CISS 410: MP3

Due before 11:59 p.m. on December 10, 2015

For this assignment, you will create a simulator for a link state routing protocol, something like OSPF as described in your book in Section 3.3.3. Your simulator will create a process for each router in the simlulated network, and those processes will communicate with each other via UDP sockets.

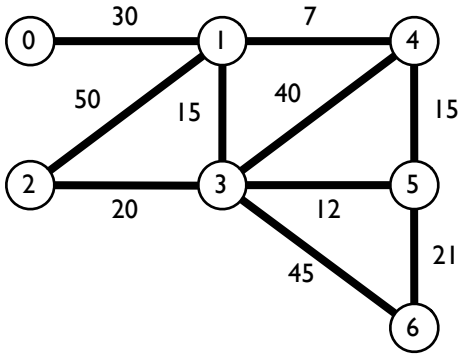
Part One: Creating the Network Topology

In this part, you will simulate the network topology by creating processes representing routers, then sending those processes information about the network.

Create a program called `ls-sim` which takes a single command-line argument:

```
$ ./ls-sim network
```

The `network` argument is the name of a file describing a network topology. An example of a network topology file and the network it describes are below:




```
7
0 1 30
1 2 50
1 3 15
1 4  7
2 3 20
3 4 40
3 5 12
3 6 45
4 5 15
5 6 21
```

The first line of a network topology file is an integer N, which means there are N nodes in the network, whose addresses are $0, 1, \ldots, N-1$. This is followed by a number of lines, one for each point-to-point link in the network. Each line has three numbers X, Y, and C, seperated by whitespace. X and Y are the routers connected connected by the link, and C is a positive integer that represents the cost of the link between X and Y.

I've provided an example network topology file in the "Part One" directory. I will test your progam with it *and other files*, so you should make up a few networks of your own

to test your program. After your program reads the topology file, it should take the following actions:

1. The program should create an additional process for each router. You can do this by calling `fork` in a loop. Each of these new processes represents a single router. For the rest of the document, I will refer to the original process as the *manager* and the new processes as *routers*.
2. Each router, immediately after being started, should `bind` a UDP port that it will use to listen for messages from other routers.
3. Each router should open a TCP connection to the manager. All communication between the manager and routers always takes place over their TCP connection.
4. Each router should send a CONFIG message to the manager with its UDP port number.
5. The manager should send a SETUP message with the following information to each router:
   (a) the router's address (a number between 0 and $N - 1$),
   (b) a connectivity table, telling the router who its neighbors are, what are the link costs to each neighbor, and the UDP port number for each neighbor.
6. Each router should set up any neccesary data structures to store its connectivity table.
7. Each router should create or overwrite a file named "N.links," where N is the address of the router. The file should have one line for each of the router's connections to a neighbor. Each line should have the format $A\ C$, where A is the address of the neighbor router, and C is the cost of the link. See the "Part One" folder for examples.
8. Each router should send a READY message back to manager to indicate that it is ready to begin simulation, then wait for a BEGIN message from the manager.
9. After the manager has received READY messages from all routers, it should send a BEGIN message to each router, telling it to begin simulation. For Part One of MP3, nothing happens at this point.
10. **Temporary:** After sending the BEGIN message to all routers, wait for a second, then close all TCP connections to the routers and exit. This behavior will change in Part Two.
11. Each router should `recv` from its manager connection. When `recv` returns and the return value is 0, the router knows that the simulation is over and exits.

Implementing these actions means there are three major design decisions you need to make:

- how to create the router processes and establish communication from the routers to the manager,

- what data structures you need in the manager (e.g., to track the sockets used to communicate with the routers) and in the routers (e.g., to store the connectivity table), and
- a simple communication protocol used by the manager and the routers to communicate. It's up to you to decide the format of the CONFIG, SETUP, READY, and BEGIN messages, but it should be *very* simple.

The links files generated by the routers will demonstrate that your manager is correctly reading the network description file, then sending the correct information to the routers. It's also a good idea to temporarily print information about the routers' UDP ports, so that you verify they've been set up correctly.

Tips

Beej often uses `AF_UNSPEC` to allow sockets to be either IPv4 or IPv6. For our projects, it's a good idea to specify that you are using IPv4 (using `AF_INET`)

How do the routers know which port to use? You could try binding random ports until one succeeds, but there is a better way: if you ask to bind port 0, the system will bind a random, unused port instead. Once you've done that, you can find the port number by using the `getsockname` function. Here's a bit of sample code that gets the port number from a socket:

```
int get_port_number(int sockfd)
{
    struct sockaddr_in ss;
    socklen_t len = sizeof ss;
    if (getsockname(sockfd, (struct sockaddr*)&ss, &len) == -1)
        return -1;

    return ntohs(ss.sin_port);
}
```

Part Two: The Link State Routing Protocol

In this part, you will simulate the link state routing protocol by having the routers send UDP datagrams to one another. The datagrams are the link state packets (LSP) of the link state routing protocol.

After you implement Part Two, your program should perform the following actions:

1. After the routers receive the BEGIN message from the manager, they send LSP messages to their neighbors.
2. Any router receiving an LSP message from its neighbor will pass it on—or not—according to the *reliable flooding* process described in the text on p. 254.

3. Each router uses the LSP messages it receives to compute its routing table using the *forward search* algorithm on p. 257.
4. After the routing algorithm has *converged*, meaning there are no more LSP messages being sent and each router has fully computed its routing table, the manager sends a PRINT message to each router.
5. When a router receives the PRINT message, it should create or overwrite a file named "N.rtable," where N is the address of the router. It should write its routing table to this file. The file should have N lines, each line having the format $D\ N\ C$, where D is a destination router, N is the address of the next hop towards D, and C is the cost to reach D. See the "Part Two" directory for examples.
6. After the router has finished writing its routing table to the file, it should send a DONE message to manager.
7. After the manager receives DONE messages from all routers, it should close all TCP connections to the routers and exit.
8. As before, when the router sees the connection to the manager is closed, it should exit.

There are several details of the link state routing protocol as described in the textbook that you **do not** need to implement:

- You do not need to implement acknowledgment or retransmission of LSP messages between routers.
- You do not need to send periodic update messages.
- You do not need to have a TTL value in each LSP.

For this part, the major design decisions are:

- the data structures needed in the routers to support reliable flooding and forward search,
- the formats for the message types described above, and
- how the manager determines that the routing algorithm has converged. You decide how this determination is made; you could simply wait a while, or you could have the routers coordinate with the manager.

Part Three: Sending Data Packets

In this part, you will simulate data packets being passed between routers in the network.

Modify `ls-sim` so that it can take one or two command-line arguments:

- If it receives one argument, that is the network topology file, and it should behave as described in Part Two.

- If `ls-sim` receives the second argument, that is a data packet description file, which the manager should open and handle as described below.

The data packet description file is very simple: it has one or more lines, and each line describes a packet that should be forwarded through the network. Each line has two numbers between 0 and $N - 1$. The first number is the source router of the data packet, and the second number is the destination router for the data packet.

After the manager has received the DONE message from each router as decribed above, it should begin simulating the data packets. For each packet,

1. The manager should print "# Trace P", where P is the number of the packet, starting with 1. Then the manager should wait a short time (no more than one second) before the next step.
2. The manager should send a SOURCE message to the source router specified for the packet. The message should contain the destination address of the packet.
3. The router should print its address to `stdout`.
4. The router should consult its routing table and send a PACKET message to the correct router, containing the destination address.
5. Any router other than the destination receiving a PACKET message should print its address, and forward the packet to the next router.
6. When the destination router receives the PACKET message, it should print its address, then send a DESTINATION message to the manager.
7. The manager prints an empty line, then repeats the process for the next packet.
8. After receiving the DESTINATION message for the final packet, the manager closes all TCP connections to the routers and exits.

The only design decision here is the formatting of the new messages—you shouldn't need any significant additional data structures for this part.

Tips

Use the `fflush` function after every print operation described above to make sure that they occur immediately. Otherwise, the print can be delayed by the operating systems, and you may not see the output in same order you created it!

Additional Specifications

Files

Your project should have the following files:

- C source files: `main.c`, `manager.c`, and `router.c`, with functions implemented in the appropriate file. You may add additional C files if useful (for example, if you have some code shared between the manager and routers), but you shouldn't have more than one or two.
- C header file: use a single header file, `simulator.h`, which you include in all of your C source files. All `#include` and `#define` directives should appear in this file (and not in the C source files). All function prototypes and struct definitions should also appear in this file.
- Makefile: this should build a single executable, `ls-sim`. It should also have a `clean` target that removes the executable and any `*.o` files.
- README: this file should have your name, the date, and a brief description of the project. If you have any messages for me, put them here. Also note it here if you did not complete all three parts of the project.
- Design Document: this file, `design.txt`, should contain descriptions of how you approached each of the design decisions described above: message formats, data structures, etc. Briefly describe your implementation and why you chose it.

  This document should be nicely formatted. **Do not** wait until the last minute to work on it. Imagine I'm looking over your shoulder and asking, "why did you do it that way?" when you describe each design decision.

Grading

5	Makefile
3	builds `ls-sim`
2	"clean" target
35	Part One
10	create a process for each router
10	routers connect to the manager
5	routers get address and port info from the manager
5	routers get connectivity info from the manager
5	routers save their neighbor info
30	Part Two
10	implement reliable flooding
20	generates correct routing tables
15	Part Three
10	implement packet forwarding
5	generate correct packet trace
5	README
5	has all required information
10	Design Document
5	describes message formats
5	describes data structures

Points will be deducted if:

- Your code is poorly written, so use descriptive variable names, write good comments, indent consistantly, etc.
- Your program doesn't work consistantly, sometimes crashing or not not exiting properly.

You get no points if:

- Your code doesn't compile.
- You "work around" the assignment; e.g. setting up the router data structures before calling `fork`.

If you fail to cite sources, you are subject to all the penalties described in the "Academic Honesty" section of the course syllabus.