Project 1: Building a Simple Software Defined Network (SDN)

1 Overview

In contrast to traditional computer networks which employ distributed routing algorithms, there is growing interest in a new way of designing networks, which is referred to as Software Defined Networks (SDN). An SDN network consists of multiple SDN switches and a centralized SDN controller. Unlike traditional networks, SDN switches do not run distributed protocols for route computation. Instead, the SDN controller keeps track of the entire network topology, and all routing decisions (path computations) are made in a centralized fashion. Routing tables computed by the controller are then shipped to the switches.

In this project, you will implement a highly simplified SDN system comprising of a set of switches and a controller implemented as user level python processes. The processes mimic the switches and controller, bind to distinct UDP ports, and communicate using UDP sockets. This project will (i) expose you to the concept of an SDN network; (ii) improve your understanding of routing algorithms; and (iii) give you experience with socket programming.

A key part of the problem is ensuring the controller has an up-to-date view of the topology despite node and link failures. To detect switch and link failures, periodic messages are exchanged between the switches, as well as between each switch and the controller as will be described in this document. The controller runs algorithms for path computation each time that the topology changes and ships the latest routing table to each switch.

The routing algorithm used by the controller to compute paths is **Dijkstra's shortest path algorithm**. Specifically, of all possible paths between the source and destination, the path with the shortest "distance" is chosen. The distance of a path is the sum of all the links' distances on the path.

2 Description

The switches and the controller emulate a topology specified in a topology configuration file. We begin by describing the format of the configuration file, next discuss the bootstrap process (the action performed by each switch process at the start of execution), path computation (how the controller computes routing tables), and the periodic actions that must be continually performed by the switch and the controller.

3 Topology Configuration File

When contacted by a switch, the Controller will assign neighbors based on information present in a configuration file. The configuration file must follow the following format:

- The first line of each file is the number of switches
- Every subsequent line is of the form:

  ```
  <switch-ID 1> <switch-ID 2> <Distance>
  ```

  This indicates that there is a bidirectional link connecting switch 1 and switch 2, with the specified distance.

Consider the switch topology given in Figure 1. Please note that the switch ID starts from 0 instead of 1.

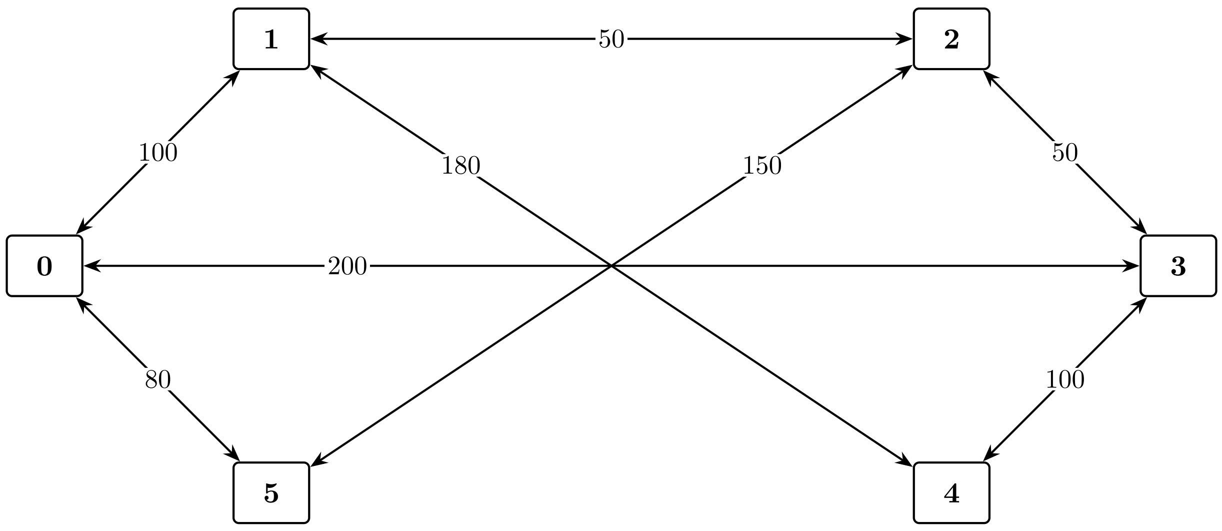

Figure 1: 6 switch topology. Each link is marked with its distance.

Here is an example configuration for the topology shown.

```
6
0 1 100
0 3 200
0 5 80
1 2 50
1 4 180
2 3 50
2 5 150
3 4 100
```

4 Bootstrap Process

The bootstrap process refers to how switches register with the Controller and learn about each other.

Note: *The bootstrap process must also enable the Controller process and each of the switch processes to learn each other's host/port information (hostname and UDP port number information), so communication using socket programming is feasible.*

1. The Controller and switch processes are provided with information using command line arguments (we require that the command line arguments follow a required format discussed in Section 9)
   (a) The Controller process binds to a well-known port number.
   (b) Each switch process is provided with its own ID, as well as the hostname and port number associated with the Controller process, as command line arguments.
2. When a switch (for instance, with ID = 4) joins the system, it contacts the Controller with a `Register Request`, along with its ID. The controller learns the host/port information of the switch from this message.
3. Once **all switches** have registered, the Controller responds with a `Register Response` message to each switch, which includes the following information:

(a) The ID of each neighboring switch.

(b) A flag indicating whether the neighbor is alive or not (initially, all switches are alive).

(c) For each live neighbor, the host/port information of that switch process.

5 Path Computations

1. Once all switches have registered, the Controller computes paths between each possible source-destination pair using the shortest path algorithm.
2. Once path computation is completed, the Controller sends each switch its "routing table" using a Route Update message. This table sent to switch A includes an entry for every destination switch (including switch A itself). Each such entry includes the next hop to reach the corresponding destination and the total shortest path distance. For the self-entry the total distance is 0. If a switch can't be reached from the current switch, then the next hop is set to -1 and the distance is set to 9999.

5.1 Shortest Path Example

In the topology example from figure 1, let us consider finding the best path between source switch 1 and destination switch 3. We can observe that the shortest path is 100, which is the path that connects switch 1 and switch 3 through switch 2.

6 Periodic Operations

Each switch and the Controller must perform a set of operations at regular intervals to ensure smooth network functioning.

6.1 Switch Operations

1. Each switch sends a `KEEP_ALIVE` message every K seconds to each of its neighboring switches that it thinks is "alive".
2. Each switch sends a `Topology Update` message to the Controller every K seconds. The `Topology Update` message includes a set of "live" neighbors of that switch.
3. If a switch, A, has not received a `KEEP_ALIVE` message from a neighboring switch B for TIMEOUT seconds, then switch A designates the link connecting it to switch B as down. Immediately, it sends a `Topology Update` message to the Controller containing its updated view of the list of "live" neighbors.
4. Once switch A receives a `KEEP_ALIVE` message from a neighboring switch B that it previously considered unreachable, it immediately marks that neighbor as alive, updates the host/port information of the switch if needed, and sends a `Topology Update` to the Controller indicating its revised list of "live" neighbors.

IMPORTANT: To be compatible with the auto-grader, we require that you use the particular values of K and TIMEOUT mentioned in section 9.

6.2 Mechanism To Handle Concurrency

As described above, a switch process has to concurrently perform two functions: (i) act on the messages received from neighbors/Controller; (ii) send `KEEP_ALIVE` and `Topology Update` messages every "K" seconds and check for dead neighbor switches. Similarly, the Controller has to receive `Topology Update` from the switches, but also periodically check whether any switch has failed.
To implement the concurrency, we require you to use **Threads** (See Threading Library in Python).

7 Simulating Failure

Since we are running switches as processes, it is unlikely to fail due to natural network issues. Therefore, to allow testing and grading your submission we require the following to be implemented.

7.1 Simulating Switch Failure

To simulate switch failure, you just need to kill the process corresponding to the switch. Restarting the process with the same switch ID ensures you can simulate a switch rejoining the network.

7.2 Simulating Link Failure

Simulating link failures is a bit more involved. We ask that you implement your switch with a command line parameter that indicates a link has failed.
For instance, let us say the command to start a switch in its normal mode is as follows:

```
python switch.py <switch-ID> <Controller hostname> <Controller port>
```

Then, make sure your code can support the following parameter below:

```
python switch.py <switch-ID> <Controller hostname> <Controller port> -f <neighbor-ID>
```

This says that the switch must run as usual, but the link to neighborID failed. In this failure mode, the switch should not send `KEEP_ALIVE` messages to a neighboring switch with ID neighborID, and should not process any `KEEP_ALIVE` messages from the neighboring switch with ID neighborID.

8 Logging

We **REQUIRE** that your switch and Controller processes log messages in a format **exactly** matching what the auto-grader requires. The logfile names must also match what the auto-grader expects. No additional messages should be printed.
To help you with generating the right format for the logs, in the starter code, we have created a starter version of `controller.py` and `switch.py` for you. Those two files contain various log functions that you must use. Each log function corresponds to a type of log message that will be explained below. **We strongly recommend that you use those functions because the auto-grader is quite picky about exact log formats. If you choose not to use those functions, we won't be able to award credit for test cases that fail owing to minor logging discrepancies.**

8.1 Switch Process

Each switch process must log:

1. When a `Register Request` is sent.
2. When the `Register Response` is received.
3. When any neighboring switches are considered unreachable.
4. When a previously unreachable switch is now reachable.
5. The routing table that it gets from the Controller each time that the table is updated.

8.1.1 Format for Switch Logs

Please refer to the starter code, where the format for each type of log message is shown in comments beside the corresponding log function.

8.1.2 Switch Log File Name

This must be `switch<switch-ID>.log`

E.g. for Switch (ID = 4): `switch4.log`

8.2 Controller Process

The Controller process must log:

1. When a `Register Request` is received.
2. When all the `Register Responses` are sent (send one `Register Response` to each switch).
3. When it detects a change in topology (a switch or a link is down or up).
4. Whenever it recomputes (or computes for the first time) the routes.

8.2.1 Format for Controller Logs

Please refer to the starter code, where the format for each type of log message is shown in comments beside the corresponding log function.

Note: To reduce log output, please do not print messages when `KEEP_ALIVE` messages are sent or received. We also do not have a log function for this.

Sample logs are available with the starter code. The sample log file is the situation where `Config/graph_3.txt` is used. Therefore, you will find one Controller log file (`Controller.log`) and three switch log files (`switch0.log`, `switch1.log`, and `switch2.log`). After all the switches know the initial topology, switch 1 is killed, and a new topology is calculated by the Controller and sent out to the switches.

8.2.2 Controller Log File Name

This must be `Controller.log`.

9 Running Your Code and Important Requirements

We must be able to run the Controller and Switch python files by running:

```
python controller.py [Controller port] [config file]
```

and

```
python switch.py <switch-ID> <Controller hostname> <Controller port>
```

or

```
python switch.py <switch-ID> <Controller hostname> <Controller port> -f <neighbor-ID>
```

The Controller should be executed first in a separate terminal. While it is running each switch should be launched in a separate terminal with the switch ID, Controller hostname, and the port.

9.1 Important Requirements

To be compatible with the auto-grader, the following are mandatory requirements:

1. You must support command line arguments in the above format. Note that the "-f" flag for the switch is a parameter for link failures (See section 7.2), and we must be able to run your code with and without this flag.
2. Please use K = 2; TIMEOUT = 3 * K (recall these parameters pertain to timers related to periodic operations of the switch and the Controller).

As mentioned earlier, you are strongly recommended to use the logging functions that we provide.

10 Grading and Submission

10.1 Grading

Grading will be based on the test configurations provided with the starter code and some hidden tests. Make sure your code is able to handle all failure and restart scenarios for full points.

10.2 Submission

Submit only `controller.py` and `switch.py` to Gradescope.
Additionally, this project will be submitted in two parts:

10.2.1 Part 1

For Part I, you only need to ensure the system works properly in the absence of link and switch failures. The set of switches, once started, register with the controller, which computes shortest paths and then sends appropriate responses to the switches. Part I will not require the use of threads. You must ensure that the following are correctly generated for different topologies when tested without failures:

- The per switch routing table (i.e. the table in switch#.log)
- The Controller's routing table (i.e. the table in Controller.log)

It is important to ensure the format of the logs is correct. Note that Part 1 is a graded submission and will carry credit.

10.2.2 Part 2

The second part requires everything described in this project document to work (including failures and re-connecting).

11 Suggested Formats for Message Passing

While you are not required to follow this format, you may find it useful to follow the format below for messages exchanged between the switches and the Controller, which we recommend.

11.1 Format for `Register Request` Message

```
<switch-ID> Register_Request
```

E.g. Switch (ID = 3) sends the following `Register Request` to the Controller when it first comes online:

```
3 Register_Request
```

11.2 Format for `Register Response` Message

```
<number-of-neighbors>
<neighbor-ID> <neighbor hostname> <neighbor port> (for each neighbor)
```

E.g. Consider the 6-switch topology given in Figure 1.

Switch (ID = 3) receives the following `Register Response` from the Controller:

```
3
0 <hostname for switch 0> <port for switch 0>
2 <hostname for switch 2> <port for switch 2>
4 <hostname for switch 4> <port for switch 4>
```

11.3 Format for `Route Update` Message

```
<switch-ID>
<dest-ID> <Next Hop to reach dest> (for all switches in the network)
```

Next hop is returned as "-1" if the destination can't be reached by the switch via any possible path.

E.g. Consider the 6-switch topology given in Figure 1. Switch (ID = 3) initially receives the following `Route Update` from the Controller:

```
3
0 0
1 2
2 2
3 3
4 4
5 2
10
```

11.4 Format for `KEEP_ALIVE` Message

```
<switch-ID> KEEP_ALIVE
```

E.g. Switch (ID = 3) sends the following `KEEP_ALIVE` to all of its neighboring switches:

```
3 KEEP_ALIVE
```

11.5 Format for `Topology Update` Message

```
<switch-ID>
<neighbor-ID> <True/False indicating whether the neighbor is alive> (for all neighbors)
```

E.g. Consider the 6-switch topology given in Figure 1. Switch (ID = 3) sends the following initial `Route Update` to the Controller:

```
4
0 True
2 True
4 True
```

12 Additional Resources

You may find the following resources helpful, which provide more information regarding socket programming in Python with UDP:

- Socket HOW TO
- Low-level socket interface
- Socket programming in Python
- Working with UDP sockets

The second part of this project also requires threading. Here are some good links for threading in Python:

- Intro to threading in Python
- Thread based parallelism

Please note that lectures will not get into details of socket programming in Python or threading. These are meant to be material you study on your own using the above resources.

Note: In traditional networks, there are important distinctions between "switches" and "routers". However, the term is used more loosely and interchangeably in the SDN context. We will use the term "switches" in this project in a looser sense, which may not exactly correspond to the lecture material.