UM EECS 489 Lab
Netimg: Remote Image Display

INTRODUCTION

This lab is a refresher for socket programming. I assume you have done some socket programming prior to this course, either in EECS 482 or equivalent experience. I further assume that you know how to build a socket program on your computing environment of choice, as covered in the first lab/discussion session of this course (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 affects of various network algorithms. Hopefully the visualization will help you understand the network algorithms we'll be studying. It could also help you detect some bugs your code may have.

You're provided with a skeleton code consisting of `netic.cpp` for the client of Netimg, `netis.cpp` for the server, a common header file, `netimg.h` and a reference Linux binary executable of the server `refnetis`. You should be able to run your client against the provided server. The provided `Makefile` builds `netis` and `netic`. You can download the support code from the course folder `/afs/umich.edu/class/eecs489/w14/FILES/`.

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:

```
% netic -s FQDN:port
```

where '%' indicates a Unix c-shell prompt, which you don't type in. The `-s` option tells the `netic` program which server to connect to. You must provide the fully qualified domain name (FQDN) of the server along with the port number the server is listening on, separated by a colon. See the testing section on how to test your code.

You can search for the string "Task 1" in `netic.cpp` to find places where "Task 1" related code must be filled in. The function `netic_sockinit(sname, port)` connects to the server whose FQDN is provided in `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 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 process. The `netic_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 `netic_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 `IM_VERS` (defined in `netimg.h`), you must terminate the process. (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 `netic_recvimage()` 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 `netic.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 `http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/ide/` (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. The server takes a single required command line argument:

```
% netis -f <image>.tga
```

The `-f` option tells the `netis` program which image file to send to the client. You replace "<image>" with the name of your TGA image file. 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.

You can search for the string "Task 2" in `netis.cpp` to find places where "Task 2" related code must be filled in. Fill in the function `netis_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 `NETIS_QLEN`. With port set to 0, the OS will assign a random, ephemeral port to the socket. So 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 `netic` to connect to the server. Finally, return the socket descriptor to the caller. The function `netis_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 `netis_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 `NETIS_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.

Finally, in the function `netis_imgsend(td, imsg, image, img_size)`, first send the image specification packet `imsg` to the client. This packet has been prepared for you in `netis_imginit()`. Then send the image contained in `image`, but instead of sending the image in one go, you are required to send it in `NETIS_NUMSEG`(+1) chunks of `segsize` (computed in the function for you). This is for future debugging purposes. The local variable `ip` points to the start of the image. You're going to send the image slowly, one chunk of size `segsize` every `NETIS_USLEEP` μseconds. This part should take 3 to 5 lines.

The server also takes a second optional argument. The `-v` option allows you to change the version number of the `imsg` packet sent. You should use it to test your client function `netic_recvismg()`, which was required to check the version field of all incoming `imsg` packet.

That's all for Task 2. It should take about 18 to 27 lines of code in total. If you want to build using an IDE, you need to add the files `netis.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 `netic.cpp`. The two tasks together should take about 35 to 51 lines of code.

TESTING YOUR CODE

You can develop your code on either Linux, Mac OS X, or Windows platform, but unless you know how to set up your firewall and other security mechanisms to allow for TCP connection to random, ephemeral ports, you will need to run the server on an ITCS linux host. These are the ITCS linux hosts available:

```
goldenaxe.gpcc.itd.umich.edu
joust.gpcc.itd.umich.edu
punchout.gpcc.itd.umich.edu
rygar.gpcc.itd.umich.edu
zaxxon.gpcc.itd.umich.edu
```

CAEN also has a number of Linux hosts. Unfortunately as of this writing, CAEN has disallowed connections to random, ephemeral port across hosts, so you can't use CAEN's Linux hosts to run your server. In all cases, you should run your client on your local host, the machine directly in front of you, be it Linux, Mac OS X, or Windows. You can also run both client and server on a single host and use `localhost` in place of the host's FQDN in the command line to `netic`.

In addition to the skeletal code and `Makefile`, we've also provided an executable binary of `netis`, called `refnetis` ("ref" for "reference"), that runs on ITCS linux hosts listed above. It is available in the course folder
`/afs/umich.edu/class/eecs489/w14/FILES/`.
It is a GNU/Linux executable, so don't try to download and run it on your Mac OS X, Windows, or even CAEN machines, which runs a different flavor of Linux. You should test your code as soon as you completed Task 1. Connect your `netic` to `refnetis`. If you have your own machine, you can also test your code locally. 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.

Here's an example test scenario. Run `netis -f BlueMarble2004-08.tga` on one window. Your OS may prompt you whether to allow the program to make or accept network connection, say yes. Then you should see something like the following printed on the window/console:

```
Image:
     Type   = RGB (1)
     Width  = 864
     Height = 432
Pixel depth = 24
Alpha depth = 0
RL encoding  = 0
netis address is yourhostname:54539
```

Next open up another window and on the same host run `netic -s localhost:54539` where `54539` should be replaced by the port number `netis` printed out for you (on the last line above) but `localhost` should be used in place of the host name `netis` reported. A window should now pop up and a map of the earth displayed slowly, one chunk at a time from top to bottom. (To see the actual download speed without our artificial slow down, try setting `NETIS_NUMSEG` to 1 at the top of `netis.cpp`.)

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. If you develop your code on Mac OS X or Windows platform, try not to use any platform dependent APIs that will prevent your code from compiling and running on another platform.