Programming Assignment 1 - Network Routing

Due: February 11 (extra credit available for February 4)

This is an individual assignment!

You are responsible for adhering to the Aggie Honor Code and student rules regarding original work. You must write and submit your own code to Mimir. Plagiarism checks will be used to check the code in addition to a manual review from the grader.

That said, we recognize that there is a certain degree of collaboration and referencing involved in the process of problem solving. In the starter code, there is a provided `report.txt` where you will answer a few questions and track all of the references you use. Use references generally to help think through a problem or find information about an error. This is especially true if you talk with peers: focus on the high-level aspects of the problem. *Do not share code! Do not copy code!*

Overview

The Internet: The Giant Computer Network

It has been stated that a single weekday edition of the *New York Times* contains more information than a seventeenth-century individual might come across in their entire lifetime[1]. Whether or not this is true, there is no denying that we live in an age of information availability. Thanks to the internet, we create and consume exabytes of information daily[2].

As vast as the internet is today, at its core, this digital age is made possible by computer networking. We may not think about it every time we make a Google search or buy something on Amazon, but what we are doing is sending and receiving digital data through a giant network. For anyone interested, computerhistory.org has a nice timeline visualizing key developments in networking[3].

This Assignment

In this assignment, you will be implementing the data structures for a highly simplified first-come-first-serve network routing program: a queue. Most networks are based on queues, in which the traffic is processed in the order it is received. Our simplified router will only have two routes: internal and external traffic.

Think of it this way: users at their local computer send network requests (called "packets") to another computer. The network consists of many nodes between your computer and the other computers. When a network request arrives at a node, it must determine where to send it next so that it reaches the destination computer. If the user was loading Google, it will forward the request to another node closer to Google's servers. If the user was loading a local resource, such as a TAMU webpage while inside the TAMU network, it will send the request to a different node, perhaps directly to the webpage server.

[1] Shenk, David. "Data Smog: Surviving the Info Glut." Technology Review 100.4 (1997): 18-26
[2] https://techjury.net/blog/how-much-data-is-created-every-day
[3] https://www.computerhistory.org/timeline/networking-the-web/

You will be given the majority of code for the simplified router and network packet. The code you must implement can be thought of as two distinct activities:

- Part a: Implement a Queue, utilizing a Doubly-Linked List data structure
- Part b: Update your data structures to support templated types

Getting Started

Mimir has the two parts of the assignment split into parts a and b.

1. Visit Assignment 01a on Mimir to download the starter code.
2. Familiarize yourself with the code
    a. You will notice that you already have the code for the NetworkPacket and NetworkRouter provided. The runner can be used once your data structure implementations have been written.
    b. The Node, DoublyLinkedList, and Queue sources need to be implemented.
3. Read through `report.txt` included in the starter code.
    a. The report file is where you can track references and sources you use in addition to class materials.
    b. As stated above, do not share or copy code, but use this as a way to help track what sources you are using. Keeping a list of references is actually quite helpful for you since it provides a catalog of useful links.

In this assignment, we can manage without a makefile by compiling all cpp source files (e.g., g++ -std=c++17 *.cpp), but you are welcome to create a makefile for yourself. It is not necessary, but it can save you time typing commands now and give you practice for future assignments with makefiles.

About the Code

The Network Router Application

NetworkPacket.h

About
NetworkPacket.h is a fully written class that simulates an internet protocol packet at a very rudimentary level.

It is loosely based on IPv4 packets, containing only the source and destination addresses, a checksum, and a data portion. The real packets have much more information. Additionally, we are representing all the data as strings, and there is no validation being performed (for instance, the checksum is not valid or checked).

However, despite all these limitations, it is still a sufficient visualization of how a packet looks and how we can operate on its fields.

Functions
In addition to default and parameterized constructors, `operator==` is defined to enable comparisons (using checksums) and `operator<<` is defined to print the packet for Mimir tests.

NetworkRouter.h and NetworkRouter.cpp

About
While the NetworkPacket is the most basic part of the application, on the opposite end, the NetworkRouter class is at the highest level. It makes use of your user-defined Queue class to "send" and "receive" packets.

As before, the network traffic is also simplified since we are not using network connectivity in this assignment. To simulate sending and receiving packets, we will use the standard input and output. This works well since the packets are being presented as strings rather than binary chunks.

The router performs a basic operation of identifying internal vs. external traffic. Internal traffic would be sent to the next relevant location inside the network, while external traffic would be forwarded to the next node closer to exiting the local network. Local network traffic is very common in homes and offices. At home, this could be your devices like your computer and printer talking to each other. In the office, a local intranet provides many internal services like file sharing, direct messaging, and more.

Functions
`receiveRequests` reads one NetworkPacket per line from the provided input stream as long as input exists. The packets are saved into the requests buffer until they can be processed.

`processRequests` dequeues one packet a time from the requests queue. It performs the operation of directing network traffic either to an internal or external buffer (more queues). The comparison operation is already implemented and consists of just a string match rather than a binary masking operation.

`sendRequests` simulates sending the packets by printing them to standard output.

runner.cpp

About
The runner file shows how a very small application can import the NetworkRouter and use it to manage traffic. This is the elegance of object-oriented programming. You can distribute the parts of the program into many different objects so that the high level application is quite clean and easy to use.

A `sample_input.txt` is provided in the starter code to test with the runner, and the "Full Program" Mimir tests show more example input and output. You can use it by piping the file to the standard input:

`./a.out < sample_input.txt` or `cat sample_input.txt | ./a.out`

You can create your own. You can also write them directly into the standard input when running the application. If you do, recall that you can press `ctrl-d` to signal the end of the input stream.

Your Queue (and therefore DoublyLinkedList and Node) must be implemented for the runner to work, so you may wish to create your own test files along the way to verify different parts of your code.

The Data Structures

As this homework is largely about review and introduction to some additional data structures, these files are the main ones that must be implemented in this assignment.

Node.h

About
The Node is the underlying object of the linked list. It contains a data member and pointers to adjacent nodes in the list. In a singly linked list, only a pointer to the next node is maintained, but in a doubly linked list, both the next and previous pointers are saved.

Use only Node.h. Since the functions are so short, it is not worth creating Node.cpp in this case.

Functions
`Node(const NetworkPacket& d)` parameterized constructor to create a Node with the data from the provided packet. You will need a default constructor to write this one.

`getData(), getNext(), getPrev()` to return the data and pointer members. Remember to set the return type to match the variable being returned.

`setData(const NetworkPacket& d), setNext(Node* n), setPrev(Node* p)` to update the Node's internal variables.

DoublyLinkedList.h and DoublyLinkedList.cpp

About
Probably the biggest single part of this assignment is the linked list. However, you may have some existing linked list code from a previous course, which can make the process of updating to a doubly linked list easier.

The doubly linked list has internal members tracking the integer length (number of elements in the list) and Node pointers to the front and back elements.

Functions
`getLength(), getFront(), getBack()` getters for the internal variables.

`insert(NetworkPacket data, int index)` should take a packet and place it at the specified index in the list. For instance, index = 0 would place it at the front. If index = 1, it would be placed right after the first element. If the length is 5 and index = 5, it would place it at the end (right after the last element at index = 4). This means that the single insert function must be able to handle all forms of list insertion: empty, front, middle, and back. Remember to update all the pointers!

If the index is out of range, throw std::out_of_range with the message "Index outside of list bounds".

`remove(int index)` removes the Node from the list at the specified index. This operation must handle removal from anywhere in the list: front, middle, or back. Remember to update all the pointers!

If the list is empty or the index is out of range, throw std::out_of_range with the message "Index outside of list bounds".

`toString()` converts the list into a string. In NetworkPacket, we used operator<< to print the object. Here, we use a direct toString() method. This function is vital for some of the grading operations. It is constructed very simply by iterating through the list from the front and visiting each element at a time.

It is easiest to use a stringstream to write the list into and ultimately convert that into a string.

After printing an element, print a single space " " to delineate betweens indices of the list. Note that you should include an extra space at the end. In other words, do not bother doing any special accounting for printing the last element of the list.

`Rule of Three` functions need to be implemented to account for the new memory being allocated for the list's Nodes.

You may add helper functions if you wish. For instance, operator==, a dedicated deleteList function, or a push_back operation may be useful, but these are not required as the behavior can directly be written in the functions that need them.

Queue.h

About
Queue is an abstract data type which restricts how to add and remove elements. New elements can only be added at the end of the current queue, and elements can only be removed from the front. This simulates first-come first-serve behavior, as in a physical line or queue. It is also called First-In First-Out (FIFO). FIFO is widely used in networks where all traffic has equal priority.

Being an abstract data type, a queue can be implemented on any linear structure. You must use the DoublyLinkedList to implement the Queue's storage.

Use only Queue.h. The implementation specific to the Queue will be trivial so long as your list is working correctly, so the functions should be quite short and not worth splitting into a separate source file.

Functions
`empty()` returns a boolean indicating if the queue is empty or not.

`push_back(NetworkPacket data)` inserts a new packet at the end of the queue.

`pop_front()` removes the front element from the queue and returns its value.

`toString()` is used for autograding and should behave like the toString() method implemented in the DoublyLinkedList.

Part A Details

Task

For Assignment 01a on Mimir, you must implement the Node, DoublyLinkedList, and Queue data structures so that they are able to store NetworkPacket data. Each of the functions for these classes is specified above.

Point Distribution

This part comprises 80% of the overall assignment score. That is distributed as follows:

- 10% - Report
  - 1% is automatically graded to remind you to update and submit the file
  - 9% is manually reviewed (remember to put references and answer the questions)
- 65% - Autograded
  - 5% for Node
  - 40% for DoublyLinkedList
  - 8% for Queue
  - 12% for the full program (runner with NetworkRouter as provided)
- 5% - Manual Code Review
  - Even though only 5% is set aside for manual review of this portion, note that autograded points can be revoked if attempts to circumvent the spirit of the assignment's autograding are discovered.
  - The goal is for these to be an easy 5 points just by commenting code, making things readable, and following the assignment goals.

In addition, 5% extra credit is available for submitting this part early, by February 4.

Part B Details

Task

For Assignment 01b on Mimir, you must make your data structures generic using templating. To make a templated class, you will replace all the instances of NetworkPacket with a generic template type. Because this change is incompatible with the code you wrote in the previous part, it is a separate assignment on Mimir.

Templated classes make use of the "template<typename T>" syntax, and instantiated objects use brackets to specify the type. For instance, Node<int> would specify a Node capable of holding an integer, while Node<NetworkPacket> would store a NetworkPacket like the Node created in the previous part.

Templates can be a little odd in C++. Because they only specify a pattern for a class or function, an instantiation in source is still needed to actually generate the corresponding functions in the object files. For instance, you will need to place "template class DoublyLinkedList<NetworkPacket>;" at the bottom of DoublyLinkedList.cpp for the linker to find an instance of the list that holds NetworkPacket objects. Those classes with only header files do not suffer this limitation

Fortunately, after writing the classes for 01a, all of the internal class logic should be resolved. For this part, you only need to focus on adapting to templated classes. The tests for each class are all-or-nothing—they either compile and link or not. In recognition of the difficulty in getting templates working correctly, this part has a larger percentage of points available from the manual code review.

Point Distribution

This part comprises 20% of the overall assignment score. That is distributed as follows:

- 10% - Autograded
    - 3% for Node
    - 5% for DoublyLinkedList
    - 1% for Queue
    - 1% for the full program
- 10% - Manual Code Review
    - This will include readability and commenting of code
    - You can gain partial credit for having some template code in place, even if it’s not to the point of running for the autograded tests

Additionally, 5% extra credit is available for small added challenges. See the Appendix for information about this.

The completed assignment for both parts is due by end of day February 11.

Appendix

Extra Credit Challenges

Blocking IP Addresses

Another application of a router beyond determining where next to send traffic (like our internal vs. external example), is when to *ignore* traffic. One popular form of cyberattack is called Denial of Service in which a remote endpoint, or many endpoints in a distributed attack, will send so many requests to a server that it becomes overwhelmed and unable to respond to any requests. These types of attacks have been used quite successfully in recent years to bring down major websites[4].

One popular method of countering a denial of service is maintaining a list of blocked source addresses. If a server is receiving too many requests from a specific location, it can temporarily or permanently add that IP address to an ignore list. Subsequent requests will not be processed.

For 3 points of extra credit, you can extend the NetworkRouter.h and NetworkRouter.cpp file to include a function `loadBlockedAddresses(std::string filename)`.

`loadBlockedAddresses(std::string filename)` reads from a file which has an IPv4 address on each line. These addresses should be stored in a list so it can be checked when processing requests if they should be sent on or ignored.

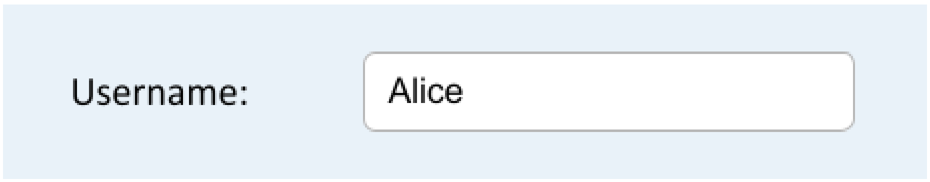
Hints:

- The Assignment 01b starter code includes a sample “blocked.txt” and runner for testing the loadBlockedAddresses behavior.
- You may use any list structure, such as a vector, to store the blocked addresses for this part. Your templated linked list would be able to store strings as well. You may need to add an instantiation such as “template class DoublyLinkedList<std::string>;” to the bottom of DoublyLinkedList.cpp for this purpose.
- You will need to add a small amount of logic in *processRequests()* to determine if the request comes from a blocked source before sending it on or not.

Filtering Packets with Potential SQL Injection

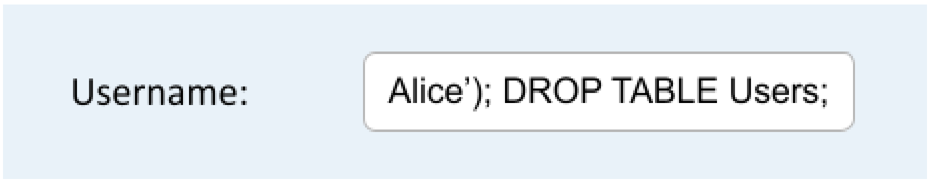
Another type of cyberattack which is perhaps less prominent today is called SQL Injection. The basic idea behind this attack is to attempt to leak database commands through a website’s interface.

For example, imagine Alice is registering a new account on a website. In the “Username” field, it presents a blank box for her to enter her preferred username. The website designers might expect this:


Username: Alice

[4] https://www.cloudflare.com/learning/ddos/famous-ddos-attacks/

But what is she were to enter something like this:



Username: Alice'); DROP TABLE Users;

Alice may be making a guess that the website is storing users in a specific kind of database with a table called "Users". Furthermore, if they're taking the user's input directly and feeding it into a database query, it might look something like this:

```
INSERT INTO Users (Username) VALUES ('<data from user form>');
```

`INSERT INTO Users (Username) VALUES ('Alice');` is perfectly innocuous and has the desired effect of adding an entry for the username "Alice".

`INSERT INTO Users (Username) VALUES ('Alice'); DROP TABLE Users; );` is decidedly not harmless, and in the language of databases, this could cause the system to delete the entire Users table. As is often the case, xkcd has a topically-relevant comic.

A method to prevent this type of attack is called input sanitization. It adds a processing step that ensures no database commands are present in the user-supplied values. There are a number of sophisticated ways to sanitize input. The most basic one is to simply find and replace dangerous symbols like semicolons[5].

For 2 points of extra credit, you can add a processing step in NetworkRouter.cpp's `processRequests()` function that causes it to ignore packet's whose data include **both** "POST" and a semicolon (;). POST requests are one of many ways to send data to a server, and we use a semicolon as a rough proxy for a potential database command.

Hint: You may "find" that std::string has a helpful search function for this.

Note: This is an extreme simplification of the problem, and we're playing with the network layers in this example. It's true that this type of raw content filtering could be performed in the days of HTTP traffic, but today, due to the preponderance of HTTPS-encrypted packet data, this type of content filtering or input cleaning is performed at the application layer rather than by a router.

[5] https://www.sqlshack.com/sanitizing-inputs-avoiding-security-usability-disasters/