchString is the String parameter passed into the method, lines is the List<String> object read from the file and returnList is the List<Integer> to be returned from the method.

This example uses the indexOf function which is available to all Strings. This searches a given String for the occurrence of another String and returns the index if it was found or a -1 if it was not found. This means if a given line within the List<String> object contains the substring it will return a value >= 0 and should be added to the return List.

Note: We are not concerned about multiple occurrences of the word within the same line. If it occurs once that line number is to be returned to the client.

TIP: I strongly recommend starting with this class and making sure you have it working. Provide it with the file name and an example word to search for and verify what it returns. I recommend doing this before worrying about the network/GUI components to the project.

Server (40 points):

The server will contain the main(String[] args) method and will instantiate an instance of the WordSearcher object defined above. After that it will open a ServerSocket and bind it to an available port. For my own system the port 1236 was open, but this may be different on your home computer.

The server will sit in a while loop performing the following steps:

1. Accept client socket connection
2. Set up input/output streams from client connection
3. Read in String from client
4. Pass String to WordSearcher object, receive List<Integer> output
5. Transmit List<Integer> to client
6. Close Connection
7. Return to top of while loop

Recommended communication format is detailed below.

If you want to include a special “shutdown” command that causes the server to exit its while loop that is fine. You may also have while(1) to make a loop that runs forever and must be shut down manually in Eclipse.

Client (40 points):

The client will create a graphical user interface (GUI) for the user to interact with. It will contain a text field where they can enter the word or phrase they want to search for, a button to transmit the text field string, and a Jlist to display the results from the server. An example is shown in Figure 1 with results having already come in from the server:

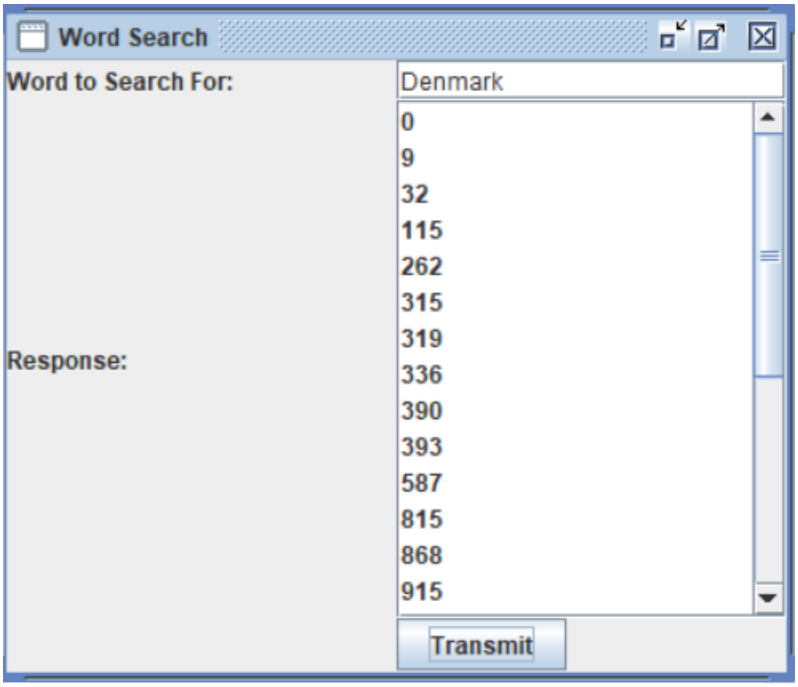

Figure 1. Example GUI with results from Server

For the JList you can use a DefaultListModel<Integer> to control the contents of the list. This would be a new public object for your JFrame that looks like:

```
public DefaultListModel<Integer> listModel;
```

When you instantiate the JList you do so with that object, such as:

```
new JList<Integer>(listModel);
```

This will allow you to directly clear the JList by calling listModel.clear() and add to the JList by calling listModel.addElement().

You will need an ActionListener tied to the Transmit button that kicks off the following steps when the button is pressed:

1. Clear the JList in the GUI
2. Read the string from the text field
    - In the example above the string was “Denmark”
    - Case sensitivity should not matter. That is, “denmark” will receive the same results.
3. Open a socket to the server
4. Send the string to the server
    - Add a “\n” to the string if you are following the recommended communication format.
5. Read the results from the server
    - If following the recommended communication format this will involve a while loop that converts each String from the server into an Integer.
    - Add the Integer values to the JList as they come in.
6. Close the socket to the server

Recommended Communication Format

You may choose whatever you want to communicate between the server and client, so long as it works. If you are already familiar with XML, HTTP or JSON in Java then feel free to use those protocols. Personally, I found it easy to work with Strings going back and forth. Figure 2 shows an example of the communication method I am recommending.

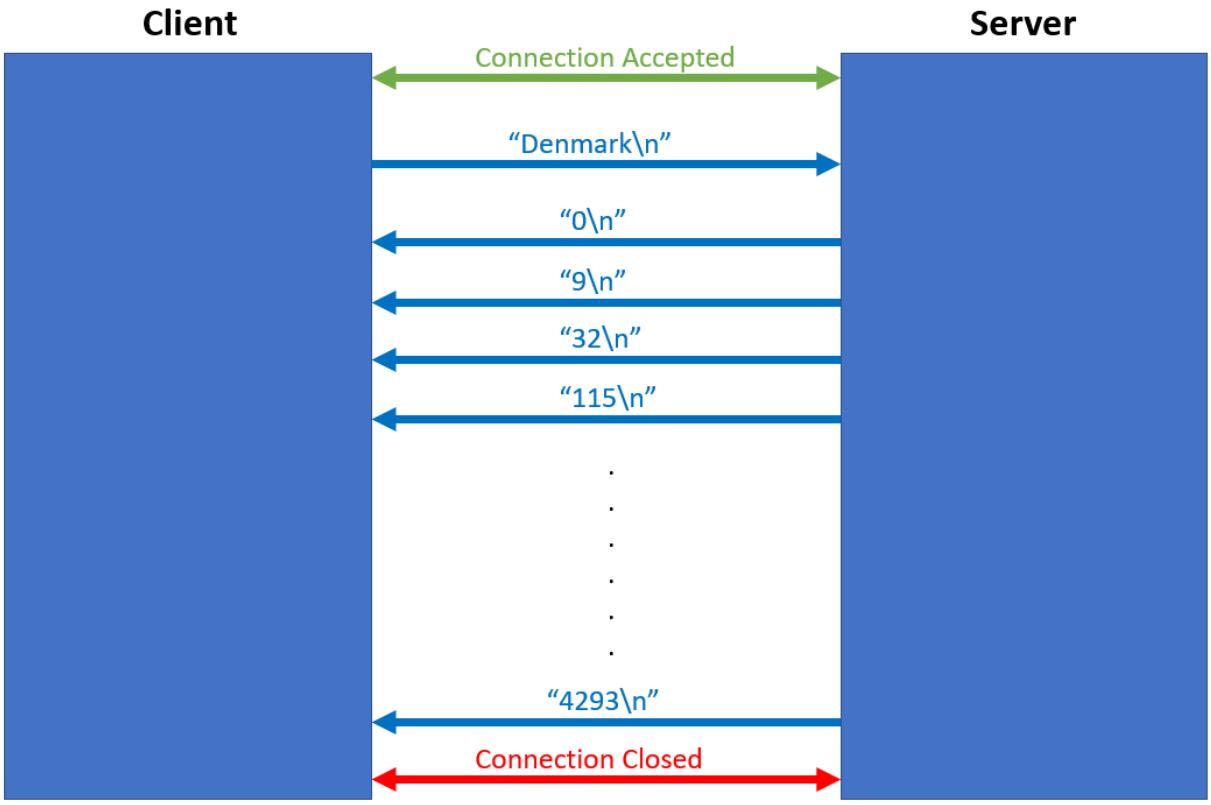

Figure 2. Example communication method. A single string ending with "\n" is sent to the server and multiple strings ending with "\n" that all contain a number are sent in the response.

By converting the Integers in the List to a String with \n appended to it, the client could call .readLine() from a BufferedReader to get the next value to place into the List. When the server has finished sending messages back to the client the readLine() call will return a null value.

For my example I used the IP address 127.0.0.1 to connect the client and server and port 1236 as it was open on my computer. You may want to try other ports, but the IP address should remain localhost (127.0.0.1).

TIP: To turn an InputStream into a BufferedReader you must apply it to a new InputStreamReader. For example:

```
BufferedReader reader = new BufferedReader(new
InputStreamReader(client.getInputStream()));
```

This will allow you to then call reader.readLine() to receive a new string from the socket. **Note that all strings must end with '\n' for this to work.** You may use any method that you want, but I found this to be the simplest.

TIP: If the server is sending several strings to the client, you will need a while loop to catch each one. The final read will be a null string which you can use to terminate the loop. In pseudocode this process on the client would look like:

```
String response = ""
while(response != null)
      response = reader.readLine()
      if(response != null)
            listModel.addElement(Integer.parseInt(response))
```

where reader is the BufferedReader set up to the socket's InputStream and listModel is the DefaultListModel that controls the JList in the GUI.

TIP: On the server side you are basically turning a List<Integer> into a series of strings. This can be done with pseudocode that looks like:

```
for(Integer x in returnList)
      String response = x.toString() + "\n"
      output.write(response.getBytes())
```

where returnList is the List<Integer> that was obtained from the WordSearcher class and output is the OutputStream from the client socket.

TIP: As shown in the example above, you can send a String directly through the OutputStream by calling the .getBytes() method. This returns the string as a byte array which the OutputStream can use in its write method.

General Advice

If you get an error from the server that it cannot bind that port or address because it is already in use, that means something is already connected to the IP/port combination you are using. If you accidentally left a copy of the server running in the background you will need to navigate to the instance that is running and close it down manually. If that does not work close down the Eclipse and restart it. If the

binding still fails, try a different port like 1236, 1237, 1238. It is also possible your firewall may prevent access though that is unlikely if you remain on the localhost IP.

PLAN OUT YOUR STEPS AND TEST AS YOU GO!

I cannot stress enough that you need to strategize on how you will tackle this project. Here are some steps I followed as an example:

1. Began with main function in the Server and a separate WordSearcher class.
2. Coded the WordSearcher initialization and FindWord method.
    a. Verified from the main function in the server that they work with an example word.
3. Created the Server code to wait for sockets.
4. Created the Client code to connect to the server and sent a default string to the server.
    a. Starting out this was just in the main method, no GUI code added yet.
    b. Verified the server got the connection and responded correctly.
    c. I commented out this code with the intention of copying it into the ActionListener once the GUI was built.
5. Constructed the GUI around the client code.
    a. I copied the test code from the main method as a skeleton of what to use in the ActionListener event.

Final advice, this is not a lot of code, but it is more than we have done in the other homework assignments. To give you an idea here are my line counts for each module:

- Word Searcher - ~25 lines
- Server – ~50 Lines
- Client - ~130 lines
- Total: roughly 205 lines of code

Not that you have to match these numbers, but if you are approaching several hundreds of lines of code, you may be over-thinking the problem.

Submission

Turn in the .java files by the due date to receive credit. Primarily I will be looking through your code and attempting to build the project on my own.

Fill free to write notes about what you know works or doesn't work. You may also want to include screenshots of the program working in case there are small compiler differences that I need to work through.

Submit early to receive early feedback. You may resubmit up to the due date. Any submissions after the due date will receive a 0.

DO NOT WAIT UNTIL THE LAST MINUTE. START NOW.