EECS 489 Lab 1: Remote Image Display

This assignment is due on Friday, 16 Jan 2015, 10 pm.

~~tl;dr~~

As this is the first assignment involving programming in this course, this description is a bit long. Please read the whole thing carefully as it sets up how the labs and programming assignments for the rest of the course will be run: how you can access the support code, the command line options used by the support code, how to test your code, what you're expected to turn in (and not turn in), and how to submit your code will all remain more or less the same for the rest of the term.

Introduction

This lab is a refresher for socket programming. I assume you have done some socket programming prior to this course, either as a project in EECS 482 or equivalent experience. I further assume that you know how to build a socket program on your development environment of choice, as covered in Lab 0. If your socket programming skill is rusty or shaky or if you're still unsure how to build a socket program using your favorite build tool, this lab gives you a chance to shore both up. In completing this lab, you may consult the sample code `server.c` and `client.c` from Lab 0.

In this lab, we will be building a simple client-server remote image viewer. When completed, the server will provide an image to the client, which the client then displays using OpenGL. This lab should provide you with direct feedback on its fitness: a correctly completed project will display a recognizable image on the client. Unfinished or incorrectly written projects probably will not. We will use this remote image display tool throughout the term to help you visualize the effects of various network protocols. Hopefully the visualization will help you understand the network protocols. It could also help you detect some bugs your code may have.

You're provided with a skeleton code consisting of `netimg.cpp` and `netimglut.cpp` for the client of Netimg, `imgdb.cpp` for the server, a common header file, `netimg.h`, and a reference Linux binary executable of the server `refimgdb`. You should be able to run your client against the provided server. The provided `Makefile` builds `imgdb` and `netimg`. You can download the support code from the Course Folder.

Even though the files provided are C++ files, I found no compelling reason to use objects/classes in this lab, so the code is more or less straight forward C. We'll use more C++ features in future assignments. You can search for the string "YOUR CODE HERE" in the code to find places where your code must go. However, you're *required* to read all comments in the source files. There is information in the comments on the expected behavior of the functions and assumptions about the structures and variables used that is not necessarily spelled out in this document. It may also be useful for you to read this document to the end first before starting to code. You may find the section on how to test your code useful, for example.

Task 1: Client Side

Your first task is to write the client code. The client takes a single required command line argument:

```
% netimg -s <server>:<port> -q <image>.tga [-v <version>]
```

where '%' indicates a Unix C-shell prompt, which you don't type in. The `-s` option tells `netimg` which `server` to connect to. Unless your DNS resolver has been set up to search for the domain of the server (as shown in lecture), you must provide the fully qualified domain name (FQDN) or IP address of the server, along with the `port` number the server is listening on, separated by a colon. The angle brackets ("< >") indicate that you are to provide the actual values for the required arguments. You don't enter the angle brackets themselves when you run the program. See the testing section on how to test your code.

The `-q` option tells the `netimg` program which image file to query the server for. You replace "<image>" with the name of your TGA image file, again you don't enter the angle brackets themselves. The file must be a TGA image file. Two sample TGA image files are provided for your use. You can view them using Apple's Preview, GIMP, Photoshop, or other image viewing tool. If you have a JPEG or PNG or other image format you would like to use, you need to convert them to a TGA file first. For example, the graphics tool `convert` that is part of the ImageMagicK package that runs on both Linux and Mac OS X can do this conversion. If your image is displayed upside down, `convert` can also flip it for you.

The `-v` option allows you to change the version number of the `iqry` packet sent. You should use it to test whether your server function `imgdb_recvqry()` is checking the version field of all incoming `iqry` packet correctly. The square brackets "[ ]" in the command line specification above indicate that the second argument is optional, you don't enter the square brackets when running the program.

You can search for the string "Task 1" in `netimg.cpp` to find places where "Task 1" related code must be filled in. The function `netimg_sockinit(sname, port)` connects to the server specified by `sname`, at the specified `port`. This is a pretty straightforward task not much different from the code in `client.c` of Lab 0. First create a new TCP socket and store the socket descriptor/handler in the global variable `sd`. Next initialize the socket with the server's IPv4 address and port number. Finally connect to the server and return to caller. If there were any error during the connect process, terminate the process. The `netimg_sockinit()` function has been commented such that it should be clear where you need to make which socket API call. Depending on the amount of error checking you do, this part of the lab should take about 10 to 13 lines of code.

Next write the `netimg_recvimsg()` that receive the packet of type `imsg_t` from the server. Store the packet in the global variable `imsg`. The structure `imsg_t` is defined in `netimg.h`. It stores the specifics of the image, width, height, etc., that we'll need to display the image. You must check the version number of the incoming packet. If it is not `NETIMG_VERS`, you must return `NETIMG_EVERS` (both defined in `netimg.h`). (Your programming assignment will be tested for checking of packet version number, so get into the habit of checking it now.) This function should take about 5 to 7 lines of code.

Finally, you're to write 2 to 4 lines of code in `netimg_recvimg()` to receive the image file itself. You'll be using the global variables `img_size` and `img_offset` to complete this task.

That's it for Task 1. You will write a total of 17 to 24 lines of code. After completing Task 1, you should test your code before continuing to Task 2. See the Testing section below for some guidelines on testing your code using the reference implementation of the server.

As you will notice, we use OpenGL/GLUT to display the image received from the server. Needless to say, you're not required to know OpenGL. All the code you're required to write deals only with network programming. Nevertheless, you need to know how to build OpenGL/GLUT code. If you build your code using the provided `Makefile`, all you need to do is type `make`. If you want to build using an IDE, you need to add the files `netimg.cpp`, `netimglut.cpp` and `netimg.h` to your project (and `wingetopt.h` and `wingetopt.c` if you're on Windows). Don't add the other files. Then you need to tell your IDE that you want to add the OpenGL libraries. If you don't know how to do this, please follow the instructions in

Building OpenGL/GLUT Programs (note that this is the IDE instructions for EECS 487, not the one you used in Lab 0 for EECS 489). If you're curious about the OpenGL code, feel free to ask (or take EECS 487 ;-).

Task 2: Server Side

Your second task is to write the server code. On the command line, the server is run simply by typing `imgdb`. The server does not have any required command-line option. Upon start up, the server initializes a TCP socket and obtain an ephemeral port number from the kernel which is prints out to the console. Then it waits for a query packet from the client, loads the requested image from file from the same folder/directory it is launched from, sends the dimensions of the image to the client, and transfers the image to the client.

You can search for the string "Task 2" in `imgdb.cpp` to find places where "Task 2" related code must be filled in. Fill in the function `imgdb_sockinit()` by first creating a TCP socket. Then bind the socket with the port set to 0. and listen for connection, with listen queue length set to the macro `NETIMG_QLEN`. With port set to 0, the OS will assign a random, ephemeral port to the socket. Obtain from the socket the ephemeral port assigned to it so that you can print it out to the screen for the user to use with `netimg` to connect to the server. Finally, return the socket descriptor to the caller. The function `imgdb_sockinit()` has been commented such that it should be clear where you need to make which socket API call. It should take about 7 to 12 lines of code to complete this function. It is not that different from `server.c` of Lab 0.

The function `imgdb_accept(sd)` accepts the connection on the socket `sd`. Later on we will be closing the accepted socket after we have finished sending the image to the client. Set the socket option so that the socket will linger for `NETIMG_LINGER` amount of time upon closing, to give time for the image to arrive at the client. This part should take about 8 to 10 lines of code.

The function `imgdb_recvqry()` receives a query packet, checks that the packet is of the correct version and type and unpacks the queried filename from the query packet. You are to write this function. It shouldn't take more than 5-8 lines of code. The definition of the query packet, `iqry_t`, can be found in `netimg.h`. The query packet carries the filename of the image the client is searching for. You may want to consult the function `netimg.cpp:netimg_sendqry()` in writing this function. (If the query packet contains the wrong version number, the support code is set up to return `NETIMG_NFOUND`, which causes the client to inform the user that the searched for image is not found. This is not the best error message for this case, but it does make writing this function simpler for you.)

Finally, in the function `imgdb_sendimg(td, imsg, image, img_size)`, first send the image specification packet `imsg` to the client. This packet has been prepared for you in `imgdb_loadimg()`. Then send the image contained in `image` to the client. However, instead of sending the image in one go, which would be too fast to observe, you are required to send it in `NETIMG_NUMSEG(+1)` chunks of `segsize` (computed in the function for you). The local variable `ip` points to the start of the image. You're to send the image slowly, one chunk of size `segsize` every `NETIMG_USLEEP` microseconds. This part should take 3 to 5 lines.

That's all for Task 2. It should take about 23 to 35 lines of code in total. If you want to build using an IDE, you need to add the files `imgdb.cpp`, `netimg.h`, `ltga.h`, and `ltga.cpp` to your project (and `wingetopt.h` and `wingetopt.c` if you're on Windows). Don't add `netimg.cpp` nor `netimglut.cpp`. The two tasks together should take about 40 to 60 lines of code.

Testing Your Code

You can develop your code on either Linux, Mac OS X, or Windows platform. The easiest way to test your code is to run both the server and client on your local host, i.e., your laptop or the desktop in CAEN labs. For example, run `imgdb` on the command line if you're running Mac OS X or Linux and are using the `Makefile` to build the program. If you're using an IDE, add the command line option to the IDE before building and running your code. If you get a prompt from your security software whether to allow `imgdb` access to the network, say yes. Then run `netimg -s localhost:port# -q ShipatSea.tga` where `localhost` should be used instead of the host name, and "port#" is the port number reported by your `imgdb`. Again, if you're using an IDE, you need to enter the client's command line option to your IDE before running the server. The advantage of testing your code on your own machine is that you can run `tcpdump` or `wireshark` and see all the packets sent out from, or arriving to, your program, or none at all, as the case may be.

In addition to the skeletal code and `Makefile`, we've also provided an executable binary of `imgdb`, called `refimgdb` ("ref" for "reference"), that runs on the following four CAEN's linux hosts: `caen-eecs489p01.engin.umich.edu` up to `p04`. These four hosts can also be referred to using their alias/CNAME `eecs489p1.engin.umich.edu` to `p4`. And if you set up your DNS resolver to search for the `engin.umich.edu` domain, you only need to enter `eecs489p1` up to `p4` to address these four hosts.

The reference server is available in the Course Folder `/afs/umich.edu/class/eecs489/w15/FILES/`. It is a GNU/Linux executable, so don't try to download and run it on your Mac OS X, Windows, or other Linux flavors such as Ubuntu. To test against the provided reference server, run the server on one of the four eecs489 hosts that CAEN has set up for this course. These four hosts allow for connection to random ports from hosts connected through UMVPN or MWireless and from the machines in CAEN Labs. If you have problem accessing these eecs489 machines from over UMVPN, MWireless, or from CAEN Labs, please let the instructor know immediately. Don't run the server on any other CAEN hosts, such as the login servers (`caen-vnc*`) as their security settings have not been set to allow for connection to random ports. You won't be able to connect to the CAEN eecs489 hosts from your home or work computer unless you're running UMVPN. You also won't be able to connect from the MGUEST wireless network. Don't try to make the client on ITCS machines as they don't have OpenGL installed.

You should test your code as soon as you completed Task 1. Connect your `netimg` to `refimgdb`. For example, `ssh` to `caen-eecs489p01.engin.umich.edu`, then run:

```
eecs489p1% cd /afs/umich.edu/class/eecs489/w15/FILES
eecs489p1% ./refimgdb
```

You should see something like the following printed on the window:

```
refimgdb address is caen-eecs48p01.engin.umich.edu:54539
```

Note the hostname and port number `refimgdb` prints out, i.e., `caen-eecs489p01.engin.umich.edu:54539` in this case. Then on your local machine open up another window and run:

```
localhost% ./netimg -s caen-eecs489p01.engin.umich.edu:54539 -q ShipatSea.tga
```

You need to replace `caen-eecs489p01.engin.umich.edu:54539` with whatever was actually printed out for you by `refimgdb`. A window should now pop up and a grayscale image of a ship displayed slowly, one chunk at a time from top to bottom. (To see the actual download speed without our artificial slow down, set `NETIMG_NUMSEG` to 1 in `netimg.h`.) Depending on your system, the image may show up upside down, it's ok.

The provided code has been built and run on Linux, Mac OS X, and Windows machines. Support for

Winsock is included in the source code. To build the code on Windows platform, you need to add the files `wingetopt.h` and `wingetopt.c` to your Visual Studio project. Do NOT use any other libraries or compiler options that are not included in the `Makefile` already. Doing so would likely make your code not portable and if we can't compile your code, you will get 0 point.

Submission Instructions

Test your compilation! Your submission must compile **without** errors.

Your "*Lab1 files*" comprises your `netimg.cpp` and `imgdb.cpp` files.

To turn in your Lab1:

1. Email your IA/GSI the SHA1's of your *Lab1 files*. Use "EECS489: Lab1 Submission" as your email's "Subject:" line. Once you've sent in your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. Upload your *Lab1 files* by pointing your web browser to Course folder and navigate to your `lab1` folder under your uniqname. Or you can `scp` the files to your `lab1` folder on IFS: `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/lab1/`.
   This path is accessible from any machine you've logged into using your ITCS (`umich.edu`) password. Please report any problems to ITCS.
3. Keep your own backup copy! Don't make any more changes to the files once you've submitted your final SHA1's.

The timestamp on your SHA1 email will be your time of submission. If this is past the deadline, your submission will be considered late. You are allowed multiple "submissions" without late-policy implications as long as you respect the deadline. Try not to email your SHA1 to your IA/GSI until you've finalized your code. You don't want to annoy them.

Turn in ONLY the files you have modified. Do not turn in support code we provided that you haven't modified. Do not turn in any binary files (object files, executables, or images) with your assignment.

Do remove all `printf()`'s or `cout`'s and `cerr`'s and any other logging statements you've added for debugging purposes. You should debug using a debugger, not with `printf()`'s. If we can't understand the output of your code, you will get zero point.