EE122 Fall 2013
Project 1 - Routing Protocols
Due: October 10, 2013, 11:59:59 PM

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 a RIP[1]-like 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/~ee122/fa13/projects/project1/project_1.tgz`

Below we describe only the class you will need to implement. Your `RIPRouter` 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 `Packet`s to and from their neighbors. The `Entity` superclass has four functions that will be relevant to you (there are a number more that you can feel free to peek at, but these should be sufficient to complete the assignment):

```
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.

    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.
```

[1] `http://en.wikipedia.org/wiki/Routing_Information_Protocol`

```
    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
    The origin of the packet.

    self.dst
    The destination of the packet.

    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.
```

3 RIPRouter Specification

Write a `RIPRouter` 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.

- `DiscoveryPacket`s are received by either ends of a link when either the link goes up or the link goes down.
- `RoutingUpdate`s 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.

`DiscoveryPacket`s 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`s with a link up event signifies a 1-hop path (direct link) to the source of the packet. Your RIPRouter 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 `RIPRouter` 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 RIPRouters 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.
- **Dealing with failures and new links**. Your solution should quickly and efficiently generate new, optimal routes when links fail or come up.

- **Implicit withdrawals**. RoutingUpdate packets should contain a list of all paths a router is willing to export. If a router previously announced a path, but a later update does not contain the announced path, the path is implicitly withdrawn.
- **Poison Reverse**. To prevent routing loops, your router should implement poison reverse, i.e., if `A`'s path to a destination `C` has `B` as the next hop then `A` should omit `C` from its update to `B`.

We will evaluate your RIP router on correctness, number of routing messages propagated, and how long it takes for new paths to stabilize in case of link failures (convergence time). Your RIP routing protocol must converge on shortest paths and must not deliver packets to hosts other than the one they were destined to. Your router should not send unnecessary routing updates – a 5% penalty will be put if the number of routing messages sent is more than twice of what our implementation sends. Also, do not accumulate updates since they would impact convergence time and your implementation can therefore fail our tests.

4 Tips and Tricks

- 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 `rip_router.py`.
- Check out the simulator guide to see how to run your tests. A sample compatibility test is provided.

We advise designing your RIP Router in phases:

1. Share routes once to develop stable paths. Don't worry about loops or link failures.
2. Send `RoutingUpdate`s to your neighbours whenever you observe an update to your own routing table.
3. Implement Poison Reverse to handle loops.

5 Bells and Whistles

5.1 Number of Routing Updates

Analyze the number of routing messages sent as a function of the network size. Use topology in Figure 1 that grows as a function of n. Plot the number of routing messages on Y-axis and n on the X-axis varying n from 3 to 50. Optimize the number of `RoutingUpdates` by being careful about when should you send them. Only the top 10% of the class with the least number of routing updates would get this extra credit.

5.2 Smart DV

Extend your RIPRouter to handle link weights. In this case, extract the link weight using `latency` field in the `DiscoveryPacket`. Note, that now `DiscoveryPackets` will be sent whenever the link goes up, link goes down, or link cost changes. This extra credit requires you to be smarter about processing DV updates so as to achieve better convergence time and lesser number of routing updates? (*H*int: when you receive an update from a neighbour, are there other entries not corresponding to neighbour's column that you can change?)

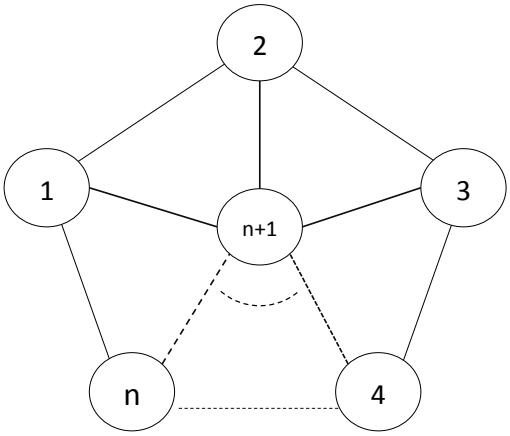

Figure 1: Topology that grows as a function of n

5.3 Link State

An alternative to Distance Vector algorithms is a Link State design. Instead of exchanging best paths, nodes exchange lists of all of their neighbors so that each node in the network has a map of the entire network topology. Each node then independently calculates the best path across the network for each destination. Implement a router that uses a Link State algorithm called `LSRouter`. Benchmark the convergence time (time between a link failure/link coming up and all of the nodes coming to a stable set of new paths) of your `LSRouter` against the convergence time of your `RIPRouter`.
Turn in: Your implementation and an analysis of the convergence times: which performs better, and why?

6 README

You must also supply a `README.txt` or `README.pdf` 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 the extra credit that would improve your router.
4. Name the extra credit options you implemented; describe what they do and how they work.

7 Downloads

Check the lecture schedule page [right-most column] 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 `rip_router.py` that implements the `RIPRouter` outlined above. It should also include your `README.txt/README.pdf` file. You may optionally include additional

files with your extra credit versions of the protocols (mention in the README). Use the following command to tar your archive (replace README.txt by README.pdf if you are submitting a pdf).

```
tar -cf project1-partner1-partner2.tar learning_switch.py rip_router.py README.txt bonus_files
```

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. You may discuss the assignment requirements or your solutions (e.g., what data structures were used to store routing tables) — *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). Assignments suspected of cheating or forgery will be handled according to the Student Code of Conduct.[2] Apparently 23% of academic misconduct cases at a certain junior university are in Computer Science,[3] but we expect you *all* to uphold high academic integrity and pride in doing *your own work*.

[2] `http://students.berkeley.edu/uga/conduct.pdf`
[3] `http://www.pcworld.com/article/194486/why_computer_science_students_cheat.html`