POLITECNICO DI TORINO

Dipartimento di Elettronica e Telecomunicazioni

Master degree thesis

Testing tool of SDN controllers' performance

Supervisors:
Prof. Andrea Bianco
Prof. Paolo Giaccone

Candidate:
Eduardo Berrueta
StudentID: 232856

POLITECNICO
DI TORINO

JUNE 2016

Contents

Abstract

The SDN controllers with the network applications running on the top of them, can be seen as "network brains". Those applications apply the control logic and they will install some commands in the data plane.

We can see that the performance of those applications is really important for a SDN controller and therefore, for the network it manages. Because of this importance, the purpose of this thesis is to adapt OFCProbe (one existing evaluation tool) in order to assess the performance of some processes on the applications running on the top of the SDN controller.

Our new tool, denoted as CPBeT, is aimed at changing the network topology in real-time, in order to load the algorithm which calculates the shortest path on the controller. CPBeT will generate a specific amount of OpenFlow control traffic through the network and it will force the controller to process large amount of packets. CPBeT will build a virtual network with virtual switches and hosts, and we will have the control over the topology, number of hosts, generation of packets, etc.

The experimental part of this work will evaluate the behaviour of L2 network application that run on the top of a Floodlight controller with a time varying topology which is the worst case for the shortest path algorithm. We will analyze the CPU and RAM usage on the controller changing some parameters of the experiments, like hosts, topology or rate of change of the topology, and we will demonstrate the effect of the computation of the shortest path algorithm on the controllers and how many devices the controller can manage before its saturation. Once the controller is saturated, we will analyze its behaviour and the reasons that cause that saturation.

Chapter 1

Summary

En los últimos años, han surgido problemas con la arquitectura tradicional de las redes de datos, tales como la escalabilidad o la manejabilidad de ellas cuando constan de muchos dispositivos. Como solución a estos problemas, la solución más importante es la nueva arquitectura de las redes llamadas SDN (Software Defined Networking), o Redes Controladas por Software, en castellano.

Estas redes constan de un dispositivo central llamado controlador, que actúa como cerebro de la red e instala las órdenes o las reglas en unas tablas alojadas en los dispositivos de la red (routers o switches). De esta forma, tenemos el plano de control de la red separado del plano de datos, lo cual facilita mucho tanto el desarrollo como el control de la misma.

Siendo esto así, es fácil apreciar la importancia de este controlador. Por ello este trabajo se ha centrado en analizar la escalabilidad y fiabilidad, así como el comportamiento de este en distintas situaciones extremas y ver en cuáles de ellas no funcionaba correctamente. El controlador analizado es Floodlight, que está desarrollado en Java.

Para analizar el controlador, hemos desarrollado una herramienta llamada CPBeT (Controller Performance Benchmarking Tool), que lo que hace es simular una topología con un determinado número de switches, y conectarla al controlador. Cuando todos los dispositivos están correctamente conectados a él, la herramienta comienza a enviar una determinada cantidad de Packets_In por segundo al controlador. Además, la herramienta cambia la topología de la red un número determinado de veces durante la ejecución del test, forzando al controlador a correr el algoritmo de shortest path con la carga de procesado que eso conlleva. Todo esto es para tratar de encontrar el punto para el cual se satura el controlador y no puede funcionar correctamente.

Una vez alcanzado este punto, se puede variar el número de switches, la topología, o el intervalo de cambio de topología y ver cómo afecta esto al controlador. Todos estos

experimentos se han hecho en este trabajo y se ha analizado el límite del controlador y tratado de aportar soluciones para que soporte topologías mayores.

Aunque sólo lo hemos hecho con un tipo de controlador, se ha probado a realizar experimentos con otros tipos y la herramienta funciona de la misma manera. Por falta de tiempo no se han podido analizar los resultados de forma fiable para otros controladores y es por ello que no se han expuesto en la siguiente memoria.

Chapter 2

Introduction

Motivation

Most people agree that Software Defined Networking is the future for the networks' design and that is why it is being investigated in the academic world and in the telecommunication industry. The most interesting thing about Software Defined Networking is the new network paradigm of dividing the control plane and the data plane, and a lot of companies have started to work with them in their data centers. Also recent projections are estimating that by 2018 the global SDN market will reach a value of $35 billion.[14]

In Software Defined Networking we have all the network control logic centralized in the SDN controller, that means that we have "silly" routers which have to ask to the controller what to do, and this shows us the importance of this controller in our network.

Knowing the importance of the controller, we should focus on it in order to evaluate the scalability of SDN in large networks. In which the controller has to process millions of control packets. Because of this amount of work, there should be a network topology with a number of packets for which the controller will be overloaded and the network will start to fail.

In this work we will try to find that topology and saturate the controller in order to understand how big could be the network for the controller to work properly. Once we found this limit, we will study the behaviour of the controller and the reason why it is failing, helping the developers to focus on the problems in order to fix it.

Organization of the thesis

The second chapter is devoted to introduce the SDN paradigm and to present Floodlight, on which the following experimentation will be based. We will explain its structure, layer 2 application and Link discovery manager, all of these are important in order to understand the experiments. Although we did some experiments with ONOS controller, we had not got enough time to find some reliable conclusions about it, so we decided not to write them on this work.

The third chapter is focused on some different tools for benchmarking the controllers, focusing on OFCProbe.

On the fourth chapter we will write about CPBeT, the tool we developed based on OFCProbe, and we will compare those two tools.

On the fifth chapter we will explain the experiments, how to do them, the environment and, of course, the results obtained for the controller. Thanks to this explanation, anybody could run the same experiments on a similar environment and get reliable results.

At last, we will dedicate one chapter to write the conclusions we have reached by this experiments.

Chapter 3

Software Defined Networking

Software-defined networking (SDN) is an architecture purporting to be dynamic, manageable, cost-effective, and adaptable, seeking to be suitable for the high-bandwidth, dynamic nature of today's applications. Communication networks are separated in data plane, control plane and management plane. The power of SDN is to separate the control and the data plane of the network. The control plane consists of the network protocols that dictate how the forwarding tables of the networking devices from the data plane should be filled out. [12]

SDN as a network paradigm was first introduced at the end of the 90s and the beginning of the 21 century, with the main idea that the new software defined networks would break the vertical integration of networks by separating the planes as we wrote above.[12]

The main reason for breaking this vertical integration is the fact that the utilization of the networks are changing year by year. For example, old design of the networks made sense when client-server computing was dominant, but such a static architecture is ill-suited to the dynamic computing and storage needs of today's enterprise data center, campuses, and carrier environments. Some of the key computing trends driving the need for a new network paradigm include:

1. Changing traffic patterns

   Within the enterprise data center, traffic patterns have changed significantly. In contrast to client-server applications where the bulk of the communication occurs between one client and one server, today's applications access different databases and servers, creating a flurry of "east-west" machine-to-machine traffic before returning data to the end user device in the classic "north-south" traffic pattern. At the same time, users are changing network traffic patterns as they push for access to corporate content and applications from any type of device (including their own), connecting from anywhere, at any time. Finally, many enterprise data center man-

agers are contemplating a utility computing model, which might include a private cloud, public cloud, or some mix of both, resulting in additional traffic across the wide area network.

2. The "consumerization of IT"

   Users are increasingly employing mobile personal devices such as smart phones, tablets, and notebooks to access the corporate network. IT is under pressure to accommodate these personal devices in a fine-grained manner while protecting corporate data and intellectual property and meeting compliance mandates.

3. The rise of cloud services

   Enterprises have enthusiastically embraced both public and private cloud services, resulting in unprecedented growth of these services. Enterprise business units now want the agility to access applications, infrastructure, and other IT resources on demand. To add to the complexity, IT's planning for cloud services must be done in an environment of increased security, compliance, and auditing requirements, along with business reorganizations, consolidations, and mergers that can change assumptions overnight. Providing self-service provisioning, whether in a private or public cloud, requires elastic scaling of computing, storage, and network resources, ideally from a common viewpoint and with a common suite of tools.

4. "Big data" means more bandwidth

   Handling today's "big data" or mega data-sets requires massive parallel processing on thousands of servers, all of which need direct connections to each other. The rise of mega data-sets is fueling a constant demand for additional network capacity in the data center. Operators of hyper scale data center networks face the daunting task of scaling the network to previously unimaginable size, maintaining any-to-any connectivity without going broke.[12]

Because of those four reasons new network architecture is needed, and Software defined networking is becoming more and more important solving the problems of the old networks.[12]

The division of the planes would result in network switches which are just simple forwarding devices, and we would have all the intelligence on a central device called controller. This controller could be a kind of a network operating system as you can see at figure 3.1.

The Control Data Plane interface running an application makes possible the separation of the control and the data plane, and provides the controller with direct control over the elements as switches in the data plane. The most common API for this interface is OpenFlow.

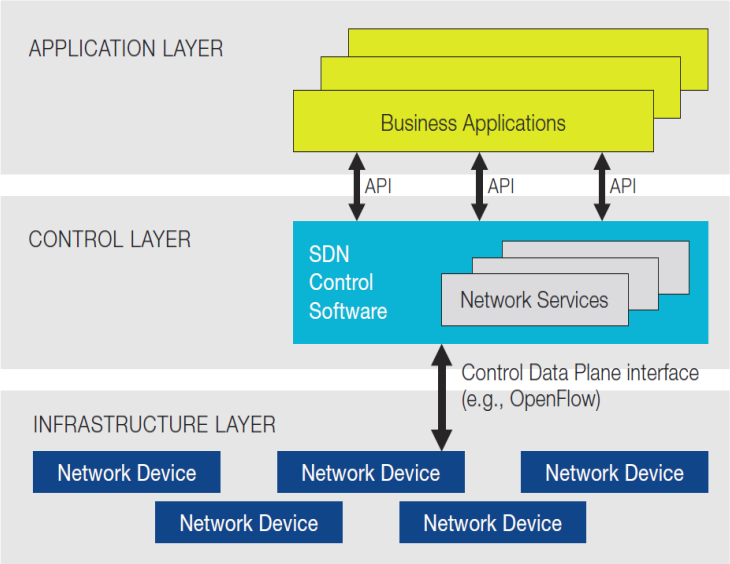


Figure 3.1: Simplified view of an SDN architecture (reproduced from [13])

OpenFlow switches contain one or more tables of rules within them, called flow tables. For each packet they want to send to another node, they watch on the flow table for the rules that match with the packet (MAC address, ports, protocols...) and if there is not rule that match it, they send a Packet_In to the controller. It processes the Packet_In and sends back to the switch a Packet_Out with the action the switch has to apply to the packet (send it by one port, send it to the controller, flood it...). Also the controller sends a Flow_Mod to the switch containing the rules and the actions for the packet in order to the switch to install the rule on its flow table. For the next packet similar to the previous one, the switch will know what to do with it and it has not to send it to the controller again.

The OpenFlow switch can behave as a classical L2 switch, router, firewall or perform another functionality, all depending on the flows installed by the controller application. The fact that the controller has an application running on the top of it allows us to program the network and adapt it according to our requirements. Instead of destination based forwarding decisions, SDN introduces flow based rules, that means that you can

choose the rules you want to apply of the packets in order to join the behaviour of different types of network devices like routers, switches, firewalls etc. This new approach makes the network programmable and helps us to solve the problems with the traditional networks we wrote above. Also it allows us to develop new software and applications in order to improve our networks faster.

The SDN controllers can work in a reactive or proactive mode, depending on when they install the rules on the switches they manage. In a reactive mode, the controller waits until it receives a Packet_In from the switches and then it installs the rule. In a proactive mode, the controller installs the rules on the switches as soon as they connect to it.

In order to install those rules on the switches, the SDN controllers send a packet to them, called Flow_Mod. The switches on the network have to send to the controller a packet called Packet_In when they do not know what to do with a specific packet. The controller has to send back to the switch a Packet_Out and it has to install new rules in that switch with a Flow_Mod.

In all the experiments we will explain on the following sections of this work, we used the controllers on a reactive mode and Layer 2 applications, so we will focus on this application and this mode.

Floodlight

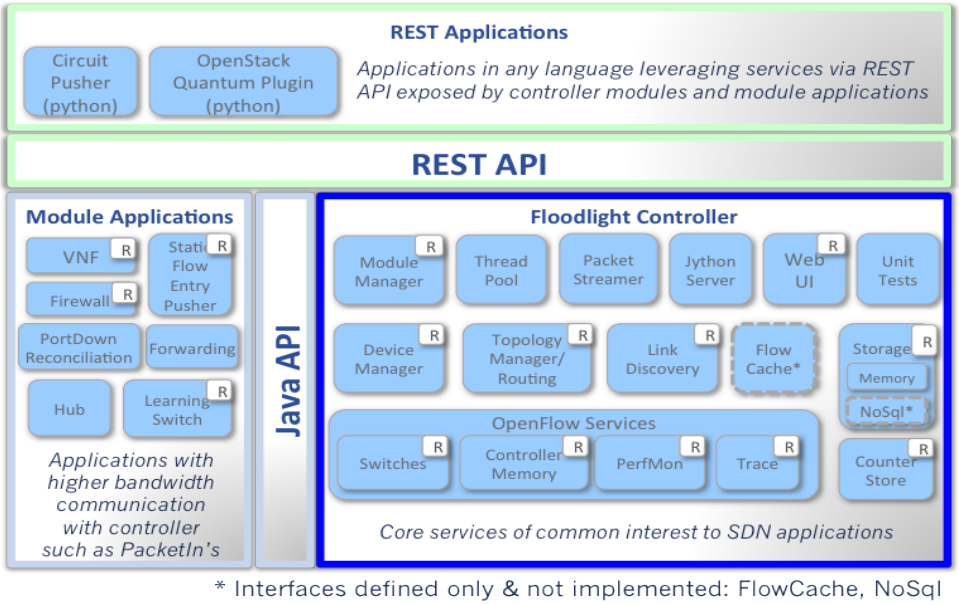


Figure 3.2: Floodlight architecture (Reproduced from [4]

Floodlight is an OpenFlow controller built on work that began at Stanford University and UC at Berkeley and now continues among a community of open source developers along with engineers at SDN and network virtualization start-up Big Switch Networks Inc. Floodlight is written in Java, and it is available via free download.

Architecture

As we can see on the figure 3.2, Floodlight can be divided on 5 parts. The first one, the modules of the controller, which are the most important thing for the controller to work properly. They should manage the packets that arrive to the controller, send the Packets_Out to the switches and install all the flow rules on them. As we will write next, there is also a module that manages the shortest paths between one node and another to save the flows information. The controller has to be aware of the changes on the topology and on the switches, and it has a global version of the network.

The second part is the Module applications. Depending on this part, the behaviour of the network will be different. We can set the layer 2 application, and the devices on

the network will behave like a Layer 2 switches. However, if we choose the forwarding application, the devices will be like routers. We can also choose the hub application, firewall application, etc. The selection of this application is really important and we have to be careful with this. For this work, we will set the Layer 2 application, and we will explain more about it in the next section 3.1.2.

The third part is called REST Applications. These applications are used by the administrator of the network in order to know the topology, the state of the links, the hosts connected to the switches, etc. It is really useful in order to manage the network and know everything about it. It is one more advantage of having the control panel of the network centralized in one machine.

Finally, there are two more parts. The APIs, Java API and REST API, whose are needed in order to program applications for the controller. The possibility of programming your own applications for the controller, gives you a lot of possibilities and make the controller really flexible and adaptable.

Layer 2 application

On this work we will focus on the layer 2 behaviour of the controller because it is the application we chose for our experiments. In the case of Floodlight, the application module is called "Learning switch". The behaviour of the switches on the topology with this application is the same as a normal Layer 2 switch, but the switch checks the rules on its rule table looking for the flow, instead of the MAC address. As normal OpenFlow switch, if there is not rule for one packet, the switch will send it to the controller, which will look for the rule and answer to the switch with a Packet_Out. The Floodlight controller will send a Flow_Mod to that switch as well as to all the switches on the path. This means that the controller must know the shortest paths between all switches. As we will see on the results of the section 6.3, this behaviour produces only one Packet_Out for each Packet_In, but more than one Flow_Mod. The number of Flow_Mods sent by the controller will depend on the number of switches on the path of the packet.

Floodlight sends periodically echo request to the switches in order to verify they are still alive. You can determine that periodicity and also you can set the timeout for a switch to be disconnected. This timeout is important in order to react when one switch is disconnected from the topology. Another timeout that we can set on the controller, is the timeout for one link to be deleted from the topology. If the controller does not see

any packet through one link and neither see it on the LLDP packets, it has to wait for this time before delete it from the topology.

Link Discovery Manager

In order to discover the links on the topology, Floodlight commands at the beginning and every 15 seconds (by default is 15 seconds but we can change it) the switches to flood LLDP packets through all of its ports. The reserved set of destination MAC addresses and ether-types used by the discovery protocol packets lets the controller to differentiate them from the other packets. When a switch receives a LLDP packet by one of its port, it sends the packet to the controller because it does not know what to do with it. The controller receives it and now it knows the two devices in this link . [10]

We can change this interval to another in order to react faster when the topology changes, but we should be careful because for large topology, decreasing the interval of LLDP packets could produce a big amount of packets on the network, and also our controller would have to process all of them.

This module has to manage all the LLDP packets. Every 15 seconds this module has to build the LLDP packets and send them to all the switches on the topology. It sends one LLDP packet per switch's port, so if we have large topology with a lot of ports on the switches, the amount of LLDP packets generated by this module could be quite important. When they send back the Packets_In, this module has to read the messages and compare the links with the topology it has on its data base in order to see if there are new links. If nothing has changed, the module has to refresh the timeout of the links on the LLDP packets and that is all. If there are some new links on the LLDP packets, the module will have to run the shortest path algorithm on all the topology in order to find the new shortest path between every nodes.

The shortest path algorithm that Floodlight implements is Dijkstra. This shortest path algorithm marks one node as the initial one, and set its tentative distance value to zero. This distance must be infinity for the other nodes. Now the algorithm has to visit every neighbours of the initial nodes and calculate the tentative distance between the initial one and each of its neighbours. Then it compares the newly tentative distance to the current assigned value and assign the smallest one. When it has considered all the neighbours of the node, it marks this node as visited and as it is visited, it will never be checked again. If the destination node is marked as visited, the algorithm has finished.

Otherwise, it has to select the unvisited node marked with the smallest tentative distance and set it as the new current node. [1]

Explained this, the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of the number of edges, and the number of vertices, using big-O notation. The running time for the simplest implementation of Dijkstra's algorithm is $O(|V|^2)$. For sparse graphs, that is, graphs with far fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently, requiring $O((|E| + |V|)log|V|)$ time in the worst case. Whatever the implementation was, the time for the Dijkstra's algorithm is not negligible for large topology, and that is why we want to stress the controller in that way.

Knowing how Floodlight works, it is obvious that the bigger the topology is, the more time Floodlight spends to calculate the shortest path between two devices. So now we are sure that there should be a topology for which Floodlight will spend so much resources and CPU and even it will saturate. We will try to saturate the Dijkstra algorithm, and that is why we will focus on the topology, making it larger and larger, instead of focusing on the amount of Packets_In per second sent to the controller.

Chapter 4

Standard tools for controller testing

As the controller is the brain of the SDN networks, we would like to evaluate the performance of these devices and measure parameters as the number of devices the controller can control properly in its topology or the amount of Packets_In that it can process. For calculating these limitations, we need some tools to emulate the network behaviour and measure the controller's reaction to the packets sent by these tools. There are few tools that try to evaluate the controller's performance evaluation.

Cbench is an open source tool which emulates a set of switches connected to a controller. They are continuously generating packets_in and trying to get packets_out or flow_mods events sent by the controller. Cbench is written in C and we can choose the number of switches or the mode in which the tests are going to be performed. The maximum number of switches allowed is 16. The most important weakness of this tool is the fact that we can not create a topology for benchmarking the controller, and for our purposes, that is a big inconvenient.

Another tool is aPET, it is based on Cbench, but it has the possibility to set one topology on the network in order to benchmark the controller in a more realistic way. aPET works between mininet and the controller, acting like a proxy at the beginning of the running in order to make the controller aware of the topology through the LLDP packets. The weakness of this tool is the fact that when you set a large topology, the tool can not send enough packets_in to saturate the controller, and also it needs mininet to build the topology and manage the LLDP packets, so it can not change the topology during the running because the topology is managed by mininet. [19]

Finally, OFCProbe is another tool with the same purpose as the two mentioned above. The main goals of this tool are explained in the next section, and we will based on this tool in order to develop our own tool.

OFCProbe

OFCProbe is a platform independent tool for benchmarking SDN controllers. It reaches five important things that are really needed for a nice evaluation of the controllers: [17]

1. Platform-Independence: This software can be executed on the most common system architectures because it is developed in Java.
2. Scalability: OFCProbe is multi-threading enabled and also its threading overhead should be reduced in comparison to another benchmarking tools.
3. Modularity:This is really important in order to adapt the tool to a possible new communication protocols version, and OFCProbe reaches this because of the separation of the program logic and the controller communication.
4. Performance Analysis: The tool generates different amounts of traffic to the controller and it can record the controller's responses allowing the analysis of the controller's behaviour in that different scenarios.
5. Detailed Statistics: This features allows the investigation of the controller's behaviour such as the latency or the CPU usage of this controller. So we can study this behaviour on different environments such as topology or number of devices connected to the controller.

Besides all of these features, OFCProbe emulates really well the network's behaviour, giving to the manager the chance to emulate topology, traffic patterns and ARP packets in order to evaluate the controllers in a more realistic environment.

Architecture

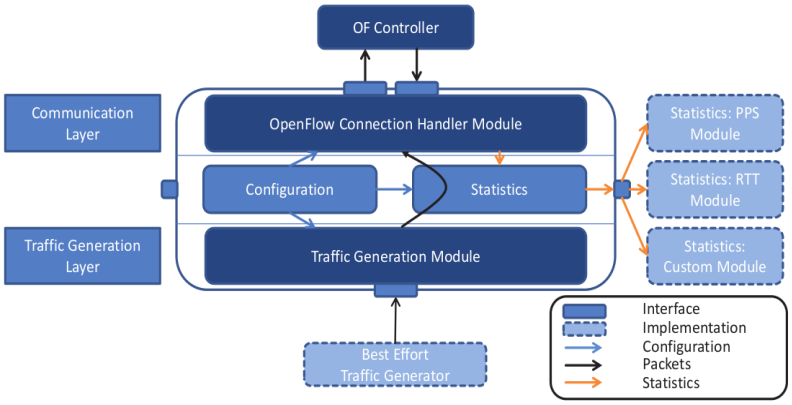


Figure 4.1: OFCProbe architecture (Reproduced from [17])

We would like to explain really well and detailed the architecture of this tool because all the experiments on the section 6.3 are done with a tool based on OFCProbe, and it follows the same architecture.

This tool is modular with interfaces between the modules because OpenFlow and SDN are evolving and changing and this is the easiest architecture to adapt the tool to these future changes. As we can see on the Figure4.1, the tool is divided on four important modules: Connection handler module, configuration module, statistics module, and traffic generation module.

1. The OpenFlow Connection Handler Module:
   This modules has three main tasks, the first one is to establish and handle the connection between the tool and the controller. For doing this, OFCProbe uses Java's NIO Selector. With this selector it can handle multiple channels in one thread.
   The second main task of this module is to manage the flow tables of the virtual switches. As the tool has to send as many Packets_In as we wanted, the flow tables of the virtual switches are not so important. Every time one virtual switch wanted

to send a packet to another virtual switch, it will sent a packet_In to the controller without check the packet on the flow tables.
The third and last task, but not the least important one, is the acceptance of messages from the "Traffic Generation Module", encapsulate them on OpenFlow messages and send them to the controller. It has to manage the replies for the controller too, and send them to the statistics module in order to process them and have them ready for further analysis. It is important to explain how this module sends the Packets to the controller in order to understand the behaviour of the tool. If you set the option "batch sending" in the configuration file, the tool will try to send all the packets on the queue before read the packets received from the controller.That means that the tool sends all the Packets together and then it waits for the replies. In following sections we will show that behaviour with some graphs in order to understand it better.

2. The Traffic Generation Module:
   This module has its own thread and it is an event-driven scheduler queue processor. The module has one queue with all the events of the simulation. An event consists on an event time, event type, and a list of virtual switches on which this event must be produced. When one event is processed for this module, it calls to different actions on the virtual switches of the event depending on the event type (ARP_EVENT, PACKET_IN EVENT, DISCONNECT EVENT, etc.). In the next section 4.1.2 we will explain how the virtual switches manage these events and the actions they should do.

3. The Statistics Module:
   Every Packet_In sent by each switch and the Packet_Out or Flow_Mod receive by them, are sent to this module in order to process them and take some information such as: arrival times, type of packet, etc. When the emulation ends and the module has all the packets sent and received by the tool, it can take much more information for each virtual switch and of course for the controller. It can show us the number of Packet per second sent and received by each switch, the RTT for each packet and switch, and also the CPU and RAM of the controller. For this last statistics, we must implement SNMP on the controller.

4. The Configuration Module:
   Thanks to this module, we can configure some tool's parameters such as the simulation time, the number of switches, the topology file, the IP of the controller, etc.

We can modify this parameters editing the configuration file who has to be in the same directory of our tool. It is really intuitive to change the values of this file because all of them are explained on that file. In the section 4.1.2 we will explain the most important ones for the experiments. On the figure4.1 we can see how this module is connected with the others in order to configure the parameters of them.

Talking about the internal behaviour of the tool, we should write about the threads running on it. As we mentioned on the previous section (Section 4.1), OFCProbe is multi-threading. We can configure the number of threads on the tool with the configuration file, but we will have always one thread for the Traffic Generation Module. The virtual switches will be able to run with its own thread or sharing one thread between more than one virtual switch.

Network behaviour

As we wrote above (Section 4.1) OFCProbe is an event-driven scheduler queue processor, which means that the traffic generation module has a queue with the events and the virtual switches in which that events has to be processed. On the figure 4.2 we can see the chain of the events, that is the steps that the emulation follows in order to benchmark the controller. Next we will explain each event and the packets generated for them.

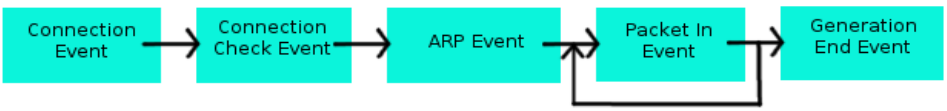


Figure 4.2: OFCProbe chain of events with ARP configuration

First of all, the module creates the Switch_Connect_Event for all the switches to connect with the controller. Also it creates the Generation_End_Event by the time of the

simulation ends. The virtual switches has to establish the connection and handshake with the controller in the correct way depending on the controller we want to benchmark. For example floodlight sends STAT_CONFIG packets and configuration packets to each virtual switch, but ONOS sends also barrier request messages to them. The tool answer this packets and after that, it is ready to start the benchmarking. Subsequently, a Connection_Check_Event is created for all the virtual switches in order to be sure that the handshake was successful and now we can start sending the first Packet_In.

Depending on the configuration of OFCProbe, the traffic generation module can start with TCPSYN Packet_In from the beginning or with an ARP Packet_In. The ARP messages will be sent for each host to all the others in the topology, and the tool will wait for the answer of that ARP messages. Meanwhile, virtual switches will answer to the ARP messages for the hosts connected to its ports with an ARP_Reply messages. When virtual switches has sent the ARP_Reply messages, the traffic configuration module creates the Packet_In event for all the virtual switches.

When a Packet_In event is executed, the module checks the type of the Packet_In and if it is a TCPSYN packet, it generates the payload of the packets, sends them to the controller and saves the bufferId generated for each Packet_In. As the module has to send one TCPSYN packet for each virtual host in the topology, the module will generate as many Packets_In as virtual hosts we had in the scenario. After that, it generates a new Packet_In event in a pre-defined amount of milliseconds. This continues until Generation_End event is reached, which ends the experiment. In the central configuration file we can set the inter-arrival time of Packet_In events.

The TCP and ARP packets generated by the module could have static or randomized address values. OFCProbe uses a pre-generated master TCPSYN packet, that contains a correct TCPSYN packet with all fields except the address fields set. This fields must be filled with generated address taken from the corresponding MAC, IP or TCP generator.

Knowing this behaviour, each virtual switch will have a queue of Packets_In generated by the traffic generation module. Depending on the size of the network (the number of hosts on it) the size of the queue could be quite large. Besides of the packets generated by the traffic generation module, when a virtual switch receives a Packet_Out from the controller, it has to check the actions for the output of the packet. If the packet is not for a host connected on this virtual switch, this virtual switch has to queue the packet on the queue of the virtual switch connected to the port indicated on the Packet_Out. In order to know which Packet_In has to be queued on the virtual switch's queue, OFCProbe has

to check the bufferId of the Packet_Out and checks if it matches with any of the bufferId it has on memory. If it matches, it can queue the Packet_In with this bufferId to the virtual switch.

Summarizing, the tool has to queue the packets on the virtual switches depending on the event reached, but packets could be queued in a virtual switches also by another virtual switch, that is how the tool emulates the behaviour of a real network and we can see the TCPSYN packets going through the network in the same way as in a real one.

It is important to write about the LLDP packets managing on the tool. As the controller sends LLDP packets to the switches in a predefined interval of time, the tool has to read them and send back to the controller the Packets_Out answering these Packets_In. OFCProbe reads the LLDP packets and creates its own LLDP reply packets emulating the network behaviour. That means that each switch has to see its connection with other switches on the topology and send back to the controller the right LLDP packets according to that topology. Then the controller will be aware of the topology created by the tool, and it will think that it is working in a real network.

Chapter 5

CPBeT

In this chapter, we will explain the tool we developed in order to benchmark the controllers. We based it on OFCProbe, but we changed few things to make the tool more useful. We called the tool CPBeT (Controller Performance Benchmarking Tool), and it is written in JAVA, using the same libraries to manage the sockets and the OpenFlow messages.

Architecture

The tool has the same parts as OFCProbe, but we completed some of them in order to get more specific results on our experiments or in order to change some behaviours in a particular environments.

1. The OpenFlow Connection Handler Module:
   The tasks of this module are the same as the module on OFCProbe, the generation and management of the packets on the tool are exactly the same, we did not need to change this part of the tool because its main goal is the generation of packets, and we need it for the benchmarking.

2. The Traffic Generation Module:
   As we mentioned on the section 4.1.1, this module has a queue of events, which will produce a different kind of packets to be sent to the controller. Besides the events we had on OFCProbe, in CPBeT, we will generate one more in order to change the network topology. With this event we will add to the tool the capability of changing the topology during the running, which is important to test the behaviour of the controller when the topology changes.

3. The Statistics Module:

OFCProbe measures the number of packets sent and received per switch, but CPBeT aggregates those statistics and we have the total number of packets per second too. This allows us to analyze the controller's behaviour knowing the total number of packets that it has to process.

4. The Configuration Module:
   Besides all the parameters we can configure on OFCProbe, we can also set for example the total number of hosts on the topology and the interval of changing the topology, these parameters allow more flexible experiments.

Network behaviour

The main behaviour of the tool is the same as the OFCProbe mentioned on Section 4.1.2. The most important difference between OFCProbe and CPBeT is that the second one will not react to the Packets_Out received by the controller when they are not LLDP packets. This difference will be explain better on the next section, where we will compare the tools. Also the chain of the events generated by the tool is different in CPBeT because as we mentioned on the previous section, the tool now has one more event to change the network topology.

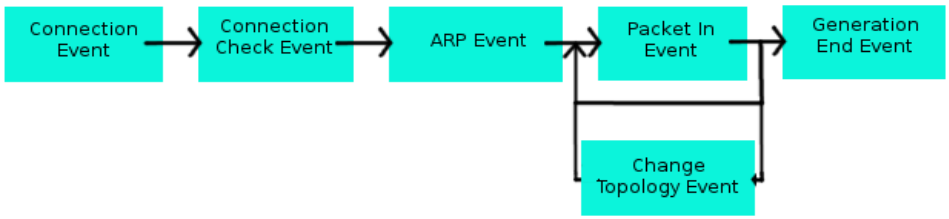


Figure 5.1: CPBeT chain of events with ARP configuration

As we can see in the figure, CPBeT has to process the Change_Topology_Event during the running. We set the interval of this event in the configuration file, and when it has to be process, the tool change some connections and continue generating the Packets_In

for each virtual switch. As we can see on the figure, we have two ways to go back again to the packet_In event. It is because sometimes CPBeT is behaving like OFCProbe, it has to generate another Packet_In without doing anything else, and sometimes it has to process Change_Topology_Event before the generation of new Packets_In.

Comparison

We will enumerate the main differences between CPBeT and OFCProbe in order to understand the different behaviour they will have during the running of the experiments and the behaviour we should expect for the controller.

1. CPBeT doesn't react to the Packets_Out received by the controller:
   This is the main difference on the tool, because we deleted the capability of acting like a real network. Now, the tool only sends an amount of Packets_In per switch but when the controller sends back a Packet_Out, the virtual switches do not do nothing. The tool checks the Packet_Out for knowing the type of the packet, because if it is a LLDP packet the virtual switch has to send a Packet_In to the controller.

   There are two main reasons for changing this behaviour. The first one is the fact that we want to saturate the controller using large topology. This means that we have large amount of switches running on our tool, and the number of Packets_Out increases really fast when the topology is larger. It causes that each thread will have to process all these Packets_Out, checking the output port, checking the bufferId, comparing it with the bufferId saved on the virtual switch and queuing the Packet_In on the other switches. So with these large topology the tool can not send all the Packets_In it should, so it is not behaving like a real network. Also, if the tool is not sending all the packets, we can not control the Packets_In sent by it, and the experiments are not as reliable as we would like.

   The second reason is that the tool, as we mentioned in section 4.1.2, saves the Packet_In with the bufferId they have. Sometimes the controller doesn't send the Packet_Out with the same bufferId as the packet_In had. It sends the Packet_Out with the Packet_In inside it. When this happens, the tool checks the bufferId of the Packet_Out but it does not match with anyone on its memory and "throws" the Packet_Out. So the behaviour of the tool is perfect just in some environments.

   We know that we made less realistic tool, it does not behave as a real network

now, but we will be able to know how many Packets_In per switch can manage the controller in a specific topology, and we can use this to compare some controllers' behaviour and choose the best one for each environment.

Although this behaviour is not the real network's behaviour, for some applications running on some controllers, this behaviour is more realistic than the OFCProbe. For example, on Floodlight running the Layer 2 applications on its top, the controller sends the Flow_Mods to all the switches on the path of the Packet_In it receives by the virtual switches (See section 3.1.2). That means that in real network, the switch who receives that Flow_Mod for the controller, will not send a Packet_In for the same flow. We will have one Packet_In for each source and destination, which is exactly the same number of Packets_In that we have with CPBeT. OFCProbe generates n Packets_In for each source and destination, being n the number of devices on the path. This behaviour is more realistic in general, but on particular applications running on the top of some controllers (as Layer 2 on Floodlight), is not realistic because the tool will produce much more packets_In than in real network. For our experiments, running on the application Layer 2 of Floodlight, CPBeT is more realistic than OFCProbe.

Summarizing, the tool is emulating a real network only if the controller sends the Packets_Out with the specific bufferId, and only with small topology, because with large ones, the tool can not process and send all the packets. That is the reason why we delete this capability, because the tool only has a correct behaviour in some cases.

2. Changes on the topology

   The main goal of these experiments is saturate the controller. We will try to find the process in the controller that takes a long time in some situations and we will try to overload it. We though that Dijkstra would take a long time if we enlarge the topology in a specific way and change the topology during the running.

   So the main reason for this change in the tool is really simple. Changing the topology we force the controller to recompute Dijkstra more times and, as it takes a long time doing it, we would be able to stress the controller in that way.

   In order to do this, we set a topology file and an interval to change it. After the generation of the Connection Event in the Traffic generation module, it generates a Change_Topology event with the interval specified in the configuration file. When the event has to be process, one link between two switches is deleted and a new one

is created between that switch and another one.

3. Set the number of hosts in the topology

   The number of hosts in the topology is the most important thing to saturate the controller, because the number of Packets_In generated by the tool is specified by this number of hosts. In OFCProbe, we can set the number of hosts but we have to choose between one host per switch, and one host per free switch's port. This is quite rigid number of hosts, and in large topology, one host per switch is too much for the controller (it is saturated since the beginning of the experiment). So we changed this fact and now in the configuration file we can choose also one number of hosts lower than the number of switches. That means that we have some switches without any host connected on it and some switches with one host.

4. Total number of Packets sent and received by the tool

   The statistics module in OFCProbe measures the number of Packets per switch, but we think that if we could see the total number of packets in the network per second, we would be able to understand better the behaviour of the controller and also analyze it with a global vision of the packets. We added a global counter of packets in this module in order to see that, and now we can see the maximum number of packets the controller can receive and process before the saturation. We can see also the behaviour of the controller when it is saturated, the number of packets it sends in this situation.

5. Adapting the tool to ONOS controller

   Although we did not benchmark ONOS controller, our objective at the beginning was to do it, but because of the lack of time, we finally could not run them. This is why we adapted the tool to this controller, and also because we wanted a flexible tool which will be able to benchmark as many controllers as possible.

   In order to know if a packet_Out is a LLDP packet, OFCProbe checks the ether-type, the destination MAC address, and some other fields of the packets. The problem is that ONOS uses different destination MAC address for sending the LLDP packets, and OFCProbe did not react to it. Consequently, ONOS could not be aware of the links in the topology. The most difficult thing was to understand why it was failing, but when we noticed that the problem was the LLDP packets, we added some lines on the source code in order to make the controller "understand" the packets and

react to them. This changes make CPBeT more flexible than OFCProbe, and now we have a tool that can benchmark more controllers.

6. Measurement of RTT, CPU and RAM on the controller

   By default we could set this capabilities on the configuration file of OFCProbe. The problem we wrote about with the bufferId of the Packet_Out sent by the controller (section 4.1.2) make us to change that capability in order to calculate the RTT on the right way. The problem is that OFCProbe calculates the RTT watching the difference between the time of the Packet_In and its Packet_Out. So if the bufferId of the Packet_Out does not match with the Packet_In's, it won't be able to calculate the RTT. For measuring the RTT in CPBeT, we have to capture the traffic with Wireshark and export it as a plain text. Then with a simple program we developed, we can calculate the RTT on a more reliable way. However, the capability of OFCProbe was not deleted and we can measure the RTT in that way in CPBeT. We just added this program in order to make better estimation of it and to be able to measure it whatever the behaviour of the controller was.

   The problem with the CPU and RAM measurement is that we need to implement the SNMP capability on the controller's machine. We would like to calculate them without that needing, and we reached it with the top command on the controller's machine. We should save the output of the command and then run another JAVA program to calculate and make the graphics of the RAM and CPU usage. the command we used for calculate the CPU and RAM usage is the following.

```
edu@edu-PC:~$ top -d 0.1 -p 9233 -b > output.txt
```

Figure 5.2: Top command used to measure the CPU and RAM usage

There is another problem with OFCProbe measuring the RTT and CPU usage of the controller. As we are trying to saturate the controller, the amount of packets sent and received by the tool is quite large. If we do too many process on the tool, we can have problems with the CPU on the machine running OFCProbe because it has to process all the packets_Out sent by the controller and calculate the time between them and the Packet_In, so we will be in the same situation we wrote about before (section 4.1.2). That is another reason for doing this measurements with other programs. This does not affect to the behaviour of the tool, and in any case, CPBeT

can saturate the controller and measure in a more reliable way the parameters in order to demonstrate that the controller is saturated and its behaviour.

7. Set the number of ports per switch

   In OFCProbe we have to set the number of ports per switch in the configuration file, but for our experiments, we have used an specific topology, and the ports that we need for each switch depend on the number of devices we have on that topology. Because of that, we deleted this capability and in CPBeT the number of ports is automatically set by the tool when it processes the topology file. This change is not really important but we would like you to know in order to understand better the methodology of the experiments explained in following sections.

Chapter 6

Experimental Evaluation

The most important goal we reached on this experimental evaluation is the saturation of the controller. As we want to show how this affects to the behaviour of the controller, we focused our graphics and our experiments on showing that. We will explain the methodology of the experiments and the results in order to allow other persons to run the experiments in the same environment. However, the tool allows us to set another parameters of the network and change the environment (topology, hosts, ports, intervals...) in order to evaluate another aspects of the controllers.

Methodology

We would like to stress the controller, and in order to reach it, we looked for some processes running on the controllers which could take quite large computational work. For the beginning, we would like to stress Dijkstra algorithm on the controller, and analyzing how it works (Section 3.1.3) we found the worst topology case for this algorithm. Then we will force the controller to run the algorithm several times and we will analyze the behaviour and the resources it consumes.

Topology

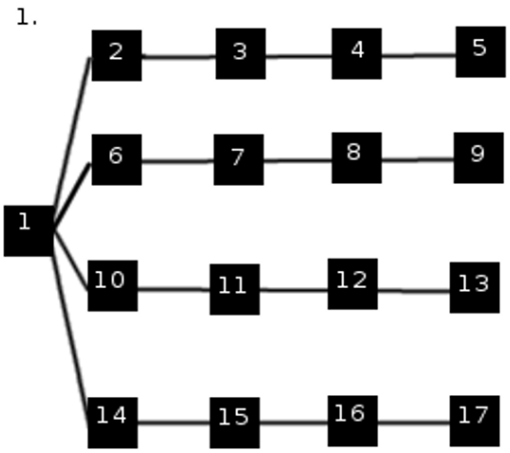


Figure 6.1: Worst topology case for Dijkstra

On the figure 6.1, we can see the worst topology case for Dijkstra algorithm just for 17 devices. From now, we will call this topology "M×N topology", where M is the number of rows and N is the number of columns of the square. So setting a M×N topology, we will have $M \times N + 1$ devices on the network.

With this topology, Dijkstra algorithm has to process all nodes in order to get the shortest path between the node 1 and the node 17. [20] This forces the controller to do some computational work every time it has to calculate the shortest path, and we will use it in order to saturate the controller.

We had to create this topology, but with much more devices in order to stress the algorithm on the controller. We did the experiments with 20×20 topology, that means 401 devices. As we mentioned on Section 7, the tool automatically set the number of ports per switch, on 20×20 topology, there are 22 ports per switch, so the controller has to send 22 LLDP Packets_Out per switch. That is a huge number of LLDP packets (around 9000 packets in total), but the tool has to reply with many fewer packets_In, because each switch has just two connections to another switches, (but the first, which has 20 connections, and the switches on the last column, who have just one). We though about this huge number of ports per switch, but we agreed that a real switch connected in a

real network, could have that amount of ports, so the controller would have to send those Packets_Out too. We want to emulate a network as real as possible, and this number of ports could be the real one.

We would like to say that the usage of the CPU on the controller is affected for this huge number of Packets_Out it has to send to the switches, and although it had not to run the Dijkstra algorithm, on the graphics of the CPU usage, we will see peaks when the controller sends this packets.

Changes on the topology

By default, as we mentioned on Section 3.1.3, the controller runs Dijkstra every time the topology changes. So if the topology is the same along all the experiment, we will not stress this algorithm. In order to overload the controller, we should change the topology during the simulation. The changes that the tool will do in the topology are explained in the next figure:

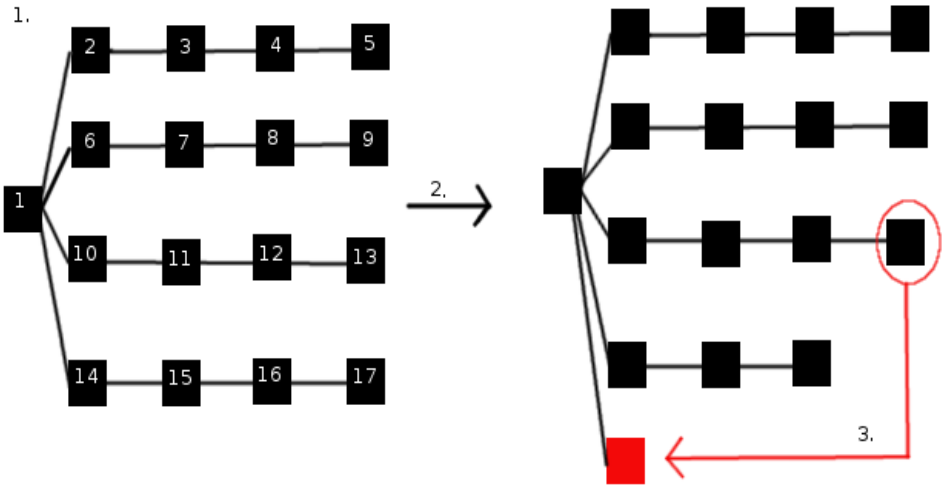


Figure 6.2: Topology changes

We think that it is not really important how to change the topology, because whatever it changes, the controller has to run all the Dijkstra algorithm again. We choose this change because doing it we maintain the structure of the topology and it is still the worst case for the algorithm. Referring to the number of Packets, it is easy to understand that the number of Flow_Mods will change with the news topologies, because the paths

between the devices are different. We can see this in the figure 6.12

We can see how the number of packets decreases every 15 seconds, when the controller notices that the topology changes. We do it every 15 seconds on the tool, and that is because the controller sends LLDP packets every 15 seconds by default. On the experiments of the section 6.3 we will change these intervals in order to demonstrate that Dijkstra algorithm affects to the CPU on the controller.

Experimental set up

Virtual machines

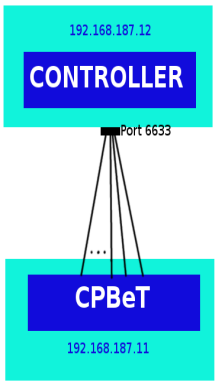


Figure 6.3: Set up architecture of the experiments

On the figure 6.3 we can see the virtual machines (one for the controller and one for the tool) used for doing the experiments. The lines between the two virtual machines represent the TCP connections the tool has to establish in order to send the packets to the controller. The tool establishes one per switch on the topology.

The two virtual machines have the same capabilities: 4 CPU cores and 3 GB of RAM memory. For futures benchmarking evaluation of controllers, it could be better if we could have more powerful machine for the tool, because as we wrote above, we had to change or delete some capabilities in the tool to avoid saturating it before than the controller.

Once we have the virtual machines running, we have to install the controllers on the controller's virtual machine. Floodlight is not difficult to install, but ONOS is. In order to install ONOS on Cent-OS (the operative system running in our virtual machines) we had to follow this tutorial [7].

Running the tool

In order to run these experiments, we have to follow these steps for running them right:

1. Check the configuration file of CPBeT
   It is important to set the parameters we want for our experiments. There are some important parameters we should set in the configuration file that we can see as follows:

```
config.openflow_version = 1
config.switchCount = 102
config.threadCount = 8
config.controllerAddress = 192.168.187.12
config.controllerPort = 6633
config.startDpid = 1
config.simTime = 180000
config.startDelay = 8000
config.stopDelay = 1
config.hasTopology = true
config.hasGraphml = false
config.graphml = Aarnet.graphml
config.checkForIndividualSwitchSettings = false
switchConfig.portCountPerSwitch = 8
switchConfig.buffersPerSwitch = 512
switchConfig.sendFlag = true
switchConfig.disableNagle = true
switchConfig.batchSending = true
switchConfig.flowTableSize = 128
switchConfig.randomizeStats = true
trafficGenConfig.scenario = TCPSYN/PCAP
trafficGenConfig.arpEnabled = true
trafficGenConfig.fillThreshold = 300
trafficGenConfig.iatType = 0
trafficGenConfig.IAT = 1000
trafficGenConfig.countPerEvent = 1
trafficGenConfig.staticPayload = false
trafficGenConfig.onlyTopologyPayloads = true
trafficGenConfig.onlyOneHostPerSwitch = true
config.TotalHosts = 10
config.TopologyInterval = 5000
trafficGenConfig.generatorTypeMAC = SERIAL
trafficGenConfig.generatorTypeIP = SERIAL
trafficGenConfig.generatorTypePort = SERIAL
trafficGenConfig.iatDistribution = Normal
trafficGenConfig.iatDistributionParamter1 = 1000
trafficGenConfig.iatDistributionParamter2 = 1
statsConfig.modules=PPS
```

Figure 6.4: Configuration File of OFCProbe

.

- The number of threads to use for the switches, it depends on the number of switches we had on the topology. For the case of 400 switches, we choose 20

threads, because we do not want to run too many threads in order to not saturate the tool. We want to have just the minimum number of them to work properly.

- The simulation time, we set it to 180 seconds because we want to see the periodicity of the LLDP packets and the peaks on the CPU usage on the controller. 60 seconds was not enough.
- The delay between the initialization of the switches and the start of the benchmark. This delay depends on the switch the tool has to initializes, for 400 devices it is enough with 10 seconds.
- The flag of batch sending. We set it "true" because we want the tool to send every queued packets at once instead of sending each message individually.
- The flag of the ARP messages, we set it "true" because we want our tool to emulate a real network, and also because we want the controller to discover the hosts before sending the TCPSYN packets_In.
- The flag of One host per switch. It should be set to true if we want to connect one host per switch.
- When we set the One host per switch flag to true, we can select the number of total hosts in the topology with this parameter.
- The interval to change the topology. The interval in milliseconds to change the topology on the tool.
- We have at last some parameters about the statistical distribution to generate the Packets_In. We set it to send one packet per second for each host in the topology.

2. On the controller's virtual machine. We should run the top command saw on the Figure 5.2, and of course, we need to run the controller we would like to benchmark. We have to be careful with the port where the controller is listening for connections and set the same port in the configuration File of CPBeT. In order to understand better the results about the number of packets, we set some parameters on the controller as the timeout for the switches to disconnect, the interval of LLDP packets and the timeout for deleting the links unused. When the timeout of one link is over, the controller runs Dijkstra again because it deletes that links and so the topology changes. We set this parameter on the source code on Floodlight and on the ONOS' CLI.

3. On the CPBeT's virtual machine. We should run the tool with the following command where "config.ini" is the configuration file and "Topology_20x20.ini" is the file with the topology we want to emulate on the tool.

```
edu@edu-PC:~$ java -jar CPBeT.jar config.ini Topology_20x20.ini
```

Figure 6.5: Command to run OFCProbe

.

4. Wireshark capture. We should remember to capture the traffic with Wireshark if we want to calculate the RTT during the experiment. If we want to use the program we developed for calculating the RTT, we should follow the following steps after the simulation. First of all we should filter the packets in order to display only the OpenFlow packets. After this, we need to export the packet dissections (On file-Export packet dissection-As plain text) and now we should select these options:

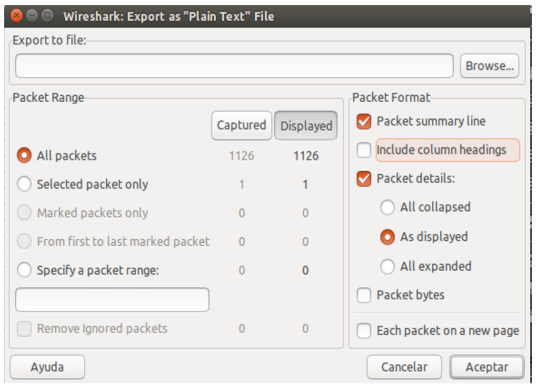


Figure 6.6: Options for exporting the Wireshark packets

.

5. Topology File. The topology file we should write as a parameter on the command to run OFCProbe (Figure 6.5) should be written like this example:

```
1:1==2:1
1:2==4:1
2:2==3:1
4:2==5:1
```

Figure 6.7: Topology file

.

The number written before the ":" is the switch number, the number after that is the port number of this switch. The "==" is the connection between the hosts. This example corresponds to the 2×2 topology as we can see.

Experimental results

In this section we will show the experiments we did in order to understand the controller's behaviour when it is saturated. Also we are going to show the number of devices (hosts and switches) the controller can manage with different topology. For each experiment, we will explain the parameters we set in the tool. Then, we will show the results with a graphic and after that, we will explain those results. First of all we will explain the main differences between OFCProbe and CPBeT running the experiments.

Comparison between OFCProbe and CPBeT

We wrote about how CPBeT works in the previous section (Section 5), but we would like to show the different behaviour OFCProbe and CPBeT have in order to understand better the reasons of these changes. In the figure 6.8 we can see the number of Packets_In, sent by the tools with different number of devices on the topology.

The red line is the number of Packets_In we can expect to be sent by CPBeT, according to the topology (linear) and the number of switches in it. We obtained this theoretical curve with the following equation:

$$h \times (h - 1)$$

Where h is the number of switches in the topology. We assumed that each switch has only one host connected. For OFCProbe, the equation is not as simple as before, because each Packet_In generated by the hosts, will generate more Packets_In on the path (remember Section 4.1.2). So for each Packet_In generated by each host, we have a number of Packets_In calculated as followed:

$$(h + 1)/3$$

This number allows us to calculate the total number of Packets_In that the tool has to generate for each topology:

$$h \times (h - 1) + [h \times (h - 1) \times (h + 1)/3]$$

and simplifying the formula:

$$h \times (h-1) \times (h+4)/3$$

With this formula we obtained the blue curve we can see on the graph, and we can notice that OFCProbe can not send all the packets it should for large topology. However, it can produce around 30000 Packets_In per second for the topology with 100 devices, which is more than CPBeT with the same topology. These is obvious because for the same topology, OFCProbe produces more Packets than CPBeT.

We can see that OFCProbe can not send all the packets_in for 31 hosts, it should send around 11000 packets and it is only sending 10000. However CPBeT is sending around 15000 packets for larger topology (120 hosts). That means we can control the number of packets for CPBeT in a large topology, but not this number of packets on OFCProbe with smaller one.

Although OFCProbe would be able to saturate the controller with smaller topology than CPBeT does, we have no control over the number of Packets that the tool sends, and that is why we prefer to develop new tool in order to allow it to work with larger topology keeping that control. Also we want to overload the Dijkstra algorithm in the controller, and that requires large topology, so we prefer to work with CPBeT.

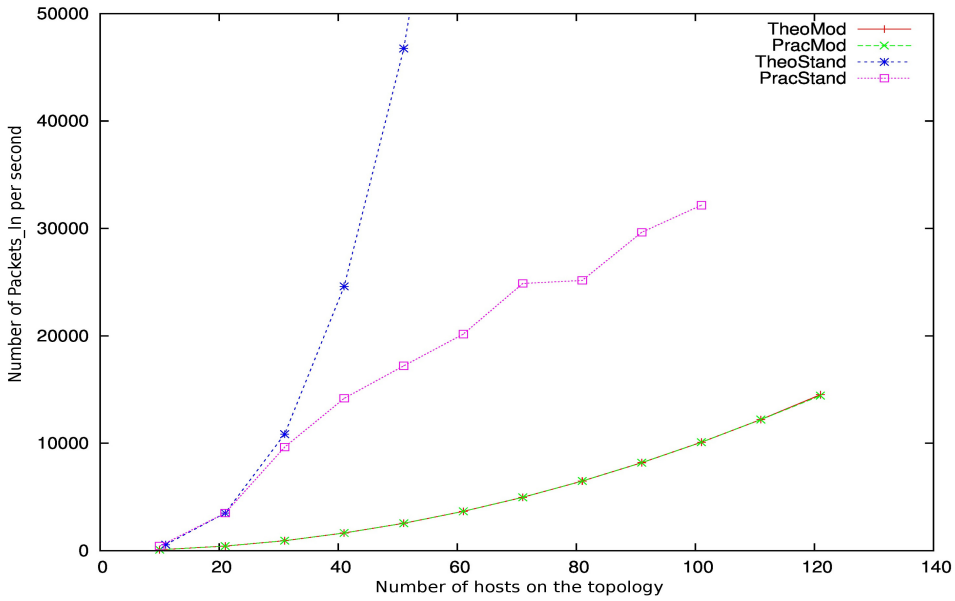


Figure 6.8: Number of Packets_In sent by the tools

Although CPBeT is working properly with large topology, we have to say that the

CPU usage on the virtual machine running the tool is as high as in OFCProbe. The problem is that if the tool receives a lot of Packets_Out or Flow_Mods by the controller, it has no enough time to send its Packets. The reason why the CPU usage is the same for the two tools is a wrong management of the TCP sockets on the tool, because the tools are always triggering the buffer of TCP sockets in order to see if there are packets in them to read. That produces a large usage of CPU. We can see on the graph below the %CPU usage on the tool's virtual machine for 41 hosts.

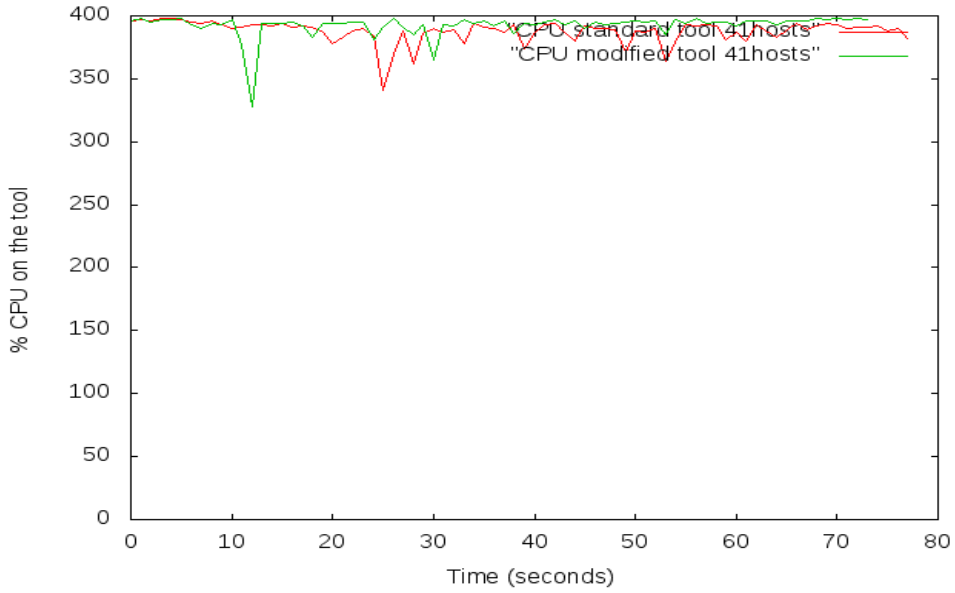


Figure 6.9: CPU in the tool's virtual machine

As we wrote before, the CPU usage in the controller is higher using the standard tool for the same topology. It is because the number of Packets_In is higher using OFCProbe. We can see the comparison between the CPU usage on the controller in the following graph:

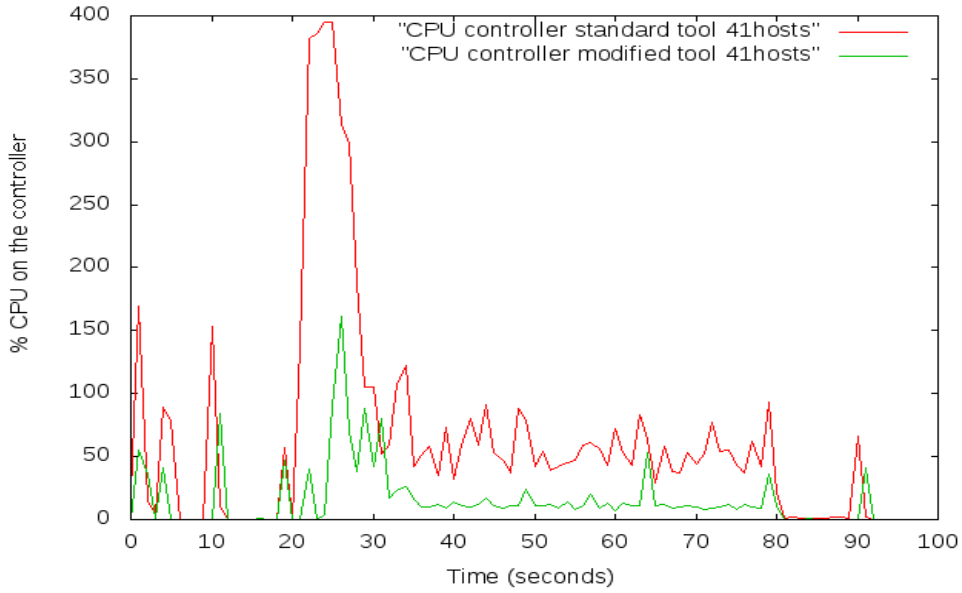


Figure 6.10: CPU in the controller's virtual machine

As we can see, the peak at the beginning of the benchmarking is much higher with OFCProbe, it is produced by the ARP packets, which are many more with that tool, because OFCProbe is answering each ARP packet and also it generates one Packet_In for each Packet_Out it receives by the controller.

The following graph shows us the reaction of the controller in terms of Flow_Mods when the topology changes. From now, the experiments will be always run with CPBeT. This graph is shown in order to notice the change on the topology produced by CPBeT, because we will not have the option of doing it with OFCProbe.

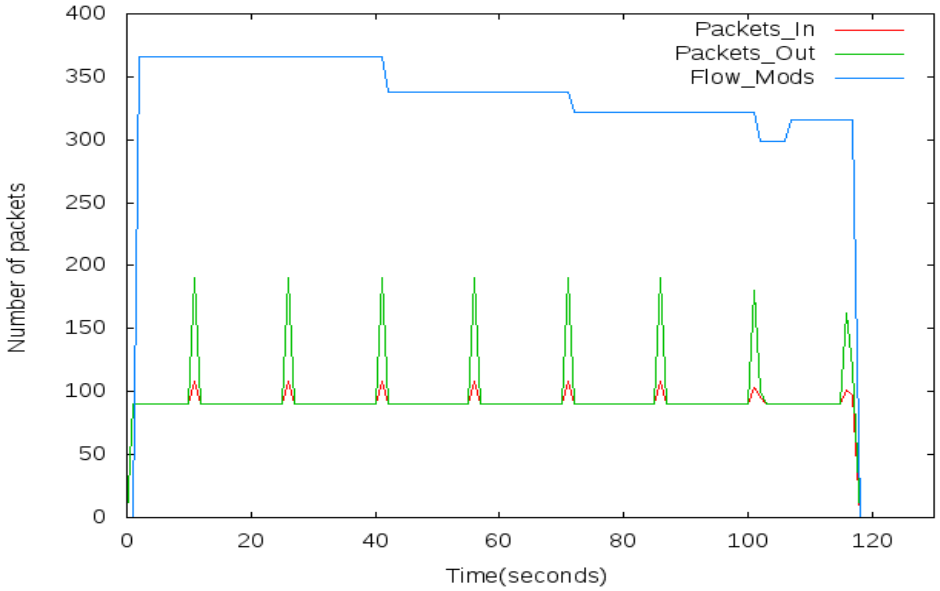


Figure 6.11: CPBeT, 9 devices

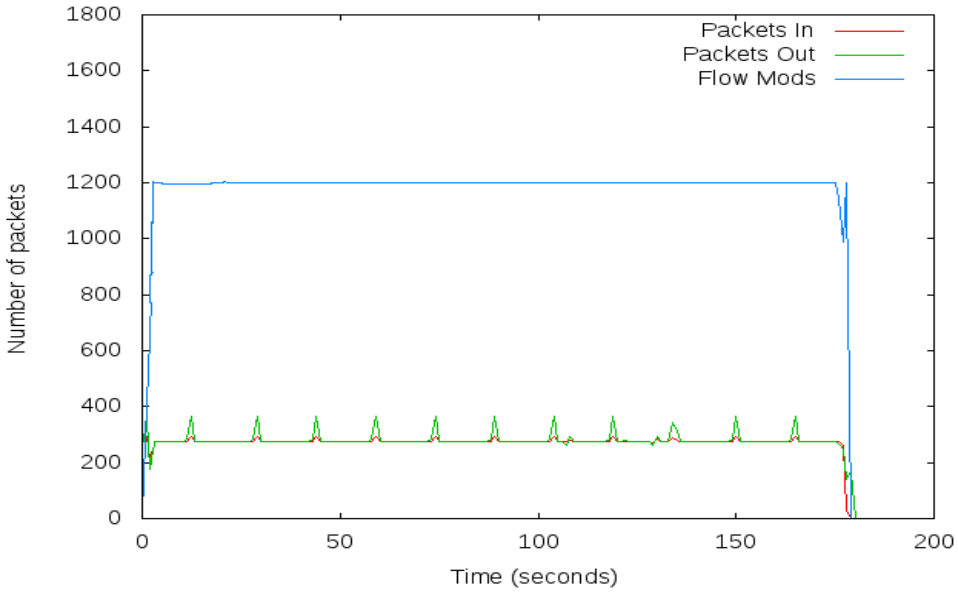


Figure 6.12: OFCProbe, 9 devices

As we can see on the Figure 6.11, as soon as the controller sends the LLDP packet, as we mentioned on the section 3.1.3, and it notices a change on the topology, it has to install different number of Flow_Mods on the switches because the paths change between some switches. In the way we changed the topology (Section 6.1.2), the number of Flow_Mods

decreases. However, on the Figure 6.12, the topology is not changing and the number of Flow_Mods is always constant, there are much more packets because the tool sends more, but the number is not decreasing along the running.

The rest of the changes can not be compare as easy as the first one, but watching these graphics we can also notice that we are showing the total number of Packets sent and received by the tool, and we can not do that with the standard tool. The other modifications we did on the tool are focused on our proposals and on the results we want to see for the experiments.

Floodlight

Dijkstra on the controller

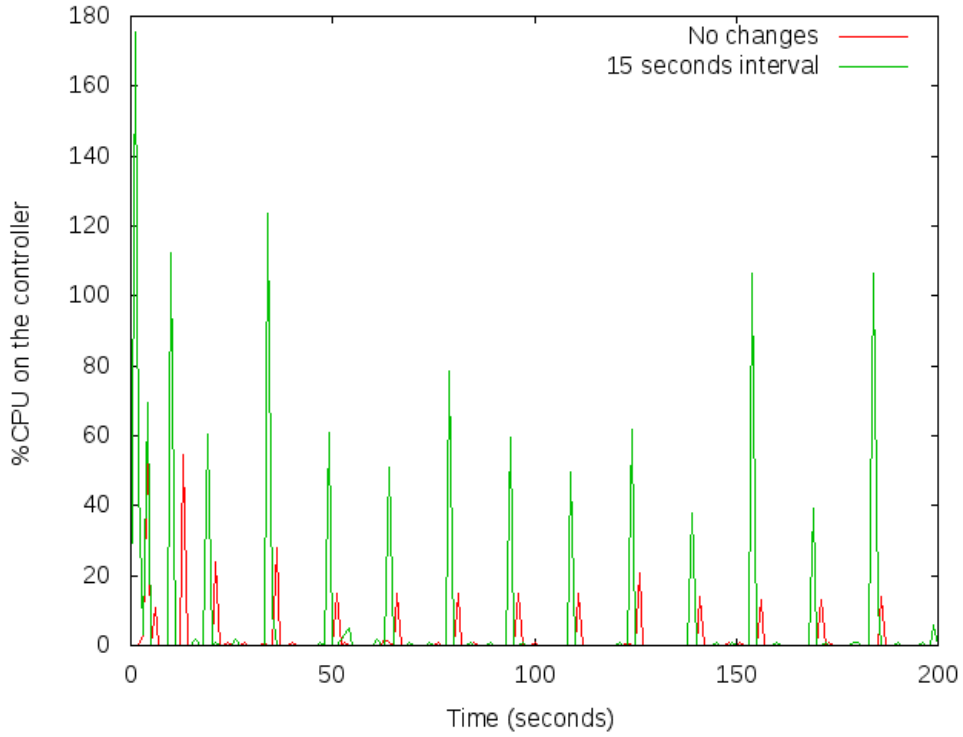


Figure 6.13: Dijkstra's effects on the CPU usage

- Topology: 20×20, same structure as showed on Figure 6.1
- Hosts: There is no hosts on the topology because we wanted to analyze only the

changes on it. The Packets_In sent by the hosts do not affect to this experiment, so we removed them.

- Topology interval: We changed the topology every 15 seconds (green line) and the second line (red) is the CPU usage without changing the topology.
- LLDP interval on the controller: It is set as default, 15 seconds.
- Time of the experiment: We set the benchmarking time to 180 seconds.

The point we would like to demonstrate with this experiment is that when the controller runs Dijkstra, the CPU usage increases. It could seem obvious, but now we know how much CPU it requires to the controller's machine. Running the controller on a virtual machine with 4 cores, we can see values for the CPU usage from 0 to 400%.

We would like to say, in order to understand the graphic, that the benchmark starts around the second 20th, so the first peaks are caused by the connection of the switches, the configuration of them and also, the first Dijkstra algorithm. As we can see, the interval between LLDP packets is 15 seconds on the rest of the experiment. The last peak, around the second 200th, is caused by the disconnection of the switches.

The most important thing we can observe on this graphic, is the difference between doing the experiment changing the topology (green line) and without doing it (red line). There is no differences during the time the controller is not running Dijkstra. Only every 15 seconds, when it is aware of the changes on the topology, we see those peaks. Seeing this, it seems that we will not be able to saturate the controller without hosts in the topology because the CPU usage is not constant, we have got only the peaks every 15 seconds.

Number of hosts

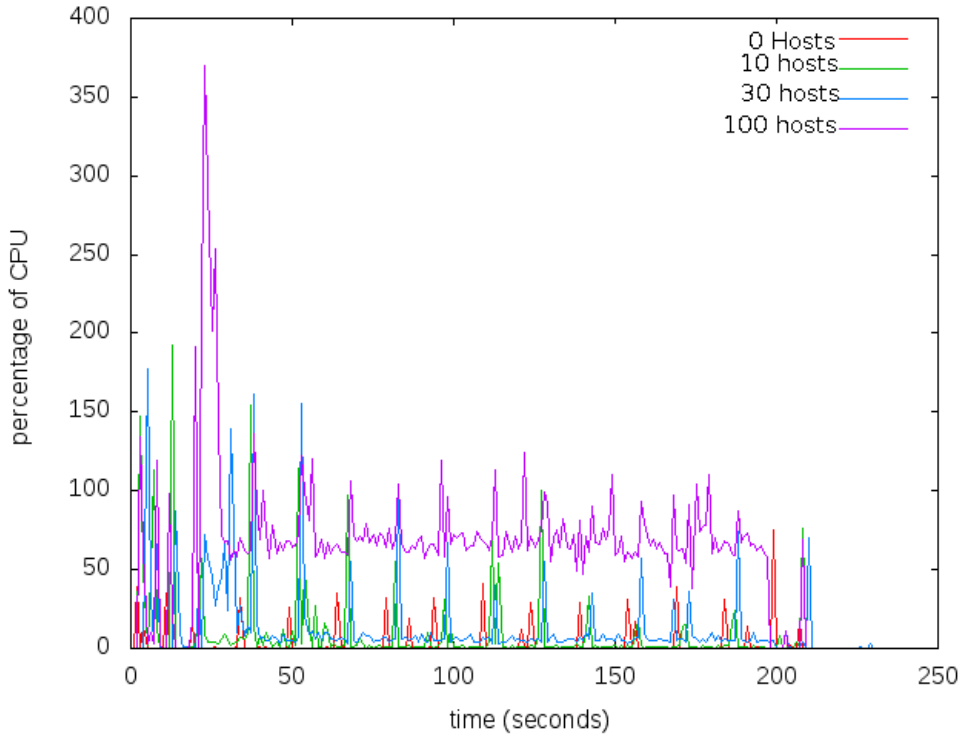


Figure 6.14: Number of hosts' effects on the CPU usage

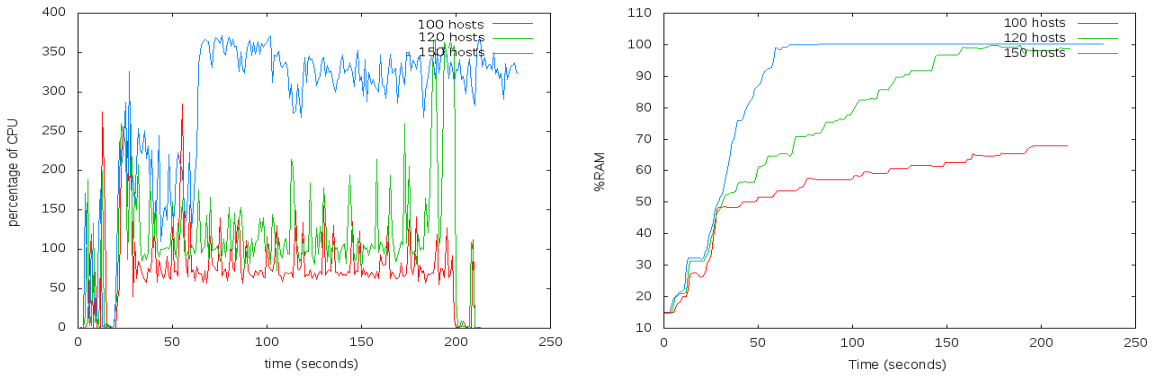
- Topology: 20×20, same structure as showed on Figure 6.1
- Hosts: we will increase the number of hosts in the topology running by running.
- Topology interval: We will change the topology every 15 seconds.
- LLDP interval on the controller: It is set as default, 15 seconds.
- Time of the experiment: We set the benchmarking time to 180 seconds.

The point we would like to show with this graphic is that the number of hosts on the topology, this means, the number of Packet_In we have through the network, affects to the controller's CPU usage. As we said on the previous experiment, this could seem obvious, but we need the graphic in order to understand how much it affects to the CPU. Also this graphic shows us that the number of hosts affects to the controller in a different way than the changes on the topology. As the second one affects only when the controller runs

Dijkstra, in this case we can see how it affects to the rest of the time. That is because we are sending the packets during the running, which makes the controller process all the packets.

Although the usage of the CPU are higher with 100 hosts on the topology, we can see the fact that the peaks produced by the Dijkstra algorithm are not growing up as the usage of the CPU does. This is an interesting fact because, as we will see later, the controller will not overload because of the peaks produced by Dijkstra, but because of the hosts in the topology.

Controller overloaded



(a) CPU (b) RAM

Figure 6.15: Controller overloaded results

- Topology: 20×20, same structure as showed on Figure 6.1
- Hosts: we will increase the number of hosts in the topology running by running.
- Topology interval: We will change the topology every 15 seconds.
- LLDP interval on the controller: It is set as default, 15 seconds.
- Time of the experiment: We set the benchmarking time to 180 seconds.

This is one of the most important and interesting experiments we did. We continued increasing the number of hosts and we could overload the controller as we can see on the figure 6.15a. When we ran the experiment with 150 hosts, we can see on the graphic

that the CPU usage goes up suddenly and it never goes down again. The controller is saturated.

As we mentioned, the purposes of these experiments were analyze the behaviour of the controller when it is saturated. Until now, we showed how we stressed the controller, and now we can see the machine of the controller overloaded. But we would like to analyze why the controller is so stressed and if there are some solutions for this.

The figure 6.15b, shows the RAM usage on the controller machine. For the beginning, we would like to remind that Floodlight runs in Java (section 3.1.1. Java runs its programs in a virtual machine (JVM), and it assigns some resources for each program. For floodlight, the JVM has 1 GB of RAM memory by default. This work is not about Java, but it is needed to explain that Java has some processes running on his programs in order to delete the objects that are not accessible any more, doing this, the JVM frees space of the RAM memory for dedicating it to another objects.[6]

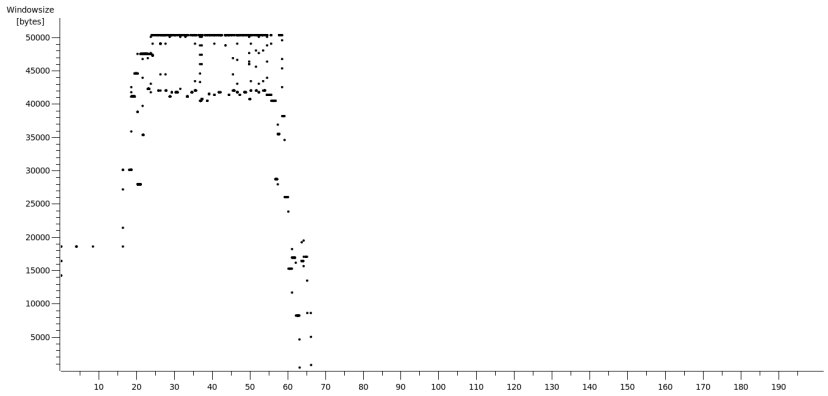
The problem with this controller is that the garbage collector of the JVM has to work all the time if we have a lot of packets and a lot of devices in the network, so it consumes all the resources in the machine. That is the reason for analyzing the consumption of the RAM memory, because comparing the two figures on figure 6.15, we can see that at the same time the CPU start to work at its full capacity and the RAM memory is full. Also we can see that the number of hosts in the topology affects in a real important way, because there is a huge different on the RAM consumption if we add more hosts.

The solution for this problem could be add more RAM memory to the controller, but it would be a short-term solution because if we added more hosts, we would have got the same problem. The best solution should be change some code on the controller in order to make it more efficient with the memory, so it does not have to ask to the machine for more.

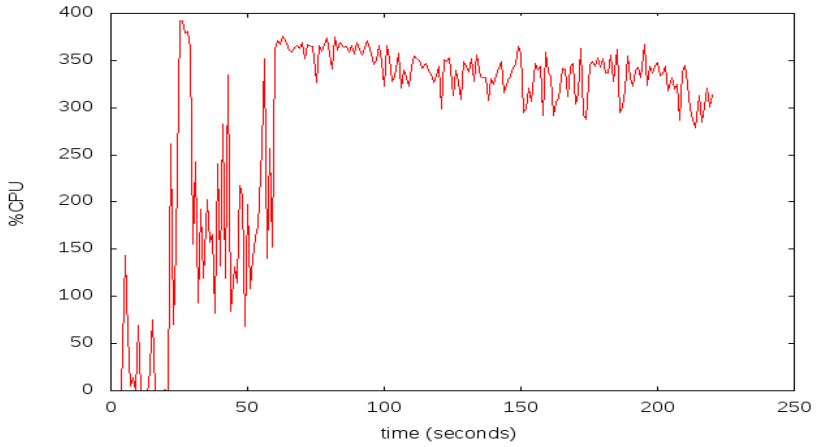
Controller's behaviour

On the previous section (section 6.3.2), we saw how to overload the controller increasing the number of hosts in the topology. Now, we would like to show with some graphics, the behaviour of the controller when it is overload. How it works and the problems it will produce for our network. The experiment we did in order to explain this, is the same as before with 150 hosts, in order to overload the controller pretty soon and demonstrate few things.

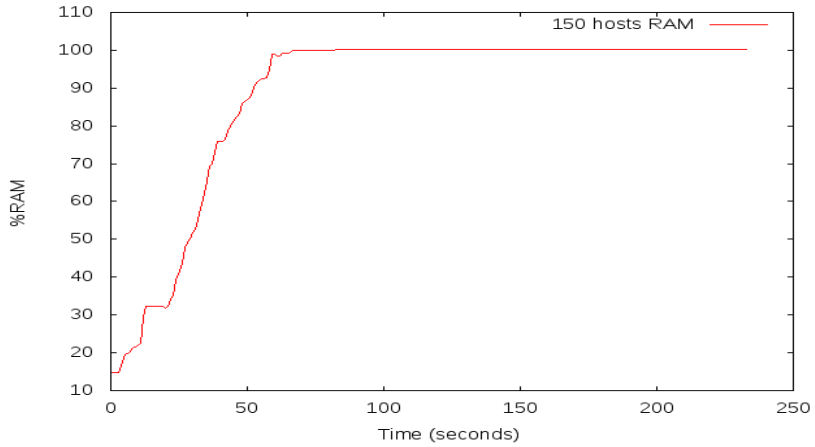
- Topology: 20×20, same structure as showed on Figure 6.1
- Hosts: 150 hosts
- Topology interval: We will change the topology every 15 seconds.
- LLDP interval on the controller: It is set as default, 15 seconds.
- Time of the experiment: We set the benchmarking time to 180 seconds.



(a) Receiver window



(b) CPU



(c) Packets

Figure 6.16: Controller's behaviour

To get started, on the Figure 6.16a we can see the TCP receiver window size for the controller during the experiment. The size is constant around 50000 bytes until the 60th second, on which the size goes down really fast until zero. This fact could show us that the controller is overload and it can not process more Packets_In because it is doing other processes (the garbage collector we mentioned on the previous section 6.3.2). But we would like to be sure about this fact comparing the three graphics.

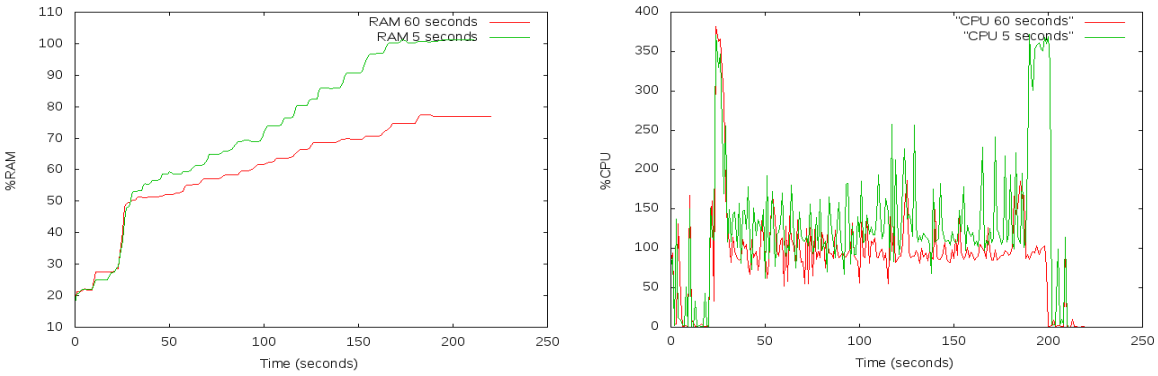
On the second Figure 6.16b, it is shown the percentage of CPU used by the controller during the running. We should notice that on the same interval as the TCP receiver window goes down, the usage of the CPU on the controller increases really fast until the maximum. This is another fact that demonstrates that the controller is saturated in that exact moment.

Finally, we would like to show the Figure 6.16, and obviously we can see that the RAM is full at the same time the controller's CPU. In fact, the lack of RAM memory is the main cause for the saturation of the controller as we mentioned on the previous experiment 6.3.2.

Summarizing, the most important thing discovered with this experiment is the fact that when we saturate the controller, it can not receive more packets, making the network totally useless, because the brain of it is saturated and it can not do anything else. We discovered the reason for this saturation as well, helping the developers to fix the problems Floodlight has with the memory.

Dijkstra and LLDP interval

- Topology: 20×20, same structure as showed on Figure 6.1
- Hosts: 120 hosts in order to overload the controller but not from the beginning.
- Topology interval: In one experiment we will change the topology every 5 seconds, and in the other we will do it every 60 seconds.
- LLDP interval on the controller: We set it in order to be the same than the topology interval mentioned above.
- Time of the experiment: We set the benchmarking time to 180 seconds.



(a) RAM (b) CPU

Figure 6.17: Changing the interval of LLDP and Dijkstra

First of all, we would like to explain why we chose those intervals for changing the topology on the tool. The 5 seconds' interval was chosen because we wanted to set a quite small interval in order to stress the controller as much as possible changing the topology. The 60 seconds interval was set because we wanted to see the difference between small interval and large one.

We would like to show with these graphics that the more times the controller runs Dijkstra, the sooner it is saturated. We can see the two lines, one with changes every 5 seconds (green) and the other with changes every 60 seconds (red). The controller sends LLDP packets with same intervals in order to make it to run Dijkstra as many times as possible and show the effects. In the graphic we can see the peaks produced by Dijkstra algorithm, there are ten on the green line between the second 50th and the second 100th (intervals of 5 seconds on 50 seconds).

As we showed on previous experiments, the controller overloads because of the lack of RAM memory on the Java virtual machine, and we can see this on the Figure6.17a, because we can see how the RAM usage increases if we set a short period for the controller's LLDP messages, but not for larger intervals, where we can see that the controller's usage increases slower and it is stable around 70% (around 700MB knowing that the total RAM memory of the Java virtual machine is 1GB). We demonstrate that running Dijkstra affects to the controller's CPU not only with the peaks seen on the first experiment (Figure 6.13), also affects to the global usage of the CPU, causing the controller to saturate even sooner. Although we should say that the Dijkstra algorithm is not the most important parameter to saturate the controller, because we have seen how the Packets_In sent by the switches

(the hosts in the topology) affect much more, increasing the CPU usage faster than the Dijkstra algorithm does. At last, our work developing a new tool in order to change the topology within an interval, was not useless, because we have seen it affects also to the controller's CPU usage. We have learnt as well the behaviour of the controller when the topology changes.

Chapter 7

Conclusions

SDN networks are being more and more important these years and we know that the SDN controllers are the brain of these networks. As the usage of SDN is increasing, we need the controllers to be reliable, and because of this needing, we developed a tool for benchmarking those controllers called CPBeT (Controller Performance Benchmarking Tool).

CPBeT will simulate a network with a predefined topology and will generate Packets_In to be sent to the controller in order to analyze its behaviour. CPBeT will change the topology in an specific interval during the running in order to load the Dijkstra's algorithm in the controller. Changing the topology and sending an amount of Packets_In to the controller, the tool will be able to saturate the controller and analyze the behaviour and the limits it has in order to work properly.

We analyzed the CPU and RAM usage on the controller, the number of Packets that the controller can send and receive during the running, and we can discover the limit of the Floodlight controller to work in the correct way, finding that it has an implementation problem with its memory on the Java Virtual Machine.

This work can continue analyzing other controllers as ONOS or OpenDayLight, which allow us to compare those controllers and to choose the best option for each topology or for each environment.

Bibliography

[1] Dijkstra's algorithm documentation.
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[2] Dijkstra ONOS.
https://groups.google.com/a/onosproject.org/d/msgid/onos-discuss/39bf8c47-4d9e-42ab-8d81-e598fe49915a%40onosproject.org..

[3] Floodlight documentation.
https://floodlight.atlassian.net.

[4] Floodlight architecture.
https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Architecture.

[5] flows ONOS.
https://groups.google.com/a/onosproject.org/d/msgid/onos-discuss/0ce15b84-be41-4328-903c-810432d8fbb6%40onosproject.org.

[6] Garbage collector Java.
http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.

[7] Install ONOS on CentOS.
https://wiki.onosproject.org/display/ONOS/Install+ONOS+on+CentOS+6.X.

[8] ONOS resource regsitrar.
https://groups.google.com/a/onosproject.org/d/msgid/onos-dev/CAPEcSOsmjLxBR7C-J%3D3c%2BmOWLyoRy9mD5VR6NNUp%3DfA2OjPavg%40mail.gmail.com.

[9] LaTEX documentation.
https://latex-project.org/guides/.

[10] LLDP.
http://vlkan.com/blog/post/2013/08/06/sdn-discovery/.

[11] ONOS wiki.

https://wiki.onosproject.org.

[12] SDN.
https://en.wikipedia.org/wiki/Software-defined_networking.

[13] SDN architecture.
https://www.sdxcentral.com/sdn/resources/inside-sdn-architecture/.

[14] SDN studies.
https://www.sdxcentral.com/reports/sdn-market-size-infographic-2013/.

[15] Introducing ONOS - a SDN network operating system for Service Providers.
2014.

[16] Disconnecting switches ONOS.
https://groups.google.com/a/onosproject.org/d/msgid/onos-dev/9bb94eef-458a-40e3-bb8b-0353a5f4c2a4%40onosproject.org..

[17] Michael Jarschel, Christopher Metter, Thomas Zinner, Steffen Gebert, and Phuoc Tran-Gia.
OFCProbe: A Platform-Independent Tool for OpenFlow Controller Analysis.
Wurzburg, Germany 2012.

[18] Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Fellow, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig.
Software-defined networking: A comprehensive survey.
October 2014.

[19] Milan Groshev.
Evaluation of the performance of OpenFlow controller.
March 2016.

[20] P. Van Mieghem and F. A. Kuipers.
Concepts of Exact QoS Routing Algorithms.