ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία

Θέμα: "Υλοποίηση και αξιολόγηση πρωτοκόλλων δικτύωσης με χρήση της OpenFlow τεχνολογίας"

Ραδουνισλής Αναστάσιος

Επιβλέπων καθηγητής : Κοράκης Αθανάσιος (Επίκουρος καθηγητής)

Συνεπιβλέπων καθηγητής : Αργυρίου Αντώνιος (Λέκτορας καθηγητής)

Βόλος, 2016

UNIVERSITY OF THESSALY

Department of Electrical and Computer Engineering

Thesis

Title: "Implementation and evaluation of network protocols using OpenFlow technology"

Radounislis Anastasios

Supervisor professor: Korakis Athanasios

Co-supervisor professor: Argyriou Antonios

Volos , 2016

Ευχαριστίες

Η παρούσα διπλωματική εργασία πραγματοποιήθηκε στα πλαίσια του προπτυχιακού προγράμματος σπουδών του τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών για την απόκτηση του αντίστοιχου διπλώματος. Αρχικά θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή και λέκτορα του τμήματος κ. Κοράκη Αθανάσιο που μου έδωσε την ευκαιρία να ασχοληθώ με το θέμα. Επίσης , θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα του τμήματος Χούμα Κωνσταντίνο , για τον χρόνο του, τις συμβουλές του και την καθοδήγηση του καθ'όλη την διάρκεια της εργασίας. Τέλος , θα ήθελα να ευχαριστήσω την οικογένεια μου και τους φίλους μου , για την στήριξη που μου παρείχαν όλα αυτά τα χρόνια.

Περιεχόμενα

Περίληψη

Σκοπός της διπλωματικής εργασίας είναι η υλοποίηση ενός Openflow Controller ο οποίος θα λύνει το switching loop πρόβλημα και θα καθιστά δυνατή την επικοινωνία ανάμεσα στους κόμβους ακόμη και εάν υπάρχει κύκλος στην τοπολογία. Αυτό γίνεται χρησιμοποιώντας τον αλγόριθμο του Kruskal για την εύρεση του ελάχιστου επικαλύπτον δέντρου και με απόρριψη των πακέτων που κινούνται στις ακμές οι οποίες δεν ανήκουν σε αυτό. Σαν βάρος ακμών χρησιμοποιούμε 3 διαφορετικές τιμές: 1)το delay κάθε ακμής, 2) το bandwidth κάθε ακμής και 3) τον λόγο bandwidth/delay.
Στη συνέχεια τρέξαμε τον controller σε 2 διαφορετικές τοπολογίες και μετρήσαμε το συνολικό μέσο RTT και το συνολικό μέσο Bandwidth για κάθε βάρος.

Abstract

The purpose of this thesis is the development of an OpenFlow controller that solves the switching loop problem and allows the communication between the nodes in non loop free topologies. This is achieved thanks to Kruskal's algorithm finding the minimum spanning tree of our topology and by dropping the packets a switch receives in a port that doesn’t belong in the tree. As weight we use 3 different values: 1) delay of links , 2) bandwidth of links and 3) the result of Bandwidth/delay of each link.
Then we run the controller on top of 2 different topologies and we measure the total average RTT and the total average Bandwidth for each weight.

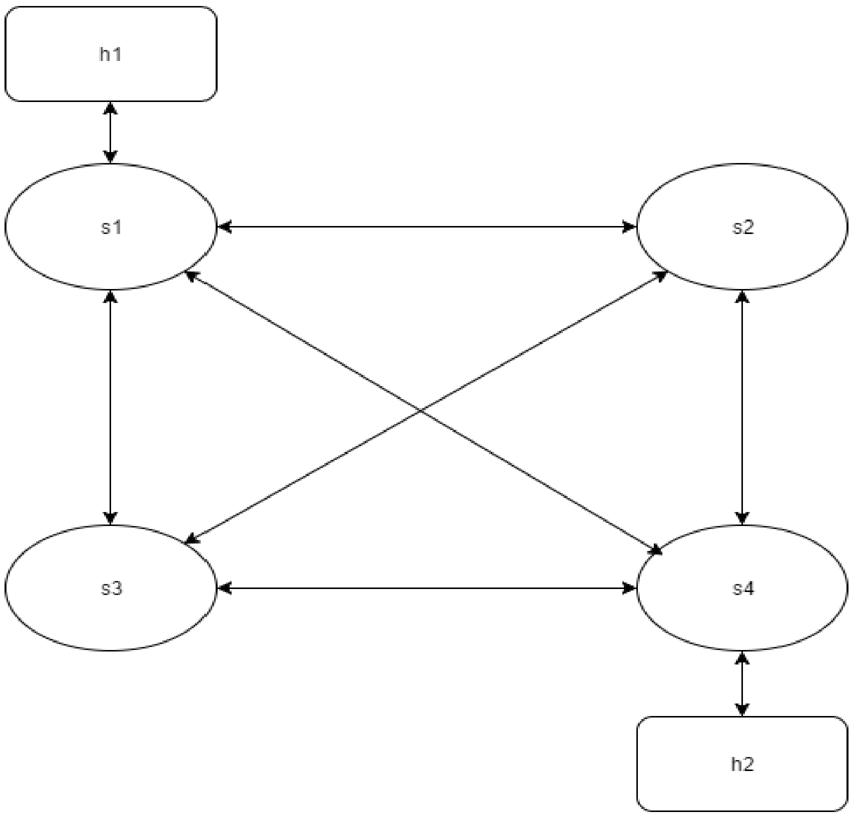
1. Introduction

1.1 Switching Loop problem

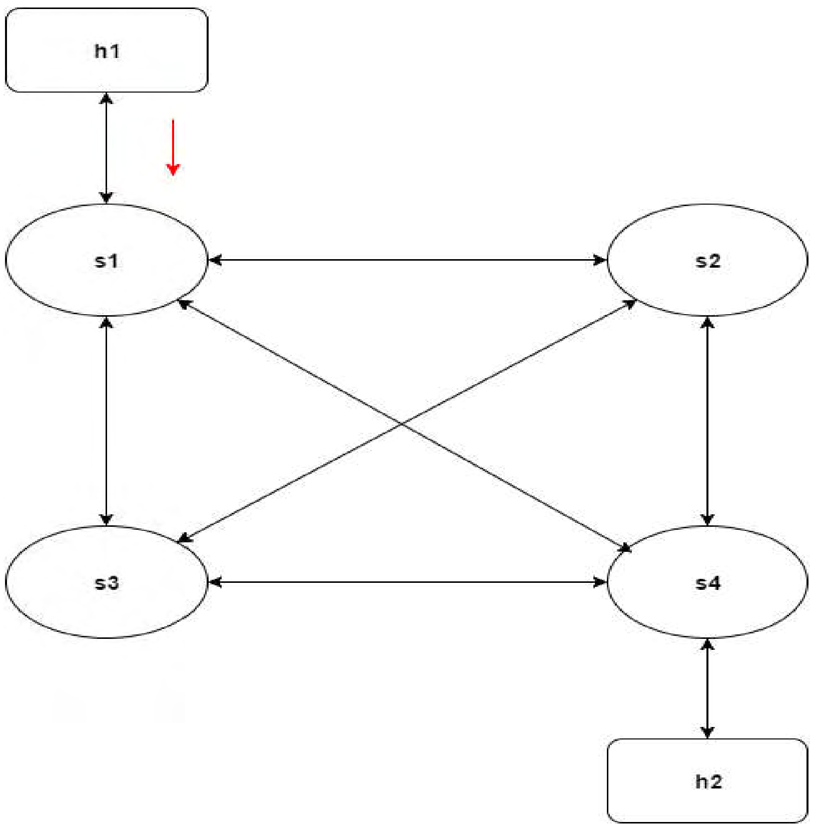
A switching loop or bridge loop occurs in computer networks when there is more than one layer 2 (OSI model) path between two endpoints. The loop creates broadcast storms as broadcasts and multicasts are forwarded by switches out every port, the switches will repeatedly rebroadcast the broadcast messages flooding the network. Since the Layer 2 header does not support a time-to-live (TTL) value, if a frame is sent into a looped topology, it can loop forever.

1.2 Example

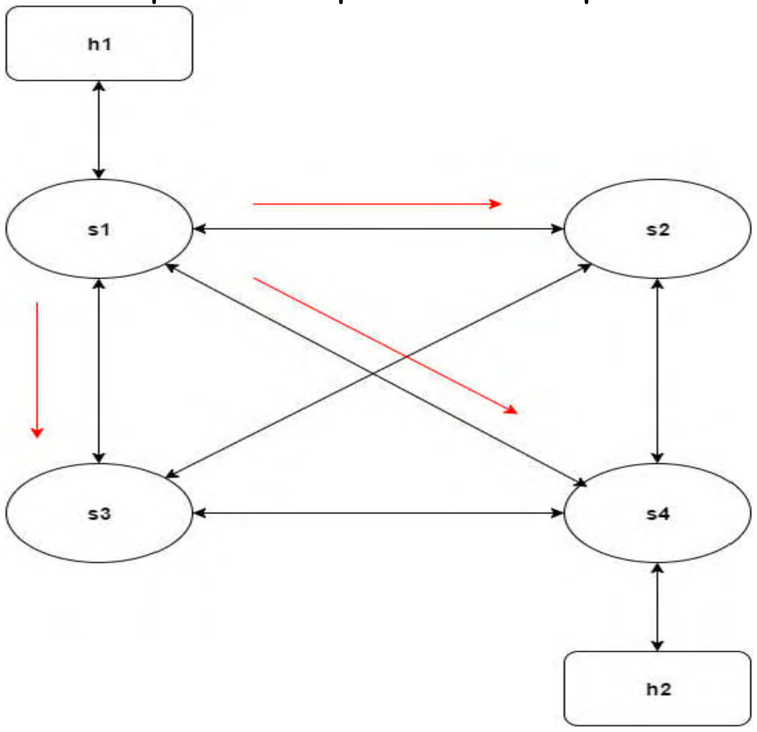
Let's examine the topology downwards. We assume that flow tables of all switches are empty.



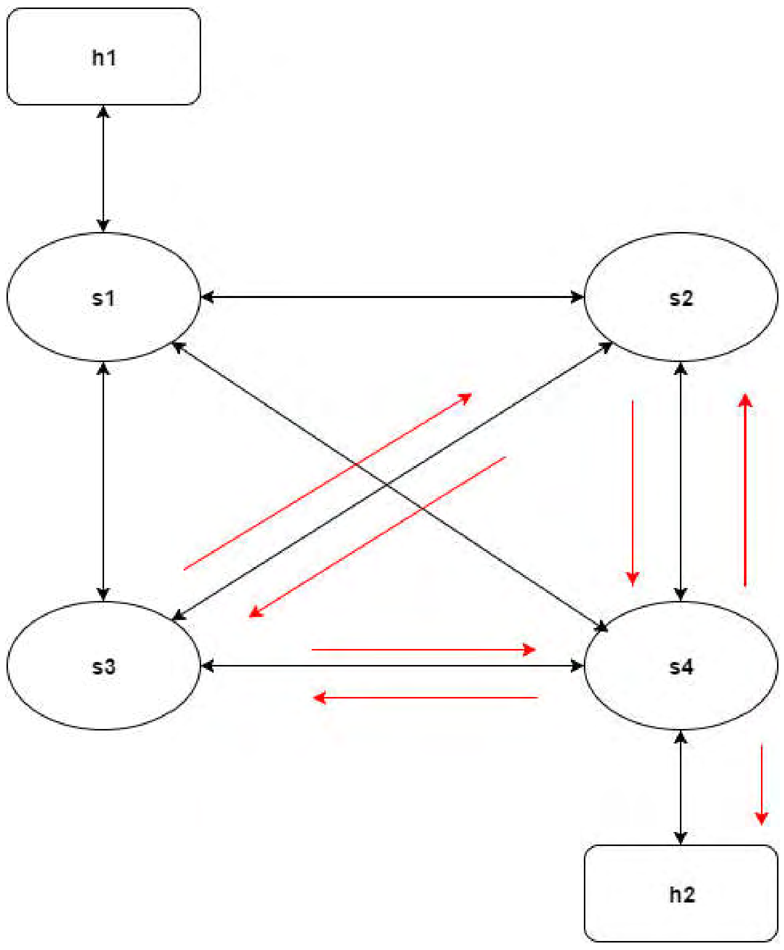
1. Full mesh square topology



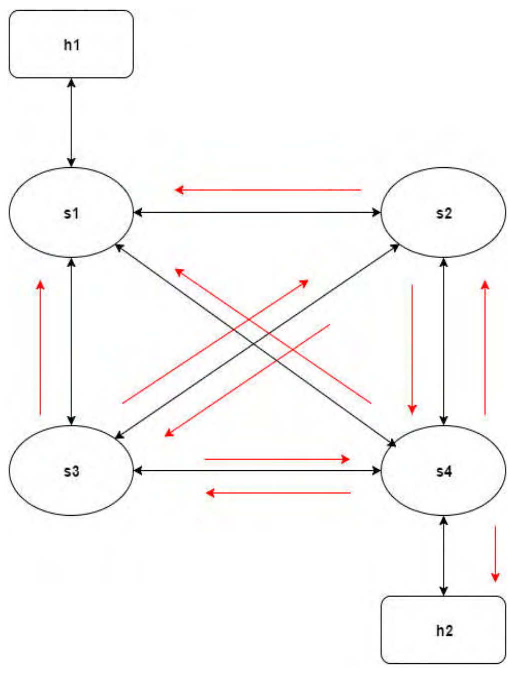
We are going to see what happens when h1 sends a packet to h2. First s1 will receive the packet, because of unknown destination MAC address the switch will forward the packet to all ports except the source port.

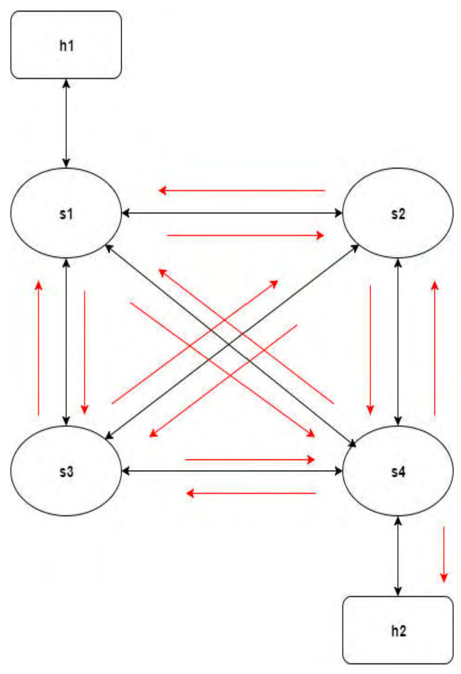


The switches will receive the packet and acting like s1 they are going to forward it to all ports. S2 will forward the packet to s3,s4, s3 to s2, s4 and s4 to s2,s3.



After that once more switches will forward again the same packet which now exists in all links.
As a result the packet will start circulating the network in a loop and since it doesn't have a TTL value (as Layer 2 packet) it will loop forever.
Also there is additional overhead because h2 will receive multiple copies of the same packet.





The same problem occurs for multicasts and broadcasts and is knows as broadcast storm . The loop creates broadcast storms as broadcasts and multicasts are forwarded by switches out every port and switches will repeatedly rebroadcast the messages flooding the network.

1.2 Spanning Tree Protocol

The Spanning Tree Protocol is a network protocol that ensures a loop-free topology for any bridged Ethernet local area network. The basic function of STP is to prevent bridge loops and esnuring broadcast radiation.
Spanning Tree consists of the following steps:

> root bridge election based on bridge ID

> root port election based on the lowest path cost to root port

> designated port election

> alternative (blocking) port election

1.3 Our Solution

To avoid switching loop we implement an OpenFlow controller that solves that problem using Kruskal's Algorithm. More precisely our controller learns the topology by forcing the switches to communicate to each other with "discovery" packets and then when a switch asks the controller what to do with a packet , the controller using Kruskal's algorithm , discovers the minimum spanning tree of the topology and blocks the ports that are responsible for loops. This happens only one time in stable topologies, when the first non-"discovery" packet arrives or when there is a change in the topology, for example, a switch connects or disconnects to the network.

Now let's explain OpenFlow and other tools that we use for the experiments and we will see how the controller exactly works afterwards.

2. Tools

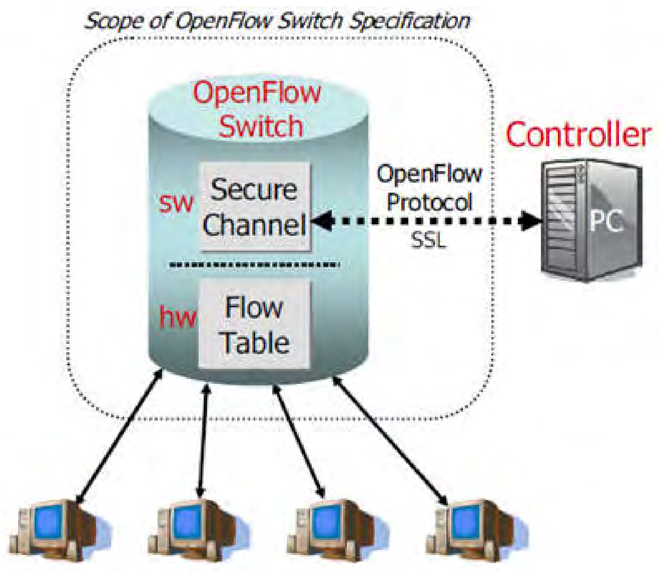
2.1 OpenFlow

OpenFlow is an open standard that enables researchers to run experimental protocols in networks we use every day. OpenFlow is added as a feature to commercial Ethernet switches, routers and wireless access points and provides a standarized hook to allow researchers to run experiments, without requiring vendors to expose the internal workings of their network devices. OpenFlow is currently being implemented by major vendors, with OpenFlow-enabled switches now commercially available

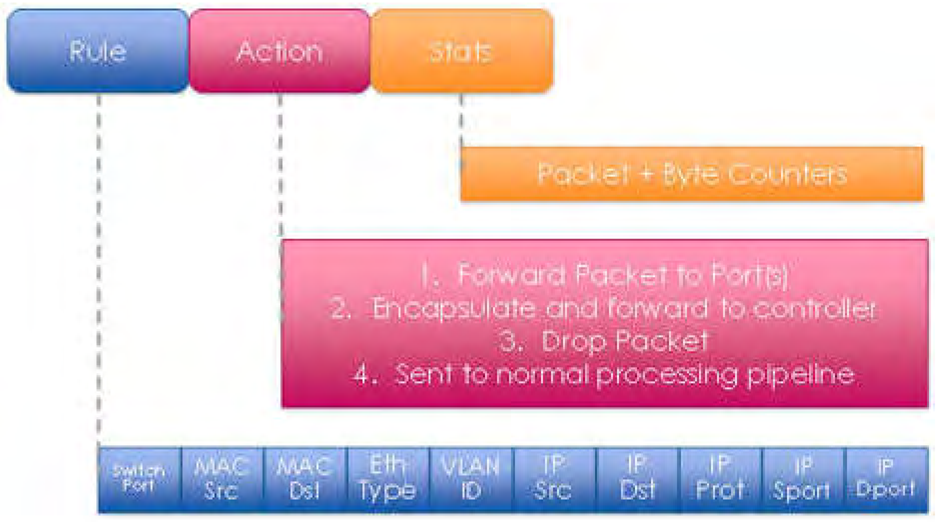
The original concept for OpenFlow begun at Stanford University in 2008. By December 2009, Version 1.0 of the OpenFlow switch specification was released. Since its inception, OpenFlow has been managed by the Open Networking Foundations (ONF), a user-led organization dedicated to open standards and SDN adoption.

OpenFlow is considered one of the first software-defined networking (SDN) standards. It originally defined the communication protocol in SDN enviroments that enables the SDN controller to directly interact with the forwarding plane of network devices such as switches and routers.
To work in an OpenFlow environment, any device that wants to communicate to an SDN Controller must support the OpenFlow protocol. Through this interface, the SDN Controller pushes down changes to the switch/router flow-table allowing network administrators to partition traffic , control flows for optimal
performance , and start testing new configurations and applications.

An OpenFlow switch is a software program or hardware device that forwards packets in a software-defined networking (SDN) enviroment. OpenFlow switches are either based on the OpenFlow protocol or compatible with it. In a conventional switch, packet forwarding (data plane) and high-level routing (control plane) occur on the same device. In software-defined networking, the data plane is decoupled from the control plane. The data plane is still implemented in the switch itself but the control plane is implemented in software and a separate SDN controller makes high-level routing decisions. The switch and controller communicate by means of the OpenFlow protocol.



Εικόνα 2. An OpenFlow switch

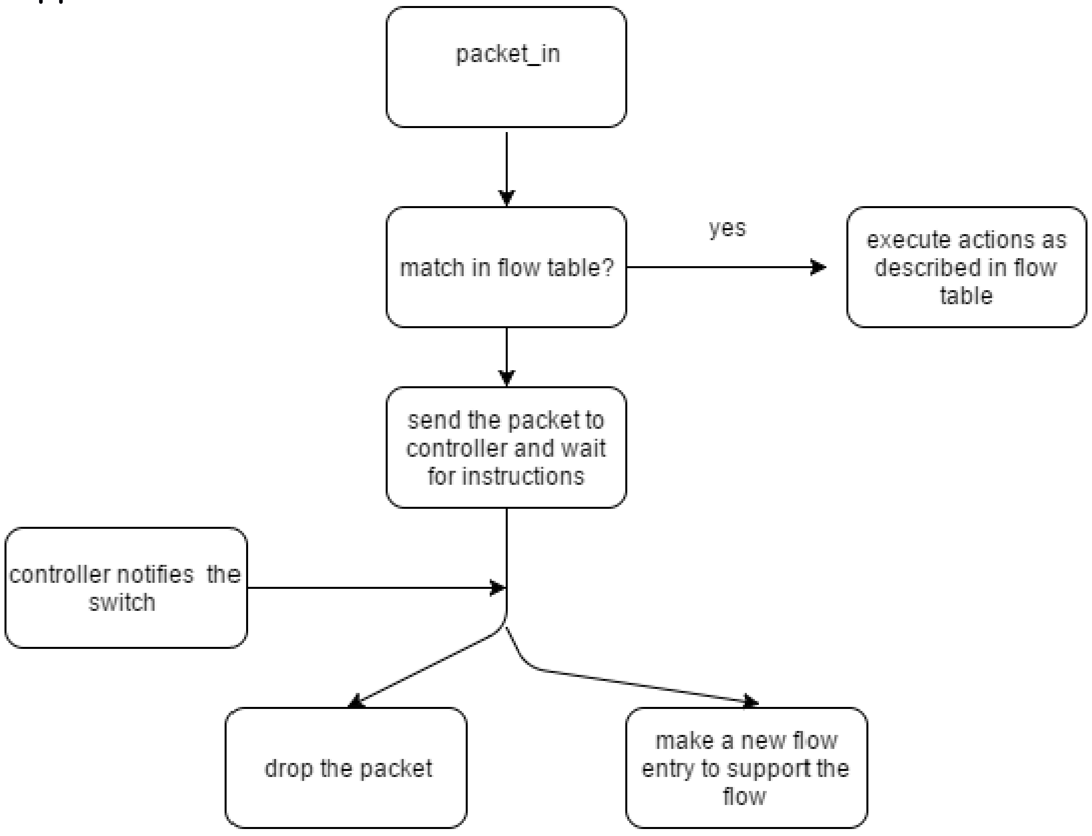


Εικόνα 3. A rule in flow-table

OpenFlow switches must be capable of forwarding Ethernet frames based in rules that are stored in one or more flow-tables. Each flow table entry contains:

- Header fields to match against packets
- Counters to update for matching packet
- Actions to apply to matching packets

When a packet arrives at the OpenFlow switch, the header fields are compared to flow table entries. If a match is found, the packet is either forwarded to specified port(s) or dropped depending on the action stored in the flow table. When an OpenFlow switch receives a packet that does not match the flow table entries, it encapsulates the packet and sends it to the controller. The controller then decides how the packet should be handled and notifies the switch to either drop the packet or make a new entry in the flow table to support the new flow.


packet_in
match in flow table?
yes
execute actions as described in flow table
send the packet to controller and wait for instructions
controller notifies the switch
drop the packet
make a new flow entry to support the flow

Εικόνα 4. General flow chart

The controller is responsible for maintains all of the network rules and distributes the appropriate instructions for the network devices. In other words, the OpenFlow controller is responsible for determining how to handle packets without valid flow entries, and it manages the switch flow table by adding and removing flow entries over the secure channel using OpenFlow protocol.

OpenFlow controllers can operate in different modes depending on:

- **Location**: we have the choice of **centralized** configuration, where one controller manages and configures all the switches , or **distributed configuration** such as one controller for each switch
- **Flow** : we can have one flow entry for each flow (**flow routing**) or one flow entry for large groups of flows.
- **Behavior**: Here there are two choices.
    - **Reactive**: The controller is designed initially to do nothing until it receives the first message
    - **Proactive**: Rather than reacting to a packet an OpenFlow controller could populate the flow tables ahead of time for all traffic matches that could come into the switch.

For our experiments we chose a centralized configuration with flow routing and a reactive behavior.

2.2 Mininet

Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support

OpenFlow for highly flexible custom routing and Software-Defined Networking.
Mininet supports research, development, learning, prototyping, testing, debugging, and any other tasks that could benefit from having a complete experimental network on a laptop or other PC.

Mininet:

- Provides a simple and inexpensive network testbed for developing OpenFlow applications
- Enables multiple concurrent developers to work independently on the same topology
- Supports system-level regression tests, which are repeatable and easily packaged
- Enables complex topology testing without the need to wire up a physical network
- Includes a CLI that is topology aware and OpenFlow-aware, for debugging or running network-wide tests
- Supports arbitrary custom topologies, and includes a basic set of parametrized topologies
- Is usable out of the box without programming
- Provides a straightforward and extensible Python API for network creation and experimentation

Mininet provides an easy way to get correct system *behavior* (and, to the extent supported by your hardware, performance) and to experiment with topologies.

Mininet networks run *real code* including standard Unix/Linux network applications as well as the real Linux kernel and network stack (including any kernel extensions which you may have available, as long as they are compatible with network namespaces.)

Because of this, the code you develop and test on Mininet, for an OpenFlow controller, modified switch, or host, *can move to a real system with minimal changes*, for real-world testing, performance evaluation, and deployment. Importantly this means that a design that works in Mininet can usually move directly to hardware switches for line-rate packet forwarding.

2.3. Trema

Trema is an OpenFlow controller programming framework that provides everything needed to create OpenFlow controllers in **Ruby**. It provides a high-level OpenFlow library and also a network emulator that can create OpenFlow-based networks for testing on your PC. This self-contained environment helps streamlines the entire process ofdevelopment and testing

Goals for Trema project:

- Provide good quality OpenFlow controller platform to researchers/developers and a continuous development, maintenance, bug fixes and user support from the project team.
- Researchers develop their own controllers on top of Trema and contribute to the community.

Ruby is an object-oriented programming language, written in C and that combine some of the best features of C, Perl and Python. Is a portable programming language and runs under GNU/Linux as well as DOS, MS Windows and MAC.

3. The Controller

In this chapter we will describe in details how our controller works.
The purpose of our controller is to solve the switching loop problem and achieve communication in non loop free topologies. So, starting, the controller must learn the topology. This is achieved by "discovery packets" that the controller forces the switches to send to their neighbors. Each switch sends to it's neighbors a packet with it's ID and a string "disc_packet". The controller provides special handling for these packets. They aren't forwarded to next switch. Each switch that receives a "discovery_packet" updates the global graph variable with an entry consisting sender's ID, receiver's ID and the receiver's port. ([sender_s id, receiver's id, message.in_port])

After little time, the controller has an overall view of the topology as a graph and knows the port numbers of a pair of switches that are neighbors.

Then the controller waits for the first packet to be delivered in a switch.

Taken as fact that the flow tables are empty in the beginning, when the switch receives the packet it will ask the controller what to do with it.

Same as flow tables, the forwarding database of the controller will also be empty so there is not an entry for the MAC destination address of the packet. Now the controller will order the switch to flood the packet out of all it's ports except the source port.

But before that, and here comes our contribution, the controller will apply Kruskal's algorithm in the topology to find it's minimum spanning tree.

Knowing the tree the controller knows which links are responsible for loops and it "blocks" the ports of each switch in the pair that consists the link.

For example if the link between (1,2) is not in the minimum spanning tree the controller will force switch 1 to drop the packets coming from the port that it connects with switch 2. Exactly the same goes for switch 2.

Now that we have a loop free topology there is no problem for a packet to loop in the network. So the switch floods the packet out of all it's ports except the source port.

Kruskal's Algorithm is used to find the minimum spanning tree of a graph(in our case of a network topology) with the least cost.

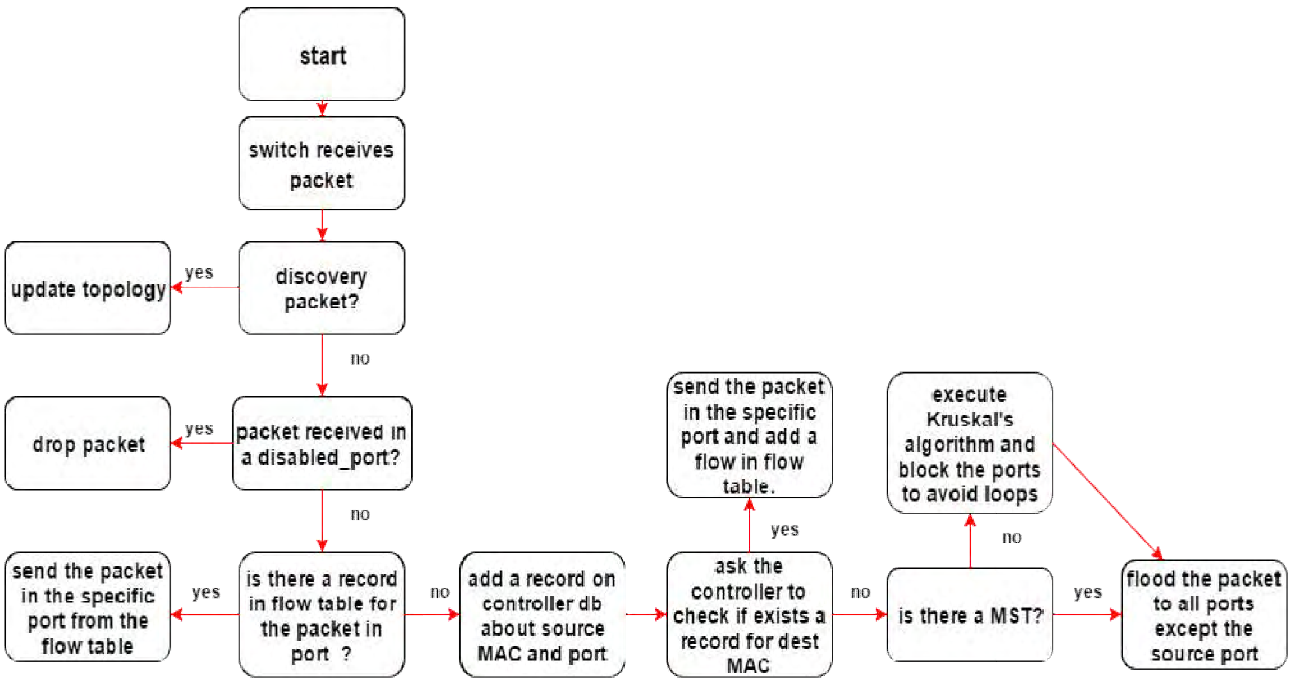
For our experiments we used three different values as weights

- Link delay
- Link bandwidth
- Bandwidth/delay value

Downwards there is a flow chart to help you understand better what happens when a switch receives a packet.

Also, it's good to know that switches in this experiment will act like an L2 switch. They will examine each packet, learn the source-port and associate it with the source MAC address.

If the destination MAC address of the packet is already associated with a port, the packet will be sent to the given port, else it will be flooded on all ports of the switch.



start
switch receives packet
discovery packet?
yes
update topology
no
packet received in a disabled_port?
yes
drop packet
no
is there a record in flow table for the packet in port ?
yes
send the packet in the specific port from the flow table
no
add a record on controller db about source MAC and port
ask the controller to check if exists a record for dest MAC
yes
send the packet in the specific port and add a flow in flow table.
no
is there a MST?
no
execute Kruskal's algorithm and block the ports to avoid loops
yes
flood the packet to all ports except the source port

The controller was built upon the multi-learning switch of Trema examples in Ruby programming language.

If there is a change in the topology (a new switch connects or a switch disconnects) the controller will run again Kruskal's algorithm to find the new MST.

4. Experiments

After the development of the controller we run several experiments and examine the overall performance of our topology depending on what we choose as a weight in Kruskal's algorithm.

As weight values we use, delay, bandwidth and the bandwidth/delay ratio of each link. These values are set in the python script we execute in Mininet to create the topology and we provide them manually to the controller.

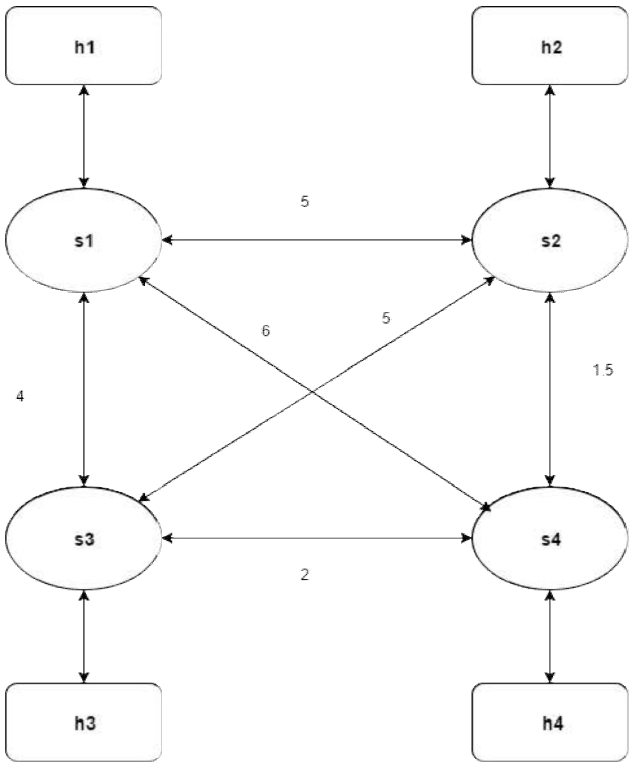
So the topologies are created in a Mininet VM using python. All nodes of each topology are wired connected. The Trema controller runs on the host OS. Host OS and Mininet VM are bridged connected.

4.1 Experiment 1.

In the first experiment we examine a full mesh square topology. Each switch is connected to a host

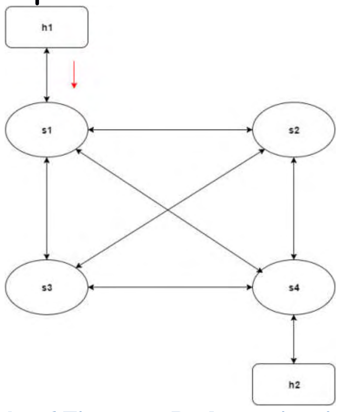
Πίνακας 1 Values of each link

Edges	Delay	Bandwidth	Bandwidth/ delay
(1,2)	5	2	0,4
(1,3)	4	1.5	0.375
(1,4)	6	3	0.5
(2,3)	5	2.5	0.5
(2,4)	1.5	0.5	0.33
(3,4)	2	1	0.5



Εικόνα 5. Full mesh square topology

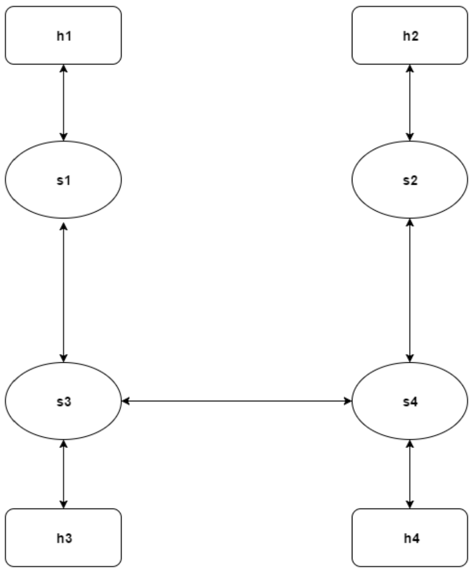
We will see what happens in the first case in details in which we choose delay as weight. For example h1 sends a ping request to h4. H1 will forward the packet to s1.



Εικόνα 6 First step. Packet arrives in s1

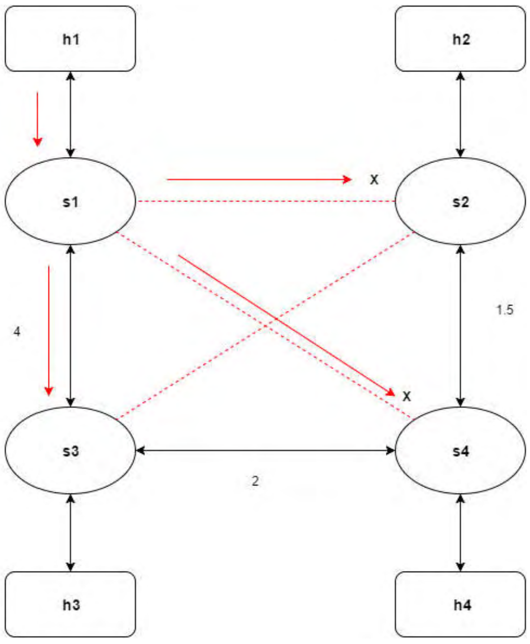
Because of the empty flow tables there will be no flow for the packet. Also there will be no match between destination's MAC address and a port, so this leads to a flood.

Before order the switch to flood the packet the controller will apply Kruskal's algorithm to the graph to remove. Below you can see the MST with delay as weight.

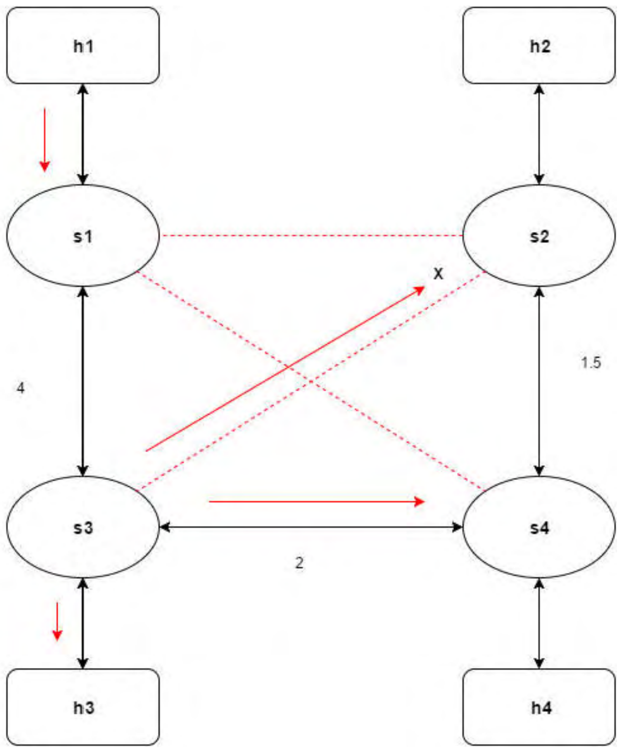


Εικόνα 7 Delay MST

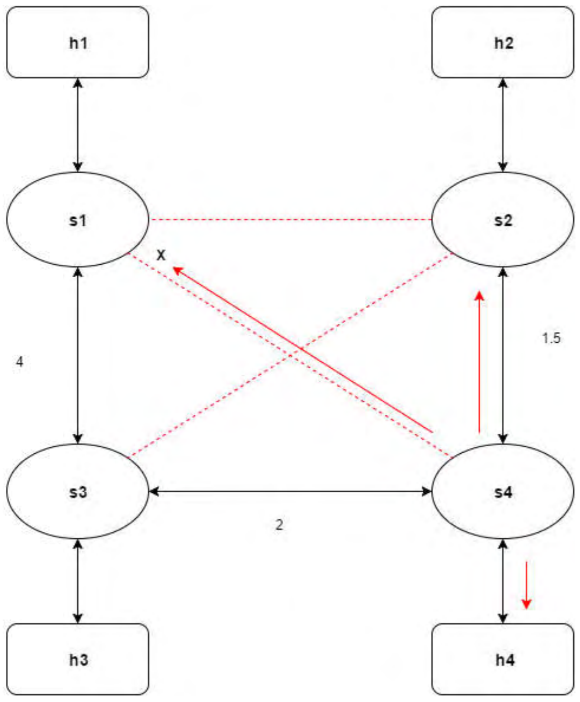
After finding the MST the controller will order the switch to flood the packet out of all it's ports except the source port.



S1 will flood the packet, but s2 and s4 will drop it because links (1,2) and (1,4) are not part of the MST. S3 will receive the packet and as s1 it will forward it out of all it's ports.



S2 will drop the packet again because link (3,2) does not include in the MST. S4 will receive the packet and same as s1 and s3 will flood the packet.

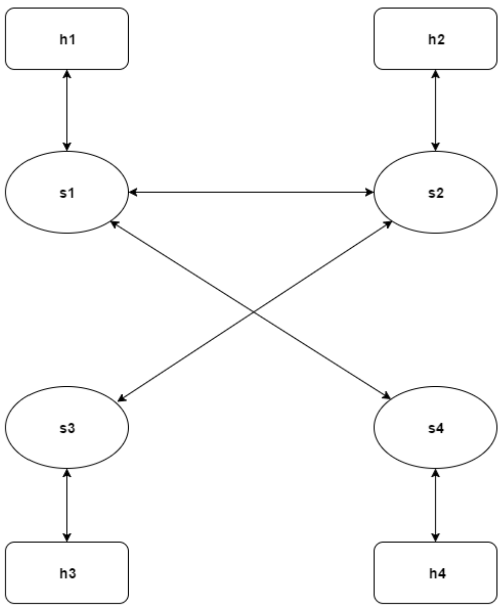


S1 will drop the packet, s2 will receive it and flood it to h2. Now the ping request will finally arrive in h4. In all the way down from s1 to s4 all switches have match the source port of the packet with the MAC address of h1.

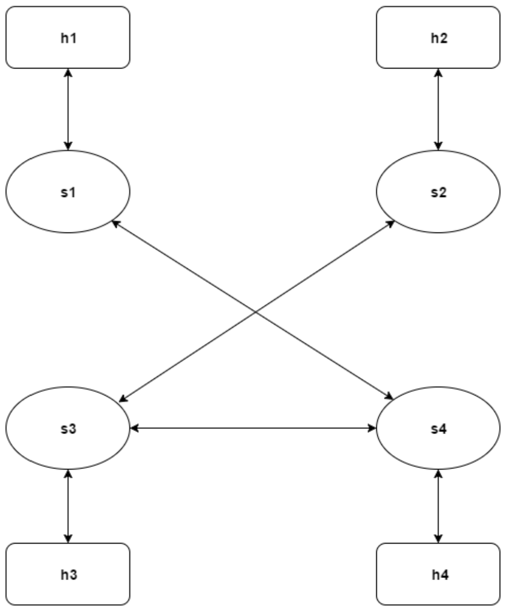
So as the ping reply has destination MAC the address of h1 the switches will know in which port to send the packet and there will be no flooding. Also flow entries will be added in each flow table. The ping reply will be forwarded to s4 from h4, then to s3, then to s1 and finally h1.

In case h2 sends a ping request to h3 the controller will not apply Kruskal's algorithm again, the MST is the same, so the packet will flood to the network.

Below there are the other 2 Minimum-Spanning-Trees.



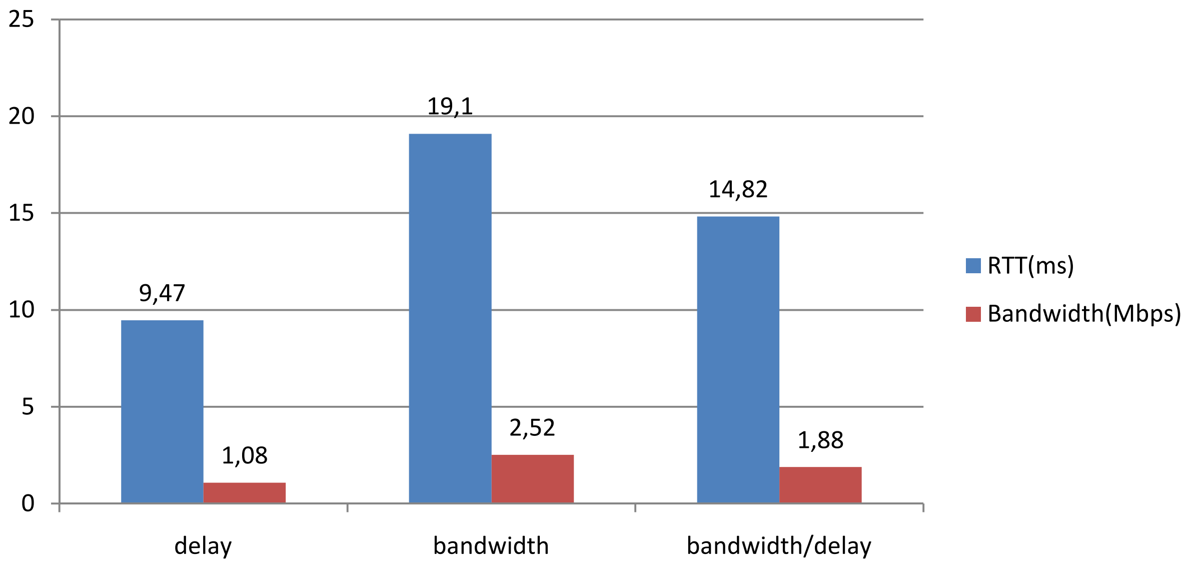
Εικόνα 8 Bandwidth MST



Εικόνα 9 Bandwidth/delay ratio MST

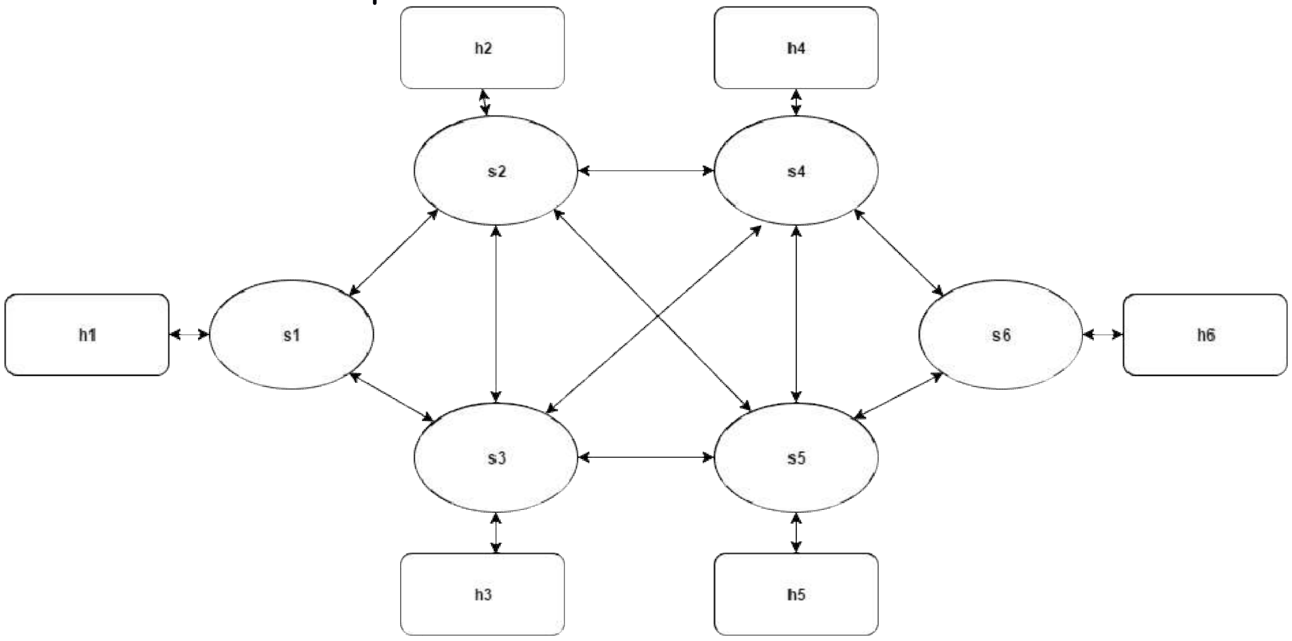
In each case we measure the total average Round-Trip-Time and the total average Bandwidth by execute ping and iperf between all hosts.

In the chart below we can see the results.

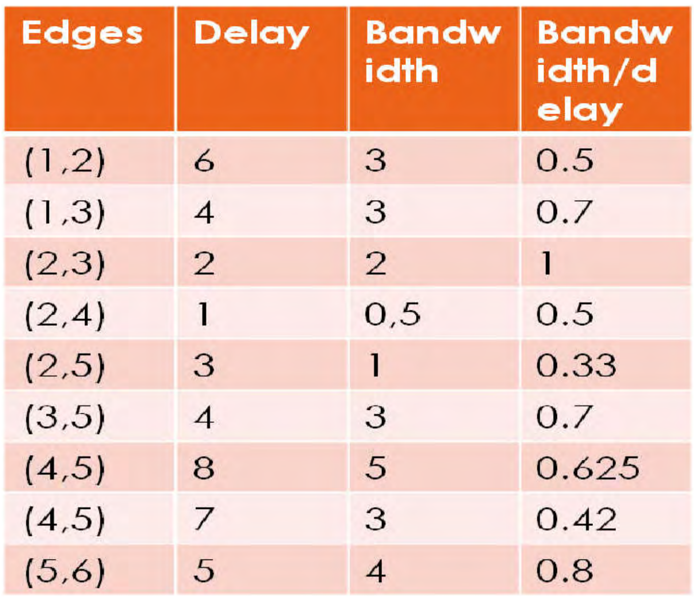


4.2 Experiment 2

For the second experiment we choose a 6-node mesh topology as shown in the picture below



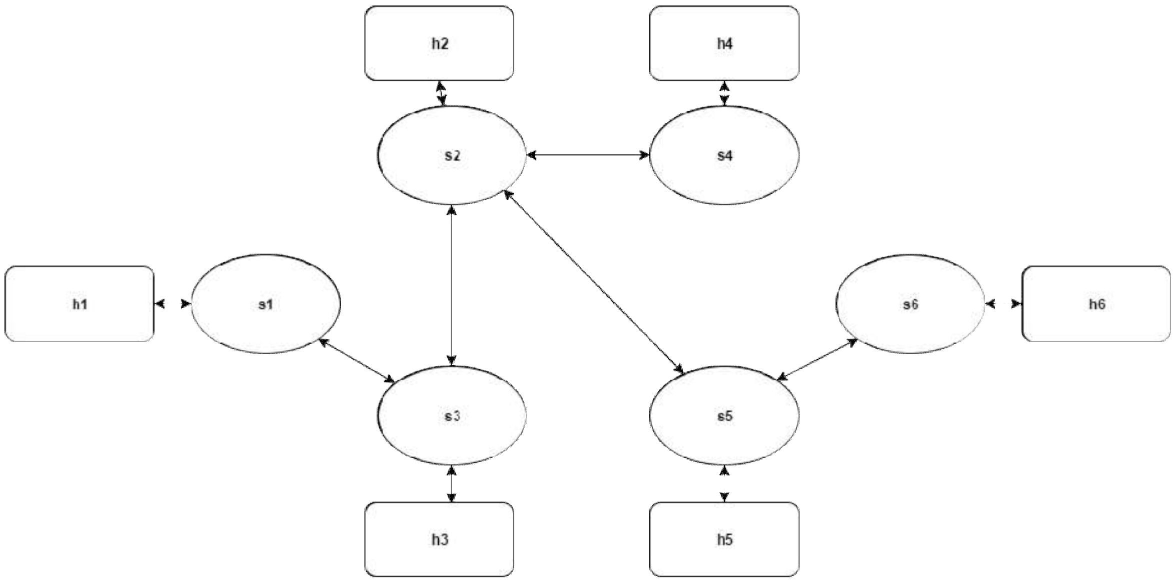
Εικόνα 10. 6-node mesh topology



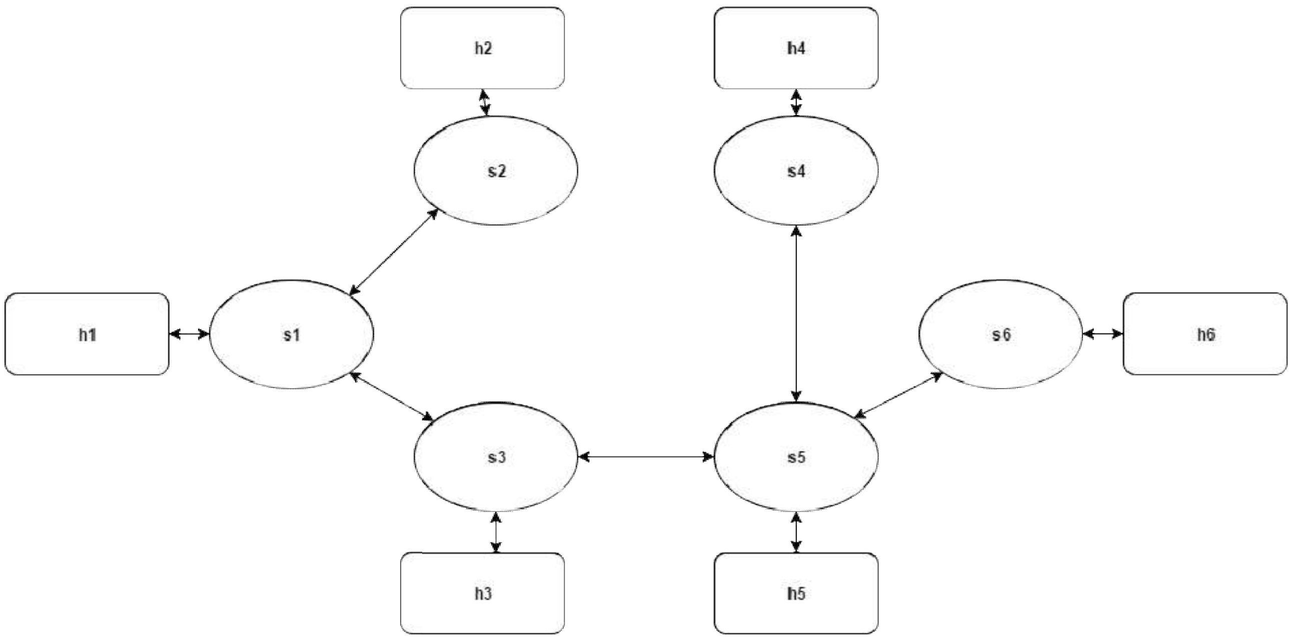
Edges	Delay	Bandwidth	Bandwidth/delay
(1,2)	6	3	0.5
(1,3)	4	3	0.7
(2,3)	2	2	1
(2,4)	1	0,5	0.5
(2,5)	3	1	0.33
(3,5)	4	3	0.7
(4,5)	8	5	0.625
(4,5)	7	3	0.42
(5,6)	5	4	0.8

Εικόνα 11. Edges of the 6 node topology

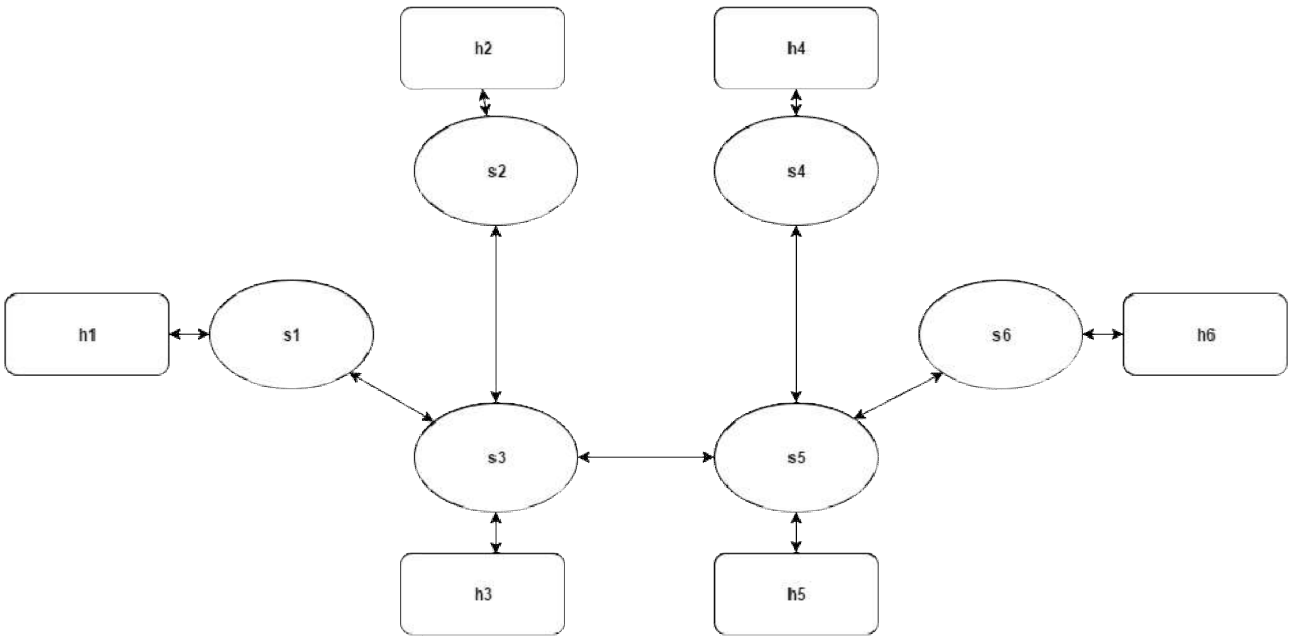
The 3 different Minimum-Spanning trees are following



Εικόνα 12. MST based on delay

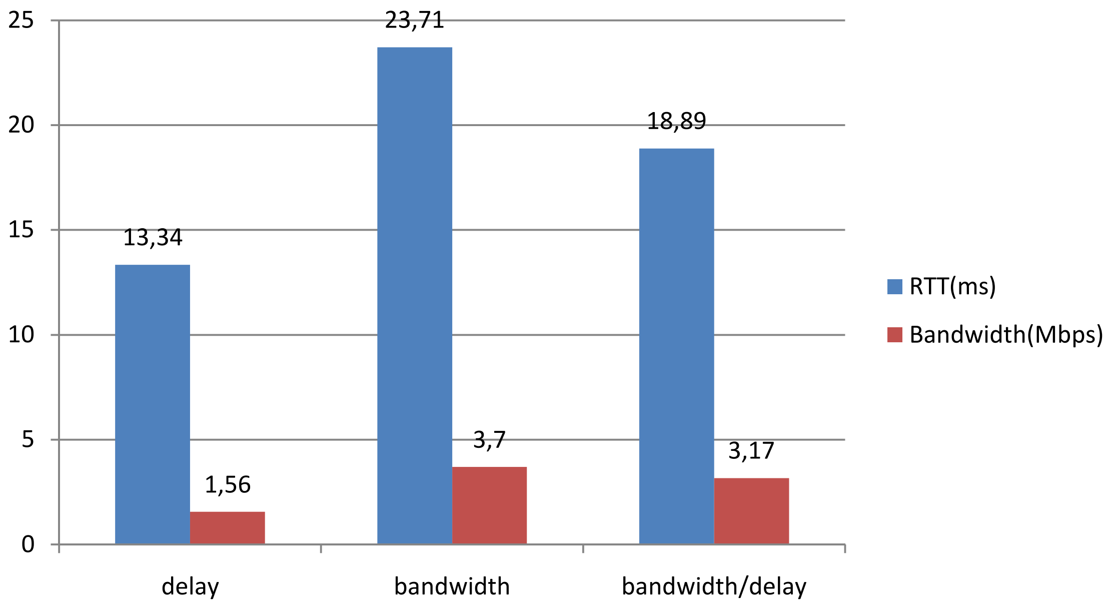


Εικόνα 13. MST based on Bandwidth



Εικόνα 14. MST based on bandwidth/delay ratio

In the chart below we can see the results.



4.3 Conclusion

As we can see, we have changes in network performance depending on what we choose as weight. If we want a fast network we should choose delay as weight, if we want a "fat" network and delay is not a problem we can choose Bandwidth as weight. If we want an average approach we can choose bandwidth/delay ratio as weight because in this case we achieve lower average RTT contrary to the Bandwidth case and better average Bandwidth contrary to the Delay case.

5. In the future

In the future we can apply our algorithm with different values as weight. Also we can measure how much time needs the controller to learn the entire topology and find it's Minimum Spanning Tree and we can compare with the time STP needs to converge.

References

[1] OpenFlow https://www.opennetworking.org/sdn-resources/openflow

[2] OpenFlow Switch http://searchsdn.techtarget.com/definition/OpenFlow-switch

[3] OpenFlow Controller http://searchsdn.techtarget.com/definition/OpenFlow-controller

[4] Switching Loop https://en.wikipedia.org/wiki/Switching_loop

[5] Spanning Tree Protocol https://en.wikipedia.org/wiki/Spanning_Tree_Protocol

[6] Mininet http://mininet.org/

[7] Trema http://www.fp7-ofelia.eu/assets/Uploads/201203xx-TremaTutorial.pdf

[8] Trema Github https://trema.github.io/trema/

[9] MultiLearning Switch https://github.com/trema/learning_switch/tree/develop/lib

[10] Open V switch http://openvswitch.org/

[11] Kruskal's Algorithm https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[12] Network Metrics
https://courses.engr.illinois.edu/cs538/sp2016/Lectures/Lecture8-2.pdf

[13] Iperf command https://iperf.fr/

[14] Ping commnad http://linux.about.com/od/commands/l/blcmdl8_ping.htm