CS 6250 – Computer Networking

Project Walkthrough

Spanning Tree Protocol

Georgia Tech

Summary of the Project

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. The project is **not** a standard implementation of the Spanning Tree Protocol, so be sure not to reference any outside material or concepts that could lead you astray.

Project Files

You will modify Switch.py, we'll go over it in a minute.

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 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. Create and send messages in Switch.py by declaring a message as

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

Project Files, continued

run.py - A simple "main" file that loads a topology file (see XXXTopo.py below), uses that data to create a Topology object containing Switch objects, and starts the simulation.

NoLoopTopo.py, SimpleLoopTopo.py, ComplexLoopTopo.py, etc - These are topology files that you will pass as input to the run_spanning_tree.py file.

Logs/NoLoopTopo.py, Logs/SimpleLoopTopo.log, Logs/ComplexLoopTopo.log – valid output files for given topologies.

Switch.py – implementing the data structure

The data structure keeps track of a switch's view of the Spanning Tree. The collection of active links from each switch's data structure is the resultant Spanning Tree. The data structure may be any variables needed to track each switch's own view of the tree.

Keep in mind that in a **distributed** algorithm, the switch can only communicate with its neighbors. A switch does not have an overall view of the tree as a whole. You should not access *self.topology* within Switch.py.

The switches are trying to learn the root, which is the switch with the lowest id, and the path to that root switch. An example data structure would include, at a minimum, a variable to store the switchID that this switch currently sees as the root, a variable to store the distance to the switch's root, and a list or other datatype that stores the "active links" (i.e., the links to neighbors that should be drawn in the spanning tree). To track the path to the root, each switch may also need to know which neighbor it goes through to get there (in the slides to follow we call this data structure variable *switchThrough*), and the distance of the path to the root. See examples to follow.

*TIPS: If you're new to Python, you'll need to understand how self works.
Do not use global variables.
Do not define your data structure in send initial message method since it will be overwritten every time the switch class is called.

Switch.py – initial messages, processing messages, logging for output

Implement the Spanning Tree Protocol:
You do NOT need to worry about sending the initial messages. You only need to worry about sending subsequent messages.

The messages are processed as a FIFO queue, but the switches do not need to push or pop on the FIFO queue since Topology.py does this for the switches as each switch calls send_msg().

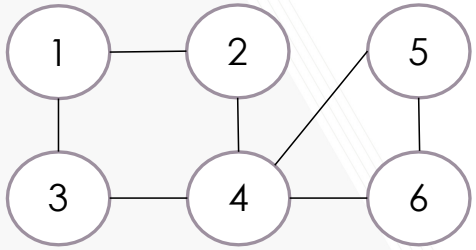
As each switch processes messages, it compares the received message data to the data in its data structure to build the spanning tree.

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

TIP: Take the time now to read all the project files, starting and ending with Switch.py and possibly reading Message.py and Topology.py twice as well.

Example:

Topology



root = id of the switch thought to be the root by the origin switch

distance = the distance from the origin to the root node

origin = the ID of the origin switch

destination = the ID of the destination switch

pathThrough = Boolean value indicating the path to the claimed root from the message's origin passes through the message's destination

ttl = number of times the message will be passed around

Empty Data Structure

Switch X	
root	
distance	
activeLinks	
switchThrough	

Empty Message Queue

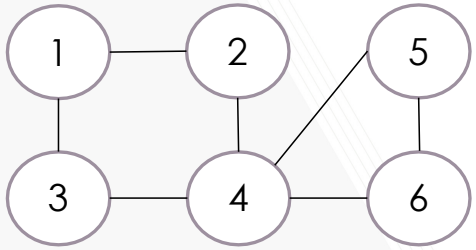
Root	Distance	Origin	Destination	pathThrough

Georgia Tech
CREATING THE NEXT

Example: Initial Data Structure and Messages sent by Switch 1

initial messages…

Topology



Data Structure

Switch 1	
root	1
distance	0
activeLinks	
switchThrough	1

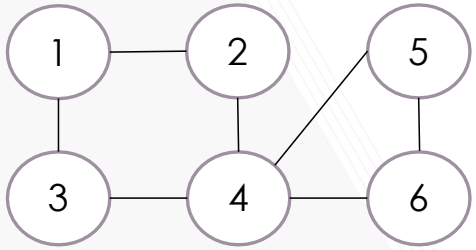
Message Queue

Root	Distance	Origin	Destination	paththrough
1	0	1	2	F
1	0	1	3	F

Georgia Tech
CREATING THE NEXT

Example: Initial Data Structure and Messages sent by Switch 2

Topology



initial messages…

Message Queue

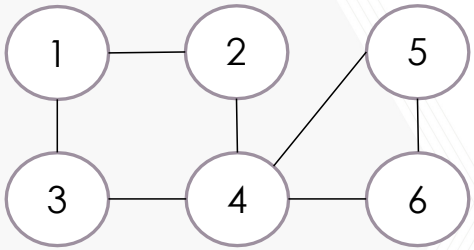
Root	Distance	Origin	Destination	paththrough
1	0	1	2	F
1	0	1	3	F
2	0	2	1	F
2	0	2	4	F

Data Structure

Switch 2	
root	2
distance	0
activeLinks	
switchThrough	2

Example: All Initial Messages

Topology



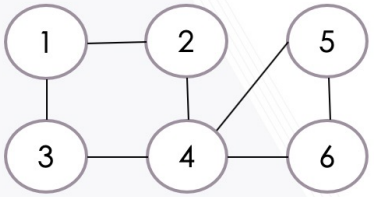
This is the FIFO message queue at the end of sending the initial messages. Note that you may see a different order of messages (the order of the messages should not matter to your algorithm).

Message Queue

Root	Distance	Origin	Destination	paththrough
1	0	1	2	F
1	0	1	3	F
2	0	2	1	F
2	0	2	4	F
3	0	3	1	F
3	0	3	4	F
4	0	4	2	F
4	0	4	3	F
4	0	4	5	F
4	0	4	6	F
5	0	5	6	F
5	0	5	4	F
6	0	6	5	F
6	0	6	4	F

Example: Switch 2

Topology



Compare message.root = 1 and self.root = 2. Update self.root to be message.root, self.distance to be message.distance + 1. Add message.origin to activeLinks, and set switchThrough to message.origin. Update neighbors with new messages

Before processing

Switch 2	
root	2
distance	0
activeLinks	
switchThrough	2

After processing

Switch 2	
root	1
distance	1
activeLinks	1
switchThrough	1

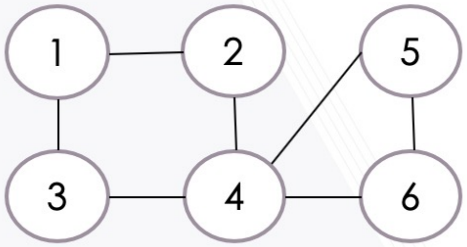
Message Queue

Root	Distance	Origin	Destination	paththrough	ttl
1	0	1	2	F	n
1	0	1	3	F	n
2	0	2	1	F	n
2	0	2	4	F	n
3	0	3	1	F	n
3	0	3	4	F	n
4	0	4	2	F	n
4	0	4	3	F	n
4	0	4	5	F	n
4	0	4	6	F	n
5	0	5	4	F	n
5	0	5	6	F	n
6	0	6	4	F	n
6	0	6	5	F	n
1	1	2	1	T	n-1
1	1	2	4	F	n-1

Example: Switch 3

Compare message.root = 1 and self.root = 3. Update self.root to be message.root, self.distance to be message.distance + 1. Add message.origin to activeLinks, and set switchThrough to message.origin. Update neighbors with new messages

Topology



Before processing

Switch 3	
root	3
distance	0
activeLinks	
switchThrough	3

After processing

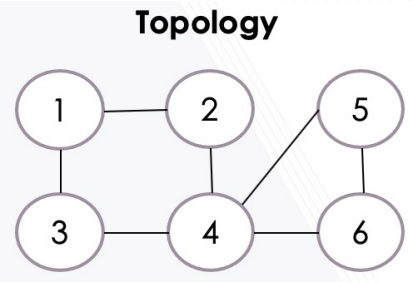
Switch 3	
root	1
distance	1
activeLinks	1
switchThrough	1

Message Queue

root	distance	Origin	Destination	paththrough	ttl
1	0	1	3	F	n
1	1	3	1	T	n-1
1	1	3	4	F	n-1

Georgia Tech
CREATING THE NEXT

Example: Final Spanning Tree



LOG FILE

1 - 2, 1 - 3
2 - 1, 2 - 4
3 - 1
4 - 2, 4 - 5, 4 - 6
5 - 4
6 - 4

Final Data Structures

switchID	root	distance	activeLinks	switchThrough
1	1	0	2,3	1
2	1	1	1,4	1
3	1	1	1	1
4	1	2	2,5,6	2
5	1	3	4	4
6	1	3	4	4

Key Assumptions for the Project

You should assume that all switch IDs are positive integers, and distinct.

These integers do not have to be consecutive and they will not always start at 1.

Tie breakers: All ties will be broken by lowest switch ID, meaning that if a switch has multiple paths to the root of the same length, it will select the path through the lowest id neighbor. For example, assume switch 5 has two paths to the root, through switch 3 and switch 2. Assume further each path is 2 hops in length, then switch 5 will select switch 2 as the path to the root.

Combining points one and two above, there is a single distinct solution spanning tree for each topology.

Key Assumptions for the Project

You can assume all switches in the network will be connected to at least one other switch, and all switches are able to reach every other switch. This is true of all starting AND ending Topologies.

You can assume that there will be no redundant links and there will be only 1 link between each pair of connected switches.

Note that when a switch deactivates a port, this port is not discarded. While the switch treats it as inactive, it can still be communicated with during the simulation.

Note that the above *examples* are only snippets of the process are are not meant to be used as a full list of rules to implement (that is what the project documentation is for). If you still have questions about the process, please post them on Ed Discussion.

CONCLUSION

Read the project description, linked materials, Ed Discussion posts and provided code
Attend TA chats as needed (there are a few every week that the project is open)
Take time to implement your solution

A good approach can be to tackle the problem in stages, and along the way either use print statements or a debugging tool to see the messages passed between switches, and to see values of switch member variables as they update when the switches together build the spanning tree for a topology by sharing information through message passing.

Incrementally add perhaps small pieces of your code at the Switch.py TODO placemarks, and observe how your code behaves. Gradually implement the Switch data structure, initial message passing, logic to process messages and code to write the logfile output according to the assignment requirements.

Sample.py is a good place to start testing your code because it is so simple. NoLoopTopo.py adds additional complexity that might challenge your initial code, and you will find even more demands on your solution from TailTopo.py and ComplexLoopTopo.py. The more topologies you can test your solution on, the more confidence you can have that your code addresses all the corner cases.

Good Luck!