EECS 489 Lab 5: Best Effort Netimg

This assignment is due on Wednesday, 18 Mar 2015, 10 pm.

Introduction

This lab introduces you to UDP-based cient-server programming. It allows you to modify the receiver window and maximum segment size of the client and the packet drop probability of the server and observe what happens when there is no flow control and no error recovery at the transport layer. You will re-build the simple client-server remote image viewer from Lab1 using UDP. You're provided with a skeleton code tar ball and a reference Linux binary executable of the client, `refnetimg`, and the server `refimgdb`. You should be able to run your client against the provided server and the provided client against your server. The provided `Makefile` builds `netimg` and `imgdb`. You can download the support code from the Course Folder. As usual, you can search for the string "YOUR CODE HERE" in the code to find places where your code must go. You may also want to consult the lecture notes on UDP socket.

Task 1: Server Side

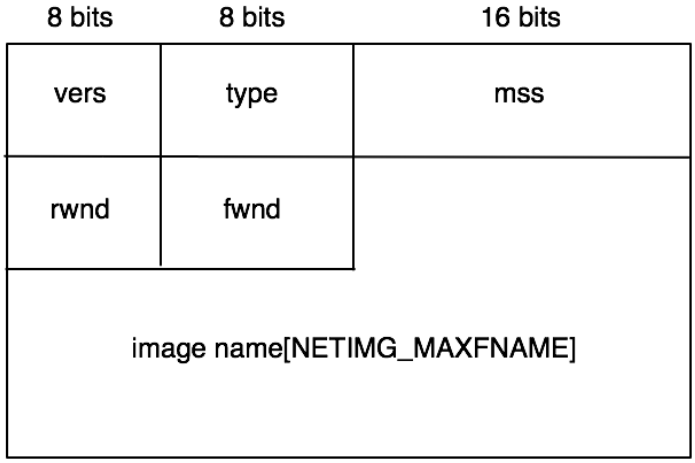
Your first task is to write the server code. You can search for the string "Lab5 Task 1" in `socks.cpp` and `imgdb.cpp` to find places where "Task 1" related code must be filled in. Fill in the couple lines of code to create a UDP socket in function `socks.cpp:socks_servinit()`. So that you don't have to re-type your code for Lab 6, the `imgdb.cpp` released for Lab 5 already includes instructions/comments for Lab 6 tasks, so be careful when you search for "Task 1" that you work only on "Lab5 Task 1" for this lab.

The server does not have any required command-line option. Upon start up, the server initializes a UDP 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.

```
% imgdb [ -d <prob> ]
```

The optional `-d` command line argument allows you to change the probability that the server drops a segment instead of sending it to the client. The probability is of type float and I recommend that you play with values ranging from 0.011 to 0.11. You can also try larger values and see how it affects the resulting image. To disable dropping, set the value to -1.0. The implemenation of this option is provided to you.

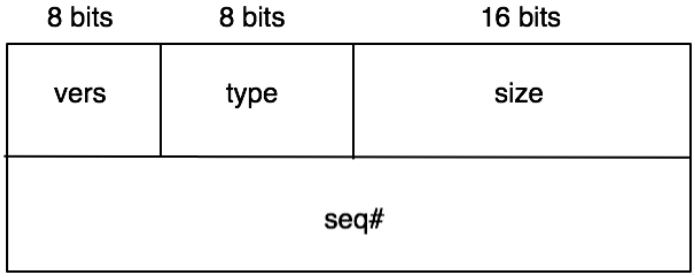
The definition of the query packet, `iqry_t`, is depicted in the following figure and can be found in `netimg.h`.



For this lab, the query packet must be of type `NETIMG_SYNQRY`, also defined in `netimg.h`. In addition to the filename of the image the client is searching for, the query message carries the maximum segment size (`mss`), the receiver's window size (`rwnd`), and the forward error correction window size, the latter two are used in Lab 6 and PA3. The function `imgdb::recvqry()` is similar to the one from Lab 3: it receives a query packet, checks that the packet is of the correct version and type and returns one of the `NETIMG` error codes defined in `netimg.h`. Unlike the one in Lab 3, this `imgdb::recvqry` calls the `recvfrom()` socket API to receive a UDP query packet. Upon successful return from `recvfrom()`, the client's address and port number must be recorded in `imgdb::client` member variable. You are to write the call to `recvfrom()`. It shouldn't take more than 2-3 lines of code.

As in Lab 3, `imgdb::handleqry()` then calls `imgdb::readimg()` to load the requested image from file. If the image is found, `handleqry()` stores the `mss`, `rwnd`, and `fwnd` information to the eponymous `imgdb` class member variables and calls `imgdb::marshall_imsg()` to prepare an `imsg_t` packet with the dimensions of the image. The definition of `imsg_t` is the same as for Lab 3 and is defined in `netimg.h`. Then `handleqry()` calls `imgdb::sendimg()` to send first the `imsg_t` packet, followed by the image itself. While we used the `send()` socket API to send the `imsg_t` packet in Lab 3, you are to write `imgdb::sendpkt()` to send the packet using the `sendto()` socket API. The destination of the `sendto()` call should be the client whose address and port number you stored earlier in `imgdb::client` in your call to `recvfrom()`. In this lab, `imgdb::sendpkt()` is a simple wrapper function of not more than 1 or 2 lines of code. It becomes more complicated in PA3.

Finally, the function `imgdb::sendimg()` sends the image contained in the provided `image` argument. The local variable `ip` points to the start of the image. Unlike in previous assignments, we will be encapsulating chunks of our image data in packets with header `ihdr_t` attached. The definition of `ihdr_t` can be found in `netimg.h`.



As usual, the first field is the version field, which must be of value `NETIMG_VERS`. The type field follows and must be of value `NETIMG_DATA`. Next comes the size field, which records the size of the attached data, in bytes, not including the header. The last field is the sequence number corresponding to the first byte of the attached data. We will use the byte offset from the start of the image in memory as the sequence

number of each byte. So the first byte of the image has sequence number 0. The *k*-th byte has sequence number *k*-1. A packet carrying a chunk of an image starting at the *k*-th byte of the image has sequence number *k*-1. Since the sequence number field in `ihdr_t` is an `int`, the largest image we can transfer is 2GB.

Prior to sending the image, you must ensure that the sender buffer can hold at least one packet of size `imgdb::mss`, as specified by the client in its `iqry_t` message. The size of the sender buffer is a socket option that you can query and set using the `getsockopt()` and `setsockopt()` socket APIs respectively. This takes about 5 lines of code.

To send the image, you are to divide up the image into chunks of `datasize`, where `datasize` has been computed for you as `mss - sizefo(ihdr_t) - NETIMG_UDPIP`. Obviously, the last chunk of the image may be smaller than `datasize`. To marshall together a packet consisting of an `ihdr_t` header followed by a chunk of data, you are NOT ALLOWED to make any copy of the image data. Instead, you are REQUIRED to use the `sendmsg()` socket API. See the comments in `imgdb::sendimg()` for further instructions. This task should take about 20 lines of code. Since we cannot tell from observing the behavior of your code whether you have used `sendmsg()` and whether you have made copies of the image data, this task will be graded by inspecting your code. If you don't know how to use `sendmsg()` or how to implement this task without copying, please consult the teaching staff. We will be weighing this task and the corresponding use of `recvmsg()` in the client side together as being worth 1/3 of the grade points for this lab. Please remember that if you cannot implement a required task, you must inform the teaching staff in your writeup for PA3 and that if you turn in a different implementation, it will be considered cheating.

That's all for Task 1. It should take about 30 lines of code in total, of which only the call to `sendmsg()` should be unfamiliar to you.

Task 2: Client Side

Your second task is to write the client code. The client takes two required command line arguments, exactly the same as in previous assignments:

```
% netimg -s <server>:<port> -q <image>.tga [ -w <rcvwin> -m <mss> ]
```

where the `-s` option specifies the server's name and port number and the `-q` option specifies the image file name. The client also behaves the same as in previous assignments, except that instead of using TCP, it uses UDP to transfer file. The optional `-m` command line argument allows you to set the client's maximum segment size, in bytes. The maximum segment size includes all headers, including the UDP/IP headers of 28 bytes. The optional `-w` command line argument allows you to set the client's receiver window size, in terms of number of maximum-size segments. The implementation of both of these optional arguments is provided to you.

You can search for the string "Lab5 Task 2" in `socks.cpp` and `netimg.cpp` to find places where "Task 2" related code must be filled in. As with task 1, so that you don't have to re-type your code for Lab 6, the `netimg.cpp` released for Lab 5 already includes instructions/comments for Lab 6 tasks, so be careful when you search for "Task 2" that you work only on "Lab5 Task 2" for this lab.

The function `socks.cpp:socks_clntinit(sname, port, rcvbuf)` creates a UDP socket and connects to the server whose name is provided in `sname`, at the specified `port`. Since this a UDP socket, the call to `connect()` in this function simply stores the server's address and port number so that subsequent calls to send and receive packets do not need to provide the server address. You are only asked to copy over the 2

line socket creation code from your `socks_servinit()` function here. Also in this function write another 3-6 lines of code to set the socket receive buffer to be of size at least `rcvbuf` bytes. Different systems set different maximum limit on the receive buffer. It is set at 1.5 MB on CAEN's eecs489 hosts. So want to retrieve the actual receive buffer size allocated to better understand the observed behavior of your code.

The function `netimage_sendqry()` has been provided to you. In addition to the filename of the image the client is searching for, the query message also carries the receiver's window size (`rwnd`) and maximum segment size (`mss`). The function `netimg_recvimsg()` is not changed from previous assignments and is also provided to you.

If the image is found, the client initializes the graphics library and put the socket in non-blocking mode before going into an infinite loop, receiving image data from the network and displaying it. In `main()` write 1 line of code to set the socket non-blocking. We'll be relying on the socket being non-blocking in subsequent assignments related to this lab.

Finally, in `netimg_recvimg()` receives the image data one packet at a time. Recall that each data packet consists of an `ihdr_t` header, followed by a chunk of data. You are NOT allowed to make any copy of the image data. Instead, you are REQUIRED to use the `recvmsg()` socket API. See the comments in `imgdb_recvimg()` for further instructions. This task should take about 20 lines of code similar and inverse to the ones you wrote for `imgdb::sendimg()`. Again, this task will be graded by inspecting your code. Please note in your writeup for PA3 if you are not able implement this task using `recvmsg()`.

That's it for Task 2. You will write a total of about 30 lines of code, most of which are very similar to the ones you wrote for Task 1.

Testing Your Code

In addition to the skeletal code and `Makefile`, we've also provided a reference implementation of both server and client that run on the CAEN eecs489 hosts. You should test your code with various sizes of receiver window and/or maximum segment. When the receiver buffer is too small, most of the arriving UDP packets will be dropped and the displayed image will only be partially completed. You should also play with the server's drop probability and observe how higher drop probability causes more gaps/streaks in the displayed image.

Your home network firewall may block UDP packets, preventing you from running your client and/or server to connect to the server/client running on the CAEN eecs489 hosts. You could use "`ssh -Y`" to connect to the CAEN eecs489 hosts and have X-window events forwarded to your local host. This works on Linux and Mac OS X (with XQuartz installed). On Windows, you can use MobaXterm. I would recommend **against** using VNC as students have reported OpenGL incompatibilities with VNC in the past. In both cases, the resulting image display becomes unbearably slow (worse with VNC). If your home network firewall blocks UDP, my suggestion is to test your client and server locally on localhost and after everything is working, then you test them on the CAEN eecs489 hosts over MWireless on campus or over UMVPN where the local network firewall doesn't block UDP. Similarly for testing interoperability against the reference implementation.

Submission Instructions

Do NOT use any libraries or compiler options not already used in the provided `Makefile`, to ensure that we will be able to build your code for grading. If we can't compile your code, you will be heavily penalized.

Test your compilation on CAEN eecs489 hosts! Your submission must compile and run **without** errors on CAEN eecs489 hosts using the provided `Makefile`, unmodified. Both your client and server must interoperate with the reference server and client respectively.

Your "*Lab5 files*" comprises your `socks.cpp`, `netimg.cpp`, and `imgdb.cpp` files.

To turn in your Lab5:

1. Submit the SHA1's of your *Lab5 files* on the CTools Assignments page. (If the URL doesn't work for you, just click the "Assignments" item on the left menu of the CTools page for EECS 489.) Once you've submitted your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. Upload your *Lab5 files* by pointing your web browser to Course folder and navigate to your `lab5` folder under your uniqname. Or you can `scp` the files to your `lab5` folder on IFS: `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/lab5/`.
   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 submission on CTools' Assignments page 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.

Do NOT turn in an archival (`.zip` or `.tgz`) file, instead please turn in your solution files individually. 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.