Project 1 – Distance Vector Routing (Simulator Guide)

This document contains a detailed description on the routing simulator you'll be using for Project 1.

1 File Layout

`simulator.py` Starts up the simulator (see section below).
`dv_router.py` Starting point for your distance vector router.
`dv_utils.py` Contains helper classes for your distance vector router implementation.
`dv_unit_tests.py` Stage-by-stage unit tests for your distance vector router.
`dv_comprehensive_test.py` Comprehensive test for your distance vector router.
`sim/api.py` Parts of the simulator that you'll need to use (such as the `Entity` class). See `help(api)`.
`sim/basics.py` Basic simulator pieces built with the API. See `help(basics)`.
`sim/core.py` Inner workings of the simulator. Keep out.
`topos/` Test topologies and topology generators that you can use and modify for your own testing.
`examples/` Examples for Entities, interacting with NetVis, automating your testing, and so forth.

2 Understanding the Simulator Command Line

The typical way to start the simulator is using `simulator.py`. This starts up a number of "modules" and then begins simulation. Modules are Python scripts which typically implement a `launch` function. A module might, for example, create a topology or run an automated test.

The `simulator.py` script itself may take command-line options. These come at the very beginning of the command line and are prefixed with two dashes (e.g., `--foo`). Command-line arguments without the two dashes specify the names of modules to be loaded (in the order in which they'll be loaded). Modules can also have options (again prefixed with two dashes). So a complete command line might look like:

```
$ python simulator.py --sim-opt1=value --no-sim-opt2 \
    mod1 --mod1-opt1=value mod2 --mod2-opt1
```

Here we're passing two options to the simulator, and loading two modules, each of which are getting one option. Note that normally options are of the form `--name=value`. For boolean options, you just do `--name` or `--no-name`. Module options are automatically passed to the module's `launch` function.

3 Micro-tutorial

Let's try a quick introduction. Since you haven't implemented a router yet, we'll use the simple example Hub entity, which just takes every packet it receives and sends it on every other link.

We'll use the `topos.rand` module to create a random topology with four switches/routers (hubs in this case), four hosts (one per switch), and three links. Since there are four switches and only three links, this topology is guaranteed not to have loops, which is important because hubs don't do so well on topologies with loops!

```
$ python simulator.py --default-switch-type=examples.hub topos.rand \
      --switches=4 --links=3 --hosts=4
```

Executing the above should give some informational text (including a listing of the names of each node in the topology—in this case, h1–h4 and s1–s4) and then a Python interpreter in the terminal. You can now open the visualizer by accessing `http://127.0.0.1:4444` in your browser. If the simulator doesn't start up, it commonly due to an `Address already in use` exception. If this occurs, it's likely that you have an old instance running which you should quit. You could also add the `--no-remote-interface` option, which disables the visualizer.

From the interpreter in the terminal, you can run arbitrary Python code, inspect and manipulate the simulation, interact with the Entities (switches, hosts, etc.), and get help on many aspects of the simulator. For example, `help(topos.rand)` will show information on the random topology generator, and `help(h1)` will give help on the first host node.

Let's start the simulation (it's initially in a suspended state unless you pass `--start` on the command line) and send a ping between two of the hosts:

```
>>> start()  # Make sure not to forget this!
>>> h1.ping(h2)
```

(If you're using the visualizer, you could also have sent the ping by selecting `h1`, pressing A, selecting `h2`, pressing B, and then pressing P.)

After a short wait, the "ping" packets will start hitting the other hosts; when this happens, a log message is printed. Since we're using a hub, the packet will show up at **all** other hosts—when it shows up at the wrong one, the log message will say so. Hosts, by default, respond to pings by sending back a pong. These will also show up in the console.

As you work on your distance vector router, be sure to launch the simulator with the correct default switch type, i.e., pass in `--default-switch-type=dv_router`, so that your `DVRouter` will be used.

4 Experimenting with Topologies

The simulator comes with a few different topologies and topology generators in the `topos` package. You can modify these, write your own based on them, etc. The point is, your router should work on arbitrary topologies, not just the ones we give you, so you might want to build your own to test on and experiment with! Here, we briefly cover how you can go about this (to gain a deeper understanding, reference the existing topologies' code).

There are two basic ways of creating a topology. The first is programmatically, by creating and linking entities. The second is by creating a `.topo` topology file which is loaded by `topos.loader` (which, internally, just does things the first way).

4.1 Programmatic Topologies

The first step is simply creating some entities so that the simulator can use them. You should **not** create nodes through normal Python instance creation. Instead, use the `.create()` factory on `Entity` like this:

```
>>> MyNodeType.create("myNodeName")
```

That is, you **don't** want to use normal Python object creation like:

```
>>> x = MyNodeType()  # DON'T do this
```

You can use the return value from `create` if you want, though (it returns the new entity). You can also pass extra values into `create`, and they get passed to the constructor (`__init__`), but be careful with this feature, since we will initialize your routers with no additional arguments!

New entities should be globally available, so you can do:

```
>>> MyNodeType.create("myNodeName")
>>> print myNodeName
```

To link this to some other Entity:

```
>>> myNodeName.linkTo(someOtherNode)
```

Optionally, you may pass in a latency for the link to override the default of one second:

```
>>> myNodeName.linkTo(someOtherNode, latency=0.5)
```

You can also `.unlinkTo()` it, or disconnect it from everything:

```
>>> myNodeName.disconnect()
```

You can use this same functionality to experiment with and interactively test your code live in the simulator.

4.2 Topo Files

There's a simple topology file format which is read by `topos.loader`.

The file format consists of lines beginning with `h` (for hosts), `s` (for switches), or `l` (links). Host and switch lines specify a node name. Link lines specify the two nodes to link and optionally a link latency.

You load these by using the `topos.loader` module with the `--filename` option on the command line:

```
$ python simulator.py topos.loader --filename=foo.topo
```

See Figure 1 for an example topology file; check the `topos` directory for more.

5 Building Your Own Tests

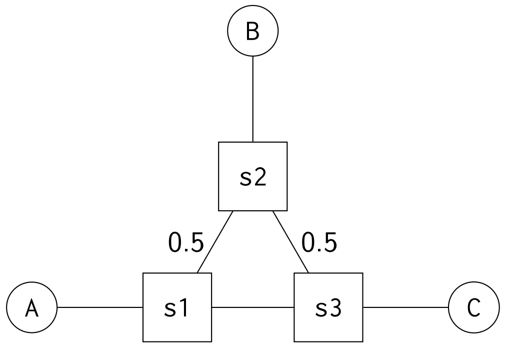
It will probably help to test your code. You can do this interactively using the Python console (using the programmatic topology manipulation features mentioned above) or through the visualizer. Another option is to write yourself some test scripts to automate the testing process. We include a few test modules to get you started in the `examples` directory.

The first of these is `test_simple.py`, which just creates a small topology and sends a couple pings, making sure the right number of pings arrive. Let's try it using the hub:[1]

[1] Note that we use the `--no-interactive` flag to start the simulator without the Python console for the purposes of testing.

```
# Triangle topology.
h A
h B
h C
s s1
s s2
s s3
l A s1
l B s2
l C s3
l s1 s2 0.5
l s2 s3 0.5
l s3 s1
```

(a) A topology file describing the triangle topology. The first line (starting with #) is a comment.



(b) Plot of the topology. Circles denote hosts, squares denote switches, and lines denote links. All links have latency 1 unless otherwise marked.

Figure 1: A triangle topology described in a topology file.

```
$ python simulator.py --no-interactive --default-switch-type=examples.hub \
      examples.test_simple
```

Running this test with the hub should fail because packets show up where they shouldn't. Running it with your distance vector router should pass it!

The `test_failure` test can be run similarly. It sends a packet, fails a link, sends another packet, and expects both packets to arrive at the destination. It'll also fail with the hub, but should pass with your distance vector router with poison mode turned on.

Using the code and design of these tests for inspiration, you can construct your own.

6 Tips

- Packets have colors in the visualizer. "Ping" packets default to random color outsides with white-ish inside. "Pong" packets flip those so you can see which "pong" went with which "ping". Route advertisement packets are magenta. If you want, you can control packet colors using the `inner_color` or `outer_color` attributes. Set them to a list containing `[redness, greenness, blueness, opacity]` where all values are between `0` and `1`.

- Log messages are generally sent to the terminal from which the simulator is run. If you are using the interactive interpreter for debugging or experimentation, this can be somewhat irritating. You can disable these by passing `--no-console-log` on the simulator command line. Note that this needs to be **before** you list any modules, or this will be interpreted as an option for the preceding module (as discussed in the section on the command line above).

- In the visualizer, entities try to position themselves automatically. You can also "pin" them in place by clicking the attached bubble. You can zoom in and out, as well as alter some of the layout forces using options in the Settings panel. The Info panel displays some usage tips.