Machine Problem 2

Unicast Routing

Due: Mar 30th, 2018 – 11:59 PM

In this assignment, you will implement traditional shortest-path routing, with the link state (LS) and/or distance/path vector (DV/PV) protocols. *You may work alone or with a partner.*

If working alone, you can choose just one of LS, DV, or PV to implement. If working in a pair, the pair 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 (clearly indicate in a readme file who did what). Submit in only one partner's SVN. Put exactly the non-submitting partner's netId (and nothing else) in a file called *partner.txt* in the *mp2* directory.

The test setup will be the same environment as in MP1, but now everything will be contained within a single VM. Your nodes will use different virtual network adapters, with *iptables* restricting communication among them. We are providing you with the same script that our testing environment uses to establish these restrictions.

This MP introduces 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.

Our Environment

Your nodes will all run on the same machine. There will be a made-up topology applied to them in the following manner:

- For each node, a virtual interface (eth0:1, eth0:2, etc) will be created and given an IP address.
- A node with ID **n** gets address 10.1.1.**n**. IDs 0-255 inclusive are valid.[1]
- 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).
- *iptables* rules will be applied to restrict which of these addresses can talk to which others. 10.1.1.30 and 10.1.1.0 can talk to each other *if and only if* they are neighbors in the made-up topology.
- The topology's connectivity is defined in a file that gets read by the script that creates the environment. The link costs are defined in files that tell nodes what their initial link costs to other nodes are, *if* they are in fact neighbors.
- 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.

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; we will test using our own. Your interaction with the manager is very simple: it sends messages in individual UDP packets to your nodes on UDP port 7777, and they do not reply in any way.

[1] Test your understanding! Why are 0 and 255 valid node IDs, when they correspond to 10.1.1.0 and 10.1.1.255? Is there a situation where it wouldn't be valid to use those as actual IP addresses that identify a host? How do you know when?

The manager's packets have one of two meanings: "send a data packet into the network", or "your direct link to node X now has cost N." Their formats are:

"**Send a data packet**":

4 bytes	2 bytes	≤ 100 bytes
command: "send"	destID (signed 16-bit int, network order)	The msg data

This message instructs the recipient to send a data packet, containing the msg data, to the node with ID destID. The data packet can have whatever format makes the most sense to you, but must take the cheapest path possible in the current topology.

Command: ASCII string "send".
destID: a 16-bit signed integer in network order.
Playload: message data

Everything after those first 7 bytes are the msg data, and should go in your data packet (remember, unlike TCP, UDP has well defined packet boundaries, so talking about "the rest of the data" gotten from a recvfrom() is meaningful). the msg data is guaranteed to be an ASCII string. It does not come with a null-terminator in the packet; it's not needed and would be wasteful.

"**New cost to your neighbor**":

4 bytes	2 bytes	4 bytes
ASCII string: "cost"	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. neighborID is a signed 16-bit integer, and newCost is a signed 32-bit integer. The first 4 bytes are literally the ASCII string "cost".

Your Nodes

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

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

Examples:

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

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. The format of the logging is described in the next section.

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. 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 that you can use for this.

That concludes the description of how your nodes need to do I/O, interact with the manager program, and stay aware of the topology. Now, for what they should actually 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 (see File Formats section).
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.

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.

File Formats

Your nodes take the "initial costs file" as input, and write their output to a log file. The locations of both of these files are specified on the command line, as described earlier. The initial costs files might be read-only.

Initial costs file format:

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

Example initial costs file:

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

In this example, if this file was given to node 6, then the link between nodes 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}. Once again, just because this file contains a line for node n, it does ***NOT*** imply that your node will be neighbors with **n**.

Log file: (See our provided code for sprintf()s that generate these lines correctly.)

Example log file:

```
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]
...
```

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.

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

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

Grading

The grading will be determined by 8 tests, which are supposed to be progressively harder and stress different elements of your code.

- 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 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).

Notes

- You must use C or C++
    - gcc 4.8.2 on Ubuntu 14.04, 64-bit
        - common mistake: libraries must go at the end of the compile command.
    - Your program must have a Makefile; running "make" should build all executables.
    - 10% penalty if your final submission needs Makefile tweaking to build correctly.

- Alone or with a partner is fine; see first paragraph.

- If you use git , do not check the .git metadata into the SVN repo: 5% penalty.
    - Do not use a public github, etc. repository. You will be held partially responsible for any resultant plagiarism.

- Your code must be your own. You can discuss very general concepts with others, but if you find yourself looking at a screen/whiteboard of pseudocode (let alone real code), you're 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 not probable", and penalties range from warnings to

recommending suspension/expulsion, based entirely on the instructor's impression of the situation.

- LS and DV/PV are two different ways to do the same thing. That means one implementation could be submitted as both, maybe with an attempt to disguise it. (Which won't hide you, but will make you look more guilty). This is cheating of the same magnitude as if you plagiarized one of the components from another group. Don't worry, a good faith effort to do a real LS and real DV cannot possibly be confused as a duplicate submission. They are extremely different approaches.
- Your nodes should not read any file other than the one it is given on the command line, and should not write any file other than its log. Having your nodes communicate via files, IPC, Unix domain sockets, etc, is cheating.
- The College of Engineering has some guidelines for penalties that we think are reasonable, but we reserve the right to ignore them when appropriate.
- You can use libraries from wherever for data structures. You MUST acknowledge the source in a README. Algorithms (e.g. Dijkstra's) should be your own.

- Your code must run on the test setup, which is just an Ubuntu 14.04 Server VM, running on VirtualBox. We will not look at your program on your laptop or EWS.

- Late penalty: 2% of total possible score per hour.

- **We'll release information on auto grader in next two weeks, around March 1st. Please be patient as we setup the auto grader.**

- **For our auto grader to work on all queued submissions – your program must complete in a specific time. So we will have timeouts on runtime on our auto grader. More info on this later.**

Link to supplemental code: http://courses.engr.illinois.edu/cs438/mp/mp2_supplemental_code.zip