ECE 595 Computer Network Systems
Spring 2017
Project I: Building a simple software defined network (SDN)
To be done in groups of 2

Due Date: 11:59 AM (noon), Tue Feb 7th, 2017

In this project, you will implement a basic SDN with a centralized controller and a simple SDN switch using socket programming. The goals of this project are (i) to provide an exercise in client-server socket programming; and (ii) to provide exposure to the basic ideas behind SDNs.

An SDN network consists of multiple switches, and a centralized controller. Unlike traditional networks, switches do not run distributed protocols for route computation in an SDN network. Instead, the 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.

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.

Given that it is logistically challenging to obtain a real network with real switches, in your implementation, the controller and each of the switches will run as separate user-level process and bind to distinct UDP ports. The communication between switches, as well as between each switch and the controller involves UDP socket programming.

The rest of the document provides more details about the project.

Bootstrap process:

The bootstrap process refers to how switches register with the controller, as well as learn about each other. Since your project will be implemented using socket programming, 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 is feasible.

1. The controller process binds to a well known port number. 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. When a switch A 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.

2. The controller responds with an REGISTER_RESPONSE message, which includes a list of neighbors of A. Specifically, for each neighbor, information returned includes: (i) the id of the neighboring switch; (ii) a flag indicating whether the neighbor is active or not. The neighbor is considered inactive if it has not yet registered with the controller (by sending it a REGISTER_REQUEST message), or if the controller

considers the neighboring switch to be dead; and (iii) for each active switch, the host/port information of that switch process .

3. On receiving a REGISTER_RESPONSE, switch A immediately sends a "KEEP-ALIVE" message to each of the active neighbors (using the host/port information provided in the response). If node *B* receives a "KEEP-ALIVE" message from switch A, it marks A as an active neighbor, and learns the host/port information for switch A. Both A and B periodically send KEEP_ALIVE messages to each other as described later in this document.

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 a traditional Dijkstra algorithm. 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.

2. Once path computation is completed, the controller sends each switch its "routing table" using a ROUTE_UPDATE message. This table includes for every other switch, the next hop to reach every destination.

Hint: The widest path algorithm is a minor variant of the Dijkstra algorithm and doable with the same time complexity. If you are unable to implement the widest path algorithm, we suggest that you at least implement the traditional Dijkstra's shortest path algorithm. This will be sufficient to get partial credit for the project, but you will however earn more credit if you can implement the widest path algorithm. You must implement either algorithm from scratch and not just use a standard library for the same.

Periodic operations and failure detection

1. Each switch takes the following operations:
    1) Every K seconds, the switch sends a KEEP_ALIVE message to each of the neighboring switches.
    2) Every K seconds, the switch sends a TOPOLOGY_UPDATE message to the controller. The TOPOLOGY_UPDATE message includes a set of live neighbors of that switch.
2. If a switch A has not received a KEEP_ALIVE message from a neighboring switch B for M * K 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.
3. 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, and sends a TOPOLOGY_UPDATE to the controller indicating its revised list of live neighbors.
4. If the controller does not receive a TOPOLOGY_UPDATE message from a switch for M * K seconds, then it considers that switch dead, and updates its topology. Likewise, if it starts receiving TOPOLOGY_UPDATE messages from a switch it previously considered dead, it updates its topology again.

5. 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. Likewise, if the TOPOLOGY_UPDATE indicates the neighbor is now reachable, the controller updates the topology to reflect that fact.

Failure handling:
When a controller detects that a switch or link has failed, it performs a recomputation of paths using the shortest-widest path algorithm described above. It sends a ROUTE_UPDATE message to all switches as described previously

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 <switchID>  <controller hostname> <controller port>
```

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

```
switch <switchID> <controller hostname> <controller port> -f <neighbor ID>
```

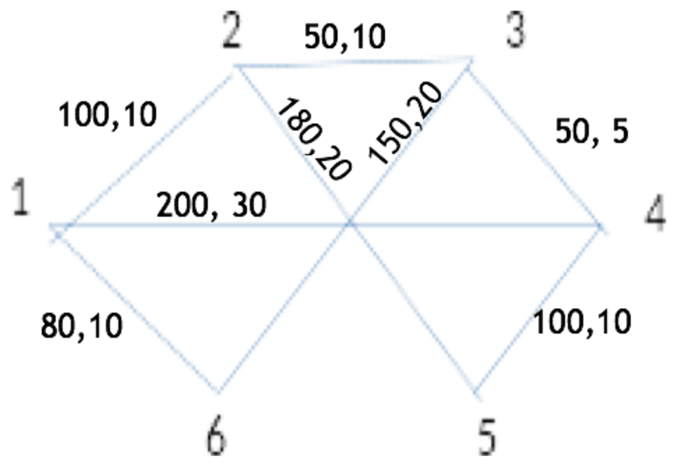
This says that the switch must run as usual, but with the link to neighborID failed. In this failed mode, the switch should not send KEEP_ALIVE messages to a neighboring switch with ID neighborID, and should not process any KEEP_ALIVE messages from the neighboring switch with ID neighborID.

Logging:
1. The switch process must log information indicating when a REGISTER_REQUEST is sent, REGISTER_RESPONSE is received, when any neighboring switches are considered unreachable, or when a previously unreachable switch is now reachable. The switch must also log the routing table that it gets from the controller each time that the table is updated.
2. Likewise, the controller process must log REGISTER_REQUEST and REGISTER_RESPONSE, when the topology is updated (a switch or link is down), and when topologies are recomputed.
3. To reduce log output, do not print messages when KEEP_ALIVE messages are sent or received in the default log mode, but please have a higher verbosity level where these messages are logged as well. The verbosity level is itself a command line parameter.
4. Make sure when you do your demo to us, the log messages are sufficiently useful for us to see that your system is operating properly.

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 the timer to expire every ‘K’ seconds to send KEEP_ALIVE and TOPOLOGY_UPDATE messages. Similarly the controller has to concurrently both perform the path computation and receive TOPOLOGY_UPDATE from the switches. Please see “Choice

of language and concurrency mechanism" section at the end for our preferences regarding which concurrency mechanism you use.

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:
    <switchID1>: <switchID2>: *BW*:*Delay*
This indicates that there is a link connecting switchID1 and switchID2, which has a bandwidth *BW,* and a delay *Delay.*
Here is an example configuration for the topology shown

```
6
1 2  100 10
1 4  200 30
1 6  80 10
2 3  50 10
2 5  180 20
3 4  50 5
3 6  150 20
4 5  100 10
```

Difference between shortest path and the widest path:
In the above topology example, let us consider finding the best path between source switch 6 and destination switch 5. The shortest path for 6-5 is **6-3-4-5** with **'path length' of 35**. Whereas, the widest

path from 6-5 is not same as the shortest path. There are two choices for the widest path (max bottleneck capacity) viz. 6-1-2-5 or 6-1-4-5 with ‘bottleneck capacity’ of 80. Either of these paths is acceptable. .

Grading:
Grading will be based on a demo. Details will be conveyed later. You should plan to include your own test cases, as well as design your project so it is easy for us to test various features during the demo. The more efficient, well thought out and smooth your demo, the higher the grade you are likely to get.
Here are some example items you must pay attention to:

- You should be able to easily run your demo with a test configuration file that we provide.
- Pay attention to what information is logged as that will be crucial to evaluating your demo. Poor logging that makes it difficult to evaluate your work will lose credit.

Choice of language and concurrency mechanism:
Our preference is you use C programming, along with I/O multiplexing using select() for concurrency. We expect that we would be able to provide most help for such an implementation. However, you are free to use Python/Java instead, and thread-based implementations if you prefer, and are more comfortable. However, the help we are able to provide with such implementations is more limited, and you should expect to be more on your own.