EE122 Routing Simulator Guide

v0.3 (19w/10/2011)

Running the Simulator

Here's the info you need to get started with the simulator:

File Layout

The simulator is organized thusly:
`simulator.py` - Starts up the simulator.
`sim/core.py` - Inner workings of the simulator. Keep out.
`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/topo.py` - You can use this to create your own topologies/scenarios. See help(topo).
`scenarios/*.py` - Test topologies and scenarios for you to use. See help(scenarios).

Getting Started

To start the simulator, run `logviewer.py` to view console logs, and `simulator.py` to interact with the simulated network:

```
$ python logviewer.py & python simulator.py
```

This pops up a new window, spouts some informational text, and gives you a Python interpreter in the terminal. In the new window, you'll see DEBUG messages output by the hosts and switches in the simulated network. From the interpreter in the terminal, you can run arbitrary Python code, can inspect and manipulate the simulation, and can get help on many aspects of the simulator.

As simulator.py comes, it simply creates a test scenario (mostly this just means a topology) and starts the simulator. You may wish to modify it to load a different scenario (including a custom one). In particular, you can modify the scenario on line 22, and the switches used in the network on line 15. By default, the simulator will load a 'linear' scenario (two hosts connected by a single hub) with a Hub switch that just forwards between the two.

To change the scenario and switches imported by the simulator, find the lines that define "switch" and "scenario" and edit them, e.g, :

```
from learning_switch import LearningSwitch as switch
from rip-router import RIPRouter as switch
import scenarios.linear as scenario
import scenarios.candy as scenario
```

You can see the source code for the Hub switch in `hub.py`, and the source code for the scenario in the `scenarios/` directory.

The scenarios in the scenario directory each contain a create() method. You might want to look at these. They create the topology that goes with the scenario. The included ones can be run with an arbitrary switch class, so you can easily set them up using a hub, a leaning switch, your RIP-like switch, etc.).

From the simulator's commandline, you have access to all the Entities created in your scenario, and you can interact with these. For example:
>>> start() ←- make sure you don't forget this
```
>>> h1.ping(h2)
```

To get started, we suggest you play around with the default scenario and default switches and make sure you understand how to send pings between hosts, add DEBUG messages to the log, and understand the DEBUG messages printed by default to the log viewer.

The Log Viewer

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 editing simulator.py -- towards the top, there is a line:
#_DISABLE_CONSOLE_LOG = True

If you uncomment this line (by removing the '#'), you will no longer see log messages on the terminal. Of course, those log messages can be really quite helpful, and you might want to see them. To rectify this, there is a standalone log viewer -- logviewer.py. Executing this program should open a separate window to display log messages.

Implementing Entities

Objects that exist in the simulator are subclasses of the Entity superclass (in api.py). These have a handful of utility functions, as well as some empty functions that are called when various events occur. You probably want to handle at least some of these events! For more help try `help(api.Entity)` within the simulator. You might also want to peek at hub.py, which is a simple Entity to yet you started.

Sending Packets

Entities can send and receive packets. In the simulator, this means a subclass of Packet. See `basics.Ping` for one such example, and see `basics.BasicHost` to see an example of how it is used. In the default simulation (a linear topology with a hub), try h1.ping(h2) to start.

Building Your Own Scenarios

You may want to test using your own topologies, and you may want to test your own events (such as nodes joining and leaving the network) within those topologies. We refer to the combination of those as a "scenario". The simulator should come with some (in the scenarios directory) to get you started, but you may want to build your own. To start:

```
>>> import sim.topo as topo
```

The first step is simply creating some Entities so that the simulator can use them. You shouldn't create the nodes yourself, but rather let the create() function in core do it for you, something along the lines of:

```
>>> MyNodeType.create('myNewNode', MyNodeType, arg1, arg2)
```

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

```
>>> x = MyNodeType(arg1, arg2) # Don't do this
```

`create()` returns the new entity, and all entities that you create are added to the sim package. So you can do:

```
>>> import sim
>>> x = MyNodeType.create('myNewNode', arg1, arg2)
>>> print sim.myNewNode, x
```

.. which will show the new Entity twice.

To link this to some other Entity:

```
>>> topo.link(sim.myNewNode, sim.someOtherNode)
```

You can also unlink it, or disconnect it from everything:

```
>>> topo.disconnect(sim.myNewNode)
```

To see the connections on a given Entity:

```
>>> topo.show_ports(sim.someNode)
```

.. this shows how the ports on `someNode` are connected to other nodes and their ports.

Your Components

As discussed in the Project 2 writeup, you will be responsible for implementing two subclasses of the Entity superclass. Some important functions from Entity are as follows (feel free to poke around in the source, but knowing these should be sufficient to implement your subclass):

```
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.
```

```
|
|  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.
```

You will also interact with the Packet class:

```
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 through.
    |
    | self.trace
    | A list of every Entity that has handled the packet previously. This is
    | here to help you debug.
```

Visualization

Finally, the simulator includes a graphical user interface. THIS IS MOSTLY UNSUPPORTED. But if you want to play with it, check it out:

NetVis is a visualization tool for the network simulator. It is written using Processing, which is basically framework for writing visualization tools in Java (with some nice shortcuts).

You should have received NetVis executables for Linux, Mac OS, and Windows. Provided you have Java installed (which should always be the case on Macs, but not necessarily on the other two), these will hopefully all just run.

If you have trouble running the GUI, the most likely explanation is that you don't have Java installed. You might try installing it. Alternately, you could install Processing (a straightforward download from processing.org) and copy the NetVis workbook folder into your Processing projects folder.

Entities, links, and packets in your simulator should show up automatically in NetVis. Generally these 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 altering some of the layout forces using options in the "adjustments" panel (click on it to open it).