Project 1 – Distance Vector Routing
Due: October 1, 2018 11:59 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 to simulated hosts on the network. By the end of this project, you will have implemented a version of a distance vector protocol that computes efficient paths across the network.

2 Getting a bit more Concrete

In much of the class material, we discussed routing abstractly, i.e., the algorithms discussed were used on graphs and computed distances between every pair of nodes. 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 other 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, i.e., its own port. Ports are typically numbered. When 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.[1]

3 Simulation Environment

You will be developing your router in Python 2.7 under a simulation environment provided by us. The simulation environment, as well as your router implementation, lives under the `simulation` directory; you should `cd` into this directory before entering any terminal commands provided in this document.

In the simulation environment, every type of network device (e.g., a host or your router) is modeled by a subclass of the `Router` class. Each `Router` has a number of ports, each of which may be connected to a neighboring entity (e.g., a host or another router). Each link connecting two entities has a **latency**—think of it as the link's propagation delay. Your `Router` sends and receives `Packet`'s to and from its neighbors.

Both the `Router` and `Packet` classes are defined in the `sim.basics` module. Relevant methods of the two classes are displayed in Figures 1 and 2; now might be a good time to skim through them. You can learn more about the simulation environment from the Simulator Guide.

Before we begin, let's make sure that your Python version is supported. Type in your terminal:

```
$ python --version
```

You should be good to go if the printed version has the form `Python 2.7.*`.

[1] 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!

```
class Router (api.Entity)
    handle_link_up (self, port, latency)
        Called by the framework when a link is attached to this router.
        port - local port number associated with the link
        latency - the latency of the attached link
        You should 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 should override this method.

    add_static_route (self, host, port)
        Adds a static route to a host directly connected to this router.
        Called by the framework when a host is connected.
        host - the host that got connected
        port - the port that the host got connected to
        You should override this method.

    handle_route_advertisement (self, dst, port, route_latency)
        Called by the framework when the router receives a route
        advertisement from a neighbor.
        dst - the destination of the advertised route
        port - the port that the advertisement came from
        route_latency - the route's latency (from neighbor to dst)
        You should override this method.

    handle_data_packet (self, packet, in_port)
        Called by the framework when a data packet arrives at this router.
        packet - a Packet (or subclass)
        port - port number it arrived on
        You should 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.
```

Figure 1: Relevant methods of the Router superclass.

```
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.
```

Figure 2: Relevant methods of the `Packet` class.

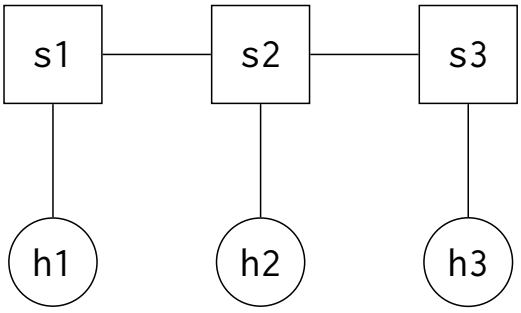

Figure 3: Linear topology with three hosts. Circles denote hosts, squares denote routers, and lines denote links. A link has a latency of 1 unless otherwise specified.

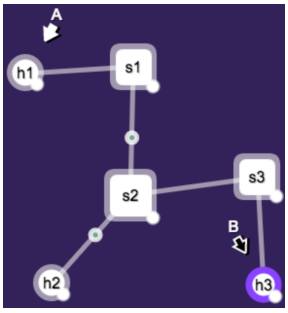

Figure 4: Sending a "ping" from the visualizer in the hub example. (Packets shown are "pong" packets sent by h3 back to h1 being flooded by hub s2.)

4 Warm-up Example: Hub

To get you started, we have provided an implementation of a **hub**—a network device that floods any packet it receives to all of its ports (other than the port that the packet came from). *The hub is already implemented and you don't need to submit anything for this section.*

Take a look at the hub implementation in `examples/hub.py`. Having no need to record any routes, the hub only implements the `handle_data_packet` method to flood data packets.

Let's try out the hub on a **linear** topology with three hosts (Figure 3):

```
$ python simulator.py --start --default-switch-type=examples.hub topos.linear --n=3
```

You can now access the **visualizer** at `http://127.0.0.1:4444` using your browser; you should see the hosts and routers displayed against a purple background. Let's now make host h1 send a ping packet to host h3. You can either type into the Python terminal:

```
>>> h1.ping(h3)
```

or you can send the ping from the visualizer by: (1) selecting h1 and pressing A on the keyboard; (2) selecting h3 and pressing B; and (3) pressing P to send a ping from host A to host B (Figure 4).

You should see the "ping" and "pong" packets being delivered between h1 and h3. You should also see both packets delivered to h2 despite it not being the recipient. This behavior is expected since the hub simply floods packets everywhere. You may also observe what's going on from the log messages printed to the Python terminal:[2]

```
WARNING:user:h2:NOT FOR ME: <Ping h1->h3 ttl:17> s1,s2,h2
DEBUG:user:h3:rx: <Ping h1->h3 ttl:16> s1,s2,s3,h3
WARNING:user:h2:NOT FOR ME: <Pong <Ping h1->h3 ttl:16>> s3,s2,h2
DEBUG:user:h1:rx: <Pong <Ping h1->h3 ttl:16>> s3,s2,s1,h1
```

Recall from class that flooding is problematic when the network has loops. Let's see this in action by launching the simulator with the topos.candy topology, which has a loop (Figure 5):

```
$ python simulator.py --start --default-switch-type=examples.hub topos.candy
```

Now, send a ping from host h1a to host h2b. You should be seeing a lot more log messages in the terminal, and the visualizer should be showing routers forwarding superfluous packets for quite a while. Oops! This is why our next step will be to implement a more capable distance vector router.

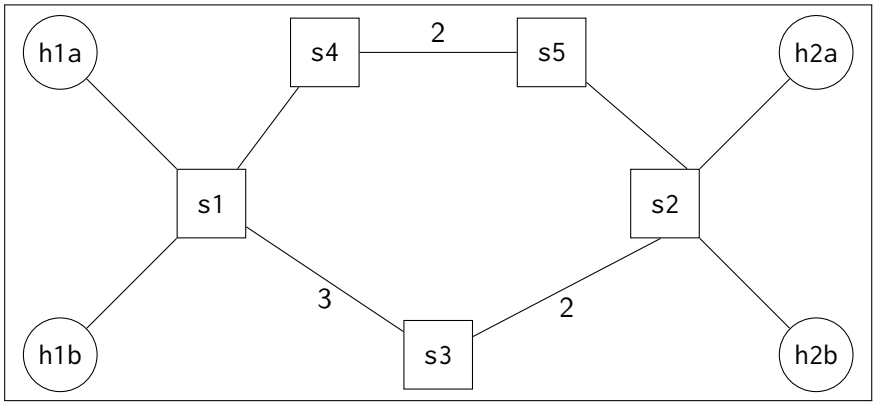

Figure 5: The topos.candy topology, which has a loop. A link has latency 1 unless otherwise specified. We'll be using this topology for demonstrative purposes throughout the project.

5 Distance Vector Router

We've provided a skeleton dv_router.py file with the beginnings of a DVRouter class for you to flesh out. The DVRouter class inherits from the DVRouterBase class, which adds a little bit to the basic Router 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 (every 5 s by default).

[2] You can see a ttl field printed for each packet. The simulator automatically assigns each packet a "time to live" (TTL)—any packet will only be forwarded up to some maximum number of times. The TTL is managed entirely by the simulator; you should **not** read or write the ttl field.

To guide your implementation of `DVRouter`, we have split the implementation process into **nine stages**, each of which covers one aspect of the router. You should follow along stage by stage, and by the end, you will have implemented a functional distance vector router!

Testing

To help you check your work, we provide you with unit and comprehensive tests. These tests will be the **only** tests that we'll use to grade your submission, i.e., there will be no hidden tests (see Section 6 for details on grading).

The **unit tests** test each aspect of the router separately and correspond to the nine stages of implementation.[3] After you finish each stage, you should run the unit tests for that stage (and all previous stages) and make sure you pass them. For example, after you finish Stage 5, you can run unit tests for the first five stages by invoking:

```
$ python dv_unit_tests.py 5
```

If your tests fail, you should read the test's documentation and source code in `dv_unit_tests.py` to figure out what the test does. You should not change any tests except for debugging purposes.

Hint: When you debug a test failure, you should read not only the documentation for a single test function (e.g., `test_handle_link_up`), but also the docstring for the enclosing class (e.g., `TestAdvertise`), which may include useful information about the common setup of all tests in this stage.

Note: The unit tests are **not** guaranteed to be comprehensive—it is possible for your implementation to have a defect in one stage that manifests itself by failing unit tests for a later stage, or for your implementation to pass all the unit tests but fail the comprehensive test.

The **comprehensive test** checks to make sure that your **completed** router has good routing behavior on pseudo-randomly generated topologies. The test proceeds in rounds; in each round, it changes the topology by adding and/or removing some random links, waits a bit for the routes to converge, sends pings from every host to every other host, and then checks to ensure that the "ping" packets arrive in time.

After you finish your implementation, you may run the comprehensive test like this:

```
$ python simulator.py --default-switch-type=dv_router \
    --default-host-type=dv_comprehensive_test.TestHost \
    topos.rand --switches=5 --links=10 --seed=1 \
    dv_comprehensive_test --seed=43
```

```
>>> start()
```

This command will launch the simulator on a pseudorandomly generated topology with 5 switches and 10 links. The comprehensive test will start running after you enter `start()` into the Python terminal. As usual, you may observe the test progress in the visualizer.

The comprehensive test will run indefinitely until a test failure occurs. If that happens, you can type commands into the Python terminal and use the visualizer to debug.

[3] There are also "Stage 0" tests that simply make sure certain parts of the skeleton code are intact. You will not be graded on the "Stage 0" tests.

The command that starts the comprehensive test comes with two random seeds (specified by the `--seed` flag). The first seed controls the random topology generation; the second controls how the test changes the topology in each round. Feel free to provide different seeds to test your router under different scenarios.

For maximum efficiency, the comprehensive test always runs your router with "poison mode" turned on (you'll understand what this means once you get to Section 5.3).

Note: The unit and comprehensive tests are subject to change. Any changes we make before the deadline will be publicized on Piazza.

Requirements

Before we get started with the implementation, let's lay down some ground rules:

- Your `DVRouter` implementation must live entirely in `dv_router.py`; **do not** add other files.
- You should **not** touch the simulator code itself or the unit tests. Nor should you write code that dynamically modifies the simulator or the tests. 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. If you're not sure about something: ask.*
- Your `DVRouter` instances should communicate with other `DVRouter` instances **only** 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!
- The constructor (`__init__`) we provide for `DVRouter` defines several instance variables for the class (e.g., `self.peer_tables` and `self.forwarding_table`). **Do not** modify or remove these definitions; you'll be using them later on. Similarly, **do not** remove any existing method definitions; you'll be filling in their implementations.
- However, feel free to add your own instance variables and/or helper methods, as long as they don't break the unit and comprehensive tests.
- Your `DVRouter` implementation must work with the unmodified `dv_utils.py` file (which contains some helper classes).
- You should not need any additional `import` statements. It would be fine for you to use, say, Python's `collections` module. However, you should **not** use (or need to use!) the `time`, `threading`, or `socket` modules. If you have questions, ask!
- You must solve this project **individually**. You may not share code with anyone, including any custom test code that you may write. You may discuss the assignment requirements or your solutions—*away from a computer and without sharing code*—but you should not discuss the detailed nature of your solution. Also, don't put your code in a public repository. We expect you all to uphold high academic integrity and pride in doing **your own work**. Assignments suspected of cheating or forgery will be handled according to the Student Code of Conduct[4].

Let's get started on implementing `DVRouter`.

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

5.1 Construct and Use the Forwarding Table

Recall from class that when a packet arrives at a router on the data plane, the router consults its **forwarding table** to determine where to forward the packet. In this first section, you will construct the forwarding table and use it to forward data packets.

Data Structures

The forwarding table is constructed from **peer tables**. For each port, your router maintains a peer table, which records all advertised routes it has received on this port. A route in a peer table is represented by a `PeerTableEntry` object[5], which has the following attributes:

- `dst`: the destination for this advertised route.
- `latency`: the latency of the route **from the neighbor** to the destination (i.e., **not including** the latency of the link on this port).
- `expire_time`: the timestamp (in seconds) at which this route expires.

For example, you may represent an advertised route to host h1, of latency 10, which expires in 20 seconds, using this `PeerTableEntry` object:

```
pte = PeerTableEntry(dst=h1, latency=10, expire_time=api.current_time()+20)
```

Note, in particular, that you should call `api.current_time()` to acquire the current timestamp.

A `PeerTableEntry` object is immutable. If you wish to update an attribute, you should create a new `PeerTableEntry` object with updated attributes.

A peer table is then represented by a `PeerTable` object, which you should use as a Python `dict` mapping a destination host to a `PeerTableEntry`. You can construct and use a peer table like this:

```
pt = PeerTable()
pt[h1] = PeerTableEntry(dst=h1, latency=10, expire_time=api.current_time()+20)
pt[h2] = PeerTableEntry(dst=h2, latency=20, expire_time=api.current_time()+20)

for host, entry in pt.items():  # <-- This is how you iterate through a dict.
    print "Route to {} has latency {}".format(host, entry.latency)
```

Your router keeps all its peer tables in the instance variable `self.peer_tables`, which is a Python `dict` mapping each port number to the `PeerTable` object for that port.

Your router will merge its peer tables into a forwarding table, which contains the route chosen for each destination. Your router will consult this table when making a decision on where to forward a packet.

A forwarding table is represented by a `ForwardingTable` object, which you should treat as a Python `dict` mapping destination hosts to `ForwardingTableEntry` objects. A `ForwardingTableEntry` object is immutable and represents the routers chosen route for a destination; it has the following attributes:

- `dst`: the route's destination host.
- `port`: the port that this route takes (i.e., a packet for `dst` should be sent out of this port).

[5] All helper classes are defined in `dv_utils.py`.

- `latency`: the latency of the route (from this router to the destination).

The router's forwarding table is stored in the instance variable `self.forwarding_table`.

Stage 1: Compute the Forwarding Table

▶ Implement the `update_forwarding_table` method to compute a new forwarding table by combining peer tables from neighbors, picking the shortest route to each destination. The new forwarding table should be stored in `self.forwarding_table`. Don't worry about populating the peer tables for now—the unit tests for this stage will populate the peer tables in `self.peer_tables` for you before calling `update_forwarding_table()`.

To compute the total latency of a route, you'll need the link latency of the port that the route goes through. You can look it up in the dictionary `self.link_latency`, which maps a port number to its link latency and is populated by the skeleton code in `handle_link_up` whenever a link goes up.[6]

Note: In case there exist multiple shortest paths to a destination, you may break the tie arbitrarily.

Note: The `update_forwarding_table` method is **not** responsible for checking for route expiry; it should simply combine **all existing** peer table entries into a new forwarding table.

Stage 2: Forward Data Packets

Using its forwarding table, your router can forward packets on the data plane. The `handle_data_packet` method is called whenever a data packet arrives at your router. ▶ Implement the `handle_data_packet` method to handle data packets appropriately.

Note: If no route exists for a packet's destination, your router should drop the packet.

Note: Moreover, to deal with routing loops, your router should treat destinations with long distances as unreachable. Specifically, ▶ the `handle_data_packet` method should **drop** any packet whose distance to destination is greater than or equal to `INFINITY`, a constant with value 16. (You should, however, leave these long routes in the forwarding table.)

Note: Never send packets back where they came from! (This undesired behavior is called "hairpinning.")

Stage 3: Advertise Routes

For routers to learn one another's routes, each router must advertise the routes in its forwarding table to its neighbors. There are many scenarios where a router should send route advertisements. For now, let's work on two of those scenarios.

Link up: When a link comes up on a port, your router should get its new neighbor up to speed by immediately advertising all of its routes to that port. ▶ Add this feature to the `handle_link_up` method.

Timer: Your router should advertise routes periodically (every time the timer fires) in order to refresh the routes and keep them from expiring. The provided timer handler `handle_timer` advertises all routes in the forwarding table by calling `self.send_routes(force=True)`. The `force` argument taken by `send_routes`

[6] The `link_latency` dictionary corresponds to the "cost table" introduced in lecture.

dictates whether to advertise **all** routes (if `force=True`) or to advertise only those routes that have changed since the last advertisement (if `force=False`). ▶ Implement the `send_routes` method for the `force=True` case. For now, don't worry about the `force=False` case—it won't be needed until a later stage on incremental updates.

To advertise a route, you should construct and send a packet of type `basics.RoutePacket`, which contains a single route. Its constructor takes a `destination` argument (the host that the route is routing to) and a `latency` argument (the distance to `destination`).[7]

▶ Be sure to implement **split horizon**, but don't worry about poisoned reverse for now.

To deal with routing loops, ▶ your router should **stop counting** at `INFINITY`, i.e., it should advertise any route with latency greater than `INFINITY` as `latency=INFINITY`. You can achieve this by capping route latency at `INFINITY` either when you create a `ForwardingTableEntry` or when you create a route advertisement.

Note: We'll add in logic to handle these route advertisements in a later stage.

5.2 Populate and Maintain Peer Tables

Now that your router can construct a forwarding table from its peer tables, we turn to populating and maintaining the peer tables themselves.

Stage 4: Add Static Routes

For each host that **directly** connects to your router, your router should record a **static route** to that host. ▶ Implement the `add_static_route` method to add a static route to the appropriate peer table; this method is called by the framework for every host that directly connects to your router. You should treat a static route simply as a route that **never expires**. To this end, you should set the expiry time of such routes to `PeerTableEntry.FOREVER`, which is a `float` constant that is larger than any finite number.

Note: You may assume, if convenient, that links directly connecting to hosts will never go down (and so static routes stay around forever), and that no host is directly connected to multiple routers.

At this point, your router should be able to forward packets between hosts that are directly connected to it. Let's try it out in the candy topology (Figure 5):

```
$ python simulator.py --start --default-switch-type=dv_router topos.candy
>>> h1a.ping(h1b)
```

Observe in your terminal and/or in the visualizer that the "ping" and "pong" packets are being delivered successfully. However, your current router cannot forward packets to hosts through other routers. Enter:

```
>>> h1a.ping(h2b)
```

and notice that the ping packet is dropped by router s1. This is because router s1 won't know of a route to destination h2b until it handles route advertisements from its neighbors. Let's implement that next.

[7] 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 `RoutePackets` should never be directly forwarded, this should probably be `None`). `destination` is at a higher layer and specifies which destination this route is for.

Stage 5: Handle Route Advertisements

Recall from Stage 3 that your router sends route advertisements to its neighbors; these advertisements got to be handled! ▶ Implement the `handle_route_advertisement` method, which is called by the framework when your router receives a route advertisement from a neighbor. This method should update any relevant peer table(s) and the forwarding table. Each time you receive a route advertisement, you should set the route's expiry time route to `ROUTE_TTL` seconds in the future (15 s by default).

Note: To get the current time in seconds, call `api.current_time()`. **Do not** use Python's `time` module.

Now, re-launch the simulator using the candy topology, wait around 15 s for the routes to converge through periodic route advertisements, and try `h1a.ping(h2b)`. You should see the "ping" and "pong" packets routed along the shortest path: h1a » s1 » s4 » s5 » s2 » h2b.

Food for thought: Can you call `h1a.ping(h2b)` at an opportune time such that the ping packet and the pong packet are forwarded along different routes?

Before you shut down the simulation, let's explore what happens when a link goes down. Remove the link between routers s4 and s5 by typing:

```
>>> s4.unlinkTo(s5)
```

and try sending a ping from h1a to h2b. Even though there still exists a route between the two hosts through router s3, you will see the ping packet get forwarded to router s4 and get dropped—s4 wants to forward the packet to s5, to which it is no longer connected! In the next stage, you will be adding functionality to handle route removals correctly so that the ping would take the alternate route.

Stage 6: Remove Routes

Your router should remove routes in two cases.

Link down. When a link goes down, your router should stop forwarding packets to that link. ▶ Implement the `handle_link_down` method, which is called by the framework when a link goes down, to remove any routes that goes through that link.

When you're done, re-launch the simulation, wait for routes to converge, remove the s4—s5 link, and send a ping from h1a to h2b (we'll refer to this as the "link down" experiment). Unfortunately, you should still see the same undesired outcome—the ping packet getting dropped by router s4. This is because even though s4 stops advertising its route to h2b after the link down event, its neighbor s1 still remembers that route and so continues to forward the ping packet to s4. To fix this problem, let's make sure that a route **expire** when it is no longer being advertised.

Timer. Previously, you assigned expiry times to your peer table entries. ▶ Implement the `expire_routes` method, which should clear out any expired routes and update the forwarding table as appropriate. The `expire_routes` method is called in the timer handler by the skeleton code.

Now, if you re-run the "link down" experiment, you should see the ping packet being forwarded correctly along the alternate route roughly 15 s after the link down event—this is when the old route has expired.

Your distance vector router now has all its basic functionality! However, it's not super efficient—for ex-

ample, route convergence takes quite a while in the candy topology. In the next (and last) portion of this programming project, we'll add in some extra loop prevention logic and some optimizations to make route propagation and removal more efficient.

5.3 Enhancements

Stage 7: Poisoned Reverse

In a previous stage, you implemented split horizon as a loop prevention mechanism. Let's get more aggressive with loop prevention by actively advertising the non-existence of a route to the port that the route takes.

▶ Augment your `send_routes` method to implement poisoned reverse—to each neighbor, advertise as unreachable any destination for which this router's route goes through that neighbor. (These would be route advertisements with `latency=INFINITY`.) You should send poisoned reverse advertisements **only if** `self.POISON_MODE` is `True`; when poison mode is off, you should still implement split horizon!

Stage 8: Incremental and Triggered Updates

Recall from previous stages that it took quite a while (at least 15 s) for routes to converge when you started the simulator on the candy topology, partly because your routers only advertised routes every time the timer fires. Route convergence would become faster if routes are advertised every time the forwarding table changes.

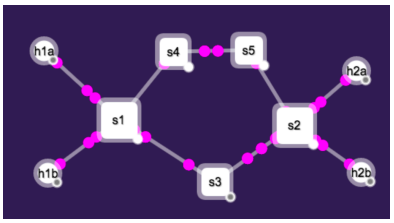

Figure 6: Triggered route advertisements are sent as soon as the simulation starts.

In this stage, you'll be implementing incremental and triggered updates—your router will advertise routes every time its forwarding table is updated (**triggered**), and will advertise only those routes that have changed (**incremental**). Don't worry about removed routes for now.

You will implement incremental updates by augmenting the `send_routes` method to handle the `force=False` case, where only route advertisements that differ from before should be sent. Here is our recommended implementation strategy:

- Maintain a "history" data structure that records the latest route advertisement sent out of each port for each destination host.
- Implement `send_routes(force=False)` so that it sends a route advertisement only if (1) it is missing from the "history" data structure, or (2) it differs from the corresponding entry in the history.
- Implement `send_routes(force=True)` to simply send all route advertisements, ignoring the history.
- In either case, `send_routes` should update the "history" data structure as appropriate.

▶ Implement the `force=False` case for the `send_routes` method. Then, ▶ send incremental triggered updates by calling `send_routes(force=False)` wherever necessary. (In how many different places do you need to insert this call?)

Hint: The timer should still advertise **all** routes as before.

To see the improved route convergence performance in action, re-launch the simulator on the candy topology, but don't pass the `--start` argument (and so the simulation doesn't start immediately):

```
$ python simulator.py --default-switch-type=dv_router topos.candy
```

Open the visualizer in a browser window (`http://localhost:4444`), and then start the simulation from the Python terminal:

```
>>> start()
```

Observe the route advertisement packets that get sent immediately (Figure 6). Wait around 5 s, then send a ping from host h1a to host h2b. You should see the "ping" and "pong" packets routed along the shortest path h1a » s1 » s4 » s5 » s2 » h2b. The route from h1a to h2b has converged to the shortest route!

Stage 9: Route Poisoning

In this stage, you'll improve incremental updates by **poisoning** any routes that are removed from the forwarding table, i.e., advertise those routes as `latency=INFINITY`. You should poison routes only when `self.POISON_MODE` is set to `True`.

Since poison advertisement packets can be dropped, your router should make sure to advertise poisoned routes **periodically** for at least `ROUTE_TTL` seconds (15 s by default). Ideally, these periodic advertisements should halt after a while (since it is of little use to keep advertising the **nonexistence** of a route), but we will not test you on this.

▶ Augment your code to implement route poisoning when `self.POISON_MODE` is set.

6 Submitting your work and Grading

Submission instructions will be posted on Piazza before the deadline. Be sure to familiarize yourself with the **late policy** outlined in the syllabus on the course website.

The grade for your `DVRouter` implementation will come from two parts:

- **90%** of your grade will come from the unit tests that we have provided to you. This portion of your grade is split equally among the nine stages (i.e., 10% for each stage). Within each stage, all tests are weighted equally.
- **10%** of your grade will come from the comprehensive test. As explained in the beginning of Section 5, the comprehensive test will only be run with poison mode turned on.

Any further details on grading will be posted on Piazza.