CS168 Fall 2015
Project 1 - Distance Vector Routing
Due: October 5, 11:59:59 PM, Midnight

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.

Your ultimate task is to implement a version of a distance vector protocol (similar to RIP[1]) to provide efficient paths across the network. Routers share the paths they have with their neighbors, who use this information to construct their own *forwarding tables*. Before getting to this, you are also asked to implement a simple learning switch as a warm-up exercise.

2 Getting a bit more Concrete

In much of the lecture/discussion material, we talk about things relatively abstractly. For example, we discuss the algorithms on sort of generic graphs and compute distances between every node. In the real world, we have both switches/routers and hosts, and you're primarily concerned with whether hosts can reach other hosts. Which is to say, for the purposes of this assignment, you need not compute routes *to other routers* – only to hosts.

Similarly, we often speak about abstract *links*. Links in the real world are often a combination of a *port* (or *interface*) on one device, a *cable*, and a *port* on another device. A device is often not aware so much of a link as a whole as it is aware of its own side of the link – its own port. Ports are typically numbered. A device sends data out of one of its own ports, the data travels through the cable, and is received by the port on the other side. The API functions in the simulator reflect this: they deal in ports, not in links. Also, don't get these confused with the logical "ports" that are part of transport layer protocols like TCP and UDP. The ports we're talking about here are actual holes that you plug cables into!

3 Simulation Environment

You can find the download link for the lastest version of the simulation environment and its documentation on Piazza.

To complete the assignment, you will be creating subclasses of the `Entity` class. Each `Entity` subclass models a different type of device in the network (like a host or your router) and 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. Here are particularly relevant methods of the `Entity` superclass. Try `help(api.Entity)` in the simulator for more info.

[1] RIP v1: `https://www.ietf.org/rfc/rfc1058.txt`, RIP v2: `https://www.ietf.org/rfc/rfc2453.txt`

```
class Entity (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 method.

    handle_link_up (self, port, latency)
        Called by the framework when a link is attached to this Entity.
        port - local port number associated with the link
        latency - the latency of the attached link
        You probably want to override this method.

    handle_link_down (self, port)
        Called by the framework when a link is unattached from this Entity.
        port - local port number associated with the link
        You probably want to override this method.

    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 method.

    log (self, format, *args)
        Produces a log message
        format - The log message as a Python format string
        args - Arguments for the format string
        Do not override this method.
```

The `Packet` class contains some fields that you should know:

```
class Packet (object)
    self.src
    Packets have a source address.
    You generally don't need to set it yourself.  The "address" is actually a
    reference to the sending Entity, though you shouldn't access its attributes!

    self.dst
    Packets have a destination address.
    In some cases, packets aren't routeable -- they aren't supposed to be
    forwarded by one router to another.  These don't need destination addresses
    and have the address set to None.  Otherwise, this is a reference to a
    destination Entity.

    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.
```

4 Warm-Up Exercise: Learning Switch

To get started, you will implement a learning switch, which learns the location of hosts by monitoring traffic. At first, the switch simply floods any packet it receives to all of its neighbors, like a hub. For each packet it sees, it remembers for the sender `S` and the port that the packet came in on. Later, if it receives packets destined to `S`, it only forwards the packet out on the port that packets from `S` previously came in on (if packets from `S` *arrived* on port 3, then port 3 must be able to reach `S` – you can send *to* `S` via port 3). We've provided a skeleton `learning_switch.py` for you to modify.

Recall from class that a learning switch is not a very effective routing technique; its greatest shortcoming is that it breaks when the network has loops. That's why our next step will be to implement a more capable distance vector router. That's also why you only need to test it on topologies without loops!

5 DVRouter Specification

We've provided a skeleton `dv_router.py` file with the beginnings of a `DVRouter` class for you to flesh out, implementing your distance vector router. The `DVRouter` class inherits from the `DVRouterBase` class, which adds a little bit to the basic `Entity` class. Specifically, it adds a `POISON_MODE` flag and a `handle_timer` method. When your router's `self.POISON_MODE` is `True`, your router should send poisoned routes and poisoned reverses (and when `False`, it should not!). The `handle_timer` method is called periodically. When it is called, your router should send all its routes to its neighbors.

Ultimately, you will need to override some or all of the `handle_rx`, `handle_timer`, `handle_link_up`, and `handle_link_down` methods. Feel free to add whatever other methods you like. Note that your `DVRouter` instances should *only* communicate with other `DVRouter` instances via the sending of packets. Global variables, class variables, calling methods on other instances, etc. are not allowed – each `DVRouter` instance should be entirely standalone!

Your `handle_rx` will need to deal with two types of packets specially (that is, two subclasses of `Packet`), which are listed below. All other packets are data packets which need to be sent on an appropriate port based on the current routing table. You can differentiate packet types using Python's `isinstance()`. Note that while you can create custom packet types for your own testing or amusement, *you should not rely on any such modifications* for your router to work. Don't add any attributes besides the ones they already have.

- `basics.HostDiscoveryPacket`s are sent automatically by host entities when they are attached to a link. Your `DVRouter` should monitor for these packets so that it knows what hosts exist and where they are attached. Your `DVRouter` should never send or forward `HostDiscoveryPacket`s.
- `basics.RoutePacket` contains a single route. It has a `destination` attribute (the `Entity` that the route is routing to) and a `latency` attribute which is the distance to the `destination`. Take special note that `RoutePacket.destination` and `Packet.dst` are not the same thing! They are essentially at different layers. `dst` is like an L2 address – it's where this particular packet is destined (and since `RoutePacket`s should never be directly forwarded, this should probably be `None`). `destination` is at a higher layer and specifies which destination this route is for.

Your implementation should perform the following:

- **Expire routes**. On receiving a `RoutePacket` messages from a neighbor, your router should remember the route for approximately 15 seconds (but not less than that), after which it should be treated as expired.
- **Handle link weights**. Your router should use minimum weight/cost/latency paths – not just hop counts. Thus, you need to locally track the link latencies you are given by `handle_link_up`, you need to compute

correct weights when sending routes to neighbors, and you need to select the lowest cost routes available when forwarding.

- **Timed updates**. In response to a timer, your router should send its routes to its neighbors. This refreshes the route entries and keeps them from expiring!
- **Triggered updates**. When a router's forwarding table changes, it should immediately send updated routes to neighbors – but should *not* send ones which haven't changed!
- **Poison mode**. When poisoning is turned on, your router should implement route poisoning and poisoned reverse. That is, if it has sent a route which is now removed (or is now unavailable because it would loop), it should send a "poison" version of that route to promote its immediate removal (rather than waiting for it to expire). When poison mode is turned off, you should still implement split horizon!
- **Don't hairpin a packet**. Never forward a data packet back out the port it arrived on... this is seldom helpful.
- **Implement infinity as 16**. Note that split horizon/poisoned reverse is not sufficient for preventing routing loops in some cases (see lecture notes for examples). To deal with such un-preventable cases, your distance vector router should treat destinations with long distances as unreachable. For this project, your implementation should stop counting at 16 and remove routes to corresponding destinations.[2]
- **Deal with changes**. Your solution should quickly and efficiently generate new, optimal routes when the topology changes (e.g., when links are added or removed).

6 Tips

- The simulator comes with a couple test topologies, a random topology generator, and a topology file loader. These are in the "topos" directory. These may be helpful for your testing!
- The simulator comes with various examples in the "examples" directory, including a `Hub` class that does basic packet flooding which you can use to get familiar with the simulation environment and visualizer.
- The "examples" directory also contains a simple test scenario to get you started writing tests.
- You should start your own router by modifying the dummy implementations provided (`learning_switch.py` and `dv_router.py`).
- If your your router is deterministic, it is more likely that your results on our grading tests will be deterministic! If that's important to you, be deterministic: avoid using random numbers, iterate over items stable/sorted ways, and so on. In general, however, this should only be an issue if your implementation is sort of borderline – if it's working entirely correctly, it should (hopefully) pass the tests reliably.
- You should not need any additional `import` statements. If you want to use something like Python's collections package, that's fine. You should definitely *not* use (or need to use!) the time, threading, or socket packages. If you have questions: ask.

7 Submitting your work and Grading

You may submit your code for testing once per day, we will run a series of tests on it, and you can see how you're doing. The tests we run daily are similar to *at least some* of the tests we will run for the final grading, though we may not test all aspects of the project in the daily tests.

[2] We will make sure that no valid path in our test scenarios exceeds this length.

Your final grade will be the greater of `70%` * `highest_daily_score` and `score_on_final_tests`. In other words, if you get 100% on your daily tests, *you will not do worse than a 70% on the project.* The one exception is that we do not run the full set of anti-cheating tests in the daily runs, so if you manage to get 100% while cheating on the daily tests, you may still get 0% on the final grading.

You will use the provided OK tool to submit your work. See `http://cs61a.org/articles/using-ok.html` for some basic info on OK.

Unfortunately, OK uses Python 3, while the simulator itself requires Python 2. (Sorry – we live in a time of great Pythonic unrest.) This means you may need to install one or the other of these even on systems with Python installed! After installing, this also means you may have to type `python3 foo.py` to run a Python 3 program and/or `python2 foo.py` to run a Python 2 program. Details will depend on your particular operating system/installation – if you need assistance, post on Piazza (or ask a GSI).

Once Python issues are sorted out, submitting should be easy. From the commandline, do something like...

```
$ python3 ok --submit
```

The first time you run OK, you will be prompted to authenticate using an email address. *This should be the address you have set in Bear Facts and is almost always your @berkeley.edu address.* If you realize later that you entered the wrong email address, you can rectify it with `python3 ok --authenticate`. If you have problems authenticating, contact a GSI.

After submission, you'll be shown a confirmation message, and your test results will be emailed to you. Remember, you can upload new code once per day. The last code you submit will be the one we use for final grading. If you want to check on your submissions, you can find them on `http://okpy.org`.

8 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. As clever as doing such a thing would be, it may not end well for you. Additionally, don't override any of the methods which aren't clearly intended to be overridden, and don't alter any "constants". *You will receive zero credit for turning in a solution that modifies the simulator itself or otherwise subverts the assignment. Don't do it. If you're not sure about something: ask.*

Your router and learning switch should *only* communicate with other instances via the sending of packets. Global variables, class variables, calling methods on other instances, etc. are not allowed – each `DVRouter`/`LearningSwitch` instance should be entirely standalone!

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 anyone 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 a public repository. Assignments suspected of cheating or forgery will be handled according to the Student Code of Conduct.[3] Apparently 23% of academic misconduct cases at a certain junior university are in Computer Science,[4] but we expect you *all* to uphold high academic integrity and pride in doing *your own work.*

[3] `http://students.berkeley.edu/uga/conduct.pdf`

[4] `http://www.pcworld.com/article/194486/why_computer_science_students_cheat.html`