LINK STATE ROUTING & FORWARDING TABLES

OVERVIEW

This assignment requires you to implement the link state routing algorithm to compute a forwarding table of a router as described in class. You must use UDP message passing between emulated routers in the network. Each router is modeled as a process executing on a host machine. Base your program on the UDP sender and receiver you implemented in previous projects.

This project requires you to write a program in ANSI C that emulates a router in a physical network where individual routers are logically connected to each other. Each router must execute the following steps to compute its forwarding table:

1. Neighbor discovery.
2. Flooding of neighbor information to all routers in the network.
3. Building of a graph representation of the router configuration and edge values.
4. Calculating the shortest path from a router to all other routers using Dijstra's shortest path algorithm.
5. Deriving a forwarding table from the results of the shortest path algorithm.

If the neighbor information changes such as the cost of a path, steps 2 – 5 are repeated to determine the new forwarding table.

For testing and evaluation purposes, the number of routers in the network should be at least 5 and the graph derived from neighbor information should be connected (no router or cluster of routers are isolated from other routers), i.e. there is a path between any two routers in the network. For neighbor discovery, you may assume that neighbor information is already known. Each process emulating the router should have access to a text file to read the router's neighbor information, and the cost to travel to each neighbor. The format of the text file is described below. In this text file, only the router's immediate neighbors may be recorded, meaning every router in the network will access a different file to perform the "discovery" step.

Your program should accept the name of the text file as entry parameter in `main()` to "discover" its neighbors (neighbor information include a router label, IP address (or hostname) of the host executing the neighbor, port number, and cost to reach the neighbor). For flooding of neighbor information you must rely on your neighbors to forward you the information they receive from their neighbors and so forth. Be aware of cycles in the network that would cause information to be forwarded endlessly. Routers forward information about their neighbors in Link-State Packets (LSPs). At the minimum, each LSP for a router must contain a hop counter, a sequence number, the router's label, and connection information to other routers including cost information. The hop counter allows a router in the network to determine if the packet needs to be forwarded to another router. A hop counter greater than 0 means the router must decrement the counter by one and forward the LSP packet to its neighbors but if the hop counter reaches 0, the router must stop forwarding the LSP packet. The sequence number tells the router if a packet is newer than the one previously received.

Part of the design of the project is deciding the structure that is used to send information to the routers in LSPs and to represent the network graph. Each router should obtain information to represent a graph in a data structure that represents the topology of the network. Once each router has heard from all the other routers and a graph has been built, the Dijsktra algorithm (see book) must be executed for each router. From the results of the

algorithm, a forwarding table must be generated and printed out on the screen. The forwarding table must have only the information that a forwarding table on the network would have. Information such as distance to reach destination and the full path to each destination is not needed. To print out the forwarding table in an easy to read format, use the labels of the routers instead of hostnames or IP addresses. Each router accepts its own label from a command line argument. Labels associated with immediate neighbors are extracted from the "discovery" text file, and labels of all other routers are extracted from LSPs that the router receives.

Your router emulation may assume that each router knows the total number of routers in the network. This means that if your router has received LSPs from all other routers in the network, the router may begin building a graph, and executing the Dijkstra algorithm, to generate a new forwarding table. The total number of routers is passed in to the router process on the command line upon startup.

DYNAMIC ROUTING

Up to this point, the network has been static. The next step is to handle a dynamic network. Each router needs to be notified if information about the cost to travel to its neighbor has changed. Set up a random event that causes a router to change the cost on a path and to send out information about the change to its neighbors. Use an input parameter to `main()` to decide whether a router will randomly change the cost or not. The change of information should be disseminated with flooding. You can develop your own method on how to represent the data and its dynamic behavior. You will also need to develop a technique to determine when it is time to run Dijkstra's shortest path and generate a new forwarding table.

TEXT FILE FORMAT

The neighbor discovery file must be formatted such that each neighbor of a router is described by a one word label, an IP address (or hostname), a port number, and a cost value as comma separate values on a single line. All neighbors of a router are listed line-by-line. The following describes the format where `<routerLabel>` is the label of the router, `<IP_address/hostname>` is the IP address or host name of the router, `<portNumber>` is the port number of the router, `<cost>` is the cost of the link.

```
<routerLabel>,<IP_address/hostname>,<portNumber>,<cost>
<routerLabel>,<IP_address/hostname>,<portNumber>,<cost>
<routerLabel>,<IP_address/hostname>,<portNumber>,<cost>
…
```

For example, a router A may have the following links indicating that the router has the neighbors B and C with respective costs of 8 and 2:

```
B,cs-ssh1.cs.uwf.edu,60005,8
C,cs-ssh2.cs.uwf.edu,60008,2
```

PROGRAM STARTUP PARAMETERS

The program implementing a network router is called `node` and executes on a single host. The program itself must be started with the following parameters:

```
node routerLabel portNum totalNumRouters discoverFile [-dynamic]
```

Here, *`routerLabel`* is a one word label describing the router itself, *`portNum`* is the port number on which the router receives and sends data, *`totalNumRouters`* is an integer for the total number of routers in the network, *`discoveryFile`* refers to the text file that contains neighbor information including the router's own label, and

the optional parameter `–dynamic` indicates if the router runs a dynamic network whose cost on a link may change.

IMPLEMENTATION SUGGESTIONS

For your implementation consider the following system calls:

socket()	**gethostbyname()**	**gethostname()**	**bind()**
sendto()	**recvfrom()**	**close()**	**select ()**
fopen()	**fclose()**	**fscanf()**	

DELIVERABLES & EVALUATION

Your project submission should follow the instructions below. Any submissions that do not follow the stated requirements will not be graded.

1. Follow the submission requirements of your instructor as published on *eLearning* under the Content area.
2. You should have at a minimum the following files for this assignment:
    a. source code for `node`,
    b. a single Makefile to compile the programs,
    c. a program report,
    d. a README file (optional if project wasn't completed)

The program report must describe your protocol for flooding and for deciding when to run Dijkstra's algorithm. In addition, the report must include the forwarding tables and the graph of a given network involving 5 routers. You must use the provided test network uploaded to *eLearning* to collect results for the report. The README file should only be included if you submit a partial solution. In that case, the README file must describe the work you did complete.

Your program will be evaluated according to the steps shown below. Notice that I will not fix your syntax errors. However, they will make grading quick!

1. Program compilation with Makefile. The options *–g* and *–Wall* must be enabled in the Makefile. See the sample Makefile that I uploaded in *eLearning*.
    - If errors occur during compilation, the instructor will not fix your code to get it to compile. The project will be given zero points.
    - If warnings occur during compilation, there will be a deduction. The instructor will test your code.
2. Program documentation and code structure.
    - The source code must be properly documented and the code must be structured to enhance readability of the code.
    - Each source code file must include a header that describes the purpose of the source code file, the name of the programmer(s), the date when the code was written, and the course for which the code was developed.
3. Program evaluation using several test runs with input of the grader's own choosing. At a minimum, the test runs address the following questions.
    - Will the *node* read the neighbor information file to "discover" neighboring nodes in the network?
    - Will the *node* flood the network with LSP packages for a limited time?

- Will the *node* build a graph from the received LSPs and run the Dijkstra algorithm to compute the shortest path to all other nodes in the network?
- Will the *node* generate and print a forwarding table produced from the shortest paths of the Dijkstra algorithm?
- Will the *node* dynamically update the network cost and force the network to rerun its protocol and Dijkstra algorithm to update the forwarding table?

Keep in mind that documentation of source code is an essential part of computer programming. In fact the better your code is document the better it can be maintained and reused. If you do not include comments in your source code, points will be deducted. I also require you to refactor your code to make it more manageable and to avoid memory leaks. Points will be deducted if you don't refactor your code or if we encounter memory leaks in your program during testing.

DUE DATE

The project is due as indicated by the Dropbox for project 3 in *eLearning*. Upload your complete solution to the dropbox and the shared drive (if available). I will not accept submissions emailed to me or the grader. Upload ahead of time, as last minute uploads may fail

TESTING

Your solution needs to compile and run on the CS department's SSH servers. I will upload your solution to the SSH servers and compile and test your solution running several undisclosed test cases. Therefore, to receive full credit for your work it is highly recommended that you test & evaluate your solution on the servers to make sure that I will be able to run and test your program(s) successfully. You may use any of the SSH servers available to you for programming, testing, and evaluation. For security reasons, the majority of ports on the servers have been blocked from communication. Ports that are open for communication between the public servers range in values from 60,000 to 60,099.

GRADING

This project is worth 100 points total. The rubric used for grading is included below. Keep in mind that there will be deductions if your code does not compile, has memory leaks, or is otherwise, poorly documented or organized. The points will be given based on the following criteria:

Submission	Perfect	Deficient		
eLearning	5 points individual files have been uploaded	0 points files are missing		
shared drive (if available)	5 points individual files have been uploaded	0 points files are missing		
Compilation	**Perfect**	**Good**	**Attempted**	**Deficient**
Makefile	5 points make file works; includes clean rule	3 points missing clean rule	2 points missing rules; does not compile project	0 points make file is missing

compilation	10 points no errors, no warnings	7 points some warnings	3 points many warnings	0 points errors
Documentation & Program Structure	**Perfect**	**Good**	**Attempted**	**Deficient**
documentation & program structure	5 points follows documentation and code structure guidelines	3 points follows mostly documentation and code structure guidelines; minor deviations	2 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor; review sample code and guidelines
Static Routing	**Perfect**	**Good**	**Attempted**	**Deficient**
Routers read correctly their configuration file to obtain their neighbor information.	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
Routers flood network to obtain router information.	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile
Routers run Dijkstra algorithm to compute spanning tree.	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile
Routers print forwarding table.	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
Dynamic Routing	**Perfect**	**Good**	**Attempted**	**Deficient**
Routers choose randomly an edge of a connected router and change the associated cost.	5 points correct, completed	4 points minor errors	1 points incomplete	0 points missing or does not compile
Routers transmit changes in network and all routers recompute forwarding table.	5 points correct, completed	4 points minor errors	1 points incomplete	0 points missing or does not compile
Reporting	**Perfect**	**Good**	**Attempted**	**Deficient**
Discusses protocols and includes results from test run.	10 points completed	7 points minor details missing	3 points mostly incomplete	0 points missing

Not shown above are deductions for memory leaks and run-time issues. Up to 15 points will be deducted if your program has memory leaks or run-time issues such as crashes or endless loops when tested on the SSH server.