Project I: Building a simple software defined network (SDN)

Project 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 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. The 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 the 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 ***widest path routing***. The widest path algorithm is a minor variant of the Dijkstra algorithm and doable with the same time complexity. Specifically, of all possible paths between the source and destination, the path with the highest "bottleneck capacity" is chosen. The bottleneck capacity of a path is the minimum capacity across all links of the path.

Project 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.

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 provide a list of ids of neighboring switches for each switch id. The first line of each file is the number of switches.
Every subsequent line is of the form:

<switch-ID 1> <switch-ID 2> *BW*

This indicates that there is a link connecting switch 1 and switch 2, which has a bandwidth *BW*.

Consider the switch topology given in Figure 1.

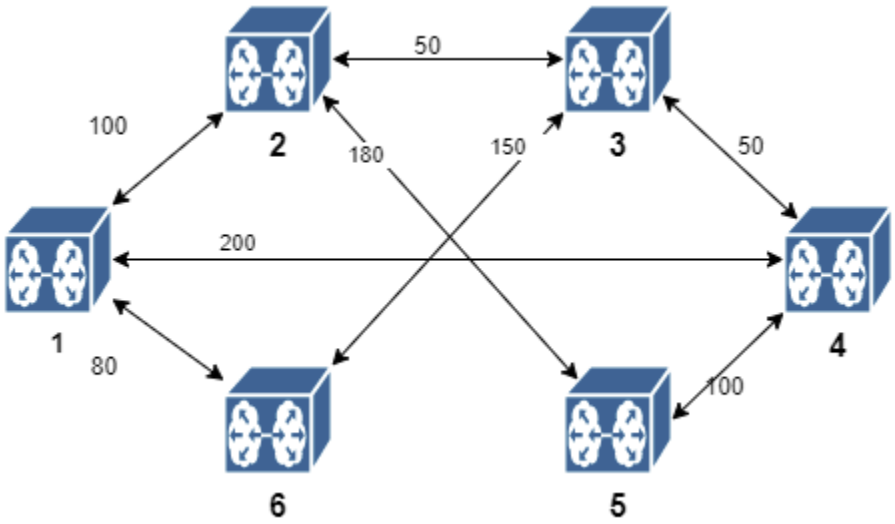

Figure 1: 6 switch topology. Each link is marked with its bandwidth.

Here is an example configuration for the topology shown.

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

Bootstrap process

The bootstrap process refers to how switches register with the controller, as well as 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 the command line arguments follow a required format discussed under 'Running your Code and Important Requirements')
    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, that the controller process runs on, 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 switch, the host/port information of that switch process.

Path computations

1. Once all switches have registered, the controller computes paths between each source-destination pair using the widest path algorithm. Note that this is NOT the traditional Dijkstra algorithm but would involve a small modification to the 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 switch (including switch A itself), and the next hop to reach every destination. The self entry is indicated by an infinite(=9999) bandwidth.

Shortest path vs Widest path Routing

In the topology example from figure 1, let us consider finding the best path between source switch 2 and destination switch 3. Assume each hop in the path has a cost of 1. The shortest path for 2-3 is **2->3** with a '**path length' of 1 hop**. Whereas, the widest path from 2-3 is not the same as the shortest path. The widest path (max bottleneck capacity) is **2->1->6->3** with 'bottleneck capacity' of 80.

Periodic Operations

Each switch and the controller have to perform a set of operations at regular intervals to ensure smooth working of the network.

Switch Operations

1. Each switch sends a Keep Alive message every K seconds to each of the 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 declares the link connecting it to switch B as down. Immediately, it sends a Topology Update message to the controller sending the controller its updated view of the list of 'live' neighbors.
4. Once a switch A receives a Keep Alive message from a neighboring switch B that it previously considered unreachable, it immediately marks that neighbor 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 particular values of K and TIMEOUT as mentioned under the "Running your Code and Important Requirements" section.

Controller Operations

1. If the controller does not receive a Topology Update message from a switch for TIMEOUT seconds, then it considers that switch 'dead', and updates its topology.
2. If a controller receives a Register Request message from a switch it previously considered as 'dead', then it responds appropriately and marks it as 'alive'.
3. If a controller receives a Topology Update message from a switch that indicates a neighbor is no longer reachable, then the controller updates its topology to reflect that link as unusable.
4. When a controller detects a change in the topology, it performs a recomputation of paths using the widest path algorithm described above. It then sends a Route Update message to all switches as described previously.

Mechanism to handle concurrency

As described above, a switch process has to concurrently perform two functions - either act on the messages received from neighbors/controller or wait for every 'K' seconds to send Keep Alive and Topology Update messages 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*).

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.

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.

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:
switch <switch-ID> <controller hostname> <controller port>

Then, make sure your code can support the following parameter below:
switch <switchID> <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.

Logging

We REQUIRE that your switch and controller processes must log messages in a format exactly as described below. Adhering to the format is essential to work with the auto-grader. The logfile names must also be exactly as shown. No additional messages should be printed.

Switch Process

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 switch must also log the routing table that it gets from the controller each time that the table is updated.

Log File name

switch<Switch-ID>.log

E.g., For Switch (ID = 4)
switch4.log

Format for Switch Logs

Register Request

Timestamp
Register Request Sent

Register Response

Timestamp
Register Response Received

Routing Update

Timestamp
Routing Update
<Switch ID>,<Dest ID>:<Next Hop> (For every Dest)
…
...
Routing Complete

Self route is also included -- e.g., Switch (ID = 4) should include the following entry:
4,4:4

Unresponsive/Dead Neighbor Detected

Timestamp
Neighbor Dead <Neighbor ID>

Unresponsive/Dead Neighbor comes back online

Timestamp
Neighbor Alive <Neighbor ID>

Controller Process

Controller process must log

1. When a Register Request is received,

2. When the Register Responses are sent,
3. When the topology is updated (a switch or link is down or up),
4. When topologies are recomputed.

Format for Controller Logs

Controller Log File name

Controller.log

Register Request

Timestamp
Register Request <Switch-ID>

Register Responses

Timestamp
Register Response <Switch-ID> (for every switch)

Routing Update

Timestamp
Routing Update
<Switch ID>,<Dest ID>:<Next Hop>,<Max Bottleneck Bandwidth>
… (For every switch-dest pair)
Routing Complete

Self route must also be included. For e.g., switch (ID = 4) must include the following entry:
4,4:4,9999
9999 indicates 'infinite ' bandwidth.

For switches that can't be reached, the next hop and bandwidth should be '-1', and '0' respectively.
E.g, If switch=4 cannot reach switch=5, the following should be printed
4,5:-1,0

Topology Update: Link Dead

Timestamp
Link Dead <Switch ID 1>,<Switch ID 2>

Topology Update: Switch Dead

Timestamp
Switch Dead <Switch ID>

Topology Update: Switch Alive

Timestamp
Switch Alive <Switch ID>

Note: To reduce log output, please do not print messages when Keep Alive messages are sent or received.

Sample logs are available with the starter code.

Running your code and Important Requirements

We must be able to an run the Controller and Switch python files by running:
python controller.py [controller port] [config file]
python switch.py <switchID> <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.

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 'Simulating Link Failure'), 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)**
3) **As mentioned earlier, all logs that you print must strictly follow the convention described.**

What you need to submit

The controller.py and switch.py codes. You are free to modify the sample config files as provided. ***Do not submit any binaries*. Your git repo should only contain source files, and no products of compilation should be included.**

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.

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.

Format for Register Request

<Switch-ID> Register_Request

Eg. Switch (ID = 4) sends the following Register Request to the controller when it first comes online.

4 Register_Request

Format for Register Response

<number-of-neighbors>
<neighbor id> <neigh_hostname> <neigh_port> (for each neighbor)

Eg. Consider the 6-switch topology given in Figure 1..
Switch (ID = 4) receives the following Register Response from the controller:

4
1 <hostname for switch 1> <port for switch 1>
3 <hostname for switch 3> <port for switch 3>
5 <hostname for switch 5> <port for switch 5>

Format for Route Update

<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.

Eg. Consider the 6-switch topology given in Figure 1.
Switch (ID = 4) receives the following Route Update from the controller (Initially):

4
1 1
2 1
3 1
4 4
5 1
6 1

Format for Keep Alive Message

<Switch-ID> KEEP_ALIVE

Eg. Switch (ID = 4) sends the following Keep Alive to all of its neighboring switches:
4 KEEP_ALIVE

Format for Topology Update Message

<Switch-ID>
<Neighbor Id> <True/False indicating whether the neighbor is alive> (for all neighbors)

Eg. Consider the 6-switch topology given in Figure 1.
Switch (ID = 4) sends the following Route Update to the controller (Initially):

4
1 True
3 True
5 True