CS 2336 – Spring 2024

Network Simulation

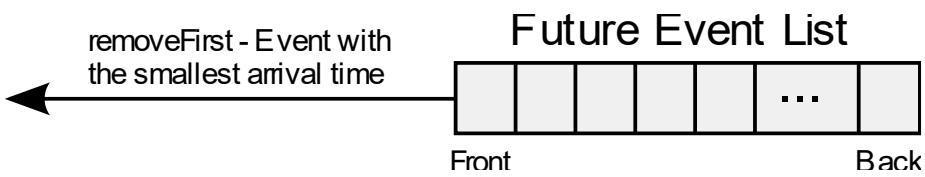
Project 1 – Timers

The theme for this semester's projects is computer networking. Whether it is your phone or your laptop, computer networks are used frequently. When designing networked programs or protocols, it may be necessary to prototype and test in a deterministic environment, ie one in which the network behavior is consistent between sessions. One such way to do this is to use a network simulator. For example, suppose you are designing an online gaming protocol which minimizes the effect of lag on the player. Suppose the protocol fails under specific high-lag conditions which rarely occur in your physical test network. You could use a network simulator to deterministically test the protocol and (hopefully) fix the issue. Other uses of network simulators include the building of very large networks virtually which would otherwise be intractable in terms of cost and size.

Through this project and the next two, you will build a functional computer network simulator! We will focus on *discrete-time event simulation*, in which everything occurring in the network is a discrete event. An event could be sending a packet from one host to another, setting a recurring timer to send packets every *n* seconds, etc. Each event occurs at a specific time, called its *arrival time*, and it is assumed that nothing happens between events (hence the term 'discrete'). Therefore, time progresses discretely each time an event occurs. Assuming the events are stored in a list structure, the steps a simulator takes can be generalized like so:

1. Remove the event with the smallest arrival time.
2. Update the simulation time to be the removed event's arrival time. This is how time progresses in a discrete event-driven simulation.
3. Execute (or handle) the event. We will assume that event execution time is zero – the only way to advance the simulation time is by removing events.
4. Add any additional events which need to occur at some future time.
5. Go back to step 1.

At the heart of a discrete-time event-driven simulator is the *future event list*, which handles ordering of events which should occur. When an event is created, it is put into the future event list in order of arrival time. All events to handle are stored in this list.



Here, `removeFirst` is the method used to remove the next event from the list (the event at the front of the list). The current time in the simulation will be updated to this event's arrival time, and the event itself will be handled, including adding additional events to the list. The next event is removed and handled, and this loop keeps going until the list is empty.

One way to visualize how this works is to consider timers. A timer is an event whose arrival time is the simulation time in which the event is inserted into the future event list + a duration. For example, assume we start a new simulation, so the current simulation time will be 0. If a timer event is created with a duration of 10, then the timer will be inserted into the future event list with an arrival time of 0 + 10 = 10. Assuming this is the only event in the list, it will be immediately removed and handled, with the simulation time updated to 10. Now our simulation time is 10. Now suppose another two timers are added to the list. The first has a duration of 17. With the current simulation time at 10, the arrival time of this event is thus 10 + 17 = 27. The second has a duration of 5. Simulation time is only increased when an event is removed, so this timer will have an arrival time of 10 + 5 = 15. Thus, the second timer will be removed first, simulation time increased to 15, timer handled, then the first timer will be removed, simulation time increased to 27, and timer handled.

Notice that the time here does not have units. If you wanted the time to represent seconds, you could have the simulator wait (maybe by calling `sleep`) the appropriate number of seconds between removing events. For the example above, the simulator would wait 10 seconds before removing the first timer event, 5 seconds before removing the next one, and 12 seconds before removing the last one. However, one nice property of discrete-time event simulators is the ability to simulate something over potentially large periods of time as fast as the events can be removed and handled. Hence, we will not be actually simulating the passing of time, just jumping to each discrete point in time.

Project Description

Implement a future event list and a timer event to use with it.

This future event list implementation will use an array (*not* an ArrayList) to store the events in the list. Whenever an event is inserted, the arrival time ordering must be preserved. Each event will have a getter method for this arrival time. When removing an event from the list, it will always be the first event in the list (the ordering property assures that the first event in the list at any given time will be the event with the smallest arrival time – the next event scheduled to be handled). Essentially what you are implementing is an array-based priority queue in which arrival time is the sort criteria.

Two interfaces are provided to help you get started: `Event` and `FutureEventList`. If you do not know what an interface is, do not worry, we'll discuss them in class, but for now you can think of an interface as a parent class in which all the methods are abstract or not implemented (a pure virtual function in C++). Each method declaration includes a comment explaining what the method should do, so please consult the comments to get a better understanding of how to implement the

methods. When you create a class and *implement* the interface, you are implementing all the methods declared in the interface.

Implementation Details:

- The future event list will store objects of type `Event`. To implement this interface, we'll create a simple concrete class called `Timer`.
    - Your `Timer` class needs to implement the `Event` interface. This entails implementing all the methods in the interface.
    - Each `Timer` object should have a unique id, starting from 0, and incrementing by 1 each time a new `Timer` object is created.
    - The constructor of your `Timer` class should have one parameter: the timer's duration. This is how long the timer stays in the future event list (ie, used to compute the arrival time and determines the order of the timer in the list)
    - Insertion time is the simulation time in which the timer is inserted into the future event list. Arrival time is the simulation time in which the event is removed from the list (arrival time = insertion time + duration).
    - The `handle` method should output to the console the timer's unique id and arrival time. Formatting should look like this (all on one line):
      `Timer <timer_id> handled (arrival time: <arrival_time>)`
        - Replace *<timer_id>* and *<arrival_time>* with the appropriate values.
    - The `cancel` method should output to the console the timer's unique id and cancelation time. The cancelation time is simply the simulation time when the cancelation takes place, ie, the arrival time of the most recently removed event. Formatting should look like this (all on one line):
      `Timer <timer_id> canceled at time: <sim_time>`
        - Replace *<timer_id>* and *<sim_time>* with the appropriate values.
    - Implement an additional method not declared in the interface called `toString`, with the following signature: `public String toString()`
      This is a special method which allows a string representation of the object to be returned implicitly in certain situations without having to call `toString` explicitly. For example, a `Timer` object named `tmr` could have its string representation output to the console like this: `System.out.println(tmr);`
      Notice that we did not explicitly call `tmr.toString()`.
        - Implement the `toString` method to output to the console the timer's unique id, insertion time, and arrival time. Formatting should look like this (all on one line):
          `- Timer <timer_id> (insertion time: <insertion_time>, arrival time: <arrival_time>)`
            - Replace *<insertion_time>* and *<arrival_time>* with the appropriate values.

- The event list itself will be of type `FutureEventList`. This interface needs to be implemented as another class. Name your class `ArrayEventList`.
    - The underlying array needs to be a primitive java array of type `Event`. Do *not* use an `ArrayList`!
    - The underlying array needs to be initialized with a length of 5 elements. When 5 events are being stored in the array and another event is added to the array, the array length should be doubled to 10 (double the number of elements the array can hold). If 10 events are being stored in the array and another event is added, double the array length to 20, etc.
    - Your `ArrayEventList` must keep track of the current simulation time. Only events removed to be handled advance the time, canceled events do *not* advance the time. The `getSimulationTime` method returns the current simulation time.
    - The `remove` method must be implemented using a *recursive* binary search:
        - You are searching the array for any `Event` object which is equal to the `Event` object given as the argument (`Event e`). This will be used for canceling events, as the event to cancel may not be the event at the front of the list.
        - When comparing `Event` objects, use the following criteria:
            - Two `Event` objects are equal if they are both the same object. To check for this, simply use the '==' operator.
            - One `Event` object is less than the other if the arrival time is less than the arrival time of the other object (and vice versa for greater than). In other words, to compare two Events to see if one is less than or greater than the other, only use arrival time, which you can get from the Event's `getArrivalTime` method.
            - There may be multiple events that share the same arrival time but are not the same object. This is the case in which the events are not equal using '==' but splitting the array does not work because the arrival times *are* equal (so there is no split to lower or upper half). Because the array is sorted on arrival time, your binary search needs to support multiple events with the same arrival time. If two `Event` objects are not equal (not the same object), but the arrival times *are* equal, then use any technique you wish to further search the array for other events which may be equal (same object) to the event you are searching for.
    - You *cannot* create additional methods in your `ArrayEventList` class outside of those methods given in the `FutureEventList` interface, unless those methods are private. You can create as many private methods as you need.
- Create a class called `Main` which contains the `main` method. The `main` method can be used to open the `events.txt` file, read from it, etc, or you can create another method (or even a class) for this task.
    - Open a file called '`Events.txt`'. This file is formatted such that each line is a command to perform regarding events (in this case timers). Each line in the file

starts with a single character, and may be followed by a space and an integer. Here are the commands:

- *I <duration>*
  - Insert a new `Timer` event into the future event list with the following duration
  - After inserting, print to the console the new timer's `toString` method return string. You can simply pass the object itself to `System.out.println` without explicitly calling `toString`.
- *R*
  - Remove the the timer at the front of the list, using the `removeFirst` method. This will be the timer with the smallest arrival time in the list.
  - After timer removal, call its `handle` method.
- *C <timer_id>*
  - Cancel the timer with id <timer_id>. Because the `remove` method does not take a timer id as an argument, but rather the `Timer` object itself, you should probably cache each `Timer` object as you create them. One way to do this is to create an array in the main method and simply append each new timer to it. Because timer id starts at 0 and increments by 1 each time a timer is created, the index can be the key to lookup the timer if you need to cancel it (array index corresponds to timer id). Lookup the timer in the array, then pass the corresponding `Timer` object to the future event list's `remove` method. For this part you are allowed to use an `ArrayList`.
  - After timer removal, call its `cancel` method.

- Once all commands are read, print to the console the following:
  - `Future event list size: <size>`
    `Future event list capacity: <capacity>`
    - `<size>` and `<capacity>` are the future event list's size and capacity, respectively. These values can be obtained by calling the `size` and `capacity` methods of your future event list object.
- Assume that the `events.txt` file is complete – no entries are missing in a message.
- Use a `Scanner` object to read the `events.txt` file.
- Have the `events.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.
- For each line in the `events.txt` file, process the command, then move to the next command, until there is nothing more to read.

Example Output #1:

Suppose your program is run with the following `events.txt` file:

```
I 12
I 99
I 51
R
I 17
I 3
R
R
I 25
R
R
```

Expected output should look like this:

```
Timer 0 (insertion time: 0, arrival time: 12)
Timer 1 (insertion time: 0, arrival time: 99)
Timer 2 (insertion time: 0, arrival time: 51)
Timer 0 handled (arrival time: 12)
Timer 3 (insertion time: 12, arrival time: 29)
Timer 4 (insertion time: 12, arrival time: 15)
Timer 4 handled (arrival time: 15)
Timer 3 handled (arrival time: 29)
Timer 5 (insertion time: 29, arrival time: 54)
Timer 2 handled (arrival time: 51)
Timer 5 handled (arrival time: 54)

Future event list size: 1
Future event list capacity: 5
```

Example output #2

Suppose your program is run with the following `events.txt` file:

```
I 20
I 52
I 27
R
I 17
R
C 1
I 31
I 20
I 2
I 5
I 72
R
R
I 7
C 4
C 5
R
```

Expected output should look like this:

```
Timer 0 (insertion time: 0, arrival time: 20)
Timer 1 (insertion time: 0, arrival time: 52)
Timer 2 (insertion time: 0, arrival time: 27)
Timer 0 handled (arrival time: 20)
Timer 3 (insertion time: 20, arrival time: 37)
Timer 2 handled (arrival time: 27)
Timer 1 canceled at time: 27
Timer 4 (insertion time: 27, arrival time: 58)
Timer 5 (insertion time: 27, arrival time: 47)
Timer 6 (insertion time: 27, arrival time: 29)
Timer 7 (insertion time: 27, arrival time: 32)
Timer 8 (insertion time: 27, arrival time: 99)
Timer 6 handled (arrival time: 29)
Timer 7 handled (arrival time: 32)
Timer 9 (insertion time: 32, arrival time: 39)
Timer 4 canceled at time: 32
Timer 5 canceled at time: 32
Timer 3 handled (arrival time: 37)

Future event list size: 2
Future event list capacity: 10
```

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 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, have fun!

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 also 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.