Homework #6

Released: 02-15-2018
Due: 02-22-2018 11:59pm

In this homework, we are going to implement three classes (`IP_address`, `Datagram` and `Node`) and three operations of the simulation system (`System::create_machine`, `System::delete_machine` and `System::connect_machine`).

The `IP_address` class represents the IP address data type; the `Datagram` class models data segments transferred by the network; the `Node` class models host machines (server, laptop and WAN node) in our simulator. The three `System` member functions support the creation and removal of machines, and the operation of connecting two machines in the network. Their corresponding commands are `create`, `delete` and `connect`.

Please do not modify `machines.h` or the lines "`friend class Grader;`" in homework 6. These are needed for grading.

1 The `IP_address` Class

IP addresses are modeled by the `IP_address` class. This class abstracts the actual representation of IP address we had in homework 5 away from the user. This class contains a data member `std::array<int, 4> ip_` which internally implements the IP address as an array of four `int`s. Please finish the implementation of the `IP_address` class:

- The constructor parses the input string into an IP address. This is the `parse_IP` function we had in homework 5.
- The `operator==` member function compares whether two IP addresses are equal. Two IP addresses are the same if their four numbers are all equal.
- The `first_octad` member function returns the first number in an IP address. For example, if the IP address is `IP_address local_addr("192.168.0.1");`, then `local_addr.first_octad()` will return 192.

2 The `Datagram` Class

The `Datagram` class represents a piece of message together with its length, source IP and destination IP. This class models a segment of data that can be transferred by the network. Please finish the implementation of the `Datagram` class:

- The constructor initializes all fields in this class by its parameters. The `s`, `d` and `m` parameters represent the source, the destination and the message, respectively.
- The `get_destination` member function returns the destination IP address of the datagram.

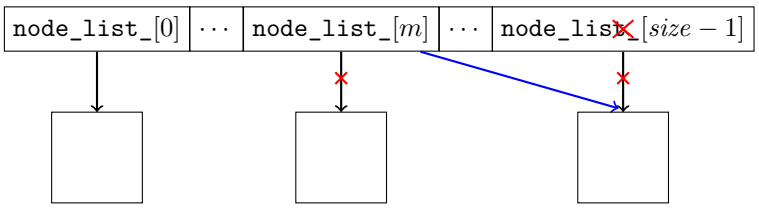
3 The `Node` Class

The `Node` class in `machines.cpp` models the machine in our simulator. For homeworks 6 and 7, all three categories of machines (laptop, server and WAN nodes) are represented by `Node`. The `Node` class contains a `name_` data member representing the name of the machine (not the type of the machine), a `local_ip_` data member representing the IP address of the machine and a `node_list_` data member containing a vector of shared pointers to the machines that are connected with the current machine. Please finish the implementation of the `Node` class:

- The `get_ip` member function returns the local IP address of a machine.
- The `connect` member function connects the current machine with a new machine. It simply appends the new machine in the connected list.
- The `disconnect` member function breaks the connection between the current machine and the designated machine. This member function removes the designated machine from the connected list.

  One way to implement this member function is to first find the index where the machine to be disconnected is stored, say `node_list_[m]`. Then assign `node_list_[m]` the value of `node_list_.back()`, and call `node_list_.pop_back()` to remove the now-repeated shared pointer reference `node_list_.pop_back()` from the end of the vector.

  In effect, this works as the following picture.



4 Some User Commands

Our entire network simulator is modeled by the `System` class. It includes some spaces to put machines (member variable `array<shared_ptr<Node>, 50> network_`) and six member functions that correspond to six commands in our simulator.

Every machine in this system has a unique IP address. Thus, when implementing commands, we will always designate a machine by its IP address. Now, we will extend our simulator to handle three of the six commands.

- Creating new machines: `System::create_machine`

  This member function finds the first empty (`nullptr`) slot in the `network_` array and creates a new machine by calling `make_shared<Node>(name, ip)`.

  The corresponding command for this member function is: "`create` *type name IP*". There are three valid types of machine: a `"laptop"`, a `"server"` and a `"wan"` node; the `type` argument must be one of three valid types.

  All three machine types are currently represented by `Node`.

- Removing existing machines: `System::delete_machine`

  This member function removes the designated machine from the system. Find the machine with the given IP address first, then invoke `disconnect` between this machine and all other machines in `network_` to disconnect them. Finally set the corresponding `network_[m]` to `nullptr`. where `m` is the index of the machine to be deleted.

  Without invoking `disconnect`, the connected machines will hold shared pointer reference to each other in `node_list_`. Thus, the machine will not be deleted if we only set `network_[m]` to `nullptr`. We must invoke `disconnect` to explicitly break the cyclic reference.

  The corresponding command for this member function is: "`delete` *IP*"

- Connecting two machines: `System::connect_machine`

  Given two IP addresses, call `connect` to connect these two machines to each other. For example, if `m1` and `m2` are the shared pointers to the two machine, then call `m1->connect(m2);` and `m2->connect(m1);`.

  Our connection between two machines will always be undirected; that is, the order doesn't matter. You can imagine the `connect` member function as plugging in the wire to the machine.

  The corresponding command for this member function is: "`connect` IP_1 IP_2"

5 Handling Errors

For error handling,

- `Node::disconnect` will never throw an exception. `Node::disconnect` simply does nothing if the given machine is not connected to this machine.
- `System::create_machine` should throw an `err_code::network_full` exception if the `network_` array is full (there are no `nullptr` elements). If the `type` argument is not one of three valid types, `System::create_machine` should throw an `err_code::unknown_machine_type` exception.
- `System::delete_machine` should throw an `err_code::no_such_machine` exception if the designated machine is not found in `network_`.
- `System::connect_machine` should throw an `err_code::no_such_machine` exception if either of the designated machines is not found in `network_`.

6 Unit Testing

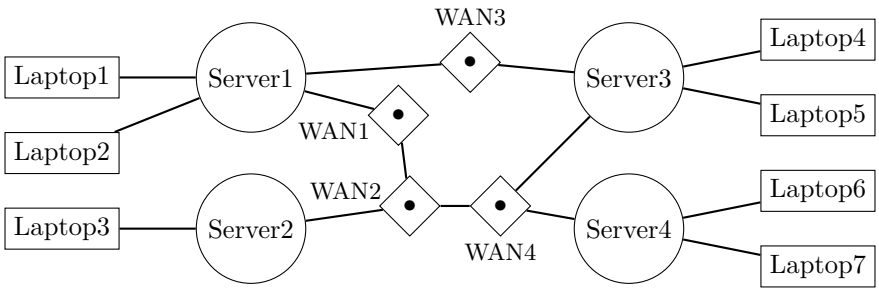
Implement comprehensive unit tests for four classes `IP_address`, `Datagram`, `Node` and `System`. Even though our unit tests cannot access private members in a class, we can still test member functions like `get_ip` and `get_destination`. We can also test for exceptions in the three operations we implemented in `System`.

To test `Node::disconnect`, you can use the `use_count()` member function of `std::shared_ptr`. To do this, first create shared pointers pointing to `Node`s, say `shared_ptr<Node> machine1`. Then, after connecting and disconnecting machines, check the result of `machine1.use_count()`. If `machine1` has correctly broken all connections with other machines, `machine1` will be the only reference to this machine and the use count will be 1. See the sample test case in the provided `networksim_test.cpp`.

Even though we cannot test `System::delete_machine` this way since we don't have access to `System::network_`, the `Node` class will print a message to `cout` upon destruction. You can at least utilize this to make sure your implementation of `System::delete_machine` is working.

Now, append the new unit tests in `networksim_test.cpp`. You have to figure out exactly what cases there might be and implement a corresponding unit test 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.

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`.