OMSCS GEORGIA TECH

Spanning Tree

CS 6250

Spring 2024

Spanning Tree

Table of Contents

PROJECT GOAL

In the lectures, you learned about 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, you 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

Download the project files from Canvas. You can do this project on your host system if it has Python 3.11.x. The project does not have any dependencies outside of Python. You must be sure that your submission runs properly in Gradescope. Gradescope is the environment where your project will be graded. **Gradescope and the VM are the only valid environments for this course.**

Part 2: Files Layout

There are many files in the SpanningTree directory, but you should **only** modify *Switch.py*, which represents a layer 2 switch. You 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 your code must be in Switch.py **ONLY**. You 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 your 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 you will code in Switch.py. The base class StpSwitch.py is the parent class to your Switch. It sends the initial messages.
- `Message.py` - This class represents a message format you will use to communicate between switches, similar to the course lectures. Specifically, you 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 you will pass as input to the run.py file.

Part 3: TODOs

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

A. Decide on the data structure(s) that you 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 your specific implementation.

B. Implement processing a message from an immediate neighbor.

1. You **do not** need to worry about sending the initial messages. You 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. You will need to decrement the ttl as you 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 your data structure and updated as needed, depending on your implementation.
4. Once this logic is complete, you will need to understand a few other things about the topologies to check your log results. For certain topologies, switches may get dropped while the algorithm is running. In this case, your 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 you 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, your 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, your algorithm should terminate. This is something you 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 you.
3. Sorted:

```
1 - 2, 1 - 3
2 - 1, 2 - 4
3 - 1
4 - 2
```

Not sorted:

```
1 - 3, 1 - 2
2 - 4, 2 - 1
3 - 1
4 - 2
```

Part 4: Testing and Debugging

To run your 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 you to test your code against. You 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.

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

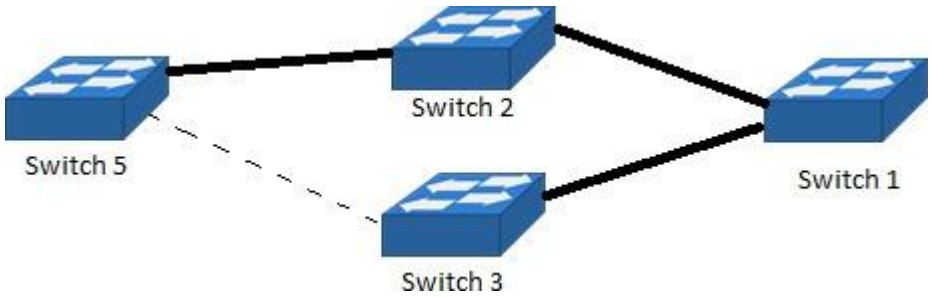
You may add print statements to facilitate debugging during your development process, but they should be removed or commented out prior to submission.

Part 5: Assumptions and Clarifications

You may assume the following:

A. **All switch IDs are positive integers, and distinct.**
   1. These integers do not have to be consecutive.
   2. They will not always start at 1.
   3. There is no maximum value beyond language (Python) limitations (which your code does not need to check for).

B. **Tie breakers**: If there are multiple paths of equal distance to the same root, the switch should choose the path through the neighbor with the lowest switch ID.
   1. Example: switch 5 has two paths to root switch 1, through switch 3 and switch 2. Each path is 2 hops in length. Switch 5 should select switch 2 as the path to the root and disable forwarding on the link to switch 3.



C. **There is a single distinct solution spanning tree for each topology.** This is guaranteed by the first two assumptions.

D. **All switches in the network will be connected to at least one other switch, and all switches are able to reach every other switch.** It will always be possible to form a tree that spans the entire topology.

E. **There will be only 1 link between each pair of directly connected switches.** You do not need to consider how STP would behave with redundant links.

F. **A switch may always communicate with its neighbors.** When a switch treats a link as inactive, the link can still be used during the simulation. "Inactive" simply means that the link will not be used for forwarding normal network traffic.

G. **The solution implemented in `Switch.py` should terminate without intervention.** When there are no more messages in the queue to process, the simulation will log the

output and terminate. Your algorithm should stop sending messages when the ttl on the messages hit 0.

H. **Your solution should not require any outside Python modules. Do not import any other modules.**

What to Turn In

Submit ONLY your `Switch.py` file to Gradescope as a single file. Do ***not*** modify the name of Switch.py. You may make an unlimited number of submissions to Gradescope before the deadline. Your last submission will be your grade unless you activate a different submission.

Before submission:

a. **Make sure your logging format is correct.** Invalid format will be marked as incorrect.
b. **Remove all print statements from your code before turning it in.** Print statements can have drastic effects on runtime. Your submission must take less than 30 seconds per topology. If print statements in your code adversely affect the grading process, your work will not receive full credit.
c. Your algorithm must converge upon the Spanning Tree within the Topology's ttl_limit.
d. **Make sure your Switch.py works in Gradescope.** Gradescope will give you immediate feedback, along with your grade, so we will not accept re-grade requests related to incorrect submissions.
e. Make sure your Switch.py has Linux-style line endings. Windows may try to put CRLF at the end of lines. If it works in Gradescope, it is fine.
f. **Helper functions:** Helper functions are fine as long as the names don't conflict with anything already in the project. If it works in Gradescope, it is fine.

After submission:

g. **Make sure your submission uploaded correctly.** Late submissions will not be accepted.
h. **Your grade in Gradescope will be your grade for this project,** with some caveats:
    a. Any Honor Code violations will result in a 0 and be referred to OSI.
    b. Any attempt to bypass or distort the autograder will result in a 0 and may be referred to OSI.
i. **Please Note**: If Gradescope receives your submission but it fails to run, we can re-run that submission after the deadline. In this case, you will miss out on the chance to get feedback before the deadline. This can happen if too many students are submitting the project at the same time, so **be sure to start early**. If Gradescope is overloaded, you can re-submit at a later time to get feedback.

j. **If,** for some reason, **you cannot submit your code to Gradescope and you are up against the submission deadline**, create a **Private** post on Ed Discussion with an ***attachment*** of your *Switch.py*.

What you can and cannot share

Honor Code/Academic Integrity: Do **NOT** share **any** code from `Switch.py` with your fellow students, on Ed Discussion, or publicly in any form. You **may** share log files for any topology, and you may share any code you write that will *not be turned in*, such as new topologies or testing suites.

All work must be your own, and consulting Spanning Tree Protocol solutions, even in another programming language or just for reference, are considered violations of the honor code. **Do not** reference solutions on Github! Do not use IDE extensions (like Github Copilot) that write or recommend blocks of code to you (autocomplete for function names is fine). For more information see the Syllabus Definition of Plagiarism. We have worked hard to provide you with all the material you need to complete this project without help from Google/Stack Overflow (Searching basic Python syntax is fine). Don't risk an honor code violation for a very doable project.

Start early, ask questions in Ed Discussion, and attend TA chat sessions if needed.

Rubric

10 pts	Correct Submission	For turning in the correct file with the correct name. You receive 10 FREE points for reading the instructions.
60 pts	Provided Topologies	For correct Spanning Tree results (log files) on the provided topologies.
30 pts	Unannounced Topologies	For correct Spanning Tree results (log files) on three topologies that you will not have access to. These cases are used to prevent students from hard-coding a solution.

Grading Note: Partial credit is **not** available for individual topology logs. The output must be fully correct to receive credit – a single link discrepancy will result in a zero for that topology.

The goal of this project is to implement a simplified version of a network protocol using a **distributed** algorithm.

The skeleton code we provide you runs a simulation of the larger network topology, and for the sake of simplicity, the `StpSwitch` class defines a link to the overall topology. This means it is possible using the provided code for one Switch to access another's internal state (self.topology). This goes against the "spirit of the project". This is also clearly labeled in the comments of the skeleton code.