



CS344

Build an Internet Router

[About](#) [Documentation](#) [Policy](#) [Schedule](#) [Source Code](#) [Staff](#)
[Teams](#) [Piazza](#) [Lectures](#)

Starter Code Overview

We have provided you with some starter code in a P4->NetFPGA project directory called `simple_router`. The directory structure is as follows:

```
simple_router/
|- Makefile
|- src/: P4 source files and static table entries used for data-plane s
|- sw/: Control-plane source files, control-plane simulation files, P4-
|- testdata/: source files for data-plane simulations
|- simple_sume_switch/
    |- test/: Full NetFPGA simulation files
    |- hw/: Top level HDL source files
    |- sw/: Software to read all SUME control regs
    |- bitfiles/: directory to store bitfiles
```

Data-Plane Starter Code

The P4 data-plane starter code is in the `src` folder.

```
src/
|- simple_router.p4 : main P4 source file
```

```
| - commands.txt : set of CLI commands to add entries to the tables spec
| - Makefile : compiles P4 program into intermediate representation for
```

simple_router.p4

The first file you will want to familiarize yourself with is called `simple_router.p4`. It contains a very simple router design with a single table that simply matches on the packet's source port and either invokes an action called `set_output_port` or `NoAction` depending on the configured table entries. Obviously this not how an internet router works so your job is as follows:

- Define additional headers
- Extend parser to parse your additional headers
- Define additional tables and actions
- Implement match-action control-flow
- Extend deparser to emit your additional headers

We will use a ternary match table for the routing table for this project rather than a longest prefix match table (LPM) because SDNet does not fully support LPM tables at the moment. We have provided some wrapper functions in the control-plane starter code for you to try and expose LPM-like functionality on top of the ternary table. Other than higher resource utilization on the FPGA, the main difference between the ternary match and LPM tables is that the control-plane must explicitly manage the priority of each entry. The priority is indicated by the entry's address, smaller addresses indicate higher priority.

The `digest_data` struct is a metadata bus that will be prepended to the packet whenever it is forwarded to the control-plane. The exact format of this struct can be defined by the P4 programmer. The starter code defines the `digest_data` struct as follows:

```
struct digest_data_t {
    bit<240>  unused;
    bit<8>    digest_code;
    bit<8>    src_port;
}
```

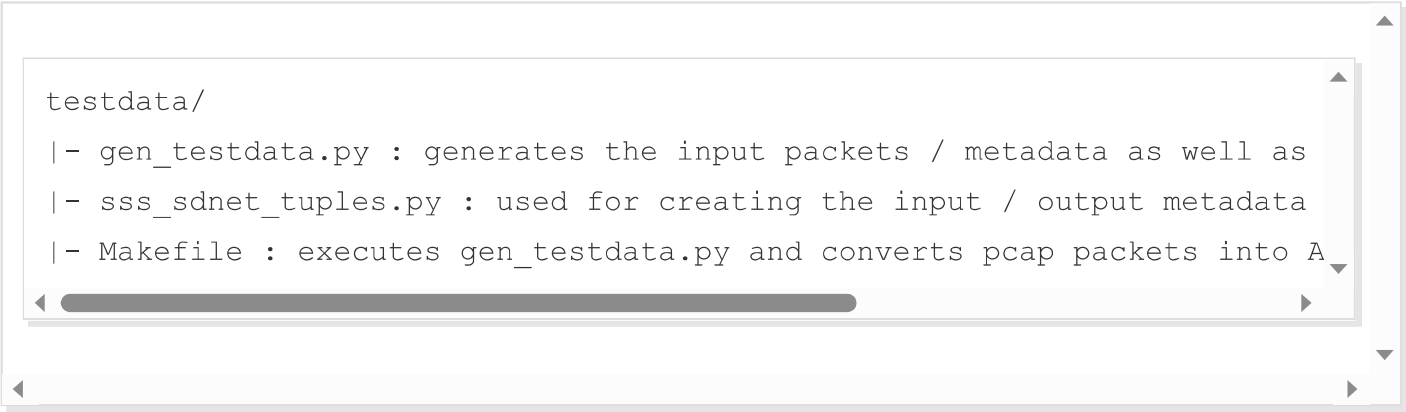
We recommend that you use this format, but you do not have to if you prefer to use something different. Keep in mind that this struct must be 256 bits wide and if you change the format then you will need to update the `sss_sdnet_tuples.py` file in the `testdata` directory, as well as the `sss_digest_header.py` file in the `sw/simple_router_sw/headers` directory.

commands.txt

This file may contain `table_cam_add_entry` or `table_tcam_add_entry` commands. See the [workflow overview](#) on the P4->NetFPGA wiki page. After compiling your P4 program and loading the bitstream onto the FPGA, this file can be `cat`ed into the

`P4_SWITCH_CLI.py` file to load table entries. e.g. `$ cat commands.txt | ${P4_PROJECT_DIR}/sw/CLI/P4_SWITCH_CLI.py`.

The main data-plane simulation tools are in the `testdata` directory.



```
testdata/  
|- gen_testdata.py : generates the input packets / metadata as well as  
|- sss_sdnet_tuples.py : used for creating the input / output metadata  
|- Makefile : executes gen_testdata.py and converts pcap packets into A
```

gen_testdata.py

This file generates the testdata used for both the SDNet and the full SUME simulation. See the [workflow overview](#) for more details on what it should produce.

We have provided a few helper functions in the `gen_testdata.py` script to help you get going. See the starter code for a description of their functionality. We have also provided you with a few baseline test cases at the bottom of the file. You will need to add more to thoroughly test your design.

You will need to update your `commands.txt` file to reflect the topology described under “Data-Plane Baseline Tests” [here](#).

sss_sdnet_tuples.py

This file defines the format of the `digest_data`, which should have an equivalent definition in the P4 program. It also imports the standard `sume_metadata`. These metadata buses are stored as python `OrderedDict()` objects. This file contains helper

functions to write these metadata buses into the `Tuple_*.txt` files in the format expected by the SDNet simulation. If you choose not to modify the format of the `digest_data` bus then you shouldn't need to make any modifications to this file.

Control-Plane Starter Code

The control-plane starter code is in the `sw/simple_router_sw` directory which has the following format:


```
sw/simple_router_sw/
|- control_plane.py
|- arp_cache.py
|- PWOSPF_handler.py
|- cp_config.py
|- hw_tables_api.py
|- routing_consts.py
|- headers/
    |- sss_digest_header.py
    |- PWOSPF_headers.py
|- simulation/
    |- sanity_check.py
    |- sim_tables_api.py
    |- topology.py
    |- test_topology.py
    |- topology_visualizer.py
|- tests/
|- utils/
    |- LPM_dict.py
    |- addr_conversions.py
```

control_plane.py

This file will contain your control plane implementation. You should update `handle_packet` to perform the appropriate actions on the packet received. You may want to create additional classes or files to handle particular aspects of the control plane's responsibilities.

To start the control plane, run `python control_plane.py`. By default, PWOSPF is disabled. To enable PWOSPF, use the `--pwospf` argument. To use the topology visualization (described below), run with `--visualize`. To configure the router with a particular topology that is defined in a json file, use `--config [filename]`.

An example config file might look like the following json file:



```
{
  "routing_table": [
    {"subnet": "12.12.12.0", "netmask": "255.255.255.0", "gw": "0.0.0.0"},
    {"subnet": "1.1.1.0", "netmask": "255.255.255.0", "gw": "0.0.0.0"}
  ],
  "interfaces": {
    "nf1": {"ip": "12.12.12.13", "netmask": "255.255.255.0", "mac": "08:00:27:00:00:00"},
    "nf0": {"ip": "1.1.1.1", "netmask": "255.255.255.0", "mac": "08:00:27:00:00:00"}
  }
}
```

To use the topology visualization you will need to implement the `topology` method, which should return your router's view of the current topology.

To configure the router from a file, you will need to implement `load_tables`, which should populate the tables given the provided interfaces and routing table. You will also need to update `parse_config_file` to use the correct action name and action data.

arp_cache.py

A skeleton class for implementing the ARP cache.

PWOSPF_handler.py

A skeleton class for implementing PWOSPF.

routing_consts.py

A collection of constants used in routing, such as ICMP types and the intervals on which to send PWOSPF updates. You may want to modify the digest codes.

cp_config.py

Provides the tuple used to configure the control plane, which includes the function used to send packets to the data plane, the api with which to update tables, and a list of the router's interfaces.

hw_tables_api.py

Wrapper class for `p4_tables_api`. It serves two primary purposes:

1. to convert between human-readable IP and MAC addresses and integers when values are read and written from the tables, and
2. to purpose a ternary match table as an lpm table because there are currently some compiler issues with lpm tables.

LPM_dict.py

A dictionary that uses longest prefix match to perform lookups. This is used for the routing table, and there is a helper in `hw_tables_api` to dump an `LPM_dict` into a p4 table.

addr_conversions.py

Provides conversions between IP/MAC addresses and integers, as well as conversions between masks and prefix lengths (i.e. `255.255.255.254 <-> 31`).

sss_digest_header.py

Contains the scapy representation of the header that is prepended to packets when they are forwarded to the control plane from the data plane. You will want to modify this if you choose to add or remove fields from `digest_data`.

PWOSPF_headers.py

Scapy objects for the PWOSPF, LSU, LSU Advertisement, and HELLO headers. The PWOSPF checksum, PWOSPF length, and LSU advertisement count fields will be calculated automatically. The PWOSPF protocol field will be set automatically if the data of the PWOSPF header is an LSU or HELLO packet. You should not need to modify this file.

sanity_check.py

Provides a set of baseline tests for the functionality of your control plane, both with and without PWOSPF enabled. There is a `TIMEOUT_TESTS` property that can be set to `False` if you would like to disable the tests that cause delays, such as the arp cache timeout test. These tests are not comprehensive, and you are encouraged to add your own.

In order for the tests to run, your control plane must have a `handle_packet` function and take in `config`, `routing_table`, and `pwospf_enabled` initialization parameters. At minimum, your `config` should allow the tables api, send function, and interfaces to be configured. (Our implementation also allows timeouts to be configurable so that timeout tests can run faster.) You also must call your arp cache table `arp_cache_table` in order to run the tests. We will use these tests to check basic functionality of your control-plane at the first milestone. You may make edits to this file, but do not fundamentally alter the essence of the functionality they are intended to test.

It is possible that, although a new control plane is instantiated for each test, your state will not be fully cleared before the start of each test. If you encounter this issue, simply clear state in `setUp`.

sim_tables_api.py

A python implementation of a subset of the p4 tables api. This is used in `sanity_check`, and you may find it useful in testing.

topology.py

A simple class that represents a topology as list of router ids, the id of the current router, and a list of links - where a link is represented by a subnet, prefix length, and list of ids of the routers that are connected to the link.

This is used in the topology visualization. If you don't use the visualization, then this class will likely not be useful for you. If you would like to use the topology visualization, your control plane must have a `topology()` method that converts from the format of your topology database to a `Topology` object.

test_topology.py

Contains a topology that can be used for testing. This is used in sanity check and in the topology visualizer. You should modify `ROUTING_TABLE` to reflect that format of your p4 routing table.

topology_visualizer.py

The Topology Visualizer provides a visual representation of your router's current view of the network topology. This can be run as a standalone program to test your control plane's PWOSPF handling, or can be run with the control plane by using the `--visualize` parameter. You will need to run the visualization from inside of VNC Viewer. Running `topology_visualizer.py` allows you to send hello and lsu packets from routers in the test topology to your router.

In order to use the visualizer, your control plane must have a `topology` method that returns a `Topology` object, and a `handle_pkt` method that handles incoming packets.

tests

The `tests` directory contains tests for the included utils and headers. You shouldn't need to modify anything in this directory, though you may want to add tests for your code.
