Machine Problem 2

Unicast Routing

Due: Thursday, April 9th, 11:59pm CST

In this assignment, you will implement traditional shortest-path routing with the link state (LS) and/or the distance/path vector (DV/PV) protocols. Your program will be run on every node in an unknown network topology to perform routing tasks to deliver the test messages instructed by our manager program. The implementation you need to submit depends on whether you work alone or as a group of two:

- If working alone:
  You can choose one of LS, DV, or PV to implement.

- If working as a group of 2:
  Your group must do both LS and DV/PV (choose one of DV or PV). The administrative, non-algorithm-logic components of LS and DV/PV are identical, so a pair can do those together. For the algorithms themselves, we recommend that you each take one.

Besides the implementation of the routing protocols, you will have to deal with the unfortunate distinction between "network order" and "host order", AKA endianness. The logical way to interpret 32 bits as an integer is to call the left-most bit the highest, and the right-most the ones place. This is the standard that programs sending raw binary integers over the internet are expected to follow. Due to historical quirks, however, x86 processors store integers reversed at the byte level. Look up `htons`, `htonl`, `ntohs`, `ntohl`() for information on how to deal with it.

1. Environment Setup

1.1 Your Group GitLab Repository

You will use `git` to maintain your code submission. In this machine problem, each group (including solo groups) is given a new repository:

```
https://gitlab.engr.illinois.edu/cs438-sp2020/mp2-mp3/group<group_num>
```

Go to your MP0/MP1 repository and check the file "mp2_mp3_group_members.txt" for your group number.
Use your UIUC login credentials to access this repository. Both you and your teammate (if a group of two) have access to this repository. We will grade your programs that are pushed to this repository. Feel free to use this repository to maintain your code during development. **For this machine problem, you must maintain your code in the *mp2* folder.**

1.2 The Auto-grader Ubuntu Machine

Here, we describe the environment the auto-grader uses to compile and test your code.

We updated the autograder to use Ubuntu 18.04.3 LTS Desktop, so you must make sure that your submitted code can be compiled (by `make`) and run on that OS version in a virtual machine. (You can search how to upgrade or use the built-in Software Updater application if your VM is using v18.04.1.) You are recommended to set up a virtual machine with the same OS version. Use `apt install` command to install necessary development tools such as `make`, `gcc`, `g++`, `clang` to ensure that you use the latest compiler versions (that the auto-grader uses). The auto-grader runs every night and hence giving you plentiful chances to discover and resolve any potential environment inconsistency issues you may have.

2. Test Environment

2.1 Network Configurations

In this MP, **everything will be contained within a single VM**. The test networks (i.e., nodes and network topologies) are constructed by using virtual network interfaces (created by `ifconfig`) and configuring firewall rules (configured by `iptables`). Your program will act as a node in the constructed network to perform routing tasks. That is, if there are *N* nodes in the test network, there will be *N* instances of your programs being started, each of which runs as one of the nodes with an ID that is associated to an IP address (see below for details).

We are providing you with the same Perl script (`make_topology.pl`) that the auto-grader uses to establish these virtual network interfaces and firewall rules. The manager program (`manager_send`) used by the auto-grader to send the test messages is also provided (see next Section). They are available at:

https://courses.engr.illinois.edu/cs438/sp2020/mp/mp2-supplement.zip

What follows summarizes the network configurations used in this MP:

- The topology's connectivity is defined in a file that gets read by the provided Perl script (`make_topology.pl`) to create the network environment.
- The initial link costs between nodes are defined in files that tell nodes (your programs) what their initial link costs to other nodes are (if they are in fact neighbors) when the nodes are being run. See Section 3 for details.
- For each node, a virtual network interface (eth0:1, eth0:2, etc) will be created and given an IP address (`10.1.1.X`).
- A node with ID `X` gets the IP address `10.1.1.X` where $0 \leq$ `X` ≤ 255.
- Your program will be given its ID on the command line, and when binding a socket to receive UDP packets, it should specify the corresponding IP address (rather than INADDR_ANY / NULL).
- To construct a test network topology, `iptables` rules will be applied to restrict which of these addresses can talk to which others. For instance, `10.1.1.30` and `10.1.1.0` can talk to each other if and only if they are neighbors in the test network topology.
- A node's only way to determine who its neighbors are is to see who it can directly communicate with. **Nodes do not get to see the aforementioned connectivity file.**
- **The links between nodes can change during the test.** This is done by setting new `iptables` rules during the test.
- **The link costs between nodes can change during the test.** This is done by letting the Manager send a cost update packet to your node (see next Section).

2.2 Our Manager

While running, your nodes will receive instructions and updated information from a manager program. You are not responsible for submitting an implementation of this manager. As mentioned above, the manager program (`manager_send`) used by the auto-grader is provided to you. Your interaction with the manager is very simple:

The manager sends messages in individual UDP packets to your nodes on UDP port 7777.
Your nodes execute the instructions upon receiving the manager's packets.
Your nodes do not need to reply to the manager in any way.

The manager's packets have two types of instructions: (1) sending a message data packet and (2) updating neighbor link cost.

Their packet formats are explained next.

(1) Sending A Message Data Packet

`4 bytes`	`2 bytes`	`≤ 100 bytes`
"`send`" (in ASCII code)	`destID` (signed 16-bit int, network order)	`msg` (some ASCII text)

This message instructs the recipient to send a data packet, containing the given message data `msg`, to the node with the ID `destID`. The message data is guaranteed to be an ASCII string. **It does not come with a null-terminator in the packet** (it would be wasteful if it does).

Note that this is the packet format you'll receive from the Manager. To deliver the given message to the destination node, you don't have to follow the same packet format. Your packet can have whatever format that makes the most sense to you. **However, you must follow the requirements stated in Section 3.3 to produce log files to pass test cases.**

(2) Updating Neighbor Link Cost

`4 bytes`	`2 bytes`	`4 bytes`
"`cost`" (in ASCII code)	`neighborID` (signed 16-bit int, net order)	`newCost` (signed 32-bit int, net. order)

This message tells the recipient that its link to node `neighborID` is now considered to have cost `newCost`. This message is valid even if the link in question did not previously exist, although in that case do NOT assume the link now exists: rather, the next time you see that link online, this will be its new cost.

3. The Assignment – Your Nodes

3.1 The Basics

Whether you are writing an LS or DV/PV node, your node's command line interface to the assignment environment is the same. Your node should run like:

```
./ls_router <nodeid> <initialcostsfile> <logfile>
./vec_router <nodeid> <initialcostsfile> <logfile>
```

For examples:

```
./ls_router 5 node5costs logout5.txt
./vec_router 0 costs0.txt test3log0
```

The 1st parameter indicates that this node should use the IP address `10.1.1.<nodeid>`.
The 2nd parameter points to a file containing initial link costs related to this node (see Section 3.2).
The 3rd parameter points to a file where your program should write the required log messages to (see Section 3.3).

3.2 Initial Link Costs File Format

The format of the initial link costs file is defined as follows:

```
<nodeid> <nodecost>
<nodeid> <nodecost>
...
```

An example of the initial link costs file:

```
5 23453245
2 1
3 23
19 1919
200 23555
```

In this example, if this file was given to node `6 (10.1.1.6)`, then the link between nodes `5 (10.1.1.5)` and `6` has cost `23453245` – *so long as that link is up in the physical topology*.

If you don't find an entry for a node, default to cost 1. We will only use positive costs – never 0 or negative. We will not try to break your program with malformed inputs. A link's cost will always be < 2^{23}.

The link costs that your node receives from the input file and the manager don't tell your node whether those links *actually currently exist*, just what they would cost if they did. In other words, just because this file contains a line for node `X`, it does **NOT** imply that your node will be neighbors with node `X`.

Your node therefore needs to constantly monitor which other nodes it is able to send/receive UDP packets directly to/from. We have provided code (partial) that you can use for this (see `monitor_neighbors.c`).

3.3 Log Files

When originating, forwarding, or receiving a data packet, your node should log the event to its log file. The sender of a packet should also log when it learns that the packet was undeliverable. There are four types of log messages your program should be logging:

```
forward packet dest [nodeid] nexthop [nodeid] message [text text]
sending packet dest [nodeid] nexthop [nodeid] message [text text]
receive packet message [text text text]
unreachable dest [nodeid]
```

For example:

```
forward packet dest 56 nexthop 11 message Message1
receive packet message Message2!
sending packet dest 11 nexthop 11 message hello there!
unreachable dest 12
forward packet dest 23 nexthop 11 message Message4
forward packet dest 56 nexthop 11 message Message5
```

In this example, the node forwarded a message bound for node `56`, received a message for itself, originated packets for nodes `11` and `12` (but realized it couldn't reach `12`), then forwarded another two packets.

To output the log messages correctly, it is recommended to use `sprint()` to from the log string. Sample code that uses `sprint()` to output correct log messages is provided in the `extra_note.txt` file in `mp2-supplement.zip`.

Our tests will have data packets be sent relatively far apart, so don't worry about ordering.

3.4 Other Implementation Requirements

Your nodes should accomplish:

- Using LS or DV/PV, maintain a correct forwarding table.
- Forward any data packets that come along according to the forwarding table.
- React to changes in the topology (changes in cost and/or connectivity).
- Your nodes should converge within 5 seconds of the most recent change.

Partition:
The network might become partitioned, and your protocols should react correctly: when a node is asked to originate a packet towards a destination it does not know a path for, it should drop the packet, and rather than log a send event, log an unreachable event.

Tie breaking:
We would like everyone to have consistent output even on complex topologies, so we ask you to follow specific tie-breaking rules.

- DV/PV: when two equally good paths are available, your node should choose the one whose next-hop node ID is lower.
- LS: when choosing which node to move to the finished set next, if there is a tie, choose the lowest node ID. If a current-best-known path and newly found path are equal in cost, choose the path whose last node before the destination has the smaller ID. Example:

  > Source is 1, and the current-best-known path to 9 is 1→4→12→9.
  > We are currently adding node 10 to the finished set.
  > 1→2→66→34→5→10→9 costs the same as path 1→4→12→9.
  > We will switch to the new path, since 10<12.

4. What to Submit

You must submit (i.e., commit and push) your code to the private GitLab repository provided to you in Section 1.1. We will only grade your programs that are pushed to the `master` branch in that repository. In your repository, it must (at least) contain the following files:

- `/mp2/readme.txt` # writes down who did what in your group
- `/mp2/testls` # if you implement LS and would like to be graded
- `/mp2/testvec` # if you implement DV/PV and would like to be graded
- `/mp2/Makefile` # the makefile

If you work as a group of two, please write down each person's responsibility in this MP.

If you work alone and only implement either LS or DV/PV, remove either `testls` or `testvec` for the one you don't implement so the auto-grader won't be testing that part.

You must create the Makefile so that execution of the command

```
make
```

compiles and generates the executable files named `ls_router` and `vec_router`, in the same *mp2* folder (if working alone, only the chosen one is required). This executable `ls_router` and `vec_router` should be able to run with arguments as specified in Section 3.1.

Finally, running

```
make clean
```

should delete all executable files and any temporary files that the Makefile or your program creates.

You may commit and push your code as many times as you like before the deadline. It's in fact a good practice to commit your code whenever there is a change that is worth noted. We will use the last commit you made for the repository when we grade your MP.

Your program cannot be graded if it has not been pushed to the repository. Fail to commit and push your code on time will result in your MP being considered late. It is your responsibility to ensure all your work is committed and pushed to your repository by the due date.

5. Grading

The grading will be determined by 8 test cases, which are supposed to be progressively harder and stress different elements of your code. If both LS and DV/PV are implemented and submitted for grading, each of LS and DV/PV implementations will be tested separately and the grade in each test case will be divided by 2 (thus 50% for LS and 50% for DV/PV).

- 5%: Send a message to a non-neighbor node in a 3-node topology
- 10%: Send a message (by shortest path, of course) to a non-neighbor in a certain small (<20 node) topology.
- 10%: Send a message to a non-neighbor node in a certain large (>50 node) topology
- 20%: Switch to a better path when one becomes available in a certain large topology
- 20%: Correctly fall back to a worse path on a certain small topology
- 15%: Correctly report a failed send (unreachable) in a certain small topology that gets partitioned
- 10%: Get a packet through after a former partition is healed, large topology
- 10%: Converge within the time limit after a major, rapid series of changes to the network, >200 node topology. (The time limit starts when the final change is made).
- Late penalty: 2% of total possible score per hour

6. Important Notes

1) You must use C or C++.
2) Your code must be runnable in 64-bit Ubuntu 18.04.3 LTS Desktop machines. This is the environment in which the auto-grader runs. You program must fulfill the given assignment requirements to get said scores.
3) If you need to use a library for data structures, you MUST get the approval of the course staff. Additionally, you MUST acknowledge the source in a README in *mp2* folder. However, algorithms MUST be your own.
4) The auto-grader will test every group's current version every night, starting at 6PM. The TAs will try their best to make sure this is the case, but bugs happen. We will make an announcement of its starting date.
   - Do a git pull the next morning to see your score in *mp2/result.txt*.
   - The score you are given before the deadline is for your reference and has no effect on your final grade.

- The late penalty is applied to the last git commit you make for anything in the folder *mp2*. If you want to make any change in *mp2* that does not change your code, please wait until the third day after the deadline to make it.

5) Do not make your private repository public. You will be held partially responsible for any resultant plagiarism.
6) You must work alone or with your teammate (for MP2 and MP3). Your code must be your own in your group. You can discuss very general concepts with other groups, but if you find yourself looking at a screen/whiteboard full of pseudo code (let alone real code), you are going too far.
   - Refer to the class slides and official student handbook for academic integrity policy. In summary, the standard for guilt is “more probable than nor probable”, and penalties range from warning to recommending suspension/expulsion, based entirely on the instructor’s impression of the situation.
   - The Grainger College of Engineering has some guidelines for penalties that we think are reasonable, but we reserve the right to ignore them when appropriate.