CS168 Fall 2014
Project 1 - Distance Vector Routing
Due: October 6, 11:59:59 AM, Noon

1 Problem Statement

The goal of this project is for you to learn to implement distributed routing algorithms, where all routers run an algorithm that allows them to transport packets to their destination, but no central authority determines the forwarding paths. You will implement code to run at a router, and we will provide a routing simulator that builds a graph connecting your routers to each other and simulated hosts on the network.

The task at hand is to implement a version of distance vector protocol to provide stable, efficient, loop-free paths across the network. Routers share the paths they have with their neighbors, who use this information to construct their own *forwarding tables*.

2 Simulation Environment

You can download the simulation environment with full documentation at:
`http://inst.eecs.berkeley.edu/~cs168/fa14/projects/project1/project1.zip`

Below we describe only the class you will need to implement. Your `DVRouter` will extend the `Entity` class. Each `Entity` has a number of ports, each of which may be connected to another neighbor `Entity`. Entities send and receive `Packets` to and from their neighbors. The `Entity` superclass has four functions that will be relevant to you:

```
class Entity(__builtin__.object)
    handle_rx(self, packet, port)
        Called by the framework when the Entity self receives a packet.
        packet - a Packet (or subclass).
        port - port number it arrived on.
        You definitely want to override this function, in class DVRouter.

    send(self, packet, port=None, flood=False)
        Sends the packet out of a specific port or ports. If the packet's
        src is None, it will be set automatically to the Entity self.
        packet - a Packet (or subclass).
        port - a numeric port number, or a list of port numbers.
        flood - If True, the meaning of port is reversed - packets will
        be sent from all ports EXCEPT those listed.
        Do not override this function.

    get_port_count(self)
        Returns the number of ports this Entity has.
        Do not override this function.

    set_debug(self, *args)
        Turns all arguments into a debug message for this Entity.
        args - Arguments for the debug message.
        Do not override this function.
```

The Packet class contains four fields and a function that you should know:

```
class Packet(object)
    self.src
    self.dst
    self.ttl
    The time to live value of the packet.
    Automatically decremented for each Entity it goes.
    self.trace
    A list of every Entity that has handled the packet previously. This is
    here to help you debug. Don't use this information in your router logic.
```

3 DVRouter Specification

Write a DVRouter class which inherits from the Entity class. The function that you will need to override is handle_rx dealing with the following three types of packets (feel free to add more member functions to DVRouter).

- DiscoveryPackets are received by either ends of a link when either the link goes up or the link goes down. Your DVRouter should never send or forward DiscoveryPackets.
- RoutingUpdates contain the routing information that is received from the neighbours.
- Other packets are data packets which need to be sent on an appropriate port based on switch's current routing table.

DiscoveryPackets are sent automatically to both the ends of a link whenever the link goes up/down (the boolean variable is_link_up signifies whether the link has gone up or has gone down). You must handle these packets properly, i.e. receiving a DiscoveryPacket with a link up event signifies a 1-hop path (direct link) to the source of the packet. Your DVRouter should maintain a forwarding table and announce its paths to using the RoutingUpdate class (defined in sim/basics.py and extends Packet) and contains a field called paths which is a hash-table mapping advertised destinations to the distance metric. Your DVRouter implementation **must** use the RoutingUpdate class as specified to share forwarding tables between routers (otherwise it will not be compatible with our evaluation scripts and you will lose points, **even if your DVRouters are compatible with each other**. Look at the RoutingUpdate implementation for more details. Your implementation should perform the following:

- **Routing preferences**. On receiving RoutingUpdate messages from its neighbors, your router should prefer (1) routes with the lowest hop count, (2) for multiple routes with the same hop count, it should prefer routes to the neighbor with the lower port ID number.
- **Update packets**. Send update packets to neighbors when the forwarding table of your router changes. Update packets can also help withdraw routes the router previous announces to its neighbors by *implicit withdrawal* or *explicit withdrawal*[1].

[1] *Implicit withdrawal* means that routes to be withdrawn are excluded in a update, so the reicever knows implicitly what routes are no longer available. *Explicit withdrawal* means that routes to be withdrawn are marked with a distance of infinity, so the receiver knows directly. If you do incremental updates, *implicit withdrawal* is not an option.

- **Avoid loops when you can**. To prevent routing loops, your router should implement techniques like *split horizon*[2], *poisoned reverse*[3]. Implementation details can differ based on how your manage updates and how your update packets look like.
- **Count to infinity problem**. Note that split horizon/poisoned reverse are not sufficient for preventing routing loops in some cases (see lecture notes for examples). To deal with such un-preventable cases, a distance vector router could treat destinations with very long distance as unreachable. For this project, your implementation should stop counting at 50[4] and withdraw routes to corresponding destinations.
- **Deal with failures and new links**. Your solution should quickly and efficiently generate new, optimal routes when links fail or come up.

4 Tips

- A `Hub` class is provided that does basic packet flooding for you to get you started with the simulation environment and the visualizer. You can start modifying the dummy implementation provided in `dv_router.py`.
- Check out the simulator guide to see how to run your tests. A sample compatibility test is provided.
- Your routers should only have reactive behaviors and their behaviors should be deterministic. Avoid using things like timers or random number generators.

5 Grading

Your distance vector router will be run against a number of test cases, and we will evaluate it based on the following aspects:

Does it work? (80 pts) Tests in this category evaluate the correctness and basic functionalities of your DVRouter. For example, the protocol must stablize fast in cases of network changes and converge on shortest paths; your router should apply techniques like split horizon or poisoned reverse to avoid looping whereever possible (and count to 50 when looping cannot be avoided); the implementation should work independent of network topologies, and should not send redundant updates (you get penalty if your router is sending too many extra updates).

Handle link weights (10 pts) Extend your DVRouter to handle link weights, as opposed to simply hop couting. In this case, extract the link weight using latency field in the DiscoveryPacket. Note that now DiscoveryPackets will be also be sent when link cost changes, in addition to link up/down.

Incremental updates (10 pts) To save network bandwidth and processing time, your router might not want to send the complete distance vector for every update. For example, if router A sends out an update of (B:2, C:1, D:3) and later has new distances of 2 to C, and 4 to D. Instead of sending out its whole distance vector of (B:2, C:2, D:4), it can choose to send only the updated entries, (C:2, D:4). Incremental updates can greatly reduce update size in a big network. To use incremental updates is more than change all update packets into a "delta" version, rethink on when to send updates.

Warning: It's your decision to work on any of the last two 10 pts tasks or not, but keep in mind that we are accepting only one version of your implementation. Warnings are that it's easier to screw up the 80 pts worth

[2]If A's path to a destination C has B as the next hop then A should omit C from its update to B. If you do incremental updates, rethink on whether/how to use split horizon.

[3]If A's path to a destination C has B as the next hop then A's advertisement to B should have it's distance to C set to infinity, so B will not route to C through A. Check out the example in lecture.

[4]We will make that the longest distance will not exceed 50 for all grading tests.

of basic tests when you choose to do the last two. Trade-off.

6 README

You must also supply a README.txt file along with your solution. This should contain:

1. You (and your partner's) names.
2. What challenges did you face in implementing your router?
3. Name a feature NOT specified in this specification that would improve your router.
4. Specify if your code can handle link weights or do incremental updates. If you have implemented any of them, describe what additional considerations need to be taken into.

7 Downloads

Check the lecture schedule page for the latest copy of the project code.

8 What to Turn In

Turn in a .tar file with both you and your partner's last names in the file name (e.g. `project1-shenker-stoica.tar` or `project1-ratnasamy.tar`). The .tar file should contain `dv_router.py` that implements the `DVRouter` outlined above. It should also include your `README.txt` file. Use the following command to tar your archive:

```
tar -cf project1-partner1-partner2.tar dv_router.py README.txt
```

9 Cheating and Other Rules

You should not touch the simulator code itself (particularly code in `sim/core.py`). We are aware that Python is self-modifying and therefore you could write code that rewrites the simulator. *You will receive zero credit for turning in a solution that modifies the simulator itself. Don't do it.*

The project is designed to be solved independently, but you may work in partners if you wish. Grading will remain the same whether you choose to work alone or in partners; both partners will receive the same grade *regardless of the distribution of work between the two partners* (so choose a partner wisely!).

You may not share code with any classmates other than your partner, including your test code. You may discuss the assignment requirements or your solutions (e.g., what data structures were used to store routing tables, test scenarios) – *away from a computer and without sharing code* – but you should not discuss the detailed nature of your solution (e.g., what algorithm was used to compute the routing table). Also, don't put your code in public repository. Assignments suspected of cheating or forgery will be handled according to the Student Code of Conduct.[5] Apparently 23% of academic misconduct cases at a certain junior university are in Computer Science,[6] but we expect you *all* to uphold high academic integrity and pride in doing *your own work*.

[5] http://students.berkeley.edu/uga/conduct.pdf
[6] http://www.pcworld.com/article/194486/why_computer_science_students_cheat.html