Project 3 (P3): Distance Vector Routing (`DV`)

Instructor: Dr. Shengquan Wang

Due Time: 10PM, 12/3/2016

The purpose of this project is to learn how distributed dynamic routing protocols like distance vector accomplish routing in practice. In this assignment, you will be asked to build a distance vector routing protocol that implements the distributed Bellman-Ford algorithm.

1 Design

Data Each router is assigned with a unique router ID (English letter). Each router needs to maintain three different data:

- All direct link cost information to all active neighbors. Only direct link cost information. If you maintain all links, it will be Link State Protocol. Then you will forfeit the purpose of this project.
- Its own distance vector. It will look like this for Router A:

  ```
  A B 4 2 C
  A C 3 1 C
  A D 6 3 C
  ```

  There are three entries in this distance vector. For each entry, the five columns are source router ID, destination router ID, distance, number of hops, the router ID for the next router in routing, respectively.
- Its neighboring routers' distance vectors.

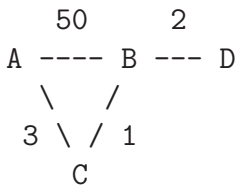
Events Each router will set up a UDP socket and should alway be listening. Each router will communicate with each of its neighboring routers through their UDP sockets:

- For any change in its distance vector, the router will propagate the updated distance vector entries to all of its neighboring routers.
- Upon either (i) receiving any distance vector entries from any neighboring router, or (ii) detecting a new neighbor, or (iii) detecting the link cost change to a neighbor, the router will use the `Bellman-Ford Equation` to update its distance vector.

The above two events might form a circular loop.

2 Testing

We consider the following network topology:



We assume all links are symmetric in terms of link cost in both directions.

Start all routers We will use the running processes to simulate routers. Start 4 processes as 4 routers in 4 different terminals. We assume they run on the same host. Each router is assigned with a unique router ID (English letter). Its running process is assigned with a UDP port number as a receiver. We assume they are

```
Router A: 1000
Router B: 2000
Router C: 3000
Router D: 4000
```

We use the same program (named as `DVRouter`) for all routers. For example with a Java program, run Router A in the shell as (the 2nd line is the displaying message)

```
prompt> java DVRouter A 1000
Router A is active. Waiting for adding links ...
```

Make sure all four routers are constantly active before adding any link.

Add all links Next, we will use a 5th terminal to run a control process (named as `Link`) to add all links. For example if we want to add a link between A and B with a link cost 50, we run it in the shell as:

```
prompt> java Link A 1000 B 2000 50
```

A message will be sent from the `Link` control process to both A and B, and then A and B will add this link to its data. The `Link` control process will terminate after this. The links should be added in the order of `AB, AC, BC`, and `BD`.

Change the cost of a link Once the system has been stabilized for a while (i.e., no more communication of distance vectors), a link cost change from 1 to 60 for B-C is detected. In order to simulate this, we will start another `Link` control process on the 5th terminal to change the link cost of B-C. We run it in the shell as:

```
prompt> java Link B 2000 C 3000 60
```

A message will be sent to both B and C. Both B and C will find an existing link B-C in their data and update its cost with 60, respectively.

In the adding or changing action, whenever a router updates it own distance vector, it should display it as the information on the console as the following example:

```
- - - - -
A B 4 2 C
A C 3 1 C
A D 6 3 C
- - - - -
```

This is the only debugging information the grader cares about.

Simulate Delay Since all routers are simulated in one host, the delay between routers are negligible. Therefore, in order to be more realistic, we will add extra delay in the communication. For any receiver UDP socket, upon receiving a message, the receiver needs to pause for $c * 10$ ms before processing this message, where c is the link cost.[1] For example, if B receives a message from A, it will wait for 500 ms.

Applying Poison Reverse Next, apply `Poison Reverse` technique in the design of the `DV`. Rerun all the above steps again. We should see fewer displaying of distance vectors and faster convergence.

3 Submission

You are going to report the following things:

(a) Describe in details how you implemented the `DV`.

(b) Write a detailed report with the screenshots of the testing results for the two scenarios: without `Poison Reverse` and with `Poison Reverse`. In addition, please report the differences in terms of the number of displaying and the time to converge. Each screenshot should include your username and the current time, which show that you did it by yourself.

(c) Specify the contribution made by each member if you work as a group.

The report should be written in a ".docx", ".doc", or ".pdf" format. All source codes should be well formatted with good comments. Submit the report and the source code files to Project P3 on Canvas. Any compression file format such as `.zip` is not permitted.

[1] Check this article "Pausing Execution with Sleep" `https://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html`