Project 2 - Routing Protocols

UC Berkeley EE 122, Fall 2011
Version 0.1

Due: October 24, 2011, 11:59:59pm

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.

To get started, you will implement a learning switch[1], which *learns* the location of hosts by monitoring traffic. At first, the switch simply broadcasts any packet it receives to all of its neighbors. For each packet it sees, it remembers for the sender *S* 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.

Recall from class that a learning switch is not a very effective routing technique: it breaks when the network has loops, cannot adapt to routing failures, and is not guaranteed to find efficient paths. Hence you will then implement a RIP[2]-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*.

Simulation Environment

You can download the simulation environment with full documentation at:
http://cs.berkeley.edu/~justine/ee122/project_2.tgz

Below we describe only the classes you will need to implement. Your `LearningSwitch` and `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 six functions that will be relevant to you:

```
class Entity(__builtin__.object)|
     |  handle_link_down(self, port, entity)
     |      Called by the simulator when the Entity self is disconnected
     |      from another entity.
     |      port - port number that was disconnected.
     |      entity - Entity to which it was previously connected.
     |      (You definitely want to override this function.)
     |
```

[1] http://en.wikipedia.org/wiki/Bridging_(networking)
[2] http://en.wikipedia.org/wiki/Routing_Information_Protocol

```
|  handle_link_up(self, port, entity)
|      Called by the simulator when the Entity self is connected to
|      another entity.
|      port - port number that is now connected.
|      entity - other Entity that is now connected.
|      (You definitely want to override this function.)
|
|  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.)
|
|  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 will be relevant to you:

```
class Packet (object):
|  self.src
|  The origin of the packet.
|
|  self.dest
|  The destination of the packet.
|
|  self.ttl
|  The time to live value of the packet.
|  Automatically decremented for each Entity it goes through.
|
|  self.trace
|  A list of every Entity that has handled the packet previously. This is
|  here to help you debug.
```

In addition, the `RoutingUpdate` class (which is found in sim/basics.py and extends `Packet`) allows routers to share their forwarding tables with each other. 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***).

Learning Switch Specification

You will write a `LearningSwitch` class which inherits from the `Entity` class, overriding the `handle_rx` function. If a packet comes in destined to an `Entity` from whom you have never seen a packet before, broadcast the packet on all ports except the port which the packet came in on. If a packet arrives destined to an `Entity` from whom you have previously received a packet, only send the packet out on the port from which packets from that `Entity` previously arrived. Your `LearningSwitch` does not need to deal with links coming up or down, or networks with loops.

RIPRouter Specification

Write a `RIPRouter` class which inherits from the `Entity` class, overriding `handle_rx`, `handle_link_down`, and `handle_link_up`. Your RIP Router should maintain a forwarding table and announce its paths to its neighbors using the `RoutingUpdate` class (which extends Packet). 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.
- ***Split Horizon with Poison Reverse.*** To prevent routing loops, your router should announce `poison reverse' announcements with hop count values of 100.

We will test your learning switch to verify that it broadcasts and learns as specified. 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.

Tips and Tricks

The LearningSwitch specification is very similar to the Hub class provided by default with the source code. Copy/Paste the Hub class to create a new LearningSwitch class, and then modify it to have “learning” capabilities.

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) Periodically send routing updates to change paths in case of routing failures.

3) Implement Split Horizon and Poison Reverse to handle loops.

Bells and Whistles

Link Weights:

RIP uses hop count as the metric to find `best' paths, under the assumption that the best paths are those with fewer hops. However, operators may choose to use other metrics, e.g: link latencies or the cost ($) of using a link. Choose two metrics[3] other than hop count and describe under what circumstances a network operator would choose to use each metric. Then design a network graph with two hosts, h1 and h2, whose `best path' is different for each of the three metrics.

Turn in: A one page file with (1) a description of your metrics, (2) why one might choose each of the metrics, (3) your network graph, and (4) the best path between h1 and h2 under each of the three metrics.

Link State Routing:

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?

I'm a superhacker:

RIP expects relatively stable links, in that they go down/come up relatively infrequently. Otherwise, routing tables do not have time to converge on good paths and packets can become lost or loop forever. Design a new routing protocol to address a scenario where each link fails/comes alive faster than your routes can propagate. Implement your protocol, and evaluate it for any or all of the following: (1) packet loss - what fraction of packets sent reach their destination? (2) latency - how long does it take for packets to reach their destination? (3) efficiency - how many entities must touch the packet (or a copy of it) before it reaches its destination?

Turn in: Your implementation and an analysis of your protocol.

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] You may use latency or cost as one of your metrics, but the other metric must be one of your own design.

(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.

What to Turn In

Turn in a .tar file with both you and your partner's last names in the file name (e.g. project2-shenker-stoica.tar or project2-katz.tar). The .tar file should contain two Python files, learning_switch.py that implements the *LearningSwitch* outlined above, and 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.

Remember, capitalization matters: if we specify something should be capitalized, make it capitalized. We specify a TAR archive: do not submit any other kind of archive (7zip, zip, rar, gzipped TAR, etc) if you want to receive credit for your submission.

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.[4] Apparently 23% of academic misconduct cases at a certain junior university are in Computer Science[5], but we expect *you all* to uphold high academic integrity and pride in doing *your own work*.

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