EECS 211 Assignment 7
Due: Friday, May 27th, 2016

In this assignment we will connect the computers into a network. We will also develop a class for holding lists of datagrams in the server and WAN machines. Program 8, then, will finish the project by adding the capability of transferring datagrams from one machine to another. At that point we will be able to create messages in laptops and send them to other laptops in the network.

Background:
Laptop computers can generate datagrams; for example, a laptop user may generate an email message to a friend. Laptops are also the end target for datagrams; that is, datagrams in our simulated network are sent from one laptop to another. When a datagram reaches its destination it is accepted into a buffer; in our project we are simulating that buffer with a pointer to a single datagram. Some time later the datagram is actually taken out of the buffer by the application the datagram is meant for. In the case of an email, for example, the email may be received by the laptop but the user does not have the email application running. The email sits in the buffer until the user starts the email application, at which point the email processor takes the datagram out of the buffer. In our project we have assumed that the incoming and outgoing buffers in a laptop can each hold a single datagram. This is not very realistic, but it makes the laptop class somewhat simpler and allowed us to perform some testing with our datagram class early in the project. We will continue to assume that each laptop can hold at most one incoming and one outgoing datagram.

Server and WAN machines, on the other hand, should be expected to handle many datagrams. Server and WAN machines receive datagrams from other machines in the network and forward those messages to other machines. A server could be connected to as many as 8 laptops, each of which is generating a datagram. Moreover, a server may be receiving incoming datagrams from one or more WAN machines. The situation is similar for WAN machines. Therefore, servers and WANs should have the capability to hold many datagrams. In fact, there should be no *a priori* limit on the number; a server or WAN should hold as many as memory allows. Moreover, one policy of fairness dictates that a server or WAN should process (i.e., attempt to forward) datagrams in the order in which they were received, first-in first-out (FIFO). Because we are assuming that servers and WANs can hold unlimited numbers of datagrams, these kinds of machines should always be able to accept an incoming datagram. However, we have assumed that a laptop can hold at most one incoming datagram. If the application for which that datagram is intended has not taken the datagram out of the buffer, that laptop cannot accept another datagram. Therefore, sometimes a server may not be able to forward a datagram to a laptop and will have to hold the datagram for a while. These features lead us to select a linked structure, a modified form of queue, to hold lists of datagrams.

A queue is a linked structure in which new elements of the list are added to the end and elements are removed from the front. One modified form of queue has new elements added to the end but allows elements to be removed from anywhere in the queue. In Program 8 we will process datagrams in order starting at the front of the queue, i.e., will process the datagrams in FIFO order. However, in a server machine, if the datagram at the front is meant to be transferred to a

laptop in that server's local area network but the target laptop cannot accept the datagram, the server will be allowed to hold that datagram but transfer other datagrams further back in the queue.

A queue is a dynamic structure, which means the "size" can change. We will implement our queue class using pointers to connect the data entries. The queue has a front and back pointer. When the queue is empty, both these pointers are NULL.

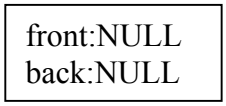


Figure 1: An empty queue.

When a new element is added, space is created to hold a data pointer and a next pointer. It is common to use an intermediate structure, a queue node. A queue node holds the next pointer and a pointer to the actual data. The actual data itself is then held in another structure, in our case a datagram object. Thus, two separate blocks are used to form a single element of the queue – a queue node and a data block. This provides a clean separation of queue operations (such as append or remove) from data operations (such as display). If the new element is the first element of the queue, then both front and back point to this new block, as shown in Figure 2.

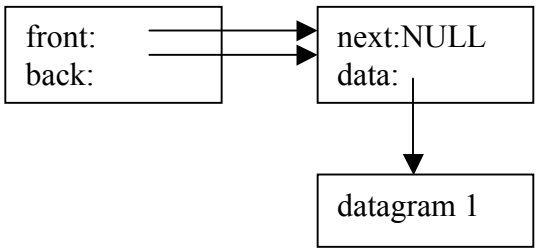


Figure 2: A queue with one node.

Additional blocks are added at the end. In each case, two nodes are used – a queue node holding just the next pointer and data pointer and an actual data block holding the data. The front pointer remains unchanged, but the back pointer changes as each new node is added to the end of the queue. After the second node is added, the picture looks like Figure 3.

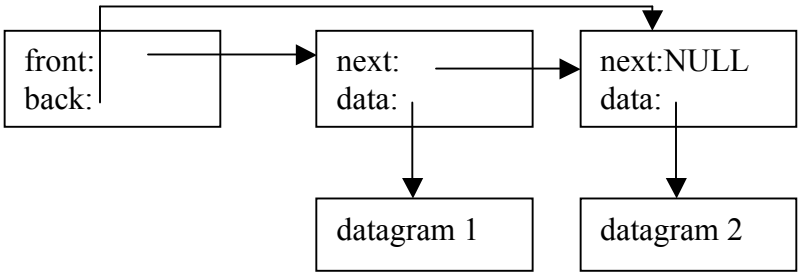
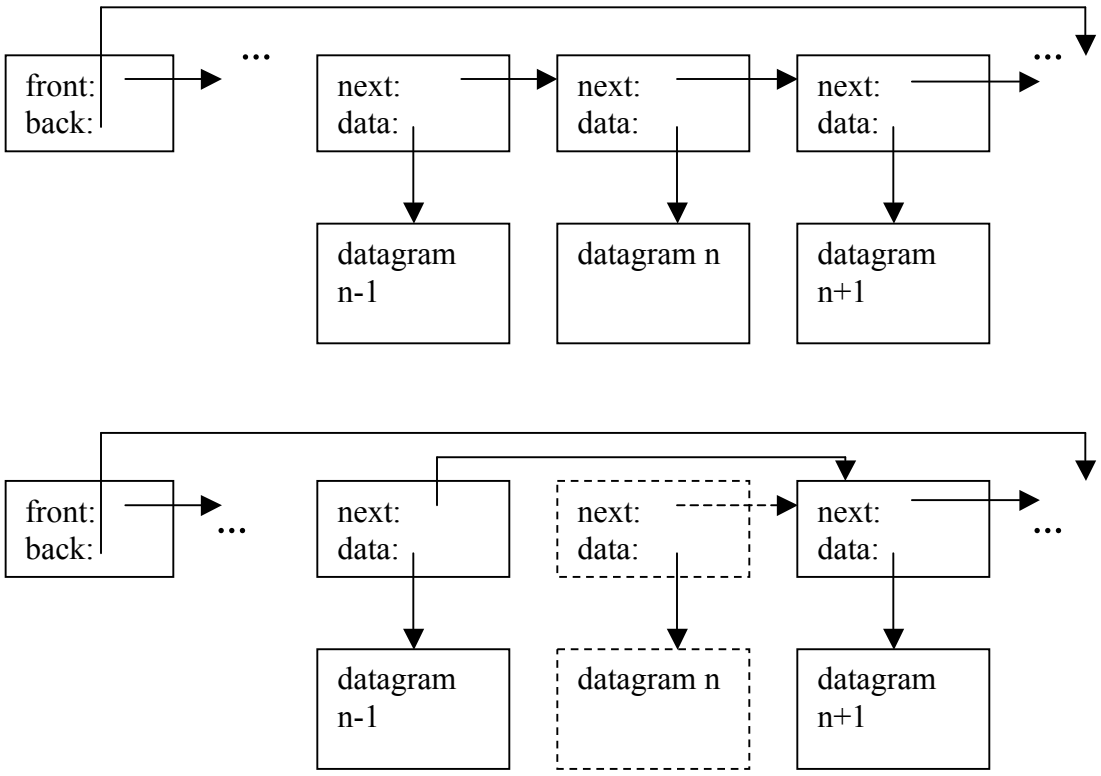


Figure 3: A queue with two nodes.

Removing nodes from a queue is slightly more involved. Consider the picture shown in Figure 4a. Suppose the node for datagram n is to be removed, and suppose that this node is in the middle of the queue, that is, there are nodes in front and back of element n. After node n is removed, node n-1 next pointer has to point to node n+1. Figure 4b shows the linkage after the removal.



Figures 4a (top) and 4b (bottom): A queue before and after removal of a node.

In general, then, the next pointer of the node in front of the removed node has to point to the same queue node as the next pointer of the removed node. There are some special cases. If the removed node is the front of the queue, then it is the front pointer that is replaced by the next pointer of the removed (i.e., original front) node. If the removed node is the one at the end, then the back pointer has to be changed to point to the node in front of the removed one.

Note, that the queue node and data block themselves still exist; they are just not part of the queue any more. The queue node should be deleted so that its memory space is reclaimed. The data pointer should be returned to the calling program so that it can use the data.

Here is a sample algorithm for appending a datagram to a queue.

append(datagram *d)

1. create a new queue node, q
2. set the data pointer of q to be d

3. set the next pointer of q to be NULL
4. if the queue was empty, set both front and back to point to q; otherwise, set the next pointer of the original back node to point to q then set back to point to q.

In program 8 we will see an algorithm for removing a queue element from the queue so that it can be transferred to another computer in the network.

Assignment:

This assignment involves writing two new cases of the main switch: CONNECT and TIME_CLICK. You'll also implement member functions of a new class msg_list and add a few new member functions to machines.cpp.

1. Change the definition of MAX_MACHINES to 10. Then add the following new constant to the definitions header file:

CONNECTION_REFUSED 204

These changes have been made in the provided definitions.h.

Notice that the printError function (in system_utilities.cpp) has been augmented to handle the new error code.

2. Notice the addition of a new member function **int isNULL()** to the **IPAddress** class. This function returns 1 if the IP address is 0.0.0.0 and 0 otherwise (changes are made in datagram.h and datagram.cpp).

3. In machines.h, notice that we've added a pure virtual function **int canAcceptConnection(int machine_type)** to the **node** class. The argument machine_type will be one of LAPTOP, SERVER, or WAN_MACHINE. Your job is to implement this function (in machines.cpp) in each of the three derived classes. A WAN machine can accept a connection with machine_type SERVER or WAN, but only if the corresponding array of IP addresses is not already full. A server machine can accept a connection with machine_type LAPTOP or WAN, but only if the corresponding array of IP addresses is not already full. A laptop can accept a connection only if the type is SERVER and only if the laptop is not already connected to a server machine (i.e., the laptop's my-server IP address is 0.0.0.0). This function returns 1 if the machine is able to accept the connection and 0 otherwise.

4. In machines.h, notice that we've added a pure virtual function **void connect(IPAddress x, int machine_type)** to the **node** class. Your job is to implement this function in each of the three derived classes. You may assume that **canAcceptConnection** has already been called to verify that the machine can accept the connection. In laptop::connect, the input machine_type can only be SERVER, so we simply need to set the server for the laptop to x. In server::connect, the input machine_type can be LAPTOP or WAN_MACHINE; x should be added to the appropriate list. In WAN::connect, the machine_type can be SERVER or WAN_MACHINE; x should be added to the approriate list.

5. The form of the connect command is:

connect *IPAddress* *IPAddress*

Implement the CONNECT case of your main switch. Form IP address objects from the remaining two tokens on the command line. If either is not a well-formed IP address, stop processing the command and set error code to BAD_IP_ADDRESS. Locate the two specified machines in the network array. If either does not exist, stop processing the command and set error code to NO_SUCH_MACHINE. Check each machine to see if it can accept the connection. If either machine is unable to accept, stop processing the command and set error code to CONNECTION_REFUSED. Otherwise, use the connect function in each machine to connect it to the other machine.

6. Notice that the display function in each machine class has been altered in the provided code so that it displays the IP addresses of any machines to which it is connected.

7. Notice the content of the new file, msg_list.h.

class **msg_list_node**:

- private data members:
  - a pointer to msg_list_node – this will be the "next" pointer described earlier.
  - a pointer to datagram
- This class has no member functions.
- This class declares the class **msg_list** as a **friend** class.

class **msg_list**:

- data members – two pointers to **msg_list_node** for front and back of the list.
- member functions
  - an initializing constructor – sets front and back to NULL
  - void append(datagram* d) – appends the datagram pointer d to the end of the list
  - void display() – displays all the datagrams in the list in a suitable and informative format

Notice that, in order for **msg_list_node** to be able to refer to **msg_list** (so that it can declare **msg_list** to be a **friend**), msg_list.h had to include the line

```
class msg_list;
```

before the declaration of **msg_list_node**.
Your job is to implement these two new classes in msg_list.cpp. Notice that the display function is provided for you, so you really only need to implement the msg_list constructor and msg_list::append.

8. Notice that the server and WAN classes have a pointer to **msg_list** as a new data member and that the constructors have been modified so that they create a **msg_list** object for the new data member. The new display functions also display the **msg_list** data member.

9. In machines.h, notice that we've added a pure virtual function **void receiveDatagram(datagram**

***d)** to the **node** class. Remember that the **laptop** class already has this function. Now we will implement this in the other two classes – **server** and **WAN**. For each of these classes, the datagram pointer should be appended to the machine's **msg_list** data member.

10. Add a new member function **void transferDatagram()** to the **laptop** class. Read about the keyword 'extern' before starting this step (there is an example of this from a recent lecture). If the outgoing datagram pointer is NULL, this function does nothing. If the laptop is not connected to a server, this function does nothing. Otherwise, the laptop machine calls **receiveDatagram** function of its server, then sets outgoing to NULL.

11. We will not implement full network message traffic until Program 8. In order to test the new class **msg_list** and the new **receiveDatagram** functions, implement the following temporary code for the case TIME_CLICK in the main switch. Have each laptop in the network transfer its outgoing datagram to its server.

Test your work with p7input.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 3 altered files to Canvas – RENAMED in the normal manner: netid_main.cpp, netid_machines.cpp, and netid_msg_list.cpp.