# Project 2 - Spanning Tree Protocol

In the lectures, you learned about Spanning Trees (see Canvas->Modules->Lesson 1-> “Looping Problems in Bridges and the Spanning Tree Algorithm”), which can be used to prevent forwarding loops on a layer 2 network (see also https://en.wikipedia.org/wiki/Spanning_tree). In this project, you will develop a simplified **distributed** version of the Spanning Tree Protocol that can be run on an arbitrary layer 2 topology. This project does not use the Mininet environment (Don't worry, Mininet will be back in later projects!). Rather, we will be simulating the communications between switches until they converge on a single solution, and then output the final spanning tree to a file.

One clarification – in the lectures, the Spanning Tree Algorithm was discussed in the context of bridges. This project uses the Spanning Tree Algorithm in the context of switches. The major result of this is that while in the lectures, not every bridge is necessarily part of the overall spanning tree, in this project, the overall spanning tree **does** include every switch in the topology.

## 1. Project Setup

Download the project from Canvas to the course VM and unzip. Ensure the files have the correct permissions. (See project 1 if you need a refresher on basic Linux commands.)

This project must be coded in Python 2.7.

## 2. Project Files Layout

There are many files in the `Project2` directory, but you only need to (and should only) modify `Switch.py`, which represents a layer 2 switch that implements our simple Spanning Tree Protocol. You will implement the functionality described in the lectures and at the links above to generate a Spanning Tree for each Switch. (Details follow in the “3. Project Outline – TODOs”.)

The other files in the project skeleton are:

- `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 code can access.
- `StpSwitch.py` - A superclass of the class you will edit in `Switch.py`. It abstracts certain implementation details to simplify your tasks.
- `Message.py` - This class represents a simple message format you will use to communicate between switches, similar to what was described in 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)
```

and assigning the correct data 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_spanning_tree.py` - A simple "main" file that loads a topology file (see `XXXTopo.py` below), uses that data to create a Topology object containing Switches, and starts the simulation.
- `XXXTopo.py`, etc - These are topology files that you will pass as input to the `run_spanning_tree.py` file.
- `sample_output.txt` - Example of a valid output file for `Sample.py` as described in the comments in `Switch.py`.

## 3. Project Outline – TODOs

This is an outline of the code to implement in `Switch.py` with *suggestions* for implementation. Your implementation must adhere to the "spirit of the project" and satisfy the labeled TODO sections in the project code per the pre-existing comments.

A. ***Decide on the data structure that you will use to keep track of the spanning tree***.

1. The collection of active links across all switches is the resultant spanning tree.
2. The data structure may be any variables 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 of the topology as a whole.
4. An example data structure would include, at a minimum:
    a. a variable to store the switch ID that this switch currently 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*" (i.e., the links to neighbors that should be drawn 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 helpful to track data needed to build the spanning tree and will depend on your specific implementation.
6. It is important to create a data structure in the correct place in Python (and most object-oriented programming languages). If you create it inside a method, every time the method is called it will be created as new. You should create a class object in the class constructor so that the data stored in the object exists for the life of the class instance that is created by Topology.py. For example self.mylist = [ ] in the constructor should create an empty list data structure and act as instance variable. But if mylist were instantiated in, say, process_messages, then it will be created every time the method is called. This could be useful in how you track which links are active to certain neighbors for any given switch.

B. ***Implement sending initial messages to neighbors of the switch*.**

1. Your implementation of send_initial_messages will be called in Topology.py for each switch in the topology before any other messages are processed and/or sent.
2. See Message.py, Topology.py, and StpSwitch.py for details on message format, message creation, and how to send messages between switches.
    a. *pathThrough* is a Boolean, not an int.
    b. In a message, *pathThrough* is TRUE if the sending switch goes through the receiving switch in order to reach *claimedRoot*. *pathThrough* is FALSE if the sending switch does not go through the receiving switch in order to reach *claimedRoot*.
3. Initially, each switch thinks it is the root of the spanning tree.

C. ***Implement processing a message from an immediate neighbor*.**

1. 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* stored in the data structure 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 may need to remove *originID* from its *activeLinks* list.

c. **Determine whether the switch should send new messages to its neighbors** and send messages accordingly.

i. This is an important design decision. There are many correct algorithms that send messages at different times.

ii. The message FIFO queue is maintained in Topology.py. The switch implementation does not interact with the FIFO queue directly, but instead uses send_msg to send messages, and receives a message as an argument in process_message.

iii. 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.

2. 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.

3. Use of “self.topology” or any of its member variables or functions is strictly prohibited and will be penalized heavily.

D. ***Write a logging function that is specific to your particular data structure.***

1. The switch should only output the links that it thinks are in spanning tree.

2. Follow the format. Unsorted/non-standard formatting can result in autograder penalties. Examples of correct solutions with correct format have been provided to you.

3. Example:

1 - 2, 1 - 3
2 - 1, 2 - 4
3 - 1
4 – 2

Not sorted:

1 – 3, 1 - 2
2 – 4, 2 - 1
4 - 2
3 – 1

# 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_spanning_tree.py SimpleLoopTopo out.txt
```

*NOTE:* "SimpleLoopTopo" is not a typo in the example command.

We have included several topologies with correct solutions (and format) for you to test your code against. You can (and are encouraged to) create more topologies with output files and share them on Piazza.

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

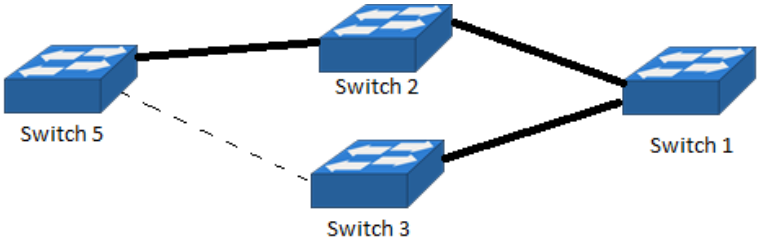
We encourage adding print statements to facilitate debugging during the development process, if they are removed/commented out prior to submission.

# 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 two 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 should behave with redundant links.
F. **The topology given at the start will be the final topology**. The topology will not change while your code is running (i.e., adding a switch, severing a connection, etc.)
G. **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 port/link will not be used for forwarding normal network traffic.
H. **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 output and self-terminate.

## 6. Submission

Submit ONLY your Switch.py in a zip file named GTLogin_p2.zip. Do not have an enclosing directory. Replace GTLogin with your GT login.

Before submission:

A. **Make sure you understand when the deadline for submission is.** Canvas shows an availability date that is after the due date – this reflects the late period. Submissions during the late period will accrue late penalties as dictated in the syllabus.
B. **Make sure your logging format is correct.** Invalid format may result in autograder penalties.
C. **Remove any print statements from your code before turning it in.** Print statements left in the simulation, especially for inefficient implementations, have drastic effects on run-time. Your submission should take less than 10 seconds to process a topology used in grading. If print statements in your code adversely affect the grading process, your work will not receive full credit.
D. **Make sure your Switch.py works with a fresh copy of the other project files.** Some IDEs or moving of files between different operating systems have been known to alter student work in unexpected ways.

After submission:

E. **Make sure your submission uploaded correctly.** Submissions after the late period expires will not be accepted.
F. **Make sure you uploaded the correct files.** We cannot accept resubmissions after the deadline has expired. Make sure you did not submit the wrong file (for instance, Switch.pyc instead of Switch.py).

## 7. Grading

| | | |
|---|---|---|
| 15 pts | Correct Submission | for turning in all the correct files with the correct names, and significant effort has been made in each file towards completing the project. |
| 60 pts | Provided Topologies | for correct Spanning Tree results (log file) on the provided topologies. |
| 75 pts | Unannounced Topologies | for correct Spanning Tree results (log file) on three topologies that you will not see in advance. They are slightly more complex than the provided ones, and may test for corner cases. |

GRADING NOTES:

1. ***Partial credit is not available for individual topology spanning tree output files.*** The output spanning tree must be fully correct to receive credit for that input topology – a single link's discrepancy will result in a zero for that topology. Additionally, we will be using many topologies to test your project, including but not limited to the topologies we provide, and checking for corner cases not exhibited in the sample topologies provided.
2. ***Using topolink or self.topology is strictly prohibited.*** The autograder checks if submissions attempt to directly access *topolink* or *self.topology*. Submissions that attempt this will receive no credit. (If you have questions about whether your code is accessing data it should not, please ask on Piazza or during office hours!)
3. The goal of this project is to implement a simplified version of a network protocol using a **distributed** algorithm. This means that your algorithm should be implemented at the network switch level. Each switch only knows its internal state, and the information passed to it via messages from its direct neighbors - the algorithm **must** be based on

these messages. ***An implementation that does not observe this may be heavily penalized.***

## 8. Honor Code/Academic Integrity

Do **not** share code from `Switch.py` with your fellow students, on Piazza, or publicly in any form. You **may** share log files for any topology, and you may also share any code you write that will *not be turned in*, such as new topologies or other testing code.

In past semesters, the most trouble we have had with students not abiding by the honor code was in this project. All work must be your own, and consulting solutions, even in another programming language or just “for reference”, are considered violations of the honor code. Start early, ask questions in Piazza and attend TA chats if helpful. While this project is challenging, most of our students have succeeded with time and hard work and have a great sense of personal achievement with this project.

# Appendix:

## FAQ:

### Spirit of the Project

The goal of this project is to implement a simplified version of a network protocol using a **distributed** algorithm. This means that your algorithm should be implemented at the network switch level. Each switch only knows its internal state, and the information passed to it via messages from its direct neighbors - the algorithm **must** be based on these messages.

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. This goes against the spirit of the project, and is not permitted. Additional detail is available in the comments of the skeleton code.

### Additional Tips and Resources

*Creating the Data Structure/Object-Oriented Programming*:

It is important to create a data structure in the correct place in Python (and most object oriented programming languages). If you create it inside a method, every time method is called it will be created as new. You should create a class object in the class constructor so that the data stored in the object exists for the life of the class instance that is created by Topology.py. For example self.mylist = [ ] in the constructor should create an empty list data structure and act as instance variable. But if mylist were instantiated in, say, process_messages, then it will be created every time the method is called. This could be useful in how you track which links are active to certain neighbors for any given switch.