EECS 211 Assignment 8
Due: Friday, June 10th, 2016 by 11am

PLEASE NOTE THE 11AM DEADLINE!!!

In this last assignment we will implement the transferring of datagrams from source to destination. We will implement a very simple protocol for picking the next computer to receive datagrams as they make their way through the network. We will also finish the msg_list class by adding a function that removes datagrams from the list.

Background:

In Program 7 we implemented functions that allowed a laptop machine to transfer its outgoing datagram to its server. In this assignment datagrams will also be transferred out of servers and WANs. That means that datagrams will be removed from the message lists in these machines. However, we will also need to be able to look at a datagram in the list to determine if that datagram can be sent to the next machine. Servers and WANs can always receive datagrams because the message list can always expand (at least until we run out of memory, which will not occur in this project). A laptop, on the other hand, can only accept a datagram if the incoming pointer is NULL. So, a laptop can always transfer its outgoing datagram (if it has one) to its server (if it's connected to one) because the message list in a server can always accept another datagram. A WAN can always transfer a datagram because it is only connected to other WANs and servers, both of which can always accept new datagrams. A server must check if the target machine is a laptop and, if so, also check if the laptop can receive a datagram.

In the case of a server, the transferDatagram function will use a local variable of type msg_list pointer to create a temporary list. Then, for each datagram on the original message list, if that datagram can be sent to the next machine, it is transferred; if not, it is appended to the temporary list. When all the original messages have been examined, the temporary list is assigned to be the message list for the server. With this method, then, we only need to add one new function to the msg_list class – a function that removes the first element of the list and returns the datagram pointer (or NULL if the list is empty).

We now consider how datagrams should be passed around the network. A laptop can only send/receive datagrams to/from its server. So, a datagram from one laptop to another laptop must travel to at least one server, then possibly to one or more WANs to another server, and finally to the destination laptop. Remember that a server is not necessarily connected to every WAN, and similarly not every WAN is connected to every other WAN. We need an algorithm to determine at each time click the next step of the path from source laptop to destination laptop. We will make some very strong assumptions and implement an extremely simple (and completely unrealistic) algorithm.

- Each of the four numbers making up an IP address is called an octad. For servers and laptops, the first octad will be called the local-area network id (LANid). We will assume that a server and all its laptops have the same LANid and that each different local area network has its own LANid different from all the other LANids in the network.

- If a datagram currently stored on a server has a destination on the local area network for that server, then the server transfers that datagram to the destination laptop (assuming the laptop actually is connected to the server at the time).
- If a datagram currently stored on a server has a destination outside its own local area network, then at the next time click the datagram is transferred to the (connected) WAN whose first octad has the minimum absolute difference with the first octad of the destination.
- If a datagram currently stored on a WAN has a destination whose LANid matches a server connected to the WAN, then at the next time click that datagram is transferred to that server.
- If a datagram currently stored on a WAN has a destination whose LANid does not match any server LANid to which the WAN is connected, then at the next time click the datagram is transferred to the WAN whose first octad has the minimum absolute difference with the first octad of the destination (note that this may take the datagram to a WAN whose LANid is 'further away' from the destination than the current WAN).

Assignment:

Implement the CONSUME_DATAGRAM case in the switch in your main function. The format of this command is

consume_datagram *IPAddress*

The IP address should be the address of a laptop computer in the network. For this case you may assume that the IP address is valid, that the computer does exist in the network and is a laptop. That is, you do not need to check for errors. This case simulates the acceptance of a datagram from the incoming buffer of a laptop into an application within that laptop.

Complete the implementation of the TIME_CLICK case in the switch in your main function. Implement the transferDatagram function in the server and WAN classes. For each computer in the network in order within the network array invoke the transferDatagram function in that computer. Note, it is possible that an individual datagram might make several "hops" in one time click. For example, if a datagram in network[0] is transferred to, say, network[3] and network[3] is a server or WAN, then that datagram will be processed again when network[3] calls transferDatagram.

The amount of data that will be displayed for a system_status command will grow quite large. Therefore, it will be convenient to be able to display information for a select group of computers in the network rather than all computers. To this end, we modify the format of the system_status command as follows:

system_status *zero, one, or more IP addresses.*

If there are no other tokens on the command line, then display all the computers in the network, as before. Otherwise, the additional tokens will be IP addresses, and you should display only those computers. Here are some examples:

```
system_status  1.2.3.4
system_status  5.6.7.8   21.22.23.24
system_status
```

Requirements and specifications:
0. Notice the laptop, server, and WAN **destructors** that have been provided in machines.cpp. Additionally, you will be writing the **deleteList** function in msg_list.cpp. The destructors are utilized in the DESTROY_MACHINE case of the main switch, and again at the end of the main function (as provided). Nothing new is necessary on your part to make use of these destructors, but just look at them to make sure you understand how they work.

1. Add a new member function **void consumeDatagram()** to the **laptop** class. This function sets the incoming datagram pointer to NULL. In the CONSUME_DATAGRAM case of the main switch, have the indicated laptop in the network invoke consumeDatagram.

2. Add a new member function **datagram *returnFront()** to the **msg_list** class. If the list is empty, this function returns NULL. Otherwise, this function removes the front element of the list, deletes the msg_list_node object, and returns the datagram pointer from that element.

3. Add a new member function **IPAddress destinationAddress()** to the **datagram** class. This function returns the destination IP address of the datagram.

4. Add a new member function **int firstOctad()** to the **IPAddress** class. This function returns the first octad of the IPAddress object.

5. Add a new member function **IPAddress myAddress()** to the **node** class. This function returns the machine's IP address. Note, this function is <u>not</u> virtual and is not overridden in any of the derived classes.

6. Add a new member function **int canReceiveDatagram()** to the **laptop** class. This function returns 1 if the laptop can receive a new datagram (i.e., incoming is NULL) and 0 otherwise.

7. Add a pure virtual function **void transferDatagram()** to the **node** class. Then implement this function in each of **server** and **WAN** classes. (Note, you have already implemented it in the **laptop** class.) In each of these functions you will have a loop that runs through the message list (dlist). Remove the front datagram, get its IP address, and then get the first octad. Then attempt to transfer the datagram according to the simple algorithm described in the **Background** section.

Make sure the TIME_CLICK case in your main switch invokes transferDatagram for **every** machine in the network. This will differ from what it did previously.

8. Change the SYSTEM_STATUS case of you main switch. You must now accept two different types of commands. In the first type, “system_status” will behave exactly as it did previously. However, now, “system_status” can be followed by any number of ip addresses. For example “system_status 1.1.1.1 1.1.1.0”. In this case it should print out the status of only the machines mentioned in the command. See p8output.txt for what your output should look like.

9. Add a new member function **void deleteList()** to the **msg_list** class. This function needs to deallocate the space allocated for every msg_list_node in the msg_list. Additionally, it needs to deallocate the space allocated for every datagram stored in the msg_list. After calling this function, both front and back should be set to NULL.

Comments, suggestions, and hints:

You may find it useful to use the built-in absolute value function **int abs(int)** in stdlib.h when comparing octads.

When testing your program it may be helpful to draw a diagram of the network on scratch paper. Just follow the creation and connect commands in the test file and draw the network by hand.

Test your work with p8input.txt.

Checking your work on the EECS Servers
Before turning in your project, its critical that you make sure that your code compiles and runs on the EECS servers. Here’s how:

1) Copy your code (all code files + the text input file + the provided makefile) to the eecs server using scp or pscp
2) Connect to the servers (ssh or putty) and in the same directory as the copied files, type ‘make’ to build your code.
3) Type **./Assignment** to run your code. Verify that the output is as you expect.

What to submit:
Submit your 4 altered files – RENAMED in the normal manner: netid_main.cpp, netid_machines.cpp, netid_msg_list.cpp, and netid_datagram.cpp.