Spanning Tree

Table of Contents

PROJECT GOAL

Spanning Trees which can be used to prevent forwarding loops on a layer 2 network (Modules->Lesson 1-> Looping Problem in Bridges and the Spanning Tree Algorithm). In this project, we will develop a simplified, distributed version of the Spanning Tree Protocol that can be run on an arbitrary layer 2 network topology. We will simulate the communications between switches with Messages. The goal is to converge on a single solution and output the final spanning tree.

Part 1: Setup

We can do this project on the host system if it has Python 3.11.x. The project does not have any dependencies outside of Python.

Part 2: Files Layout

There are many files in the SpanningTree directory, but we should **only** modify *Switch.py*, which represents a layer 2 switch. We will implement the functionality of the Spanning Tree Protocol to generate a Spanning Tree for each Switch.

The files in the project skeleton are described below. DO NOT modify these files. All of the code must be in Switch.py **ONLY**. We should study the other files to understand the project.

- `Topology.py` - Represents a network topology of layer 2 switches. This class reads in the specified topology and arranges it into a data structure that the Switch can access.

This class also adjusts the topology if any changes are indicated within the XXXTopo.py class.

- `StpSwitch.py` - A base class of the derived class we will code in Switch.py. The base class StpSwitch.py is the parent class to the Switch. It sends the initial messages.
- `Message.py` - This class represents a message format we will use to communicate between switches, similar to the course lectures. Specifically, we will create and send messages in Switch.py by declaring a message as:

  ```
      msg = Message(claimedRoot, distanceToRoot, originID,
                    destinationID, pathThrough, ttl)
  ```

  and assigning the correct value to each input. Message format may NOT be changed. See the comments in Message.py for more information on the data in these variables.
- `run.py` - A "main" file that loads a topology file (see XXXTopo.py below), uses that data to create a Topology object containing Switches, and runs the simulation.
- `XXXTopo.py,` etc. - These are topology files that we will pass as input to the run.py file.

Part 3: TODOs

This is an outline of the code we must implement in `Switch.py` with *suggestions* for implementation. The implementation must adhere to the "spirit of the project": it must be a **distributed** solution.

A. Decide on the data structure(s) that we will use to keep track of the spanning tree.

1. The collection of active links across all switches is the resulting spanning tree.
2. The data structures may be variable(s) needed to track each switch's own view of the tree. **A switch only has access to its member variables. A switch may not access its neighbor's information directly – to learn information from a neighbor, the neighbor must send a message.**
3. This is a distributed algorithm. The switch can only communicate with its neighbors. It does not have an overall view of the spanning tree, or the topology as a whole.
4. An example data structure should include, at a minimum:
   a. a variable to store the switch ID that this switch sees as the *root*,
   b. a variable to store the *distance* to the switch's root,
   c. a list or other datatype that stores the "*active links*" (only the links to neighbors that are in the spanning tree).
   d. a variable to keep track of which neighbor it goes through to get to the root (a switch should only go through one neighbor, if any, to get to the root).
5. More variables may be used to track data as needed to build the spanning tree and will depend on the specific implementation.

B. Implement processing a message from an immediate neighbor.

1. We **do not** need to worry about sending the initial messages. We only need to worry about the sending and processing of subsequent messages.
2. For each message a switch receives, the switch will need to:

    a. **Determine whether an update to the switch's root information is necessary and update accordingly.**

    I. The switch should update the *root* stored in its data structure if it receives a message with a lower *claimedRoot.*

    II. The switch should update the *distance* stored in its data structure if a) the switch updates the *root*, or b) there is a shorter path to the same root.

    b. Determine whether an update to the switch's active links data structure is necessary and update accordingly. The switch should update the *activeLinks* if:

    I. The switch finds a new path to the root (through a different neighbor). In this case, the switch should add the new link to *activeLinks* and (potentially) remove the old link from *activeLinks*

    II. The switch receives a message with *pathThrough* = TRUE but does not have that *originID* in its *activeLinks* list. In this case, the switch should add *originID* to its *activeLinks* list.

    III. The switch receives a message with *pathThrough* = FALSE but the switch has that *originID* in its activeLinks. In this case, the switch should remove *originID* from its *activeLinks* list.

    c. Determine when the switch should send new messages to its neighbors and send the messages.

    I. The message FIFO queue is maintained in Topology.py. The switch implementation does not interact with the FIFO queue directly, but uses the send_message function, and receives messages as arguments in the process_message function.

    II. When sending messages, *pathThrough* should only be TRUE if the *destinationID* switch is the neighbor that the *originID* switch goes through to get to the claimedRoot. Otherwise, pathThrough should be FALSE.

III. The switch should continue sending messages to its neighbors until the ttl on the Message reaches 0. We will need to decrement the ttl as we are processing the Messages.

3. Other variables may be helpful for determining when to update the root information or the *activeLinks* data structure and can be added to the data structure and updated as needed, depending on the implementation.
4. Once this logic is complete, we will need to understand a few other things about the topologies to check the log results. For certain topologies, switches may get dropped while the algorithm is running. In this case, the algorithm should adjust accordingly and create a Spanning Tree for the new topology. The Topology class will restart the message process if a change occurs. This is handled for we already. The final Spanning Tree should match the results of the new Topology, not the starting one.
   a. The switch that is dropped should never split the original topology. That means that the final Topology will remain connected and there will only be one resulting Spanning Tree.
   b. The switch that is dropped could be the original root, the algorithm should adapt accordingly.
   c. The Topology file will include the ttl_limit and drops. The ttl_limit is the starting ttl for each message in the Topology. Once the message has been passed around that many times, the algorithm should terminate. This is something we must implement. The drops indicate which switch(es) will be dropped to change the topology. **C.** Write a logging function.

1. The switch should only output the links that are in the spanning tree.
2. Follow the below format (# - #). Unsorted or non-standard formatting will result in penalties. Examples of correct logs with the correct format have been provided to we.
3. Sorted: Not sorted:

Sorted:	Not sorted:
1 - 2, 1 - 3	1 - 3, 1 - 2
2 - 1, 2 - 4	2 - 4, 2 - 1
3 - 1	3 - 1
4 - 2	4 - 2

Part 4: Testing and Debugging

To run the code on a specific topology (SimpleLoopTopo.py in this case) and output the results to a text file (out.txt in this case), execute the following command:

```
python run.py SimpleLoopTopo
```

“SimpleLoopTopo” is not a typo in the example command – don’t include the .py extension.

We have included several topologies with correct solutions for we to test the code against. We can (and are encouraged to) create more topologies and test suites with output files and share them on Ed Discussion. There will be a designated post where students can share these files.

We will only be submitting `Switch.py` – the implementation must be confined to modifications of that file. We recommend testing the submission against a clean copy of the rest of the project files prior to submission.