CS/ECE 438: Communication Networks — Fall 2007
Machine Problem 2 — Due: 9:00 PM, Friday, November 9th

Implementation of Unicast Routing Protocols

Please read all sections of this document before you begin coding.

Motivation

This machine problem moves forward from point-to-point communication issues to implementation of a distributed algorithm to control an internetwork. In particular, you will implement a link state, OSPF-like protocol and a distance-vector, RIP-like protocol. The purpose of the project is to give you some working experience with threads and socket programming and familiarize you with unicast routing.

Guidelines

For this MP, you are encouraged to work in teams of two, although you may work alone if desired (using the same grading scale, and thus doing roughly twice as much work). We recommend that you collaborate on the infrastructure and that each partner choose one of the protocols to implement. Please find a partner as soon as possible (within a week). We will allocate a few minutes of the lecture one week after the distribution date for this MP to pair off people without partners. If you attend the lecture and make your lack of a partner known to us, we will pair you with another person or will make suitable accommodations. If you do not attend the lecture or fail to make your lack of partner known to us, you may have to work alone.

Collaboration, discussion, code sharing, and any other form of group work with persons other than your MP2 partner is forbidden. Please indent and document your code. Use meaningful names for variables and follow other style guidelines that enhance the clarity of your code. Follow the guidelines in the **Hand In** section below when turning this assignment.

Project Description

You will simulate a network topology where nodes are made out of threads and you will use socket IPC (inter process communication) to simulate links. For this project, you must use pthreads, not Unix processes. Your program will read a topology from a file, distribute the topology information to all the nodes which in turn will establish connections (links) with each other according to the topology they received. The nodes will then run a link state unicast routing protocol to create unicast routes and use these routes to send data packets. Specific nodes will then be informed of changes in link costs or of link breaks. The nodes will then re-run the routing algorithm to reestablish the routes. The process will similarly be done for a distance vector routing protocol. You should demonstrate the correct operation of the unicast protocols by dumping the routing tables at each node and producing trace files that show the correct routing of messages.

This project has three parts:

1. You will first write a simple program that takes in a description of network connectivity (a connectivity table) and spawns several threads. Each of these threads corresponds to a simulated router in the network.

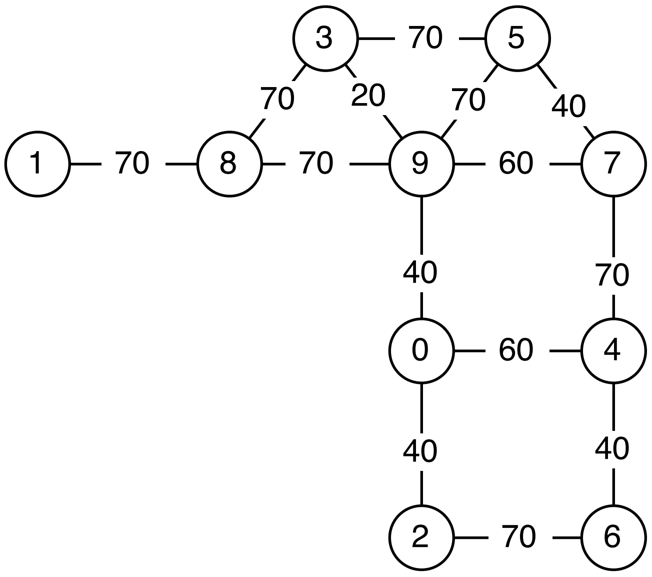
2. Once each simulated router has figured out its connectivity table, it starts executing the routing algorithm. Then, some sources will send data.
3. Next, some nodes will be informed of changes in link status, forcing a recomputation of the routes. Following this, some sources will send data along the potentially new route.

The output of your program will have two parts:

1. The unicast routing table of each node in the network for each protocol
2. A trace of some data packets

Part 1: Configuring the network

To configure the network, you will implement a program called the manager, which takes a routing algorithm and three file names as arguments. The first file contains a network topology description. (The second and third files are described below.) The format of the first file contains an integer followed by a table. The first line of this file contains a single integer N. This means that there are N nodes in the network, whose addresses are 0, 1, 2, ..., N-1. This line is followed by several lines, one for each point-to-point link in the network. Each line has 3 integers separated by white space: X, Y, C. X and Y are node numbers between 0 and N-1, and C is a positive integer that represents the cost of a link between X and Y. Here is a sample network and topology description:



```
10
0 9 40
0 4 60
0 2 40
1 8 70
2 6 70
3 9 20
3 8 70
3 5 70
4 7 70
4 6 40
5 9 70
5 7 40
7 9 60
8 9 70
```

After the manager reads the network topology description, the following things must happen: The manager must spawn one thread for each node in the network. In the subsequent description, we use the term router to denote this thread. To implement the protocol, each router must listen on a datagram (UDP) socket. Before the protocol message exchange can happen, the router must be told several pieces of information:

- It's own address (a number between 0..N-1)
- Its connectivity table (who its neighbors are, what are the link costs to each neighbor, and the UDP port number for each neighbor).

You must implement a simple protocol between the manager and each router so that the manager can tell the router this information. You should design the message formats for exchanging this information. One way of doing this is:

- After each router starts up, it opens a TCP connection to the manager (figure out how it does this).
- Then, the router sends a message to the manager telling it what its own UDP port is.
- The router then sends a request to the manager to be assigned a node address, and to be given a connectivity table.
- At this point, each router starts exchanging routing messages, as described below.

Part 2: Implementing the Routing Algorithms

The goal of the second part is to implement the routing algorithms discussed in class. Once this part is implemented, you should be able to have each router process (after it has received its connectivity table) send routing updates as specified by each protocol.

To implement this functionality, you are expected to design your own data structures, and to design your own packet formats. Note that your packet formats should be very simple. You may assume that node failures do not happen and that link failures only occur as indicated by the manager. In particular, this means you do not need to implement procedures that simulate these failures. You should carefully read and understand the protocol processing rules. It is useful to work out the algorithm on different simple topologies. You need not implement periodic update routing messages. Rather, each router, after it receives its connectivity table from the manager or an update from the manager, sends out a routing update. Whenever it receives a routing update from a neighbor that causes a change in its own table, the router sends an update. You should figure out a way to determine when the routing exchange has converged. This can be as simple as waiting for several seconds, or up to a minute or by coordinating with the manager.

Once the routing tables have converged, a source will send some data to a destination. The configuration file (the second argument to the manager program) for this part will contain two items: (a) the router id of the source, and (b) the router id of the destination. An example input might be the following:

```
1 3
4 0
```

The manager needs to tell this information to the related nodes through the existing TCP connections. There should be enough time gap between each round of sends so that their output does not interfere with each other.

The final file indicates changes in the status of specific links and should look similar to the first file, with each line having 3 integers separated by white space: X, Y, C. X and Y are again node numbers between 0 and N-1, and C is a positive integer that represents the new cost of a link between X and Y, or C could be -1 to represent an infinite link cost (i.e., a broken link).

The manager tells this information to both nodes X and Y, which then propagate any necessary routing updates to their neighbors.

The manager should alternate between sending a message between a source and a destination and then informing nodes of link status updates. I.e. the manager should send the message described in the first line of the messages file, then process the first update in the updates file, wait for the tables to converge and then send a second message. If one file has more lines than the other, execute the remaining lines in sequence. We recommend that you first implement and test your protocols without the updates and then add them in later.

After both files are read, you should ensure that all your processes terminate gracefully (i.e. the TA does not have to use the kill(1) program to terminate your processes).

Design

You should structure your makefile so that the result of the make command is a single executable, called manager with the following syntax:

```
manager ls topo.dat messages.dat updates.dat > out.tr
```

This program takes these parameters:

- "ls" or "dv" specifies which algorithm to use
- "topo.dat" is a file containing a network topology description,
- "messages.dat" is a file containing the source-destination pairs,
- "updates.dat" is a file containing the link status updates.

We will not tell you in advance what topology description, sources and updates we intend to use. You may, however, assume that all input files will be syntactically correct and will contain only the required data (i.e., no unrelated characters or comments).

To aid us in evaluating your program, your software should output the following files containing the information specified below. Be sure to adhere to the format suggested below, otherwise you may not get credit for your submission:

- There should be one file named "ports". This file should have exactly as many lines as there are nodes in the network. Each line has two positive integers X and Y, separated by a single space character. X is the address of a router, and Y is the UDP port number assigned to the node. The lines should be sorted in increasing order of router address (that is, the first line should be for router 0, the second for router 1 and so on).
- For each router and each protocol, there should be one file for each convergence instance of the routing table. The name of this file should be the address of the corresponding router followed by ".ls" for link state and by ".dv" for distance vector. Thus, your output should have 2N files named 0, 1, 2,..,N-1 for each convergence instance. These files should contain the routing table of the corresponding router for the particular protocol.
  - The format of each file for link state routing is a sequence of lines. Each line depicts a routing table entry for a single router, say A. The format of a line is:

    X Y C : P1,P2,P3 . . .

    where X is the address of the destination router, Y is the address of the next hop from A towards X, C is the cost to reach X from A, and the remainder of the line describes the simple path from A to X (including A and X).

  – The format of each file for distance vector routing is a sequence of lines. Each line depicts a routing table entry for a single router, say A. The format of a line is:

    X Y C

    where X is the address of the destination router, Y is the address of the next hop from A towards X, C is the cost to reach X from A.

- A file that contain records of traces of each data packet forwarded through the router. The records will consist of a sequence of node ids (integers), each on a separate line. Records are delimited by a line that begins with a "#" character, perhaps followed by some comments. Make sure you call fflush() after each print statement to ensure data is written immediately. The file should be produced by the routers printing to stdout as shown in the program invocation above. (Note that you can print error or debugging messages to stderr; those will appear on your screen even if you redirect the stdout into the trace file.) The format of the trace files should be as follows:

```
# Trace 1
source id
unicast router id
unicast router id
...
destination id

# Trace 2
source id
....
```

  Note that the sequence of "router id" entries may change each time you run the program. This is normal.

File Layout

Your project must have the following:

- *Header file*: This file contains the declarations of the data structures, all the #include's and #define's. This header file alone is then included in the other C files. Note that by convention all header files have a "".h" suffix.

- *C files*: The whole project should be broken up into at least three C files, one for each part (as described above). If you have a good file hierarchy in mind you can break up into more files but that break up should be logical and not just spreading functions into many files.

- *Makefile*: Now that you have multiple files, you will need to create a makefile so that you can do all the compilations by just typing "make". Your makefile should also take care of cleaning up the directory, i.e. the command "make clean" should remove all the old *.o files and all the binary files. Read the make(1) man page for more information. Your Makefile will produce an executable called `manager`. If this executable is not produced, we will not test your program and you will receive **no correctness marks**.

- *README*: This file should contain a short description of the layout of your project, i.e., a brief description of each source file including its purpose. This should also include any notes to the TA as to how to run your program. Also, please specify who your partner is, or say that you are not using a partner.
- *Design Document*: This part should briefly describe and justify your design and implementation decisions, including any data structures used to support frame formats. Discuss any other interesting aspects of your implementation. As always, you should acknowledge the sources if you copied sections of your code from outside courses (even if you modified it later). And remember to write your name in the beginning of this file.

Hand In

Place your source code in your MP2 directory along with a Makefile that, on execution of the command make, causes the executable code for the program manager to be generated by the compiler.

When you get the C programs, README, and Makefile files within the directory MP3 in good working order, you are to transfer the directory electronically as follows. First, remove all object files and executables from the directory, leaving only the source code, the Makefile, and the README file. **Submission of object files or executables will reduce your grade**. Next, from within the directory, type

```
~ece438/Handing/handin
```

You may handin multiple times; each subsequent submission will overwrite previous ones.

MP 2 Grading Scheme

Total of 100 points possible.

Initial Setup (10 pts)

- Address and port for each router (5 pts)
  - Each router gets their address from the manager.
- Connectivity table for each router (5 pts)
  - Each router will get their connectivity table from the manager.

Implementation of LSR (30 pts)

- Routing table, without implementation of dynamic update (10 pts)
- Packet routing trace, without dynamic update (10 pts)
- Routing table, with dynamic update (5 pts)
- Packet routing trace, with dynamic update (5 pts)

Implementation of DVR (30 pts)

- Routing table, without implementation of dynamic update (10 pts)
- Packet routing trace, without dynamic update (10 pts)
- Routing table, with dynamic update (5 pts)
- Packet routing trace, with dynamic update (5 pts)

Design Document (15 pts)

General Behaviour (15 pts)

- All router threads exit and clean up gracefully
- Code should be well commented

Deductions

Points will be deducted for the following:
- Makefile does not create all the necessary executables (-10 pts)
- No partner described in README file (-10 pts)
- Handing executable and/or object files (-10 pts)
- Poor code quality (-5 pts)

No Credit

No credit will be received:
- if UNIX processes are used instead of posix threads
- if any attempt to subvert/avoid the problem is used
- if there are compilation or link errors
- for failure to cite sources, i.e., plagiarism