Lab 1: Avoiding Loops when Flooding

Due: September 23 at 12:01 AM

Early Submission: September 16 at 12:01 AM

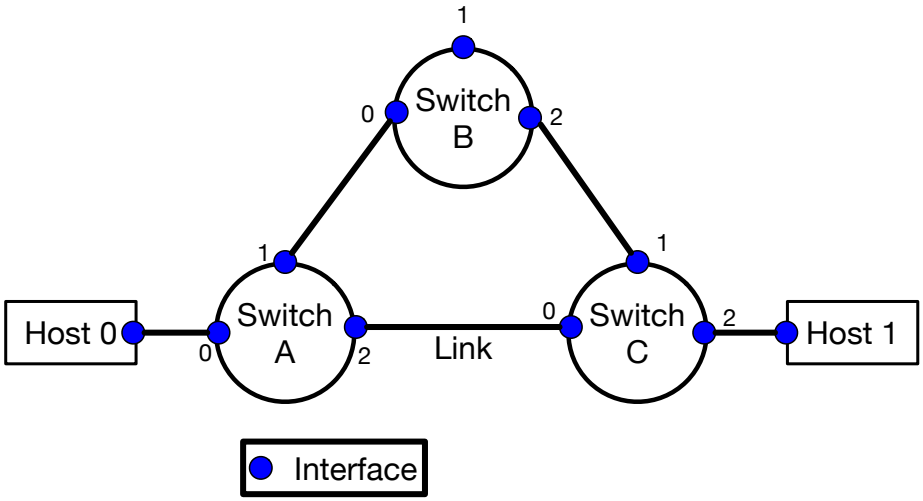

Figure 1: A simple network topology

In this lab you need to develop a **protocol** that allows switches to **forward** packets from one host to another using flooding even when the network has loops. You will be implementing this protocol in a Python based simulator we have provided you for this purpose.

Getting the Code

To acquire the simulator and lab infrastructure go to https://classroom.github.com/a/Q-DvzbX_, and accept the assignment. Doing so will create a repository for you to work on your project. Once done clone the repository using `git`, go into the cloned repository and then execute:

If on Mac OS X or Linux

```
>  ./setup.sh       > . bin/activate
```

If on Windows:

```
> setup       > Scripts\activate
```

In either case when you are done working on the project you can run `deactivate` to leave the project environment.

You only need to run `setup` **once**, the next time you need to work on the project just run `. bin/activate` (or `Scripts\activate`).

While reading through the rest of this document you might find it convinient to have the simulator documentation and `lab1.py` open.

Simulator Mechanics

The simulator we have provided simulates networks consiting of a set of simple switches and hosts. Both switches and hosts are assigned **unique names** (IDs) and consist of a set of interfaces.

> For now a packet is just a message (i.e., some set of bytes) with some metadata attached. In the case data packets (explained below) this metadata indicates the sender and the receiver.

The switches in our simulator make a distinction betweeen **data** and **control packets**:

- A *data packet* is one that has been sent out by a host and is destined to another host. When a switch receives a data packet it **floods* it, i.e., sends it out all interfaces except for the one on which it was received. This is the same mechanism that we discussed in Lecture 1 and 2.
- A *control packet* is one that has been sent by a switch in order to implement a protocol. When a switch receives a control packet it calls the `process_control_packet(self, sw: sim.SwitchRep, iface_id: int, data: Any)` function defined on line 37 of lab1.py. At this point you can choose to handle this packet as you desire.

Each switch consists of several interfaces, for example in the figure above Switch A, B and C have three interfaces each. An interface might not be connected to anything (e.g., Interface 1 on Switch B), or connected to a link attached to an interface on another switch or host (e.g., Interface 2 on Switch A is connected to Interface 1 on Switch B).

Any interface on a switch can be set to be either **UP** or **DOWN**. When an interface is *DOWN* the switch will neither receive any data packets sent on the link connected to that interface, nor will it *send* any data packets out that interface. In particular this means that no packets will be flooded out interfaces that are set DOWN. This is similar to adding interfaces to the deactivated set from lecture 3.

All interfaces are initially set as UP, i.e., at the beginning of the simulation all interfaces are set up.

Finally, when initializing a switch the simulator will call the `initialize` function on Line 31 of lab1.py. It is **important** that you send out some **control** packets in this call, since otherwise the simulator does not automatically send out any control packet. You can send out control packets by calling `sw.send_control(iface,`

`packet)` as shown in line 28 of lab1.py. `sw` in this case is a `SwitchRep` object, which you can read more about in the documentation.

The Problem: Working Around Loops

Consider what happens when **Host 0** in the figure above sends a **data** packet addressed to **Host 1** in the topology above:

- First, Switch A receives the packet on Interface 0, and send it out Interface 1 and 2. Recall, that flooding a packet involves sending it out all ports **except** for the one on which it was received.
- Next two copies of the packet are received at two switches:
  - Switch B receives the packet on interface 0 and floods it by sending out a copy on interfaces 1 and 2.
  - Switch C similarly receives a copy on interface 0 and floods it out interface 1 and 2.
- Next we have three copies of the packet in play in the network:
  - Host 1 receives a copy of the packet from Switch C and is happy to have received the message sent by Host 0.
  - Switch B receives a copy on Interface 2, which it floods by sending copies on Interface 0 and 1.
  - Switch C receives a copy on interface 1 which it floods by sending copies on interface 0 and 2.
- Next we again have 3 copies of the packet in play:
  - Host 1 receives another copy of the packet from switch C and is confused about why.
  - Switch A receives a copy on Interface 1 that it floods by sending copies on interface 0 and 2.
  - Switch A also receives a copy on interface 2 that it floods by sending copies on interface 0 and 1.
- Next we have 4 copies of the packet:
  - Host 0 receives **two** copies of the packet and is not sure why.
  - Switch B receives a copy on interface 0, which it floods out interface 1 and 2.
  - Switch C receives a copy on interface 0 which it floods out interface 1 and 2
- This goes on forever.

As we can see the current network allows packets from Host 0 to be received at Host 1. In fact it is so successful at delivering packets that Host 1 will receive an unbounded number of copies of this packet. However, once a packet enter the network it will keep being forwarded forever and will never exit the system.

In this **lab** you need to design a protocol that ensures that:

- Packets from any host can reach any other host.

- Packets eventually leave the network, i.e., packets cannot loop around forever.

You **should not** handle failures in this lab.

Some Thoughts on Developing a Protocol

High Level

The simulator mechanics (which we discussed above) limit the amount of control you have over how **data** packets are forwarded. The only way you can impact the forwarding of data packets is by **setting** interfaces down or up.

Setting an interface down is **equivalent** to removing a link, since no data packets are **sent** or **received** on an interface that is down. This means that the protocol you need to design is one that takes a graph with loops and removes a set of links so as to produce a graph **without loops**.

It is important however to ensure that in producing a graph without loops you do not disconnect portions of the graph: for instance, in the example topology, it might appear appealing to set Interface 1 on Switch A and B down, thus eliminating the loop. However this would disconnect Switch B, which might not be a good idea in an alternate case where some host (Host 3) is connected to Switch B.

We suggest you start by sketching out what an algorithm for producing a loop free graph from a loopy graph might look like when you *have* access to the entire graph. You should then see what information this algorithm requires and use it to produce a distributed protocol.

Low Level

Some low-level hints about the code:

- You might want to start by figuring out what information you need to include in the `Data` class defined on Line 9 of lab1.py.
- Each switch has an ID, which you can access by calling `sw.id` from `process_control_packet` or `initialize`.
  - We guarantee that IDs are unique strings.
  - As a result we also guarantee that all IDs in the simulation have a total order. What this means is that if switch 1 has ID `id1`, switch 2 has ID `id2`, and switch 1 and 2 are not the same switch, then either `id1 < id2` or `id2 < id1`.
  - This in turn means you can order all of the nodes in the simulation, ordering nodes in this manner is essential to successfully completing this assignment.

- You can set interfaces up or down from either the `initialize` or `process_control_packet` functions in lab1.py.
  - `sw.iface_down(id)` will set interface `id` down.
  - `sw.iface_up(id)` will set interface `id` up.

Some Additional Constraints and Getting Started

It is very important to ensure that the protocol you develop **terminates**, i.e., you need to ensure that arriving at a loop free graph takes you *no more* than a finite number of messages. The simulator will error out if this is not the case.

> If you are on Windows please substitute all occurences of `python3` with `python`, this is an annoyance because of recent upgrades to Python.

You can test your code by running:

```
> python3 lab1.py topos/4-clique.yml
> python3 lab1.py topos/4-loop.yml
> python3 lab1.py topos/dumbell-loop.yml
```

If run on the inital code stencil or using a bad protocol this should result in an assertion failure in `setup.check_algorithm()` (line 64 in the stencil code). `check_algorithm` is a function that checks whether your protocol has produced a graph where all hosts are connected and the graph is loop free.

Generally we believe that if you can successfully pass this check using the three topologies we have included then your code is probably correct. We however encourage you to develop your own topologies. We provide information on this below.

Debugging Problems

We have provided a set of Jupyter based interactive tools that you can use to debug problems with this assignment. In addition the `Tracer` class (described in the documentation) also provides text based debugging support. To use the Jupyter based debugging run the following from the project repository after you have activated the virtual environment:

```
> jupyter notebook
```

This should open a browser window with a file browser. In the browser window go an click on `Debugging.ipynb`. You should now be able to reexecute all cells in this notebook. The cell with `draw_connectivity_over_time` represents the set of links that are active (i.e., links where both interfaces connected to the link are up) as you process different control messages. You should aim to remove any loops in this graph. The cell with `draw_active_links_over_time` represents

the same information, except that it colors links red rather than removing, thus making visual comparison easier at times.

We draw these graphs using Matplotlib and NetworkX, however I (Panda) am not very good at figuring out visualizations of this form. If you have an alternative approach that looks better, we would greatly appreciate your contributions to this.

Tooling Help

IDEs such as PyCharm, VSCode, etc. contain support for developing and testing software run in virtual environments. While we neither require nor recommend the use of these tools, if you are already using them, this support might be helpful.

The virtual environment within which you are working on the project includes two tools that you might find useful in checking the correctness of your code:

- The first, `mypy` can be run by calling: `> mypy lab1.py` `mypy` is a Python linter that makes use of type annotations to check the correctness of your code. We have annotated all of the functions in the simulator, and strongly recommend you do the same. This document provides a good overview of how to annotate your code, and the mypy documentation can be of futher help. Linting and type-checking your code are really helpful in avoiding many common problems which might otherwise require extensive testing to uncover.
- The second, `black` can be run by calling `> black lab1.py` Black formats your code based on an opinionated standard that results in readable code. Using black to format your code will make it easier for any one else who is trying to read and understand your code, and might even allow you to debug faster.

Both `mypy` and `black` have plugins that allow them to be incorporated with a vareity of IDEs, including the ones we mentioned above, `vim` and `emacs`.

The course staff will require that you use both of these tools before asking us for help, since doing so would allow us to avoid spending time figuring out errors and issues that can be automatically detected.

Submitting the Project

You can submit the project by pushing your code to the Github repository that you created in the Github classroom link. As with all labs in this class submitting a correct version one week after the lab goes out (i.e., by September 16) gets you 10% extra credit. If you chose to **submit early** please fill out this form. Note, you need not fill this form if you are submitting normally.

Appendix: Topology Format

If you look at `topos/4-loop.yml`, you will find a file that looks like:

```
hosts:
    - h0
    - h1
switches:
    - s0
    - s1
    - s2
    - s3
nifaces: 5
edges:
    - [s0, s1]
    - [s1, s2]
    - [s2, s3]
    - [s3, s0]
    - [h0, s0]
    - [h1, s3]
```

This is a standard YAML file, with the following schema:

- `hosts` is a list of named hosts, in this case we have two `h0` and `h1`.
- `switches` is a list of switch names, in this case we have 4 `s0` through `s3`.
- `nifaces` is the number of interfaces each switch should contain. In this case we set each switch to have 5 interfaces.
- `edges` is a list of pairs of nodes between which a link should be created. In general we assume that the network is **multigraph** and thus allow multiple links between a pairs of nodes.