Project 2 Spanning Tree Protocol

INTRODUCTION BY HEAD TA STACIA STOKES

View the Presentation

Click here to view the associated introduction video

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.
- This project is different from the previous project in that we're not running the simulation using 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.

*Layer 2 = Data Link Layer (Know all the OSI Model)

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

Time: 1:45

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 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.
  - Create and send messages in Switch.py by declaring a message as msg = Message(claimedRoot, distanceToRoot, originID, destinationID, pathThrough)

Time: 4:29

Project Files cont.

- 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.

Project Files – Switch.py

- data structure : keep track of the spanning tree. The collection of active links across all switches 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. It does not have an overall view of the tree as a whole.
- 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).
- More variables may be helpful to track data needed to build the spanning tree.
- The switches are trying to learn the root, or the switch with the lowest id and the path to the switch. To track the path to the root, each switch may need to know which neighbor it goes through to get there and the distance of the path to the root. To output the spanning tree, the switch also needs to know whether it is on the path its neighbor takes to get to the root.

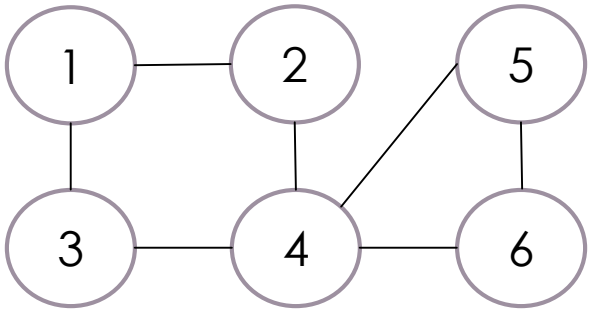
Time: 6:15

Project Files – Switch.py – Processing Messages

- Implement the Spanning Tree Protocol by writing code that sends the initial messages to neighbors of the switch, and processes a message from an immediate neighbor
- The messages are processed as a FIFO queue
- As each switch processes messages, it compares the received message data to the data in its data structure to build the spanning tree
- The switches do not need to push or pop on the FIFO queue since Topology.py does this for the switch as each switch calls send_msg
- Write a logging function that is specific to your particular data structure
- Take the time now to read all of the project files, starting and ending with switch.py and possibly reading message.py and topology.py twice as well.

Time: 8:38

Example: Topology, Data Structure, Initial Messages



Data Structure

Switch X	
root	X
distance	0
activeLinks	
switchThrough	X

Time: 11:07

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

*# **distance** = the distance from the origin to the root node*

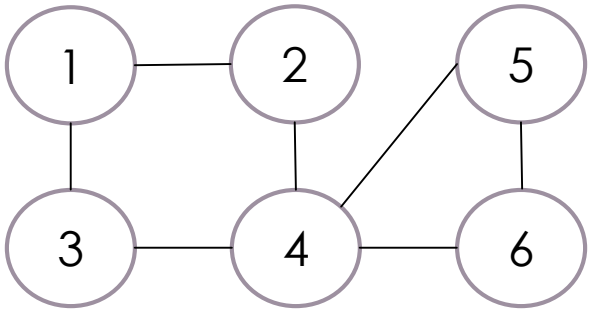
*# **origin** = the ID of the origin switch*

*# **destination** = the ID of the destination switch*

*# **pathThrough** = Boolean value indicating the path to the claimed root from the origin passes through the destination*

Root	Distance	Origin	Destination	paththrough

Example: Topology, Data Structure, Initial Messages



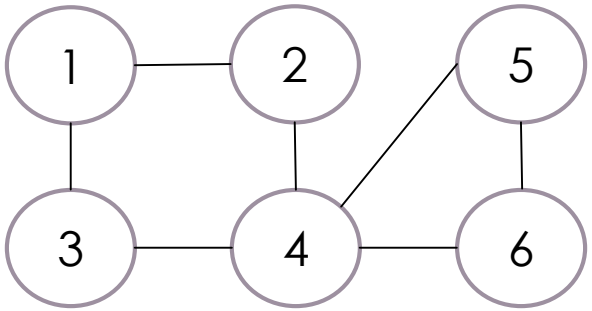
Data Structure

Switch 1	
root	1
distance	0
activeLinks	
switchThrough	1

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

Time: 17:42

Example: Topology, Data Structure, Initial Messages



Data Structure

Switch 2	
root	2
distance	0
activeLinks	
switchThrough	2

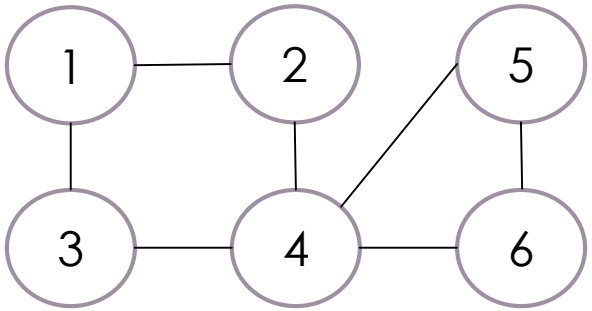
Root	Distance	Origin	Destination	paththrough
1	0	1	2	F
1	0	1	3	F
2	0	2	1	F
2	0	2	4	F

Time: 19:10

Example: Topology, Data Structure, Initial Messages

- (omitting initial messages for switches 3, 4, 5, 6)

Example: Topology, Data Structure, Initial Messages



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	3	F
4	0	4	2	F
4	0	4	5	F
4	0	4	6	F
5	0	5	4	F
5	0	5	6	F
6	0	6	4	F
6	0	6	5	F

- This is the FIFO message queue at the end of sending the initial messages.
- Note that you may see a different order. This is a simplification of the processing.

Time: 20:13

Example: process_message(self, message)

- Compare message.root = 1 and self.root
- Update self.root to be message.root, self.distance to be message.distance+1. add message.origin to activeLinks, and set pathThrough to message.origin
- Which neighbors do you send_msg?

Before process

Switch 2	
root	2
distance	0
activeLinks	
switchThrough	2

After process

Switch 2	
root	1
distance	1
activeLinks	1
switchThrough	1

Time: 20:49

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	5	F
4	0	4	6	F
5	0	5	4	F
5	0	5	6	F
6	0	6	4	F
6	0	6	5	F
1	1	2	1	T
1	1	2	4	F

Example: process_message(self, message)

- 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 pathThrough to message.origin
- send_msg to which neighbors?

Before process

Switch 3	
root	3
distance	0
activeLinks	
switchThrough	3

After process

Switch 3	
root	1
distance	1
activeLinks	1
switchThrough	1

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

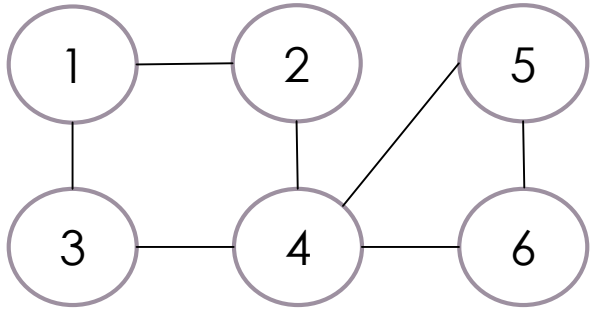
Time: 25:48

Example: process_message(self, message)

- (omitting processing many messages)

Example: process_message(self, message)

- **These is an example message that may not be found in the queue**



Root	Distance	Origin	Destination	paththrough
1	2	4	6	F

- Compare message.root = 1 and self.root = 1
- Compare message.distance+1 = 3 and self.distance = 4
- No update to self.root
- Update self.distance, self.activeLinks, self.switchThrough
- send_msg to which neighbors?

Before process

Switch 6	
root	1
distance	4
activeLinks	5
switchThrough	5

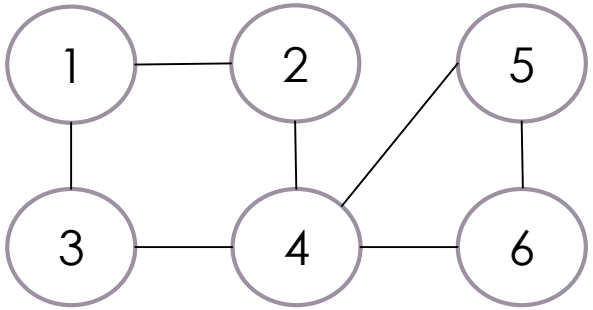
After process

Switch 6	
root	1
distance	3
activeLinks	4, 5?
switchThrough	4

Time: 26:21

Example: process_message(self, message)

- **These is an example message that may not be found in the queue**



Root	Distance	Origin	Destination	paththrough
1	3	6	4	T

- Compare message.root = 1 and self.root =1
- Compare message.distance+1 = 4 and self.distance = 2
- No update to self.root or self.distance but pathThrough = True, update self.activeLinks
- send_msg to which neighbors?

Before process

Switch 4	
root	1
distance	2
activeLinks	2
switchThrough	2

After process

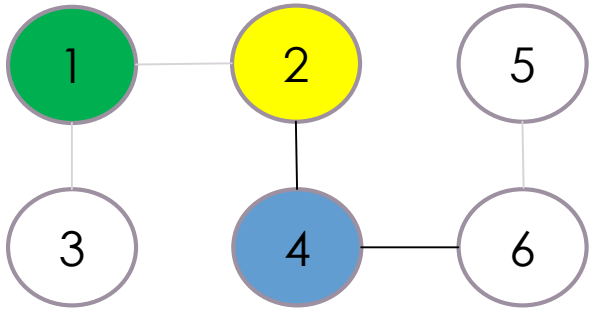
Switch 4	
root	1
distance	2
activeLinks	2,6
switchThrough	2

Time: 30:10

Conceptualizing a Spanning Tree from Switch 4's Perspective

- **Conceptionally, this is how Switch 4 currently views the Spanning Tree**
- **At some point, Switch 4 needs to receive a message from 5 with pathThrough = True to activate that Link and add 5 to its activeLinks**
- **Switch 4 doesn't know anything about the greyed out links because they're too far away**

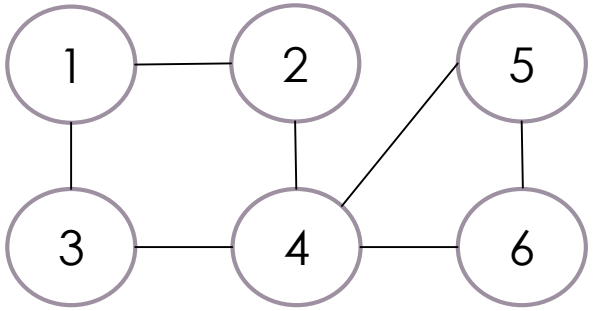
Switch 4	
root	1
distance	2
activeLinks	2,6
switchThrough	2



Time: 31:38

Example: process_message(self, message)

- **These is an example message that may not be found in the queue**



Root	Distance	Origin	Destination	paththrough
1	1	2	4	F

- Compare message.root = 1 and self.root =1
- Compare message.distance+1 = 2 and self.distance = 2
- Tiebreaker:

message.origin < self.switchThrough

- Update self.activeLinks, switchThrough
- send_msg to which neighbors?

Before process

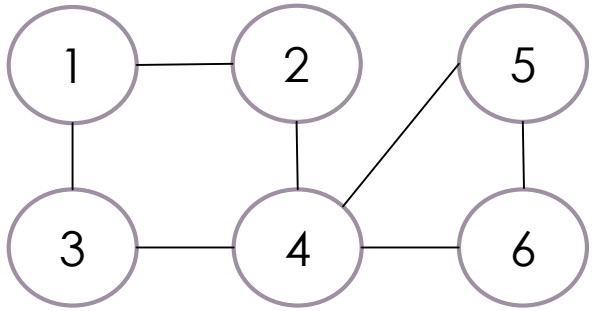
Switch 4	
root	1
distance	2
activeLinks	2,3,6
switchThrough	3

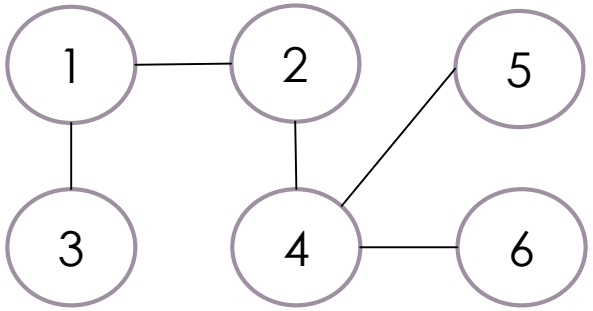
After process

Switch 4	
root	1
distance	2
activeLinks	3,6
switchThrough	2

Time: 34:02

Example – Final Spanning Tree





- Final Active Links/Spanning Tree/Data Structures

SwitchID	Root	ActiveLinks	switchThrough
1	1	2,3	1
2	1	1,4	1
3	1	1	1
4	1	2,5,6	2
5	1	4	4
6	1	4	4

Time: 35:56

Key Assumptions

- 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 at 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 and disable forwarding on the link to switch 3.
- Combining points one and two above, there is a single distinct solution spanning tree for each topology.

Time: 36:19

Key Assumptions

- 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.
- You can assume that there will be no redundant links and there will be only 1 link between each pair of connected switches.
- You can assume that the topology given at the start will be the final topology and there won't be any changes as your algorithm runs (i.e adding a new switch).
- Note that when a switch deactivates/blocks a port, this port is not completely discarded. While the switch treats it as inactive, it will still be communicated with during the simulation.

Time: 37:10

Conclusion

- Read the rest of the Project Description
- Take your time
- Good luck!