CS 2336 – Spring 2024

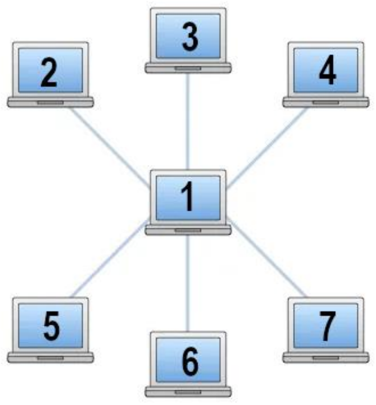
Network Simulation

Project 2 – Network Traffic / Host Pinging

In this project, you will expand upon your `FutureEventList` class by incorporating it into a simple network simulator capable of simulating single-hop (neighbor-to-neighbor) network traffic. This project will comprise the following tasks:

- Implement a linked-list version of the `FutureEventList`.
- Implement a new event representing a network message.
- Implement a simple host capable of sending and receiving network messages.
- Implement a custom Exception class for Event-related exceptions.

The network you will simulate is that of a star topology:



The host in the center is directly connected to the rest of the hosts. Network traffic (messages) traverse between hosts. This can be viewed as neighbor-to-neighbor message passing. For example, we can send messages from Host 1 to any other host, whereas any other host can send messages to Host 1.

The hosts in your simulated network will send messages, called *ping requests*, to neighboring hosts, in which the neighboring hosts will reply with another message type called *ping responses*. When the response to a ping request is received, the host will compute the *round-trip time (RTT)*. RTT is defined as the time interval from when a message (ping request) is sent to when a response to that message is received. For example, Host 1 sends a ping request to Host 5. Host 5 then responds by sending a ping response back to Host 1. The time interval between when Host 1 sent the request to when Host 1 received the response is the RTT of the message. This is also called *pinging* a host.

Implementation Details

Unlike in Project 1 in which you were given interfaces to implement, in this project you will be given abstract classes in addition to interfaces. These abstract classes are partially implemented, and you will need to extend them to complete the implementation. Please see each class for further information on what each method should do. Here are additional implementation details:

- The `FutureEventList` interface remains the same. You will need to implement a new class, `LinkedEventList`, which will have the following properties:
    - The implementation needs to use a *doubly-linked* list. You'll need to additionally implement a node class for the list nodes. You will need to have a field in your `LinkedEventList` class for the head node, but you will *not* need a tail node.
    - The `remove` method should *not* use a binary search. Binary search doesn't make much sense to use on a linked list, so use a linear search instead (use linear search for both `insert` and `remove`).
    - The `capacity` method should return the size of the linked list – the number of nodes currently in the list. This means we will define `size` and `capacity` as being the same.
- The `Event` class is now an abstract class. Certain functionality such as event id have been implemented for you.
    - An updated `Timer` class has been provided for you.
        - Look through this class implementation to gain a better understanding of how it works. Your code will not directly create and use these `Timer` events, but you'll work with methods that create and use them. This is very similar to a real-world project in which some code is already implemented, and you'll need to figure out how it works so you can implement your code properly.
    - You will need to create a new event class called `Message` by extending `Event`:
        - This will represent a network message.
        - In addition to implementing the abstract methods of the `Event` class, you will need to implement the following methods:
            - `getMessage`: returns a string representing the network message – we'll call this the *string message*.
            - `getSrcAddress`: returns an int representing the source address (sender host) of this message.
            - `getDestAddress`: returns an int representing the destination address (receiving host) of this message.
                - Source / destination hosts will be neighbors.

- `setNextHop`: this method has two parameters: the destination `Host` instance, and the distance between source and destination host. This is used by the `sendToNeighbor` method in the `Host` class. To simplify things, we assume that 1 distance = 1 simulation time (ie, if two connected hosts are a distance of 5 from each other, it'll take 5 simulation times units for a message to traverse from one to the other).

- We will use the *string message* as an indicator to determine what type of `Message` it is:
  - One string value will denote that the message is a *ping request*, and another to denote that the message is a *ping response*.
  - These string values will allow your `SimpleHost` to figure out the message type (ping request or ping response).
    - This enables you to use the 'switch on string' that Java supports!
- Make sure you understand how the implemented `Host` class methods work with `Message` objects. You cannot modify `Host`, so if your `Message` objects are not compatible with `Host` class methods, you'll need to make the appropriate changes in your `Message` class.

- An abstract class called `Host` has been provided. This class is partially implemented with functionality for processing events between hosts. You will create a child class called `SimpleHost`, implementing the abstract methods from `Host`. You need to additionally implement the following method:
  - `sendPings`: this method takes three parameters (all `int`):
    - `destAddr`: destination address of host to send ping requests.
    - `interval`: amount of time to wait between sending ping requests. For example, if this is 10, then send a ping request to the destination address every 10 simulation time units (starting at time 10).
    - `duration`: total amount of time in which the host will send ping requests. For example, if this is 35, then send pings until 35 simulation time units have passed.
  - The `sendPings` method will be called at the beginning of a simulation when the simulation time is 0. Thus, an example of using `sendPings` with `destAddr=5`, `interval=10`, and `duration=35`. At simulation time 10, send ping request, at time 20, send ping request, at time 30 send ping request, at time 35 stop sending ping requests.
    - You may want to use timers for this – the `Host` class has methods to work with timers on your behalf.
  - Your `SimpleHost` class needs to handle sending ping requests, receiving ping responses, and handling ping requests which are received:

        - Determine the message type by looking at the value of the *string message* (ie, come up with a unique string for each message type).
        - If receiving a ping request, you'll need to reply with a ping response.
        - If receiving a ping response, compute the RTT.
    - The `Host` class has many methods implemented for you – look through this class to get an understanding of how it works, and which methods may be useful.
    - Some methods throw instances of `EventException`. You will need to implement this exception class by extending a base exception class – your new exception class should be an *unchecked* exception.
- Create a class called `Main` which contains the `main` method. The `main` method can be used to open a `simulation.txt` file, read from it, etc, or you can create another method (or even a class) for this task.
    - Open a file called '`simulation.txt`'. This file will be numbered (`simulation1.txt` for example) and is formatted according to the following:
        - The first line is a single integer, representing the host address in the middle of the star topology. We'll call this the *first host*.
            - When you instantiate a `SimpleHost`, you'll need to also call a method in Host to give your host its address and a reference to the `FutureEventList` object.
        - The next *n* lines contain two integers, the first being the address of a host connected to the first host, and the second being the distance between them.
            - Keep reading these until you read a -1 for the host address, that's the sentinel value to stop.
            - For each host read, you'll need to add the first host as a neighbor – the `Host` class has a method for this. Remember that these connections are bi-directional, so you'll need to also add each host read as a neighbor for the first host.
        - The remaining lines in the file are to bootstrap the simulation by providing information on which hosts should send ping requests. These lines contain four integers: address of host which will send ping requests, address of host to receive ping requests (will be a neighbor), time interval in which to send consecutive ping requests, and the total time duration in which the host should be sending ping requests.
            - After reading each line, you can pass the appropriate information to the relevant host – this will bootstrap the

simulation, ie, populate the `FutureEventList` with initial `Events`.

  - Have the `simulation.txt` file in the current working directory. This means that when you open it, you do not need to put any directory prefix, only the filename itself. If working in IntelliJ IDEA, the file should be in your project root folder (not in the src folder – that is where your code is). If working in OnlineGDB, files are by default created in the current working directory.
  - Once you have read the file and bootstrapped the simulation, the only thing left to do is 'run' the simulation. For discrete event driven simulations, this entails popping the first event from the `FutureEventList` in a loop until there are no more events left to pop. You'll notice that simply doing this produces a sequence of `Message` sending and receiving which looks like it could have come from a real network!
    - Remember to call the `handle` method of each event you pop, otherwise your simulation will stall.

Assumptions

- A host will only send ping requests to one other host.
- The time interval between sending ping requests will always be greater than the RTT to send them. For example, if sending a ping request to a host results in an RTT of 5, then the interval of sending ping requests will be greater than 5. This way you will always receive the ping response for a ping request before having to send the next ping request.
- The `simulation.txt` file is complete – nothing is missing.
- You *cannot* modify the `Event`, `Timer`, `Host`, or `FutureEventList` classes.

Example Output #1:
Suppose your program is run with the following `simulation1.txt` file:

```
5
6 2
7 3
-1
5 7 10 28
```

The file above states that the first node has address 5, and is connected to two hosts with addresses 6 and 7 respectively. Here, host 5 will send ping requests to host 7 starting at time 10, every 10 simulation time units, until time 28.

Expected output should look like this:

```
[10ts] Host 5: Sent ping to host 7
[13ts] Host 7: Ping request from host 5
[16ts] Host 5: Ping response from host 7 (RTT = 6ts)
[20ts] Host 5: Sent ping to host 7
[23ts] Host 7: Ping request from host 5
[26ts] Host 5: Ping response from host 7 (RTT = 6ts)
[28ts] Host 5: Stopped sending pings
```

The output format is as follows: the value in the brackets is the simulation time in which the output took place (`ts` is just a made-up unit for the simulation time). Next is the host address which generated the output, followed by the output itself.

Example Output #2:
Suppose your program is run with the following `simulation2.txt` file:

```
5
6 2
7 3
-1
5 7 10 35
```

Expected output should look like this:

```
[10ts] Host 5: Sent ping to host 7
[13ts] Host 7: Ping request from host 5
[16ts] Host 5: Ping response from host 7 (RTT = 6ts)
[20ts] Host 5: Sent ping to host 7
[23ts] Host 7: Ping request from host 5
[26ts] Host 5: Ping response from host 7 (RTT = 6ts)
[30ts] Host 5: Sent ping to host 7
[33ts] Host 7: Ping request from host 5
[35ts] Host 5: Stopped sending pings
[36ts] Host 5: Ping response from host 7 (RTT = 6ts)
```

Example Output #3:
Suppose your program is run with the following `simulation3.txt` file:

```
5
6 2
7 3
-1
5 6 5 31
```

Expected output should look like this:

```
[5ts] Host 5: Sent ping to host 6
[7ts] Host 6: Ping request from host 5
[9ts] Host 5: Ping response from host 6 (RTT = 4ts)
[10ts] Host 5: Sent ping to host 6
[12ts] Host 6: Ping request from host 5
[14ts] Host 5: Ping response from host 6 (RTT = 4ts)
[15ts] Host 5: Sent ping to host 6
[17ts] Host 6: Ping request from host 5
[19ts] Host 5: Ping response from host 6 (RTT = 4ts)
[20ts] Host 5: Sent ping to host 6
[22ts] Host 6: Ping request from host 5
[24ts] Host 5: Ping response from host 6 (RTT = 4ts)
[25ts] Host 5: Sent ping to host 6
[27ts] Host 6: Ping request from host 5
[29ts] Host 5: Ping response from host 6 (RTT = 4ts)
[30ts] Host 5: Sent ping to host 6
[31ts] Host 5: Stopped sending pings
[32ts] Host 6: Ping request from host 5
[34ts] Host 5: Ping response from host 6 (RTT = 4ts)
```

Example Output #4:
Suppose your program is run with the following `simulation4.txt` file:

```
3
1 2
2 1
5 11
7 9
-1
1 3 5 17
2 3 5 18
3 1 6 25
7 3 30 90
```

Expected output should look like this:

```
[5ts] Host 1: Sent ping to host 3
[5ts] Host 2: Sent ping to host 3
[6ts] Host 3: Sent ping to host 1
[6ts] Host 3: Ping request from host 2
[7ts] Host 3: Ping request from host 1
[7ts] Host 2: Ping response from host 3 (RTT = 2ts)
[8ts] Host 1: Ping request from host 3
[9ts] Host 1: Ping response from host 3 (RTT = 4ts)
[10ts] Host 1: Sent ping to host 3
[10ts] Host 2: Sent ping to host 3
[10ts] Host 3: Ping response from host 1 (RTT = 4ts)
[11ts] Host 3: Ping request from host 2
[12ts] Host 3: Sent ping to host 1
[12ts] Host 3: Ping request from host 1
[12ts] Host 2: Ping response from host 3 (RTT = 2ts)
[14ts] Host 1: Ping request from host 3
[14ts] Host 1: Ping response from host 3 (RTT = 4ts)
[15ts] Host 1: Sent ping to host 3
[15ts] Host 2: Sent ping to host 3
[16ts] Host 3: Ping response from host 1 (RTT = 4ts)
[16ts] Host 3: Ping request from host 2
[17ts] Host 1: Stopped sending pings
[17ts] Host 3: Ping request from host 1
[17ts] Host 2: Ping response from host 3 (RTT = 2ts)
[18ts] Host 2: Stopped sending pings
[18ts] Host 3: Sent ping to host 1
[19ts] Host 1: Ping response from host 3 (RTT = 4ts)
[20ts] Host 1: Ping request from host 3
[22ts] Host 3: Ping response from host 1 (RTT = 4ts)
[24ts] Host 3: Sent ping to host 1
[25ts] Host 3: Stopped sending pings
[26ts] Host 1: Ping request from host 3
[28ts] Host 3: Ping response from host 1 (RTT = 4ts)
[30ts] Host 7: Sent ping to host 3
[39ts] Host 3: Ping request from host 7
[48ts] Host 7: Ping response from host 3 (RTT = 18ts)
[60ts] Host 7: Sent ping to host 3
[69ts] Host 3: Ping request from host 7
[78ts] Host 7: Ping response from host 3 (RTT = 18ts)
[90ts] Host 7: Stopped sending pings
```

Submission Instructions:

Please submit your code in a single zip file. Do *not* submit the classes provided to you, but only your classes: `LinkedEventList`, your node class, `Message`, `SimpleHost`, `EventException`, and `Main`. This must be one class, one file. Each class has to be in a separate .java file named the same as the class.

Tips to Approach this Project:

This doc is a specification for how to write the project but does not explain every aspect of it. If something is not defined in the specification, you can implement it how you like.

Do *not* collaborate with other students, this is an individual effort.

Be sure to start early, and work on it consistently. If you get stuck, work on some other part of the project or take a break from the project entirely. If you find yourself stuck and want to ask for help, your course instructor, your course TA, and the CSMC are all available to provide guidance. But most important (as with the last project), have fun!

Additional tips:

- There are no `toString` implementation requirements in the project. The `Event` class does contain a `toString` implementation, and I encourage you to use it when debugging, for example to see if the events in your `LinkedEventList` are being inserted in proper order.
- Start with simple networks first – one with only two hosts, where one host sends a single ping request to the other. Make sure the other host responds to it. Then code your way up to more difficult scenarios.
- Read the Javadoc documentation in the classes provided so you understand not only what the methods you implement are supposed to do, but also what the implemented methods do.
- For output which occurs at the same simulation time, the ordering can be different than what is shown above in the example output.

Additional References

If you want to learn more about network simulation, please check out OMNeT++: https://omnetpp.org/

OMNeT++ is an open-source, fully functional discrete-time event simulator written in C++. The source code can give you ideas on how to structure your own code, and may give inspiration for awesome extensions you can do to these projects (for example visualizing the network simulation). A secondary goal of these projects is to give you something you can extend and turn into a personal project.