In this homework, we will implement data transferring functionality in our simulator. Data transferring will be modeled by passing raw pointers to the `Datagram` class around. We will implement 4 new member functions of the `Node` class in Section 2 and three system operations in Section 3. Together, they model the start, ongoing and the end of a data transfer process.

Please do not modify `datagram.h`, `machines.h` or the lines “friend class Grader;” in homework 7. These are needed for grading.

**Data Transfer Process**

The data transfer process we are modeling allows the user to transfer a piece of message from the source machine to the destination machine even with these two machines might not be directly connected. The process starts with the source machine allocating a new instance of `Datagram` containing the message to be sent. The `Datagram` is then being transferred repeatedly from one machine to another, moving closer toward its destination. When the `Datagram` arrives at its destination, it will be placed in the receive buffer in the destination machine, waiting for the user to process it.

We will implement 4 new member functions of `Node` in section 2 to simulate this process. Though the given algorithm for sending datagrams is pretty naive, we guarantee that every datagram will be able to arrive at its destination.

- `Node::allocate_datagram` allocates a new instance of `Datagram`.
- `Node::send` sends the datagrams in the send buffer to the best-suited machines connected to the current machine.
- `Node::receive` receives a datagram from another machine. The current machine might or might not be the destination of the incoming datagram. If the current machine is not the destination, the datagram will be sent out by later invocations of `Node::send`.
- `Node::release_datagram` processes the received message and frees the datagram in the buffer.

**Raw Pointer Memory Management**

In our system, an object that possesses a non-null raw pointer to a `Datagram` object takes the responsibility to free that `Datagram` upon the destruction of the owning object itself. We also need to ensure that no two classes can have raw pointers to the same `Datagram` at the same time.

Exactly two classes can own raw pointers to `Datagrams` in this homework. A `Node` can own a raw pointer `Datagram*` in `Node::incoming_`. A `ListNode` in a linked-list can also own a raw `Datagram` pointer in `ListNode::data`. When a node `Node` or a `ListNode` is being destructed, it needs to delete the `Datagram` if it has a non-null raw pointer to that `Datagram`.

If we ever want to pass the raw pointer to a `Datagram` to another class, we have to set the original raw pointer to the `Datagram` to `nullptr`. For example, we can extract the raw pointer from a `ListNode` below, pass it to another `Node` class. Then, we must set the original `head->data` to `nullptr`. This not only represents that the ownership of the the `Datagram` has been transferred to `*m`, but also prevents `head` from erroneously deleting the `Datagram`.

```cpp
// data_list_ is a List and m is a shared_ptr<Node>
shared_ptr<ListNode> head = pop_front(data_list_);

m->receive(head->data); // Now m possesses the raw pointer to head->data,
head->data = nullptr; // so we set head->data to nullptr
```
1 Linked-List Library, Revisited

We will reuse our linked_lib in homework 4 to implement the data buffer in Node. The type of data field has been changed to Datagram* to store raw pointers to the Datagram class.

Copy your homework 4 solution to linked_lib.cpp in the provided code. Complete the implementation of the constructor and the destructor of ListNode. Upon creation, initialize the ListNode::data field with nullptr. Upon destruction, delete the allocated Datagram if the ListNode::data field is non-null.

2 Transferring Data Across the Network

Two new data members are added to the Node class. One data member is data_list_ of type List (from homework 4) representing the send buffer; the other data member is incoming_ of type Datagram* representing the receive buffer.

Implement the four new member functions and the destructor below. They model the creation, dispatching and delivery of datagrams. Be sure to transfer the ownership of Datagram*.

- In ~Node(), delete the Datagram pointed by incoming_ if it is not nullptr.
- The allocate_datagram member function initiates a data transmission. It allocates a new Datagram in the memory and pushes the Datagram* to back of the linked-list data_list_.
- The release_datagram member function returns the message of the received datagram in the incoming_ buffer by calling Datagram::get_msg if the incoming_ buffer is not nullptr or an empty string otherwise. The datagram, if exists, will be deleted and the incoming_ buffer will be set to nullptr again.
- When node_list_ is not empty, send member function sends out every Datagram* in the ListNodes in data_list_. If the destination of the Datagram is in the connected machines, send the datagram to that machine. Otherwise, find the connected machine whose IP address’s first octet is closest to the destination’s first octet, and send it there. Send the datagram by calling the recipient machine’s receive member function with the datagram. If the recipient throws an err_code::recv_blocked exception, keep the datagram in data_list_. Finally, return the number of successfully sent Datagrams. One possible way to implement this is given in Section 6.
- In receive member function, there are two possibilities. If the destination of the Datagram is the current machine (compare Datagram::get_destination() with Node::local_ip_), either it will be delivered or blocked, depending on whether the incoming_ buffer is nullptr or not. In particular, if incoming_ is nullptr then it should be set to the incoming Datagram; otherwise throw an err_code::recv_blocked exception to indicate that the incoming_ buffer is full.
  If the destination of the Datagram is not the current machine, then the Datagram is pushed to the back of the linked-list data_list_.

3 Some More User Commands

Extend the simulator to handle three more commands in this homework.
• Sending New Datagrams: System::allocate_datagram
    Invoke allocate_datagram on the designated machine to initiate a data transmission from IP_{src} to IP_{dst} with data “message”. The corresponding command for this member function is “send IP_{src} IP_{dst} message”.

• Consuming Datagrams: System::release_datagram
    Invoke release_datagram on the designated machine to “process” the data and end the data transmission. The corresponding command for this member function is “recv IP”

• Time Ticking: System::time_click
    Invoke send on all machines to route datagrams one step further. The corresponding command for this member function is: “tick”

4 Handling Errors

For error handling,

• System::time_click, Node::send, Node::allocate_datagram and Node::release_datagram will never throw an exception. Node::send simply does nothing when there are no connected machines.

• Node::receive should throw an err_code::recv_blocked exception if the destination of the input Datagram is the current machine and Node::incoming_ is not nullptr.

• System::allocate_datagram and System::release_datagram should throw an err_code::no_such_machine exception if the designated machine is not found in network_.

5 Unit Testing

Write comprehensive unit tests to check the behavior of the Node class for the data transfer functionality. See networksim_test.cpp for an example of how to check that Node::send sends the datagram to the best connected Node specified in Section 2.

Append the new unit tests in networksim_test.cpp. You have to figure out how to setup unit tests to check the behavior of Node::release_datagram, Node::send and Node::receive to ensure that the required functions work properly. We will also grade on the completeness of unit test coverage in the form of self-evaluation as in Homework 4.

6 Hints: Node::send

One trick is to use another linked-list to keep the Datagrams that failed to be sent. While data_list_ is not nullptr, we call pop_front, extract the Datagram, and find the best node to send to. If Node::receive failed, we push the Datagram to the back of the new linked-list. At the end of Node::send, replace data_list_ by the new linked-list

new_list <- empty linked-list
while data_list_ is not nullptr, do
    List head <- pop_front data_list_
    
    # Take the ownership of head->data. Setting head->data to nullptr is very important.
    # Otherwise, head will delete the datagram after it goes out of scope.
Datagram* d = head->data
head->data <- nullptr

If the machine d->get_destination() is in node_list_, let m be that machine. Otherwise, Find the node m in node_list_ that minimizes the difference between d->get_destination().first_octad() and m->get_ip().first_octad(). If there are ties, choose whichever you like.

Call m->receive(d). If m throws an exception, pushes d to the back of new_list

data_list_ <- new_list

Appendix: Project Introduction: Homework 5-8

In this project, we are going to build a tiny network simulator modeling a small system that has laptops, servers and WAN (Wide Area Network) nodes. We will also model datagram transmission between them. A laptop must first be connected to a server. A server can connect multiple laptops, building a LAN (Local Area Network) between them. A server can also be connected to multiple WANs, in which case it will be able to transfer datagrams indirectly to other servers and finally to other laptops outside LAN. A WAN node can connect not only to arbitrary servers, but also to other WAN nodes.

Starting from homework 6, we will implement one class for each of the constructs in this system: a System class for the entire network system, a Datagram class for datagrams and machine classes Laptop, Server, WAN_node for laptops, servers, and WAN nodes respectively. The System class will have member functions corresponding to network operations. These include: sending and receiving a datagram on a Laptop, adding and removing machines from the network, and a time ticking function for servers and WAN nodes to route datagrams one step toward their destination.

The simulator, aside from the System class modeling the entire network, also contains a command line interface to interact with the user. The user can enter commands to control the system and view the status of the network system. In this homework 5, we implemented three utility parsing functions that help the command line interface convert input strings into commands and accompanying data in order to invoke the corresponding member functions of the System class.

In provided the code, main.cpp and interface.cpp implement the command line interface. In main.cpp, the main function repeatedly reads a line from the user, parses the input into tokens by the tokenize function, and calls execute_command to perform the corresponding operations. If an error is thrown, it catches the error code err_code and prints an error message.

In interface.cpp, the execute_command function first identifies the input command by searching through the command_syntaxes list, match the command string and obtain the cmd_code for the input command. execute_command then parses the accompanying data (some by parse_IP and invokes the member function of System.