INFO8011: Network Infrastructures
Software-Defined Networking

C. Soldani

1 Introduction

In this assignment, you will extend your learning switch obtained during the OpenFlow Tutorial to implement the control logic for a simplified Clos network.

1.1 What is a Clos network?

When you build a data-center network, a conventional topology is a hierarchical tree. A number of compute nodes are co-located inside a rack, and connected through a top-of-rack switch. This switch also has one uplink connection towards an aggregation switch, connecting several racks. The aggregation switches themselves have uplink connections to core switches, and so on.

Such a topology is known as a fat-tree. It allows for an efficient communication, but it requires bigger and bigger capacity links (and switches) as we go up in the tree. Even if you try to co-locate servers as best as possible so that they will be close together in the data-center, you will still have communication between racks and to the outside internet. The number of flows going through links will increase as you go up in the tree, requiring higher capacity links and switches.

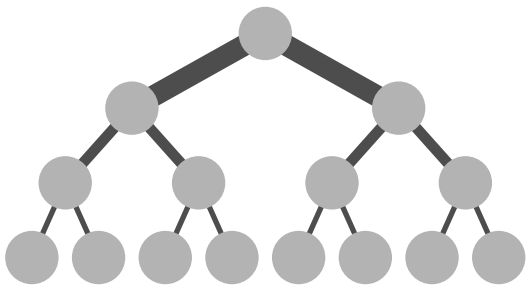

Figure 1 – The traditional fat-tree.

The problem with the fat-tree approach is that high-capacity switches and links are very expensive. You can generally buy several low capacity switches for the price of one high-capacity switch. Also keep in mind that you need to duplicate those high-capacity switches for redundancy, so that the data-center connectivity can be maintained if one switch fails. However, those additional switches are generally not used, leading to a waste of expensive resources.

To alleviate those problems, one can ditch the traditional tree-based topology in favour of a Clos network. A Clos network is a multistage switching network, first formalized by Charles Clos in 1952 in the context of circuit switching for phone communications. In a Clos network, we use a larger number of switches than in an equivalent fat-tree network, but all switches are identical *commodity* (*i.e.* cheap, readily available) switches, for an overall decreased cost. Moreover, we naturally obtain redundancy in the network, which diminishes the need for additional, generally unused switches.

Clos networks can also be used in packet-based data-centers, which is what was proposed by Al-Fares *et al* in their paper A Scalable, Commodity Data Center Network Architecture (PDF). Such a network is illustrated in figure 2.

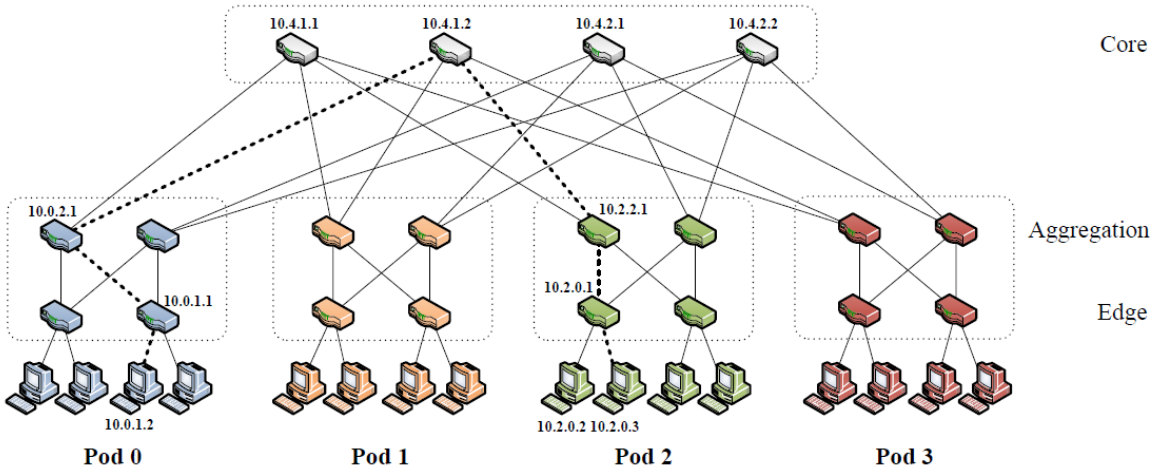

Figure 2 – Al-Fares *et al* topology.

Note that the network is no more a tree (it can contain cycles), and that uplink bandwidth is now split between multiple low-capacity links. We advise you to read that paper. Note that, confusingly, the authors are also calling their Clos-like network a *fat-tree*.

1.2 A simplified Clos-like topology

In this assignment, you will implement the control logic for a simplified Clos-like network. The topology will consist of a layer of *core* switches, connected to a layer of *edge* switches, which are connected to a set of *hosts* (figure 3).

2 Implementing three control policies

You will experiment three different control policies for our simplified Clos-like network.

2.1 A simple tree

A first way to handle such a topology in your SDN controller is to build a *spanning tree* for it, in order to avoid cycles. You will not need to support the actual *spanning tree protocol*, but you will implement a similar logic in your controller, by avoiding the use of some links and switches. For the example in figure 3, the topology would look like the one in figure 4.

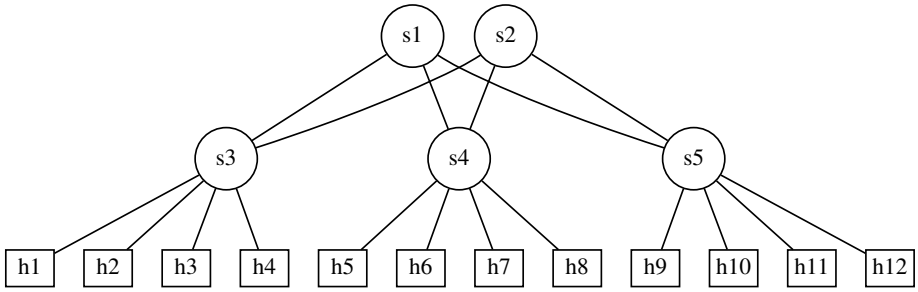

Figure 3 – Simplified Clos-like topology with two core switches `s1` and `s2`, four edge switches `s3` to `s5`, and four hosts per edge switch.

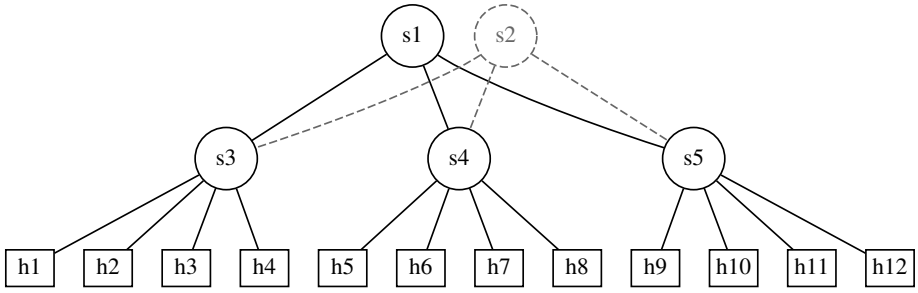

Figure 4 – The spanning-tree control policy ignores some links and switches.

The simple tree approach is not very efficient, because:

- edge switches have a single up link instead of several, which can cause congestion;
- only one of the core switches will be actually used.

2.2 Using VLANs

To improve the situation, assume that each host belongs to a single tenant, and is only allowed to communicate with other hosts belonging to that tenant. You can group hosts of a tenant in a VLAN. You can then use different trees for different VLANs, leading to a better utilisation of links and switches.

In our running example, assuming the host VLANs are given by their color in figure 5, we could use two trees:

- One tree rooted at `s1` for the red and green VLANs.
- One tree rooted at `s2` for the cyan and magenta VLANs.

This approach is more efficient, but is not optimal because some host pairs might communicate much more than others. In our example, if red and green machines exchange data heavily while cyan and magenta ones rarely communicate, `s1` links will be congested while `s2` links have plenty of capacity left. In practice, this could be alleviated by measuring average link use by VLAN, and reallocating VLANs to core switches from time to time, but it is still not very flexible (bandwidth requirements will vary with time).

This approach is also more restrictive, as hosts can only communicate with hosts from the same

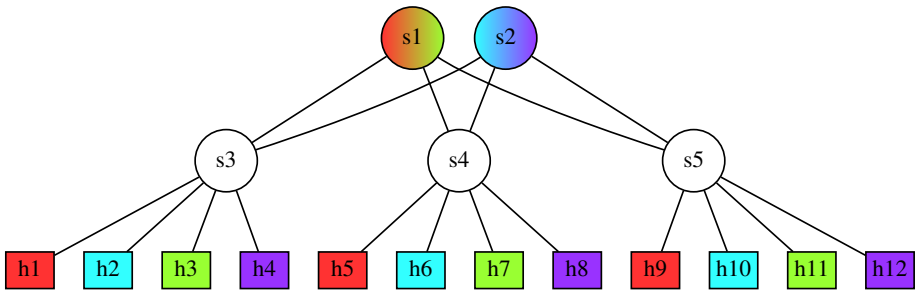

Figure 5 – Core switch allocation according to hosts VLANs.

VLAN. However, it must not necessarily be the case. Rather than using VLANs, you could simply partition hosts into disjoint sets, and then choose the uplink core switch according to the source or destination host. For example, a flow going from `h2` to `h5` would be sent on link `s3` – `s2` according to the source color (cyan), while the response packets going from `h5` to `h2` would be sent through the link `s4` – `s1` (as the source `h5` is red).

2.3 Adaptive routing

For this policy, you will adaptively send new flows to the less loaded core router. You will first need to monitor switches using `PortStatsRequest` messages. Then, when you see a new flow (here, you will also consider protocol and source/destination ports, not just MAC addresses) in an edge switch, you will forward it to the core switch with which the up link is the less loaded.

In our example topology (figure 3), let's say that `h1` initiates a new HTTP connexion towards `h7` while last measurements indicate that `s3` – `s1` is at 6 Mbps while `s3` – `s2` is at 1.5 Mbps. `s3` would choose the less loaded link `s3` – `s2` (and thus core switch `s2`) to forward the packet, and install a new rule for that flow.

3 Practical details

3.1 Implementation of the policies

You will implement aforementioned three policies using the POX controller as installed in the updated mininet VM provided on eCampus (in lab 2).

Your controllers should be named:

- `tree.py` for the tree-based policy;
- `vlans.py` for the VLAN-based policy;
- `adaptive.py` for the adaptive policy.

You can use additional python files if needed.

For the VLAN-based policy, provide a file `tenants.py`. This python module must exports a dictionary `vlans` mapping the Ethernet address of hosts (with type `EtherAddr`, as provided by `packet.src` in the tutorial) to their VLAN. The module must also export a variable

`n_vlans` indicating the number of different VLANs. The user is in charge of modifying this file according to his topology and VLANs assignments.

All your code will be run from the `pox/pox/misc` folder on the VM.

3.2 How to test your solutions

To help you test and compare your policies, you can use provided files to create the example topology in mininet, and do bandwidth measurements between various pairs of hosts:

- `clostopo.py` file is a class that builds a simplified Clos-like topology.
- `test.py` is a mininet script that you can run (with `sudo`) to build the example topology and run the measurements.
- `client.py` is a flow generator that is run on client nodes by `test.py`.

Note that your policies should work not only for the example topology, but for any topology which is generated by `ClosTopo`. You can test alternate topologies by modifying the `test.py` script, or by launching mininet CLI using the `ClosTopo`:

```
sudo mn --custom ~/clos-test/clostopo.py --mac --link tc \
    --topo clostopo,nCore=3,nEdge=4,nHosts=3,bw=10 \
    --switch ovsk --controller remote
```

3.3 How to discover the topology

Ideally, your SDN controller should be able to discover the topology, and react to live topology changes (*e.g.* a switch or links that fails). pox comes with built-in topology and discovery modules (see `openflow.discovery`), which can help you monitor the topology entirely from pox.

You can however assume that the general shape of the topology will be the one of our simplified Clos-like network, *i.e.* each host is connected to a single edge switch and edge switches are themselves connected to one or more core switches.

Alternatively, you might re-use the `ClosTopo` class in your pox controller, and initialize it with the same arguments (number of core and edge switches, number of hosts per edge switch) than when launching mininet (pox will pass command-line arguments to your controller launch function). You will likely need to extend the `ClosTopo` class to expose the information needed for your controller.

4 Evaluation

Assignment evaluation will be based on the following criteria:

- **Functionality**. Your three controllers must behave according to the corresponding policy. They should work for any topology generated from `ClosTopo`, not just the one in the examples. You will obtain a bonus if your controllers correctly react to live topology changes.

- **Coding style**. This assignment is not given in a programming course context. However, at this step of your studies, we assume you are able to provide elegant solutions to complex problems. The coding style (and, consequently, your solution elegance) will be part of the grading.
- **Documentation**. In the fashion of other popular programming languages, Python comes with a powerful tool for documenting code: `Docstring`.[1] We ask you to fully document your code with respect to Docstring standards.
- **Report**. You must write a short report[2] describing:
  - the general overview of your three solutions;
  - anything special you had to do to fulfil the requirements, which is not obvious from **documented** code;
  - how you tested your policies, and what you observed;
  - your feedback about this assignment:
    * Where were the main difficulties?
    * How many hours of work did it take (per student)?
    * What would you suggest to improve this assignment?

5 Submission

The submission of your lab is subject to the following rules:

1. an archive named **Team_xx.zip** (where **xx** must be replaced by your TeamID), which contains all required files,must be uploaded on the submission platform (see https://submit.montefiore.ulg.ac.be)
2. the deadline is **November, 29th, 08h00** (hard deadline).

[1] See, for instance, https://www.datacamp.com/community/tutorials/docstrings-python
[2] See LaTeXtemplate on eCampus.