**Lab 1: Remote Image Display**

As this is our first assignment involving programming, this description is a bit long. Please read the whole thing carefully as it sets up how labs and programming assignments work in this course: how you can access the support code, how to set command line options, how to test your code, what you're expected to turn in (and not turn in), and how to submit your code, these 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](http://web.eecs.umich.edu/~sugih/courses/eecs489/common/notes/ide/). 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 up both. 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. The visualization tool could also help you detect some bugs in your code.

**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 (if you're using bash, the prompt will likely be '$" instead of '%'). 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 for a sample test case.

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 packets 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 socks.cpp and netimg.cpp to find places where "Task 1" related code must be filled in. The function socks\_clntinit(server, sname, reuse, self) connects to the server, either at the IPv4 address given in the sin\_addr field of the server argument, or, if the sin\_addr field is 0, at the host name given by the sname argument. One of IPv4 address or host name MUST be specified. In either case, the sin\_port of the server argument must be valid and must contain the port number of the server. This is a pretty straightforward task not much different from the code in client.c of Lab 0. First create a new TCP socket. Set the socket option so that the socket will linger for NETIMG\_LINGER amount of time upon closing, to give time for outstanding data to arrive at the destination. If the server's IPv4 address is not provided, obtain its address from the server's name, which must be provided in sname. Next initialize the socket with the server's IPv4 address and port number. Finally connect to the server and return the connected socket descriptor to caller. If the call to connect() fails, close the newly created socket and return SOCKS\_UNINIT\_SD. For any other error, terminate the process. The socks\_clntinit() function has been commented such that it should be clear where you need to make which socket API call. The third argument of the function, reuse is used in Lab 2, and the fourth argument self is used in PA1, you can ignore them for now. Depending on the amount of error checking you do, this part of the lab should take about 16 to 20 lines of code. If you're using Windows, the API to close a socket is called closesocket() instead of just close(). The provided socks\_close() wrapper function makes the system-appropriate call to close a socket.

Next write the netimg\_recvimsg() function to receive the return packet from the server. The return packet must be of data type imsg\_t. Store the packet in the global variable imsg. The structure imsg\_t shown here is defined in netimg.h.

It stores the specifics of the image: width, height, etc., which we will 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 9 lines of code.

Finally, you're to write 3 to 5 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 about 28 to 33 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 your code deal only with network programming. Nevertheless, you need to know how to build OpenGL/GLUT program. If you build your program using the provided Makefile, all you need to do is type make (on Linux and Windows systems, the first time you build an OpenGL/GLUT program, you may need to first install GLUT). If you want to build using an IDE, you need to add the files netimg.cpp, netimglut.cpp, socks.cpp, socks.h, 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 link with OpenGL libraries. If you don't know how to install GLUT or how to link with OpenGL libraries, please follow the instructions in the EECS 487 course note, [Building OpenGL/GLUT Programs](http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/ide/). 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 it 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 socks.cpp and imgdb.cpp to find places where "Task 2" related code must be filled in. Fill in the function socks\_servinit() by first creating a TCP socket. Then bind the socket with the self argument passed into socks\_servinit(). This argument MUST have its sin\_addr and sin\_port variables set by the caller. Once the socket is created and bound, listen for connection, with listen queue length set to the macro NETIMG\_QLEN. If the provided sin\_port is 0, the OS will assign a random, ephemeral port to the socket. Obtain from the socket the ephemeral port number assigned to it and update the provided self variable with the assigned port number. Since the socket was bound to INADDR\_ANY, you can't use getsockname() to find out the address of the socket (it will return 0's). Instead, to obtain the address of the current host is to call gethostname() and then call gethostbyname(). Store the FQDN returned by gethostname() in the provided variable "sname". Finally, return the socket descriptor to the caller. The function socks\_servinit() has been commented such that it should be clear where you need to make which socket API call. As with socks\_clntinit(), the reuse argument of the function is used in Lab 2, you can ignore it for now. It should take about 7 to 13 lines of code to complete this function. It is not that different from server.c of Lab 0.

The function socks\_accept(sd) accepts the connection on the provided socket sd. 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. If the argument remote is not NULL, store the connected client's address and port number in it (this feature will be used in Lab 2 and PA1). Return the socket descriptor for the accepted connection. This part should take about 7 to 9 lines of code.

The function imgdb::recvqry() receives a query packet, checks that the packet is of the correct size, of the expected version number and type (NETIMG\_QUERY), and that queried image name is not longer than NETIMG\_MAXFNAME. The definition of the query packet, iqry\_t, shown here, and the error code to send corresponding to the aforementioned errors can be found in netimg.h.

The query packet carries the filename of the image the client is querying. You may want to consult the function netimg.cpp:netimg\_sendqry() in writing this function. It should take about 10 lines of code.

Finally, the function imgdb::sendimsg(td, imsg) sends the image specification packet imsg to the client. Most of the content of this packet has been filled in for you by imgdb::marshall\_imsg(), called by imgdb::handleqry(). But all the integer fields are still in host by order. So you need to convert all integer fields of imsg to network byte order. You MUST also set the im\_vers field to NETIMG\_VERS. Note that the im\_type field of the packet has been set by the caller and should not be modified in imgdb::sendimsg(). Next, the function imgdb::sendimg(td, image, imgsize, numseg) sends the image contained in image to the client. In sending the image, instead of sending it in one go, which would be too fast to observe, you are required to send it slowly, one chunk every NETIMG\_USLEEP microseconds, for a total of numseg(+1) chunks (computed in the function for you), each of which is of size segsize. The local variable ip points to the start of the image. Completing these two functions should take 6 to 8 lines of code.

That's all for Task 2. It should take about 30-40 lines of code in total. If you want to build using an IDE, you need to add the files imgdb.cpp, socks.cpp, netimg.h, socks.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 no more than 70 to 75 lines of code.