Fall 2011 - CSEE W4119 Computer Networks
Programming Assignment 2 - Simple Link State Routing

Prof. Gil Zussman

due: 12/12/2011, 11:00 AM, EST
Please start early, no extensions will be given

1 Introduction

The objective of this programming assignment is to implement a simplified version of a Link-State routing protocol that runs in a static network. You are required to write a program which builds the nodes' routing tables based on the information (network topology and distances/costs between neighboring nodes) flooded within the network. The UDP protocol along with the *PI* algorithm should be used to exchange the routing information among the nodes in the network. The *Dijkstra* algorithm should be used to build the routing table at each node.

2 Network Model

We assume that all the nodes run on the same machine, and they all have the same IP address. Nodes can be identified uniquely by their (UDP listening) port numbers, which are specified by the user. The port numbers must be > 1024. The maximum number of nodes to support is 16. The links among the nodes in the network and the distances (non-negative integer) between two directly connected nodes are specified by the user upon the activation of the program and stay static throughout the session. Thus, there is no need to support dynamic link/node status changes. The link distance is the same for either direction (e.g., between node A and B, $\text{Distance}_{A \to B} = \text{Distance}_{B \to A}$.)

3 Protocol

For this assignment, we will use the UDP protocol to exchange the routing information among the nodes.

4 Command Syntax

The program name should be `lsnode`. For each node in the network, the command syntax is:

```
$ lsnode <local-port> <neighbor1-port> <distance1> <neighbor2-port> <distance2> ... <last>
```

`lsnode`	Program name.
`<local-port>`	The UDP listening port number of the node.
`<neighbor#-port>`	The UDP listening port number of one of the neighboring nodes.
`<distance#>`	The link distance to the `<neighbor1-port>`.
	Keep listing the pair of `<neighbor-port>` and `<distance>` for all your neighboring nodes.
`<last>`	Indication of providing information regarding the last node in the network. Upon the input of the command with this argument, the verification message exchanges among the nodes should kick in.
`exit (ctrl-C)`	Exit the program.

5 Program Steps

5.1 Step 1: Input Network Model Information

The first step is to input the network model information per node. Note that you will execute as many instances of the program (`lsnode`) as the number of nodes in the network, and you will input the 'partial' network topology information that each node should know, i.e., port number of neighboring (single-hop) nodes and the costs of the links connected to them. Assume that the network is *static* and has *no hierarchy* (e.g., network connected to other subnets). For each node, you will input the names of the neighboring nodes along with the costs to reach them, as specified in the command syntax section. Suppose we have the network topology described below.

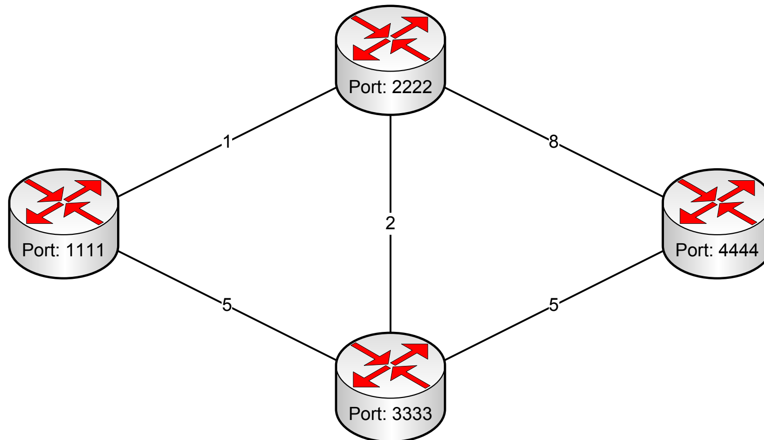

Figure 1: Network Topology Example

Then your initial inputs to process the network topology information will be (from the command line prompt):

```
$ lsnode 1111 2222 1 3333 5
$ lsnode 2222 1111 1 3333 2 4444 8
$ lsnode 3333 1111 5 3333 2 4444 5
$ lsnode 4444 2222 8 3333 5 last
```

Each line is executing one `lsnode` program for one node along with its network topology information. Based on these inputs, every node should have the initial link-state information locally.

5.2 Step 2: Verify the Link Status

The second step is for all nodes to verify the link status. Verifying the link status, here, means for each node, (1) it can reach the neighboring nodes specified in Step 1, and (2) the cost information that it has matches with the information that the neighbors have.

In the example above, as soon as the command for node `4444` is entered, node `4444` should start sending queries to its neighbors, i.e., `2222` and `3333`. Each node (except `4444`) should start querying its neighbors as soon as it receives the first query from one of its neighbors.

We do not specify the hand-shaking protocol nor the format of these messages as long as you can verify the link status. During the process, if any of the nodes finds (1) the link is broken, or (2) the cost information does not match with its neighbor's cost, the program instance for that node should report the problem and quit. (This can happen if your UDP packet exchange fails, or the network model you input in Step 1 has some flaw.)

5.3 Step 3: Broadcast Routing Information (Message *Flooding*)

After the node verifies the links' status, it should wait for 1 second (to make sure all the nodes finished the verification process) and then start broadcasting its link-state information to the rest of nodes. You should use some version of the *PI* algorithm covered during the class to manage the process. Again, we do not specify the format of the messages, but you **should not** use any built-in broadcasting functionality.

5.4 Step 4: Wait for Flooding to Settle and Run *Link-State* Algorithm

Since each node does not have the full knowledge the network topology, it can not tell whether it has already received messages (link-state information) from all nodes within the network. For simplicity, each node should wait for 5 seconds after receiving a new topology message. If no messages arrive during the 5 seconds time period, then the node should determine that the flooding is finished and start generating its routing table by using *Dijkstra* algorithm.

At the end of Step 4, each node should output a routing table in the format like Table 4.3 in the text book (page 379).[1]

5.5 Step 5: Send a data packet

Once the routing tables are created the nodes should use them in order to send one data packet from the node entered last (indicated by "last") to the farthest away node in that node's routing table. The packet should be forwarded along the shortest path between the two nodes.

6 Status Messages

The status messages should be displayed at least for the following events. The example of the messages are also shown below.

1. Message sent (will be used for all type of messages including the forwarding of the data packet):
   `Message sent from Node (Port xxxx) to Node (Port vvvv)`

2. Message received (will be used for all type of messages including the forwarding of the data packet):
   `Message received at Node (Port vvvv) from Node (Port xxxx)`

3. Link failure:
   ```
   Node (Port vvvv) can not reach Node (Port xxxx)
   Link failure occurred. Program terminated.
   ```

4. Link cost mismatch:
   ```
   Link cost from Node (Port vvvv) to Node (Port xxxx) is 123.
   Link cost from Node (Port xxxx) to Node (Port yyyy) is 124.
   Link cost does not match. Program terminated.
   ```

5. Link-State message flooding start:
   `Message flooding started.`

6. Link-State message flooding finish:
   `Message flooding finished.`

7. After 5 seconds time window, summary of link-state information received at Node (Port xxxx):
   ```
   Node (Port xxxx) received the following link-state information:
   source Node (Port xxxx): Neighbors Node (Port vvvv) (Cost 123), Neighbors Node (Port vvxx) (Cost 111)
   ```

[1] Additionally, you could store the "node information" in XML format, or you could go more creative by displaying the complete topology graphically using java classes, or you could do something else altogether! Creative ideas will fetch you more points.

```
source Node (Port yyyy): Neighbors Node (Port vvyy) (Cost 123), Neighbors Node (Port yyxx) (Cost 122)
⋮
```

8. Beginning routing table construction:
   `Node (Port vvvv) starts routing table construction.`

9. Routing table (at the end of the process):
   See Table 4.3 in the text book (page 379).

7 Program Structure

The program should run an independent process per node. The program structure is left for you to design and implement, but make sure you test it thoroughly.

8 Submission Instructions

8.1 Submission Package

It is strongly recommended to work on either C or Java, both available at CS department CLIC or EE ILAB. Your submission package should include the following deliverables.

- `README.pdf (or .txt)`: The file contains the project documentation, program features, usage scenarios, brief explanation of algorithms or data structures used, and the description of any additional features/functions you implemented.

- `Makefile`: Make sure your source codes are compilable. Include `Makefile` to build your application. If written in C, the name of the output file must be `lsnode` and the class file in Java must be `lsnode.class`. Also, all input parameters MUST be command line arguments.

- Your source code: Do not include any executables in the folder. We will only run executables that we compile using 'make'.

- `test.txt`: This file should contain some output samples on several test cases. This would help others to understand how your programs work in each test scenario. It is optional to include this, as part of your `README.pdf` document.

The submission should be made via Courseworks. Zip all the deliverables mentioned above, and name the zip file as `<your UNI>PA2<CLIC or ILAB>.zip` (e.g. `gz2136PA2ILAB.zip` for Professor Zussman.) Upload the file in the Courseworks, under `Class File -> Shared Files -> PA2`.

8.2 Grading Policy

In the grading of your work, we will take the following points into account.

- The documentation clearly describes your work and the test result.

- The source codes are complied properly by using the `Makefile` and generate appropriate output files.

- The programs run properly, including 1) take appropriate commands and arguments, 2) handle different situations and support required functions, and 3) display necessary status messages.

□