Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare

LUCRARE DE DIPLOMĂ

Aplicaţie Grafică pentru Generarea de Pachete

Conducător Științific:
Prof.Dr.Ing. Răzvan Rughiniş
As.Drd.Ing. Mihai Carabaş
Ş.L.Dr.Ing. Laura Gheorghe

Autor:
Oana Niculăescu

București, 2014

University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Graphical Packet Generator

Scientific Adviser:
Prof.Dr.Ing. Răzvan Rughiniş
As.Drd.Ing. Mihai Carabaş
Ş.L.Dr.Ing. Laura Gheorghe

Author:
Oana Niculăescu

Bucharest, 2014

This page is intentionally left blank.

Abstract

It is always a challenge to keep the state of the network consistent and to be able to say at any moment what packets are getting through it. Software Defined Networking comes in aid and offers an alternative to the old way networking administration is done, offering the administrators the possibility to control the network through a custom written software. We propose in this paper the development and testing of an application that will help with the solving of this task. Our goal is to show the advantages and disadvantages that a Software Defined Application has at the moment.

Contents

List of Figures

Chapter 1

Introduction

As Andrew Tannenbaum was noting in his work [1], computer networks and the computer industry has seen great development over a short span of time. They have evolved from the centralized unit, where a single large machine was serving all the needs of a company, to largely distributed systems, that are located all over the world.

But, what actually changed for the computer network industry is the dimension of the hardware. If 40 years ago computers were occupying an entire room, now we can have our own router at home and keep a real system in our pockets. What did not actually change are the principles that are governing the computer industry. Today, we are still using protocols that were written 30 years ago, when the first Internet connection was established [13].

The network industry innovation came at the speed of the hardware innovation, at how much smaller and faster the routers and switches were getting. There was an improvement in the speed of processed packets per second for the equipments, but the same protocols are being used. So, the concepts behind networking are those from the beginning. But, back then computer networks were not designed for the amount of traffic that is experienced nowadays.

1.1 Context and Motivation

The domain of computer networks is closed to innovation. One of the reasons for the slow innovative process is that computer networks are made out of switches, firewalls, routers - complex equipments that are running a routing protocol. In order to be able to run a routing protocol, all of those equipments are implementing a distributed system [21].

A representation of this situation, of an industry that for every new routing protocol needs to implement another distributed system, can be seen in Figure 1.1 [21]. At the base of the networking industry we have the specialized hardware, from different vendors, each with a specific set of features. On top of that we come and add a network Operating System, that will facilitate the use of the hardware functionalities and will present the user with information about the machines. Usually the operating system is vendor specific. And then, over the operating system, different routing protocols are implemented. The core of every routing protocol is the distributed system.

Even though those distributed systems might be very similar between two routing protocols, they are different enough to ask for a stand alone implementation. They have different particularities and can not be integrated in a common layer between the operating system and the actual routing protocol.

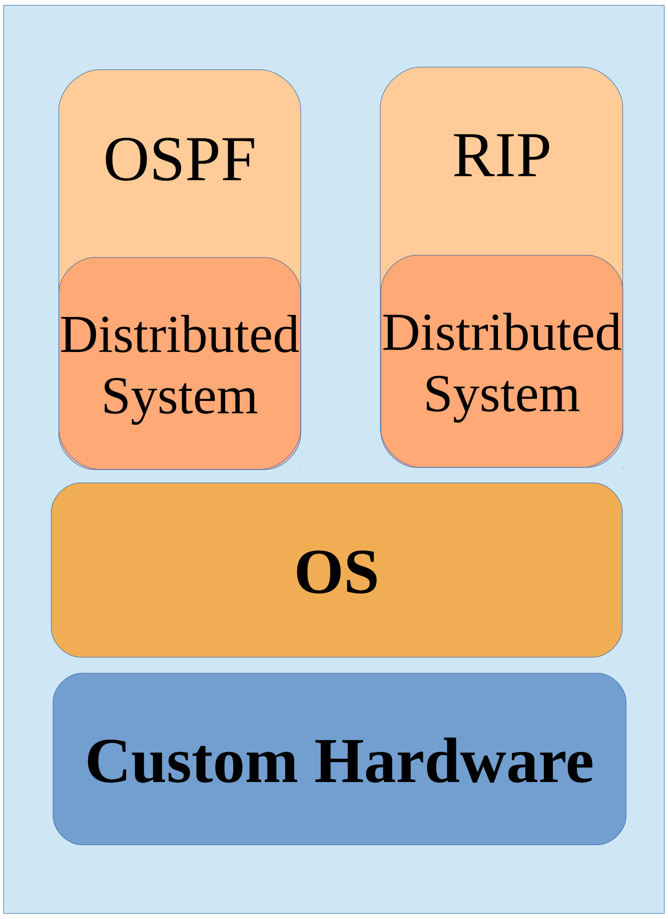


Figure 1.1: Vertically integrated, closed protocol development

Developing a new distributed system is a long lasting, trial and error kind of process. And, after the initial distributed system is created, it has to wait for years until that protocol becomes a standard and then it is deployed in production. After all this work, the developers hope the new protocol will be adopted pretty fast, though that may not happen at all, a good example in this case is the IPv6 protocol.

Even though we ran out of IPv4 addresses in 2011 [25] and IPv6 has been standardized since 1998 [26] it has not been adopted as fast and by as many users as they initially predicted. The reason for the low adoption rate of IPv6 protocol is that it is based on changes that must come from the equipment vendors. They have to create new equipments that will support the IPv6 protocol - that means they have to invest in new technology and push back the actual technology that is bringing them revenue.

That is a lengthy process and every vendor of network equipment should do it. So the industry chose to go along with methods that will prolong the transition period from IPv4 to IPv6 for as long as possible. This corroborated with the slow process of devising distributed systems makes the industry closed to innovation. Besides that, another reason why the innovative process is so slow, is that the industry is based on proprietary hardware. Every vendor has its own set of features [21] that are based on protocols and specifications written 30 years ago [13].

Besides the lengthy process of standardizing new protocols, the networking industry has to deal with other hurdles too. For example to set up a network today, administrators have to log in every equipment and set up routing policies, test, adjust and test again. This process is repeated every time a new policy is being set up or a new equipment is being added to the network. Other than that they have to deal with different environments and interfaces for equipments from different vendors or even between the same vendor's equipments. This makes interoperability impossible and innovation almost close to zero.

A solution to some of those problem could be represented by Software Defined Networking (SDN). SDN is a new paradigm that hopes to come and change the way networking is done and how computer networks are managed.

1.2 Project Description

Software Defined Networking is represented by the separation of the control plane from the forwarding plane [17]. The control plane is a software controller situated between the network operating system and the routing protocols, that are deployed in the network.

In light of this definition, we propose a way of testing the capabilities of the SDN paradigm by implementing and testing a packet generation application - Graphical Packet Generator. The application uses a virtual machine by way of simulating the use of real life routers. That means that security or performance concerns will not be addressed within this work.

What we propose to analyze in this thesis is the use of SDN when it comes to implementing an application that will permit us to generate packets from a central point, and the limits of the API that will be used in developing this application. We will discuss about the different aspects of the software implementation and the restrictions the API presents, restrictions that at the moment make the development of software a hard job. We will also address the fact that this is just a virtual machine simulation and on real life machines the software might not work as expected.

The project testing is based on a graphical application that will connect to a virtual network, made out of routers. After connecting to the network, the software will use a routing protocol to automatically discover our network topology and extract useful information, like interfaces names and interfaces MAC address.

After having those informations about the network topology, we can generate traffic inside the network, providing a source MAC address and a destination MAC address, as well as source and destination IP addresses. There is the possibility of generating ICMP, UDP or RAW IP traffic, with a custom payload in the transmitted message.

Later in this thesis we are presenting different stages of the application, with a series of restrictions imposed by the API used for developing it. The final limits section addresses mainly restrictions and hurdles that came up when we tried to set up a new virtual network, on which we deployed the application. Another problem we talk about in this last section is the hardships we encountered when we wanted to have a general view of the network. We needed such a view in order to be able to discover the whole topology, and later to generate packets inside the topology.

More details about the work that needs to be done and other restrictions are discussed in the packet generation section (Chapter 4.4). Some of those limits are the impossibility of generating level 3 packets. We were able to generate only level 2 packets. The packet generation is possible just in some of the API's languages, so an inconsistence with the API is another problem. At the same time, the packet generation process is not straightforward, and in order to be able to inject packets you have to build them from the ground up in code, with no facility to help you and ease the process.

In order to be able to develop this project we used a virtual machine with router virtualization capabilities. We created a virtual network inside the machine and deployed our application inside that network. The Graphical Packet Generator is developed using the onePK API [7], provided by Cisco inside the virtual machine. The API offers the programmer the possibility to hook inside a router and be able to call the router's operating system functions.

1.3 Project Objectives

In our work we are trying to offer answers for a few questions. The questions are about how and why is our SDN solution relevant, how can we use the onePK SDN API to implement a Graphical Packet Generator application and what are some of the problems we faced during the development process. The answers to those questions are outlined as main objectives next.

The general goal of the project is the design and implementation of a Graphical Packet Generator application, that is consistent with SDN principles. The software will run in a controlled environment, like a virtual machine.

We will show a way of constructing a new topology inside a virtual machine. Also, we expose some of the limits presented by the environment and the way the topology is created.

Another objective of our work is connecting to a virtual network element. Our aim is to have a functional connection established between the application and one of the routers. Then, we obtain pertinent information about the state of the equipment.

Our next goal is implementing and testing a way of automatically discovering the network topology, after being able to connect to a network element. We explain some of the restrictions of the API and the application when it comes to automatically topology discovery.

And last but not least, testing the possibilities of implementing a packet generating software, by offering a SDN implementation using the onePK API. We are going to highlight the limits encountered during the development phase and what can and can not be done using the API.

The main motivation behind the Graphical Packet Generator project is the better understanding of Software Defined Networking and how it works. Also, we want this project to present in an objective way the achievements and limits of the API used in developing the software. The limits of the API will be shown during the implementation process. We will be talking about the available solutions but also about improvements that could be added to the API, in order to make it more accessible and usable.

The intent of this project is not to offer an unique way of doing networking administration. The project wants to offer an alternative view on networking and computer networks, and tries to analyze the overhead added by such an application in a network, albeit a virtual one. We also try to analyze the ease of use and the constraints imposed by such a paradigm.

In the next sections we are going to further discuss what Software Defined Networking is, or how it is understood in the industry at the moment. We are going to present different SDN flavors and the differences between them. And we will present a Software Defined Networking application and its limits as stated by the objectives of this thesis. The work in this thesis will concentrate on the use cases of the API in the context of packet generation and automatically discovering the network topology. We are going to present how we are able to develop such an application starting with smaller use cases applications and what their constraints are.

Chapter 2

Background

The raise of the Internet and the unprecedented high amount of data that is transient through networks nowadays, makes networks administration hard. The impossibility to automate the most part of setting up a network and the need to be able to have physical access to the equipment in order to be able to log in on every one of them and do the setting up, made the networking industry a domain where progress is slow.

We are going to present next some background information on how current networks are working and to give an alternative to the old way of doing things, by using Software Defined Networking.

2.1 Networking Basics

Before we are going to talk about what the application architecture is, we have to know how the current networks are working. Every process of a router or switch from a network can be assigned to one of the following three conceptual planes of operations: management plane, control plane and forwarding plane. The management plane configures the control plane and offers the network methods to access it. The control plane is where every decision is being made and decides the policies on which packet forwarding is done. And the forwarding plane is the one that moves packets from the input interface to the output interface.

To understand those planes we have the example of an user that logs inside a router command line interface - CLI (the management plane) using SSH (secure shell) and is configures a routing protocol for the router to exchange information with its neighbors (the control plane), which gets installed in its local routing table (the forwarding plane).

Everything happens at the level of one device and we are talking about a network of many independent devices, that is supposed to take a consistent routing decision without actually having the whole picture of the network. Packets are being exchanged by neighboring routers, those routers base their routing decisions solely on their own routing table. So, for the whole routing process to be consistent the neighboring routing tables should be consistent.

Software Defined Networking (SDN) comes and changes this paradigm. SDN means that the control plane is decoupled from the forward plane. So, instead of each node having to take routing decisions about forwarding packets, a centralized software based controller is taking those decisions for them, the controller knowing the full network topology.

One definition of SDN could be "A network in which the control plane is physically separate from the forwarding plane and a single control plane controls several forwarding devices." [22]. That describes the separation between the forwarding plane and the control plane, the former forwards the packets according to a set of rules - a match-action rule - and the latter decides what to do with the packets and effectively permits the set up of rules.

Having this definition in mind we can come up with an use case for Software Defined Networking. We have a centrally managed network, with specific requirements. This means that putting together some equipments, like routers or switches is not enough. Those equipments should be able to perform certain tasks and take consistent decisions. The problem is that we have no abstractions or layers, no easy to use APIs, no easy way to manage the network. SDN is the idea to solve this problem.

A first important benefit of SDN, that gives it an advantage over the distributed control plane, is that it can take forwarding decisions globally at a higher level of abstractions. The forwarding decisions are not taken individually, by each equipment. A central controller takes them, taking in consideration the situation in the whole network. A representative example in this case is a level two network, that has loops, or redundant paths. To be certain that the final topology has no loops we usually use a spanning tree protocol - this is a process on the control plane level. We need such a protocol because individual equipments can not know the entire topology. But, if we would move all the control plane level functionalities to a central controller, this one could be able to see the entire topology. In this situation the controller will base its forwarding process on a consistent known state. This situation is represented in Figure 2.1.

A second advantage of a central controller is the fact that it provides an easy to program interface. It is important to have such an abstraction as a program interface because using this other applications can control the network and its resources. The packet forwarding decisions can now be taken by these applications. We could have mobile devices in a network that get moved to another network. The controller can be programmed to migrate also the QoS policies or the firewall rules at the same time as the devices is passed to the other network. This way an administrator will not have to reconfigure manually both the source and destination networks, and the states in both of those networks will be kept consistent.

2.2 A Short History of SDN

Software Defined Networking has been mentioned more often lately, as well as Software Defined Anything (SDx). The last few years have seen an ascending trend into calling different technologies Software Defined. But no one really explains what software defined networking for example means. That is mainly because the term is so new and different parties have been stretching it into a marketing symbol, in order to be able to encompass their different objectives and strategies.

SDN has its roots in the way old telephone networks were functioning [11]. There was a distinctive separation between the control plane and the forwarding plane. So the technology that SDN is based on has evolved over the past 20 years. The aim of this long time development was to make computer networks more programmable, and SDN borrowed this concept also from old telephones networks, even though those were more programmable at the data plane. SDN gives the user that kind of flexibility at the control plane.

The history of SDN can be divided in three development stages: the active network stage, the stage where the control and data plane are separating and the adoption of the first network operating systems at large [11]. Besides those three stages of SDN evolution, we should also mention the virtualization stage, that was one of the first real life use case for SDN.

The early stages of SDN started at the beginning of the 1990's, once computer networks and Internet became widely spread inside universities and researching circles. Those who had access to the Internet wanted to open up the networking industry to innovation by exposing a set of functionalities from each individual equipment with the help of an API [11]. This was the

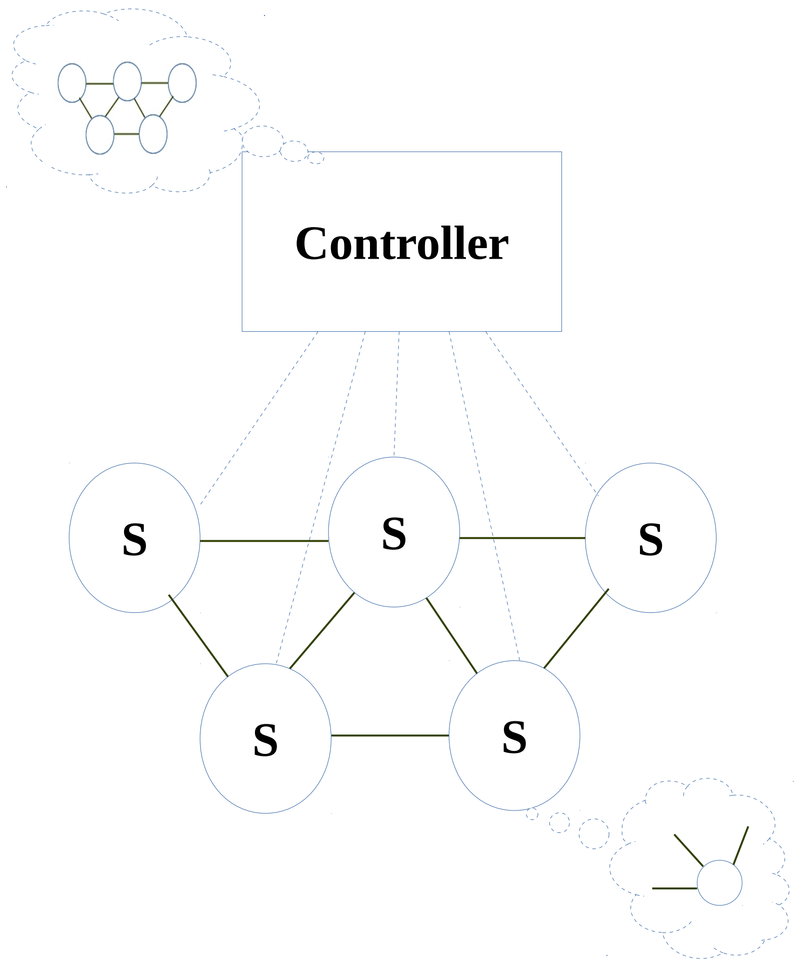


Figure 2.1: A central controller that has a complete image of the network

beginning of the "active networking". They developed two programming models for this type of networks: the capsule model [28] and the programmable router/switch model [2]. The first model meant that the code that would be executed on each equipment was carried in-band inside the data packets – coda was carried by data packets without any sequence prefixes, the second model meant that the code was established by out-of-band mechanisms [11] [2].

Active networks are of great importance because they sustained the idea of more easily programmable computer networks. The motivation behind SDN and the development of networking APIs was that innovation at the speed of hardware innovation was slow, and the paradigm should be shifted towards faster paced innovation, at the speed of the software. So even though the emphasis in the case of active networks was on programmability at the data plane level, it opened the road to innovation at the control plane too.

Another plus that came out of the active networking phase was identifying the need for a unified middle box layer. The functions of the middle boxes would be used as a unified entity with the help of an API. This vision was not accomplished with active networks, but opened the door for the different SDN APIs.

The separation between the control and data plane came as a demand from the industry. The use case for this separation was the need for traffic engineering - or the strict control of the data path. The separation between the two give way to two major models: an open interface that lays between the control plane and data plane, such as ForCES (Forwarding and Control Element Separation) [29] and the logically centralized control of the network, represented by RPC (Routing Control Platform) [3]. The effort to separate the two planes was an innovation added in the networking industry with an emphasis put on helping network administrators and not end-users. But there were some derived concepts from this attempt that will be adopted later by the SDN paradigm. The ForCES model proposed a centralized controller that will use an interface to interact with the data plane. Also, from the RCP SDN borrowed the distributed state management.

In the middle of the 2000's, Stanford create the Clean Slate program that is the base of SDN and gave us the first SDN implementation, that will later become an open standard, the OpenFlow standard. OpenFlow relayed on the use of existing hardware, a fact that made it almost instantly deployable. The standard adopted all the earlier ideas that were researched in the domain, and gave network administrators the possibility to control the network using an API and having a clear separation between the control and data plane.

2.3 Different SDN Solutions

There are at least four totally different Software Defined Network solutions today, and more may come along. Current solutions include: a "software-switch-router" model, an "overlay-network" model, an "API-and-Protocols" model, and the purist OpenFlow model.

The “software-switch-router” model is a software implementation of current switching/routing and no special SDN features at all.

The “overlay-network” or the “Nicira” [24] model is a solution proposed by VmWare. In this solution SDN is deployed as a new virtual level on top of current equipment level [11] [24]. In general, an overlay network is a network that runs on top of another one. In this category we can add cloud provider networks, content delivery networks - CDNs, virtual private networks – VPNs.

The "API-and-Protocols" model [5] is a distributed one and it is implemented by Cisco, the goal of this model is to add features to current devices/protocols.

The purist OpenFlow standard [20] is managed by the Open Networking Foundation and is derived from the original standard developed at Stanford. We will talk about the differences between the last two models further in this section.

OpenFlow is a flavor of the SDN architecture. It is a simple protocol for remote communication with a router data path. The first draft of OpenFlow [20] was written on the principles of Ethane [4], with a focus on network management and access control. The idea behind Ethane is to allow network administrators to declare policies for the whole network using a central controller. The policies implemented are used by the controller to check every new generated data flow against a set of rules.

The initial protocol was able to establish rules for packet routing inside the network, to forward packets to a central controller for processing - initially every first packet from a new flow had to be forwarded to the controller - and to drop entire flows of packets. OpenFlow is a set of rules and actions, with a match action executed on a data flow and a rule.

But all of those things could have been done before, without having to add another “protocol” on the network infrastructure. The difference is that now it can be done in a centralized way using a control interface or an API. It will not solve all network problems and it is still dependent on the capabilities that are supported by Ethernet - meaning that with OpenFlow, Ethernet will still lose packets when congestion occurs - seeing that Ethernet and IPv4 are best effort protocols, but it offers an alternative way to see and maintain the network in a consistent state.

Cisco Open Network Environment (ONE) is the Cisco SDN solution and includes Cisco onePK, Cisco One Controller and a few top layer network solutions [5]. Cisco onePK, short for One Platform Kit is a SDN implementation offered as an API by Cisco. More exactly is a set of programming libraries. Those libraries let programmers build applications that are integrated and interact with Cisco manufactured devices.

There are several problems that onePK tries to solve. Some of those problems are the need to have greater and more precise control over the flows and routes in a network, the need to extract particular packets for modification and re-injection, the need to offer a real Quality of Service functionality and to add new services to the network without having to replace all the hardware.

While OpenFlow is the initial theoretical standard from which most of the network equipment producing company started to develop their own SDN strategy, onePK is a concrete implementation of it. The basic idea behind it is the same as OpenFlow - separation of the data and control planes.

There are a few service sets provided by this API, some of them are basic service sets - that provide capabilities of the application to hook into the packet flow and extract packets that can be modified and then can be injected back into the flow, or the possibility of configuring Access Control Lists, QoS policies, routing policies. The extended or optional service set provides means to access the virtual terminals on the devices and path trace possibilities.

2.4 onePK API

To be able to offer the facilities of programmatically controlling the network we need a SDN implementation. Our application - the Graphical Packet Generator - is being developed using the Software Defined Networking solution proposed by Cisco, the onePK API [8].

There are several reasons why we need an API for our application. Kyle Forster said about OpenFlow that is “like x86 instruction set” [16], what he meant is that the technology is very powerful but at the same time is very hard to get a grasp of it. We know we can use it to create a powerful computer, but when we are faced with the fact that we need to create an operating system or a game things are becoming less obvious.

What onePK API actually does, is that it creates an interface between the “x86 instruction set” and the programmer, giving him tools that he can control more easily. OnePK allows programmers to write applications that can make direct function calls to the routers operating systems.

It gives us the possibility to change routes in the network, make policy decisions - restricting or permitting access to information -, enforce QoS and interact with packets by changing their source/destination, dropping or injecting them. It enables the use of those x86 instructions like without actually knowing assembly. We can think about it like C/C++ and other high level programming languages that let software engineers program applications.

Applications written using the onePK API can be hosted in one of the following places: in a Linux environment, on the network element itself, on a dedicated hardware blade inside the same equipment as the network element and it can also be deployed on an end-host server, meaning it can be completely decoupled from the hardware that it will control.

Sessions are used for establishing the communication between the application and the network element and are one important aspect of a onePK application. A session is created by using the IP address or the hostname of the network element. To be able to add a session between the two you need to be able to perform authentication - providing a valid username, password and authentication certificate. After the session has been established, it will be referred using a session handle.

To be able to give applications different functionalities onePK uses service sets. There are base service sets and extended or optional service sets. The base service sets provide the minimum level of functionality and are supposed to be implemented by most of the future equipments. This set lets the user to hook into the packet flow and generate new packets (`Data Path`), set up routing and ACL rules using the `Policy` and `Routing` Service Sets, connect to a network element and obtain information about it (`Element` Service Set), discover the topology of the network using the `Discovery` Service Set, and register callbacks events and functions using the `Configuration` and `Event` Service Sets.

Chapter 3

Solution Design

In this thesis we are proposing the testing of the SDN API by the way of implementing an application that will give the user the possibility to automatically discover the network topology and to generate traffic inside the network. Next, we are going to present the general design of the application with a description of its functionality and analyze some of the design choices we had to make during the development. In the following section we will present in detail each part of the construction of the application and analyze the limitations that the API presents with concrete use cases.

The Graphical Packet Generator is a software application designed for generating traffic inside a virtual network. The application we developed was tested and run inside a virtual environment, the All-in-one-VM [9], which provides a copy of the onePK SDK and one Network Device Emulator (NDE) - Virtual IOS (vIOS). The vIOS instances are emulating each an ISR G2 router, which is the only supported vIOS device that the onePK API has at the moment.

The application was written using the Java and C programming languages. The onePK API has implementations for both these languages. The choice to write the application in a combination of Java and C was determined by the API itself. The onePK API divides its functionalities in different categories called service sets. And the developers of the API have not made the `DataPathServiceSet` (DPSS) available in any other API implementation than the one for the C language, even though there are API implementations for the Java and Python programming languages as well.

In order to be able to connect to the virtual devices, the user has to provide a correct set of credentials, a network element root certificate and a certificate for Java Secure Socket Extension [23]. We need to provide these elements because the only connection that it is available between a onePK application and a virtual network element is a secure connection, TLS connection [27].

The application is made out of three main parts, two of them could be used as standalone programs. A first part is the topology discovery option of the program, where an automatically discovery of the network is done. A second program is the actual packet generation, that is represented by a graphical interface, the driver program and the C module. The driver program is the one that calls native functions from the C onePK API and generates packets.

Between those two modules of the application, we developed the main module. This module establishes a connection to the network element, calls automatically the topology discovery module for the user and starts the Graphical User Interface (GUI) after. Some of the details of the implementation can be visualized in Figure 3.1. The figure represents a simple description of the component classes of the application, with only the relevant fields present.

We are now going to further explain the design of the application and some of the choices we had to make during the implementation. Those choices were mostly influenced by the constraints presented by the API. The main program of the application is the one that is responsible with the establishment of a connection between the onePK API and the virtual router. The program receives the credentials for connecting to a routing element and also the certificates for the TLS secure connection with their client keys. The application must receive the C and Java certificates, because later we are going to connect to the network element using a C written program too. The application also receives an element IP address or hostname in order to be able to connect to that virtual element.

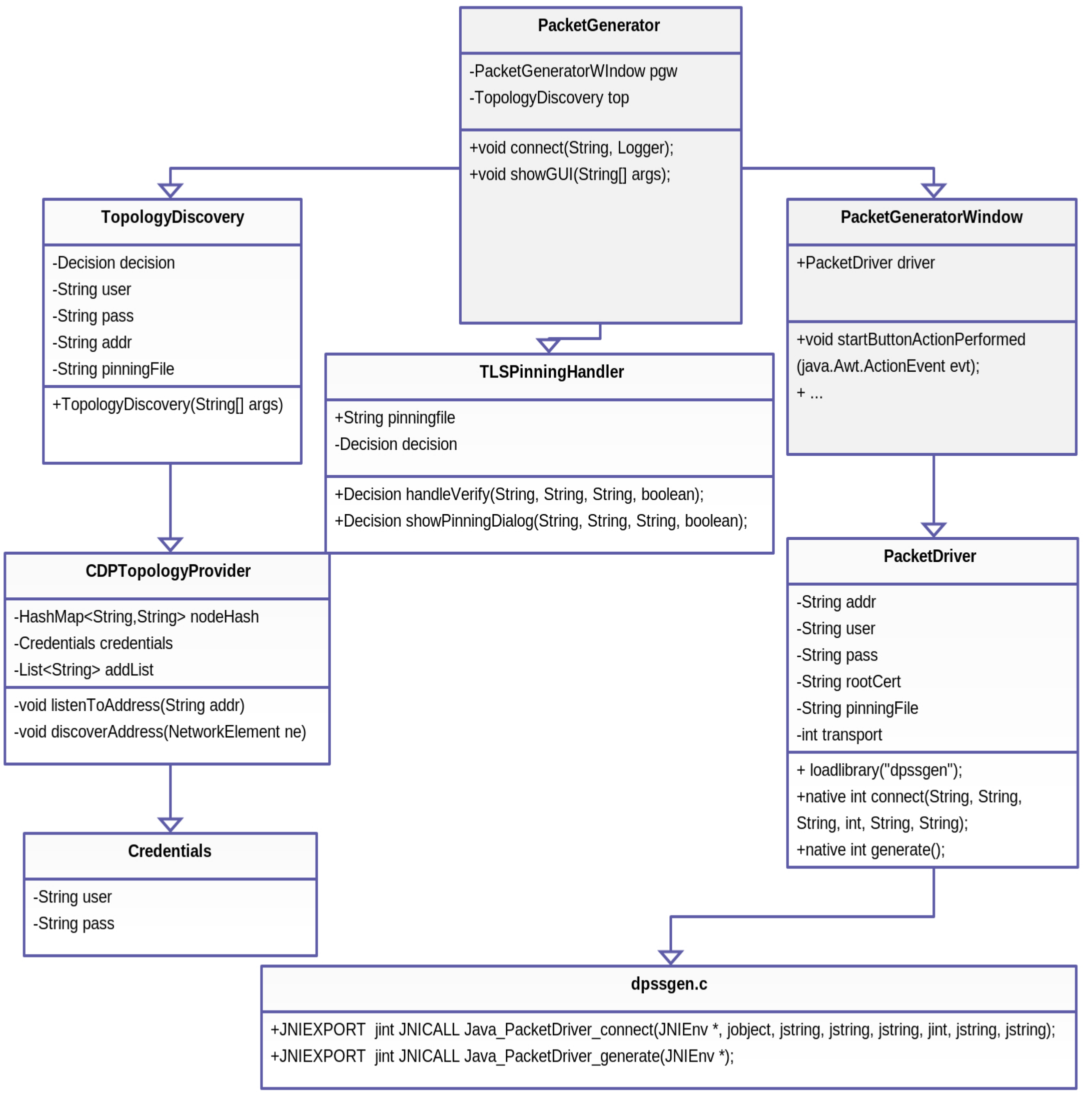

PacketGenerator
-PacketGeneratorWIndow pgw
-TopologyDiscovery top
+void connect(String, Logger);
+void showGUI(String[] args);
TopologyDiscovery
-Decision decision
-String user
-String pass
-String addr
-String pinningFile
+TopologyDiscovery(String[] args)
PacketGeneratorWindow
+PacketDriver driver
+void startButtonActionPerformed (java.Awt.ActionEvent evt);
+ ...
TLSPinningHandler
+String pinningfile
-Decision decision
+Decision handleVerify(String, String, boolean);
+Decision showPinningDialog(String, String, String, boolean);
CDPTopologyProvider
-HashMap<String,String> nodeHash
-Credentials credentials
-List<String> addList
-void listenToAddress(String addr)
-void discoverAddress(NetworkElement ne)
PacketDriver
-String addr
-String user
-String pass
-String rootCert
-String pinningFile
-int transport
+ loadlibrary("dpssgen");
+native int connect(String, String, String, int, String, String);
+native int generate();
Credentials
-String user
-String pass
dpssgen.c
+JNIEXPORT jint JNICALL Java_PacketDriver_connect(JNIEnv *, jobject, jstring, jstring, jstring, jint, jstring, jstring);
+JNIEXPORT jint JNICALL Java_PacketDriver_generate(JNIEnv *);

Figure 3.1: Graphical Packet Generator – the structure of the modules

After reading the property file, where those initial input values are stored, or parsing the command line arguments, the application connects to the virtual network element using the credentials provided. In this part of the program we are setting up Differentiate Service Code Point [10] values, that are the way Cisco routers, the provider of the onePK API, are dealing with the Quality of service problems. What this means, is that based on those values the traffic is treated by the network's routers based on the Type of Service field (ToS).

The main part of the application, the `PacketGenerator` module in Figure 3.1, has a reference to a `PacketGeneratorWindow` that will instantiate a graphical object of the application and another reference to the `TopologyDiscovery` module. The connect function from the `PacketGenerator` module is the one that will be used both for the topology module, as well as for the graphical interface. Only one instance of the connection will be established for both of those modules. A seemingly redundant connection will be made in the C library but we will talk about this later in this paper.

The `TLSPinningHandler` module is used for the overriding of the TLS certificate verification. In the module we are implementing a mechanism by which certain hosts are white-listed for TLS verification. It is a way of making a connection succeed even though the verification of the certificate fails. With the condition that the certificate presented in the TLS pinning file is the same as the host certificate that wants to establish a new connection.

One of the most important modules of our application is the `PacketGeneratorWindow` module, where we are implementing the graphical interface of the application and, at the same time, the module that connects the main application with the C developed module. The graphical part of the module is implemented using Java Swing [19], actually a subclass of it - JFrame, and the application presents the user with a single-do-it-all window. Inside the window the user can select or insert the source and destination MAC address for the packets and the source and destination IP addresses and the type of packets it wants to generate.

By default the onePK API allows the creation of only three types of packets: ICMP packets [14], UDP packets [12] and IP Raw packets. The last category of packets means that the program can transmit and receive any Internet Protocol (IP) packets without a specific transportation level. The user can also add a payload to the generated packets, the same for all the packets generated in a series.

Besides the actual graphical interface initialization, this module instantiate a driver module for the C program too. The driver module will be instantiated on a new thread in the application, the reason we chose to use a separate Java thread for the C application is that we want the interface to be still responsive to user commands even though we started the packet generation process.

The `PacketDriver` class is the one that will make the connection with the C program and implements the `Runnable` interface in Java. We choose this method of implementation over the other option of extending the `Thread` class in Java, because this presents a series of advantages for our program.

First, is the separation of concerns principle, implementing `Runnable` being the correct way of doing it from an Object Oriented Point of view. Composition is the better solution. We give the thread something to run and we are do not modify the thread to do our binding. Second, Java supports only single inheritance, so we can extend just one class. This is not relevant for the `PacketDriver` class, since we are not going to extend this class beyond this at the moment, but for the `PacketGenerator` class this makes a difference. The `PacketGenerator` class already extends the JFrame class, so we could not have extended a second class if we wanted to use the `Thread` option instead. So, in order to be consistent we decided that we will use the `Runnable` interface implementation, instead the Thread class extension.

The `PacketDriver` class only contains the definition of the connect and generate methods defined in the C part of the program. The call to this functions is done using Java Native Interface [18]. This framework allows Java code running in a Java Virtual Machine (JVM) to call native code. The native code must be packed as a library and can be written in other programming languages, like the C programming language.

The `DataPathServiceSet` library that is created using the C onePK API is loaded in the driver program, that will later call the native functions. Inside the library we are using onePK

API functions from the service set to connect and generate new packets inside the topology. In this situation we are not going to use the connection established initially by our application with the virtual network element. We need to establish a new connection to the topology and for that we receive the transmitted parameters from the top application to the connect method. We can not use the initial connection to the network due to an API limitations, there is no work around to be able to share the same session handle and session connection between two different implementations of the API at the time this application was written.

The automatically topology discovery part of the application is done using a routing protocol for the discovery of the neighbouring routers of the router we are connected to. We are using Cisco Discovery Protocol (CDP) [6] for discovering the neighbours of our router and their interfaces, that will become available destination interfaces. The `Topology` class just instantiates the object and calls on the CDP event listener in order to register the up and down events for neighbouring interfaces.

CDP has some limitations, like the discovery of the routers can not be done directly for routers that are at a greater distance than one node. So the protocol only discovers the direct neighbours and for further discovering it needs to be run recursively. We were constrained by the API to use this protocol, since we could not find a viable SDN alike alternative for automatically discovering the topology.

The topology discovery is done by a series of shutting down and bringing back up again the interfaces. The CDP protocol listens on all the interfaces for up and down interface events and register those events from the neighbouring routers. The `Credentials` class saves the logging information need to connect to a network element. Another connection should be established to the network element in order to be able to command neighbouring interfaces to go up and down.

The `Topology` module returns to the main module a list of available interfaces from the neighbouring routers. The available interfaces will be used as an option in the graphical user interface, for the user to be able to choose a destination IP address.

With the Topology module we conclude our short presentation of the design architecture of our Graphical Packet Generator. We tried to present the main structure of the application with its functionality and some of the design decision we had to take and the motivation behind them. In the next section of our thesis we are going to present the application implementation with emphasis on the testing and evaluation part. We are going to pay careful attention to the evaluation of our solution and the possibilities the API offers us to create it. The next section will point out the limits this API presents and how we tried to overcome them.

Chapter 4

Solution Implementation and Experimental Evaluation

In this section we are going to discuss in detail about the main parts of our application. We are going to present the build process of our application from the ground up and we are going to show and test the capabilities of the API by introducing smaller applications that helped us, in the end, to create our Graphical Packet Generator. The main intent of this part of our work is to analyze the available features in the onePK API that will help a programmer to automatically discover the network elements inside a topology and be able to generate traffic inside that topology.

In order to show the experimental value of our project we are going to construct a four network elements topology with the help of the virtual IOS system. We will connect to one of those network elements and will do an automatic topology discovery, to get the addresses of the network elements. Then we will generate traffic ICMP/UDP/IP RAW between the network element we connected to and another element in the topology.

We decided to break down our application in use cases, so that at any time in the development of the application the programmer has a functional product. In the light of this fact the use cases we are going to discuss are the introducing and construction of the topology, making a connection to a network element, options for automatically discovering the topology and, we will conclude, with the packet generation inside the topology.

4.1 Introducing the Topology

A network topology can be defined in the virtual environment using an XML-based programming language. The topology will represent the base on top of which our future applications will run. Because we are using an XML-based language the topology correctness can be verified using a XSD schema before the actual start of the network. The use of such a language makes the definition of the topology standard across Cisco platforms.

Before we are going to discuss the actual creation of a new topology, we have to introduce some concepts. A network element is the one that hosts a onePK service provider, it could be a real life router/firewall equipment or a virtual one. Since we are working with virtual devices, for us, it will be a virtual router. The network elements are Network Device Emulators or NDEs, meaning they are standalone virtual machines that run on top of a hyper-visor and simulate the real life network elements.

A network topology is the one that describes the arrangement of the network elements or nodes and how they will communicate with each other using the links. The configuration file for the network topology is required by the virtual machine cloud engine in order to be able to start the virtual network. The network topology file is composed of two main type of elements: network elements and connection elements.

The first section of the network topology definition file defines NDEs, internal Local Area Networks (LANs) and external Network Interface Controllers (NICs). The NICs help us connect to external routers or switches. Nodes inside a network can be of four types: simple, detailed, asset and segment. The simple type nodes are non-distributed routers. The detailed type are distributed routers, and at the moment they are not supported by the virtual machine that the API is running on. The asset nodes make possible the connection between a virtual network/node and a physical router/switch. The virtual interface is bridged to the real interface. And the segment type is used when two or more interfaces need to be connected to each other in a LAN.

A node element has the following attributes: name, type and vmImage attribute. The name is the name of the router element, the type could be any of the four types presented previously - except those that are not supported by the virtual machine, and the vmImage attribute is a mandatory one and points to the virtual image of the NDE element.

Besides those attributes a node element has an interface child, that is optional - a node can have no connection to other nodes in the topology. The interface child has a name attribute that helps identifying the node the network. Following, in Listing 4.1, is an example of a simple node configuration from our topology.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<topology xmlns="http://www.cisco.com/VIRL" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" schemaVersion="0.3" xsi:schemaLocation="http://www.cisco.com/VIRL␣http://cide.cisco.com/vmmaestro/schema/virl.xsd">
    <node name="router1" type="SIMPLE" subtype="vios" location="188,263" vmImage="/usr/share/vmcloud/data/images/vios.ova">
        <extensions><entry key="bootstrap␣configuration" type="String">/home/cisco/vmcloud-example-networks/3node/router1.con</entry><entry key="import␣files" type="String">/home/cisco/vmcloud-example-networks/3node/router1.p12</entry></extensions>
        <interface name="GigabitEthernet0/0"/>
        <interface name="GigabitEthernet0/1"/>
    </node>
</topology>
```

Listing 4.1: A representation of the network element in the topology file

The second part of a topology configuration file is composed of topology connection pairs. A pair is represented by a source and destination router, between which a link is established. The source and destination routers are presented with their index in the configuration file and the interfaces on which the connection is established is presented with a number too. The numbers start at 1.

In Figure 4.1 is a representation of the topology we are using for our application. The node 6 and node 4 are representing a local area network, and an exterior local area network. Node 5 represents the bridge node – not present in the figure, that establishes a connection with the virtual machine on which the IOS images are running.

As it can be seen we created a four routers topology, where router 1 has connection to all the other routers in the topology. Some of the restrictions we had to keep in mind while creating this topology are: the router to which we are going to connect our application to has to have connections to all the other routers in the network - we will talk more about this in the next section and the fact that there are no other virtual operating systems for the routers except the ISR G2 router - this is not enough to get the real feeling of the capabilities of the API.

Another important drawback for this section was the general process of constructing a new topology. We would have preferred to be able to construct the topology in a more dynamic way, a drag and drop interface where to create the network elements and the connections between them and in the background the application would add the XML elements. We think this would have been a good solution especially because we are inside a virtual environment and the construction and overcoming the limits of the network design stage should not be of great concern to us. Alas, even this stage presented restrictions.

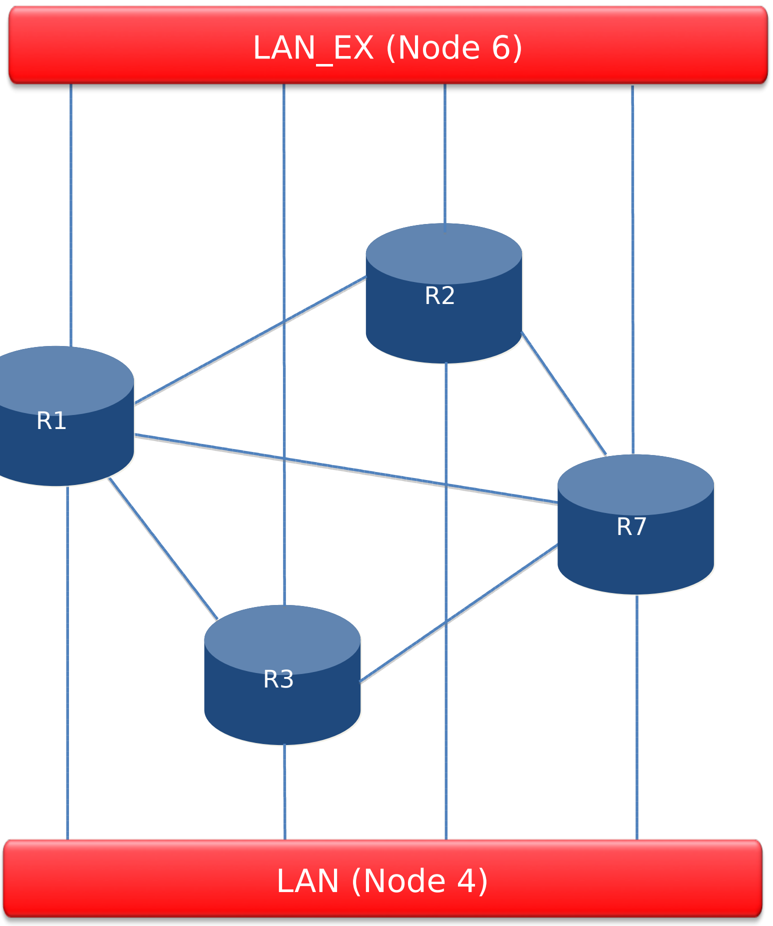


Figure 4.1: The network topology representation

4.2 Connecting to a Network Element

In this section we are going to present the way our application connects to a network element. After we have the network topology up and running we use one of the elements to connect our application to it. In order to be able to connect to the topology we should provide the application with network element credentials and the address of the element. The credentials are represented by an user name and a password used to connect to the element and also a TLS certificate.

The use of the TLS certificate is marked as optional in all of the virtual machines versions but it is actually mandatory. We consider this a drawback for the purpose of our application, because it adds overheat to the application, an unjustified overheat when we are running the application on the same machine where the network resides.

The TLS option has to be set up on the router element we are connecting to, enabling the onePK TLS communication, and also inside our application. We are going to establish a one direction secure communication, with the authentication enabled only on the device, not in the application too.

To be able to connect to the network element we will use the element hostname or IP address. We obtain a `NetworkApplication` instance and set the name of this application obtaining at the same time a `NetworkElement` inside the application from the IP address. After transforming the string provided for the IP address to an INET address, the application tries to connect using the user name and password provided and authenticates using the trust-store certificate and password. After successfully connecting to the network element we obtained a session handle and a virtual network element inside the application, which we can use to access the real router. The flow can we visualized in Figure 4.2 [7].

The established session to the network element will be used in the next parts of the application - for the automatic topology discovery and the traffic generator subprograms. We do not need to establish new connections for every subprogram, the session once opened will stay that way until we stop the running of the program or we use the disconnect command.

The restrictions we faced in this part of the application were mainly security ones. The API claims to be using the TLS options for an increase in security, but the session of the connection between an application and the network element stays open indefinitely. We think it would have been a good feature for the application to disconnect after some preset time of inactivity. Also, the pure socket connection is an option in theory for connecting to the network element, but it is not working nor it is recommended to be used. For virtual applications or applications that reside on the same blade as the network element, the pure socket option could be a viable one - it would save time at connection and will add a smaller overhead.

4.3 Automatically Discovering the Topology

One important aspect of our application is the automatic discovery of the topology of the network we are using. It is important for a few reasons: for one it proves the utility of such a program, if you are able to discover the network topology from a central application you are able to keep the state of the network consistent on some levels, and it gives us the possibility to test the API's capabilities with a complete application. We do the discovery of the topology and we use the results from the discovery to the second part of the application, where we generate traffic between two routers.

In our application we are using one of the Service Sets that is contained in the onePK API to do the topology discovery. We are using the Discovery Service Set that enables Service Discovery and Topology Discovery. The later provides a way to discover the topology as seen by a protocol

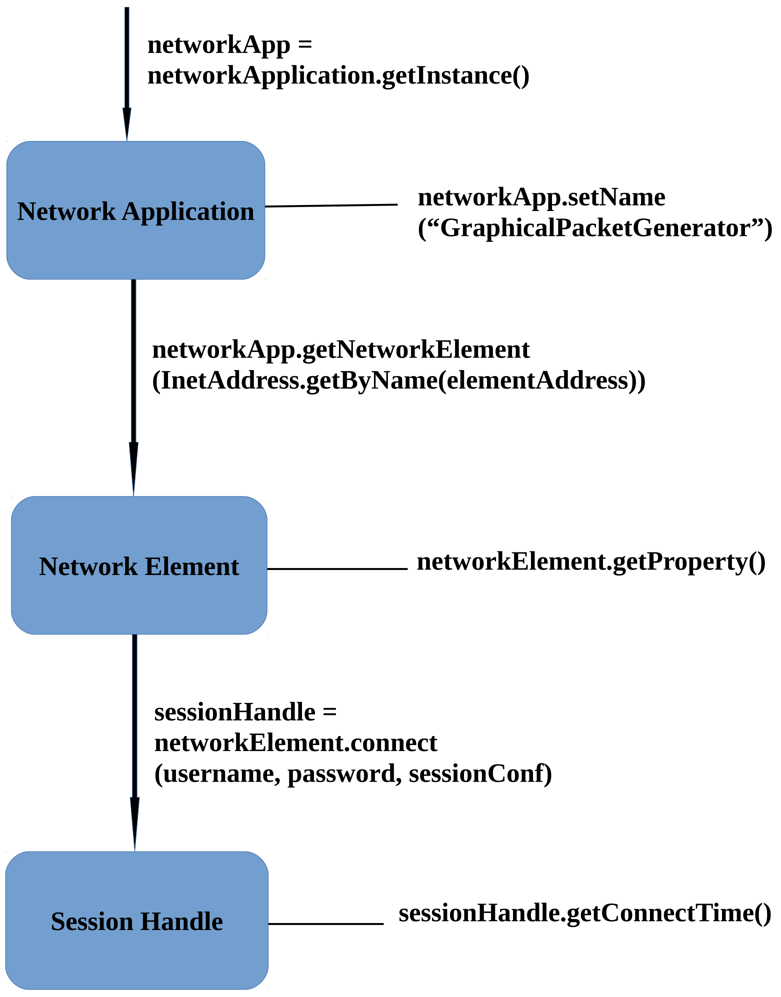

networkApp =
networkApplication.getInstance()
Network Application
networkApp.setName
("GraphicalPacketGenerator")
networkApp.getNetworkElement
(InetAddress.getByName(elementAddress))
Network Element
networkElement.getProperty()
sessionHandle =
networkElement.connect
(username, password, sessionConf)
Session Handle
sessionHandle.getConnectTime()

Figure 4.2: Connecting to a network element

and lets the application to obtain the topology as a graph. Before stopping on this method, we tried another solution to discover the topology - using a system monitor method, where we enabled a callback filtering function on the whole network and waited for up and down signals to be transmitted to the interfaces. Both of those methods are using the Cisco Discovery Protocol. We chose the first method because it creates also a graph for the topology, even though both of this are based on methods that filter the interfaces. But we will detail both these methods further in this section.

The `Topology Discovery Service Set` gives the developer a mechanism to discover the topology as seen by a protocol. The service set is a set of APIs that include objects to define the topology graph and functions that can be registered as callback to listen to topology change events. There is a series of data structures that are involved in the topology discovery process: edge, topology events, topology filters, graphs, node connectors and actual topology structures.

An edge is the path as seen by the protocol from one interface (head) to the another one (tail). The topology events represent the events that take place inside the topology every time a change occurs. The topology filters are used to filter the events that occur in the topology, and the graph represents the graph of the network as seen by the protocol from head node. A node connector is the identity of a network interface, its name or IP address. All of this elements are used in our actual solution for the topology discovery problem.

Both of our solutions use the Cisco Discovery Protocol (CDP) in their implementation. The network elements send out CDP advertisements to a multicast address from each connected interface. The advertisements are received by other CDP speaking equipments that refresh their internal table, where the CDP information is stored. After a specified time, if no more advertisements are received from one of the neighbours the information that came from that neighbour expires and it is deleted from the current node's table.

One drawback of the CDP is that it can discover only neighbours that are one hop distance away. This means that in order to be able to discover neighbours that can be found two or more hops away the protocol must run recursively. This is a situation less than ideal, considering that in order to be able to run the protocol from the onePK API one must do something similar with the actions in Figure 4.3 [7]:

- create an object that represents the topology;
- initialize the previous object for node A;
- connect to node A and obtain the neighbours;
- do the same thing recursively for the children of node A;
- connect to the leaf nodes and obtain their children - stop if no more children.

This kind of approach is time consuming and it does not scale for big networks. The onePK API promised support for other protocols too EIGRP, OSPF with the possibility of getting the graph of the network as seen by those protocols. Unfortunately support for those protocols is not implemented and the only way to do the topology discovery is using this level two protocol, CDP. We found this to be one of the major restrictions of the onePK library and this is one of the reasons why in our test topology one of the nodes (router 1) is connected to all the others - to minimize the time for topology discovery.

The first method we tried as a solution for the topology discovery problem was a system monitor method. After connecting and authenticating on a router from our network the program was processing all of the addresses of the interfaces and sub-interfaces of the network element. We did that in order to avoid the processing loops when the same element will be discovered via its neighbours. Every router was sending out CDPEvents turning its on interfaces up and down, and for every CDPEvent received from a router, the neighbouring router was processed similarly. In the end, every network element starting with the initial router on which we logged on and continuing with all the other ones were discovered recursively and had a system monitor instantiated, monitor that reacted at every interface change.

We were monitoring events that happened on all the interfaces, events with a priority between 0 and 8. This means that we were monitoring all the events on all the interfaces. In order to be able to monitor those interfaces for events, we used a filter that matched the criteria mentioned previously.

Even though this method gave us the expected result - a list of IP addresses for the direct neighbours of our connecting router, we tried a second method for topology discovery. The main reason behind it was the testing of the other components of the Topology Discovery Service Set, namely the graph, edge and node connector components. The topology in this situation is represented by a list of edges that indicate the connectivity between two nodes. The nodes are connected by a node connector and the topology object returns the graph as seen by a protocol, in this situation CDP - shows only the one distance away neighbours.

We chose to obtain just the one node topology and not to connect to each child node and obtain an incremental one. In this situation a new topology object would have been instantiated and

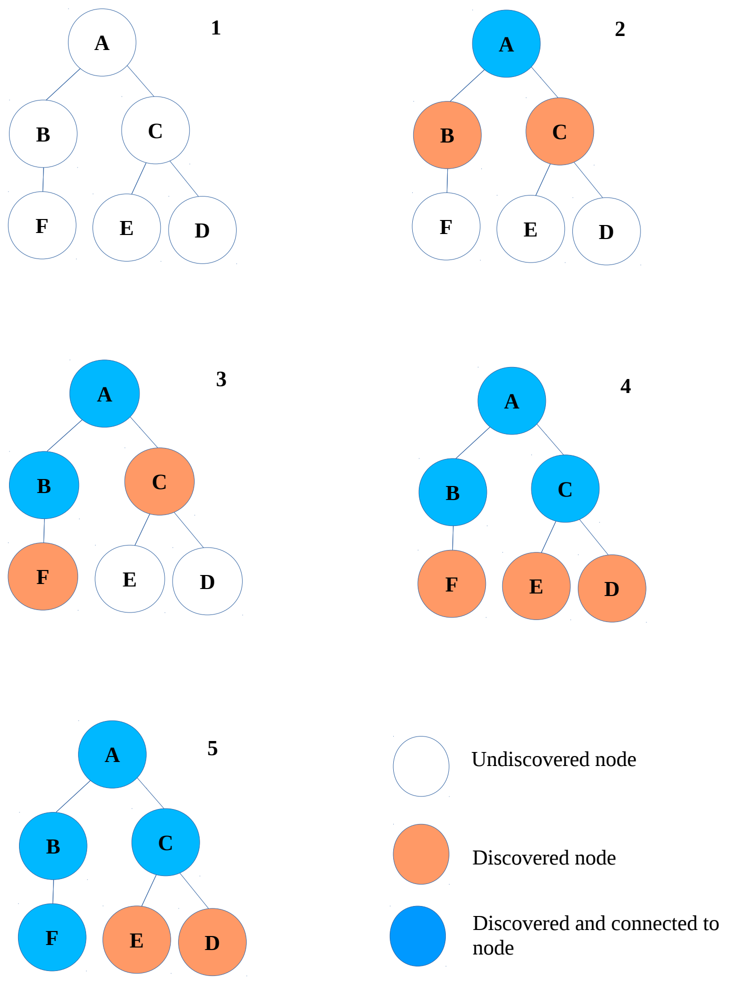


Figure 4.3: A Breadth First Approach to CDP Topology Discovery

associated with the discovered node.

We registered a topology event for this approach too. The topology event notifies the program if edges have been added or deleted from the network. It looks out for new connections coming up or for old connections going down. The way this method works can be seen in the Figure 4.4 [7].

As can one observe in the figure we initially log into a router element and obtain a session handle to communicate with that element. We initialize a topology object, that offers us the information from the topology updated. This means that after every event that takes place inside the topology this object will be automatically updated to reflect it. Using the topology object we get a snapshot of the topology graph as a graph object. The graph object is an updated version of the topology made out of edge objects.

An edge object offers us information about the path discovered between one router element and another. The edge objects are represented by connector objects, one for each node that it connects. The connector objects represents the interfaces of the connected nodes. The node connectors are actually pointing to the discovered nodes inside a topology. This means that if we discover a node connector we discovered a node.

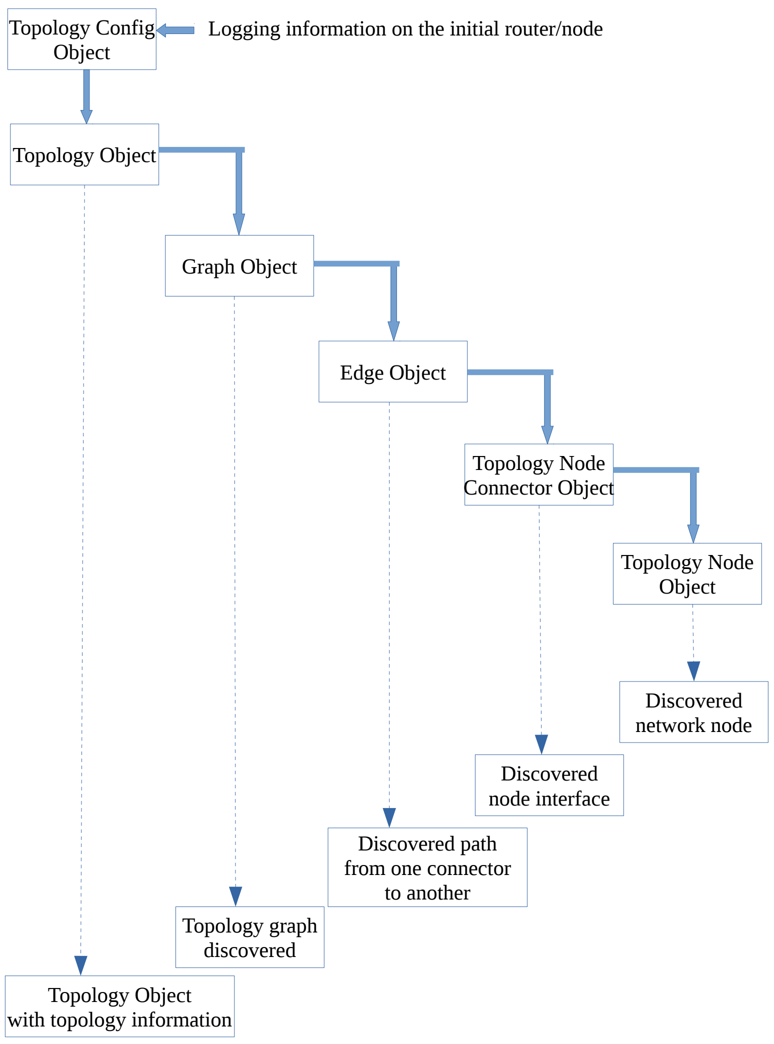

Topology Config Object
Logging information on the initial router/node
Topology Object
Graph Object
Edge Object
Topology Node Connector Object
Topology Node Object
Discovered network node
Discovered node interface
Discovered path from one connector to another
Topology graph discovered
Topology Object with topology information

Figure 4.4: Internal organization of the discovery subprogram and how the topology is obtained

The difference between the two approaches is that in the first situation we are not getting as a result the graph of the topology, we are only receiving the IP addresses of the neighbouring routers. Using the second method we are receiving a complete topology graph from the perspective of the router we connected to. Also, the discovery of the neighbouring devices is done subtly different. In the first case we generate up and down events for the outgoing interfaces, events that are registered by the counterpart interfaces from the other end. In the second case, we use CDP and let the Discovery Service Set to do its job.

Another big restriction of the API that we met in this part of our development was the impossibility of getting the MAC address of an interface without actually creating a session handle and connecting to that element. We would need the MAC addresses for the next part of the application, the packet generation subprogram, and connecting to each network element each time we want to generate a packet is not a feasible thing. At the moment there is no possibility for us to get that MAC address other than having access to the actual router.

4.4 Packet Generation Inside the Topology (ICMP/UDP/IP RAW)

The packet generation part of our program is the most important one. It shows that the application comes full cycle and provides the user with an useful feedback. What we wanted initially to implement in this section was a program that receives source and destination IP addresses and generates between the two packets of different types. Theoretically, the API supports generation of ICMP, UDP and IP Raw packets, later we will see that generation for these kind of packets is not fully supported though.

To be able to generate packets using the API we had to use the `Data Path Service Set` (DPSS) provided by it. One of the biggest restrictions of the onePK API is that the DPSS functions are available only in the C implementation, even though there are Java and Python implementations as well. So to be able to call the native C functions inside our Java program we used Java Native Interface (JNI), that let us run our code inside the Java Virtual Machine and call at the same time the C functions. We implemented the packet generation in C, created a system library with those functions and loaded the said library into our Java program.

DPSS gives us the possibility to hook into the packet flow and extract or modify packets from that flow. The service permits two main operations on the packet flow: copy the packets and divert or punt them. When copying a packet flow, each packet is duplicated to the controller. So the controller and the destination of that packet flow receive each one packet. The difference between copying and diverting is subtle. When we divert it, we send it to the application and do not let it continue on its initial way until is returned from that level. At the application level the packet can be modified or unmodified. When we use copy on a packet, the application can not modify the packet, it can only examine it.

Knowing we can use DPSS to generate packets, we came up with a few ways we could do it:

- using a next hop action with an explicit next hop IP address;
- using a divert action, calling the `onep_dpss_inject_raw_packet` function with the original packet and location set at PREROUTING - this would rebuild the packet's level two header;
- using a divert action, calling the `onep_dpss_inject_raw_packet` function with the original packet and location set to OUTPUT, but we will modify the level two header before injection.

After analyzing the solutions we came up with, we realized that the first option was not supported by the onePK API, is not listed in the documentation as something that can be done with the API. The second option is not what we are looking for. With this option packets will be routed based on the entries from the routing table. So what we are left in the end is the third option, but the drawback is that we have to build the level two headers ourselves, before the injection of the packets.

Besides that, we encountered another problem. There is not a viable way to find out the MAC address of the next hop from the application, in order to be able to generate the level two header ourself. We could use another service set, the Virtual Terminal Service Set (VTY SS) to be able to run "show ip arp" and grab the MAC address of the next hop, but we would have to run separately a ping first to that destination.

This is counterintuitive to our purpose, we want to grab the destination MAC address in order to generate ICMP packets from the application, but we have to run first ICMP separately. In light of those restrictions, we decided to complete the field for the destination MAC address manually, for now.

Before we can present the DPSS application integrated in our initial Java program, we have to present the setting up we needed to do in order to be able to run the service set. To be able to run a DPSS program we must have the dpss_mp main process running. The process has a configuration file where we specify the IP address for the application gateway (LOCAL_IP), the user name and the group this user resides in on the machine we are running the process, the transport type between the application controller - the DPSS process - and the network element we are connecting to, the only supported option is raw - GRE (Generic Routing Encapsulation [15] tunnel). In this same section we set up a sender id, that must be the same with the sender id we are going to set when we will enable the DPSS option on the router we are connecting to. The configuration file looks like the listing in Listing 4.2.

```
LOCAL_IP 10.10.10.1
LOCAL_PORT 9999
MAX_CLIENTS 15
GROUP_NAME cisco
USER_NAME cisco
TRANSTYPE gre
ONEP_SENDER_ID 10
```

Listing 4.2: The configuration file for the dpss_mp main process

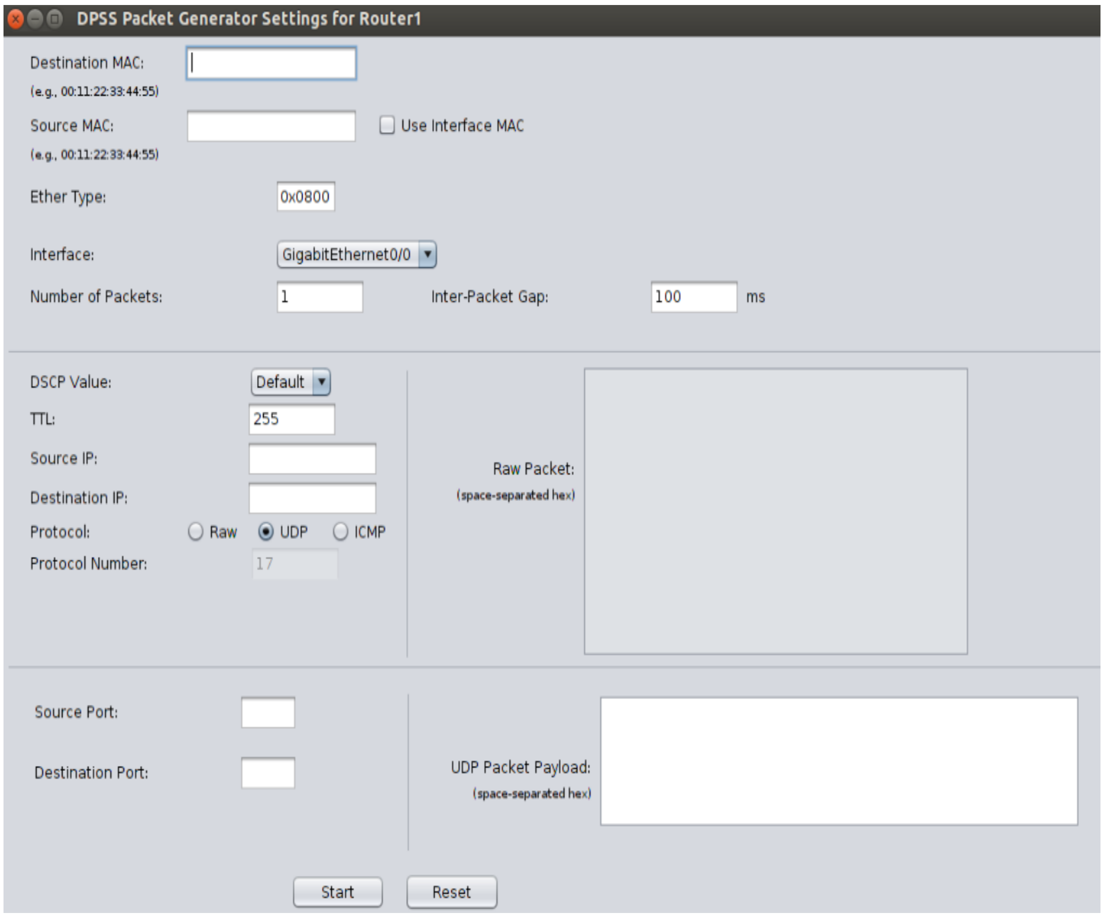
Now that we presented the setting up for the use of the data path service set we will show the Java interface, that can be seen in Figure 4.5. The connection between the graphical interface and the driver native C library is done using JNI. The native library is loaded using JNI and the function calls are used inside the Java application. Using the graphical interface the user can set up the construction of the new level two header, those parameters would be transmitted to the JNI function call - that will enable the native C function. The steps we took in creating a C DPSS application are:

- application initialization;
- DPSS initialization;
- register packet callback handlers;
- run a DPSS packet loop.

We are going to detail those steps and what it can and cannot be done with the software. The application initialization was done by passing the credentials from the Java application into the native C connect function call. That way we obtained another session handle and another connection to the network element. To initialize the DPSS application, we had to create our own level two packet header. For that we needed a source and destination MAC address, as well as a source and destination IP address. The MAC address we had to insert by hand in the graphical interface, but the source and destination IP were provided by the automatically topology discovery option. The IP addresses are the addresses of the neighbouring routers.

Inside the interface we can modify other parameters too, those will be set up and verified inside the DPSS library in our program. We can use the DSCP value to change the priority of the traffic inside the network. Modifying the DSCP value we modify the Type of Service (ToS) byte, that encompasses the DSCP value (the DSCP value is only 6 bits long, while ToS is 8 bits long). After modifying the value and starting generating packets we use Wireshark to sort the traffic by the DSCP value and see how it travels through the network.

We used the callback functions for getting a feedback from the destination router of an ICMP packet. The problem with the API is that ICMP packets are not fully supported, so even though we register a callback function on the router we connected to the network, the echo replies the callback function will see will be incomplete. Meaning, that we will not be able to tell from which sequence number did we receive an echo reply.


DPSS Packet Generator Settings for Router1

Destination MAC:
(e.g., 00:11:22:33:44:55)
Source MAC: Use Interface MAC
(e.g., 00:11:22:33:44:55)
Ether Type: 0x0800
Interface: GigabitEthernet0/0
Number of Packets: 1 Inter-Packet Gap: 100 ms
DSCP Value: Default
TTL: 255
Source IP:
Destination IP:
Protocol: Raw [✓ UDP] ICMP
Protocol Number: 17
Raw Packet: (space-separated hex)
Source Port:
Destination Port:
UDP Packet Payload: (space-separated hex)

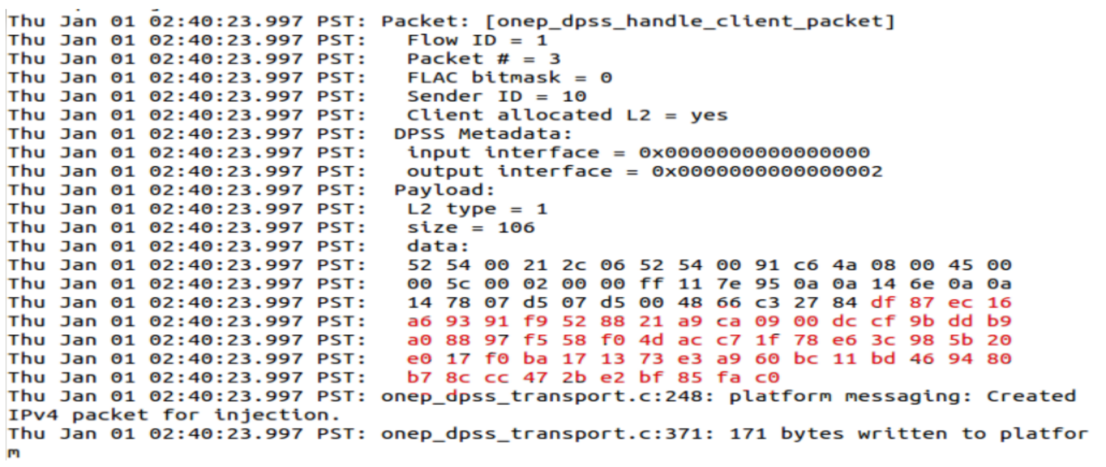
Figure 4.5: The user interface for the Graphical Packet Generator

For all types of packets we can provide a packet payload - bytes that will get transported across the network. A problem persists with the packet payload for the ICMP packets, the packets get transmitted across the network but with a null payload - another limit of the API. To visualize the packet payload we used Wireshark again. At the moment, the API does not offer any method to get a feedback about UDP or IP Raw packets, we cannot know if the packets were received by the destination or not. The only way to gain such a feedback is by creating another connection to the destination router and there registering callback functions.

We tested the application using the onep_dpss_inject_ip_raw function, injecting different types of packets ICMP/UDP with different dimensions for payload. The output of one of the transmitted packets can be seen in Figure 4.6 - where we got a caption from the dpss_mp main process with the debug option turned on. To obtain this output we ran the program, generating 3 UDP packets, with router one as source and router two as destination, and a payload of 64 bytes.

In Figure 4.6 we can see that we have created an IPv4 packet, that is ready for injection, and the bytes have been written to the platform. The payload of the packet is highlighted, in the caption. Also we get other information too, like the sender ID and the sequence number of the packet in this generation sequence (Packet # in the caption).

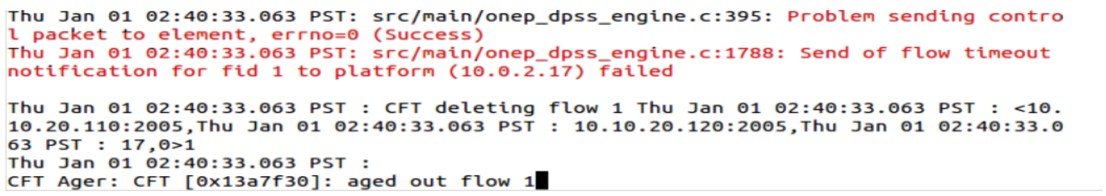
As it can be seen in the Figure 4.7 (the same packet has been transmitted but now we get a feedback about what happened with it, in the dpss_mp process) there was a problem with our packets. The information about the problem is not detailed, but if we would have logged on the



```
Thu Jan 01 02:40:23.997 PST: Packet: [onep_dpss_handle_client_packet]
Thu Jan 01 02:40:23.997 PST:   Flow ID = 1
Thu Jan 01 02:40:23.997 PST:   Packet # = 3
Thu Jan 01 02:40:23.997 PST:   FLAC bitmask = 0
Thu Jan 01 02:40:23.997 PST:   Sender ID = 10
Thu Jan 01 02:40:23.997 PST:   Client allocated L2 = yes
Thu Jan 01 02:40:23.997 PST:  DPSS Metadata:
Thu Jan 01 02:40:23.997 PST:   input interface = 0x0000000000000000
Thu Jan 01 02:40:23.997 PST:   output interface = 0x0000000000000002
Thu Jan 01 02:40:23.997 PST:  Payload:
Thu Jan 01 02:40:23.997 PST:   L2 type = 1
Thu Jan 01 02:40:23.997 PST:   size = 106
Thu Jan 01 02:40:23.997 PST:   data:
Thu Jan 01 02:40:23.997 PST:   52 54 00 21 2c 06 52 54 00 91 c6 4a 08 00 45 00
Thu Jan 01 02:40:23.997 PST:   00 5c 00 02 00 00 ff 11 7e 95 0a 0a 14 6e 0a 0a
Thu Jan 01 02:40:23.997 PST:   14 78 07 d5 07 d5 00 48 66 c3 27 84 df 87 ec 16
Thu Jan 01 02:40:23.997 PST:   a6 93 91 f9 52 88 21 a9 ca 09 00 dc cf 9b dd b9
Thu Jan 01 02:40:23.997 PST:   a0 88 97 f5 58 f0 4d ac c7 1f 78 e6 3c 98 5b 20
Thu Jan 01 02:40:23.997 PST:   e0 17 f0 ba 17 13 73 e3 a9 60 bc 11 bd 46 94 80
Thu Jan 01 02:40:23.997 PST:   b7 8c cc 47 2b e2 bf 85 fa c0
Thu Jan 01 02:40:23.997 PST: onep_dpss_transport.c:248: platform messaging: Created
IPv4 packet for injection.
Thu Jan 01 02:40:23.997 PST: onep_dpss_transport.c:371: 171 bytes written to platfor
m
```

Figure 4.6: dpss_mp main process caption for an UDP flow

destination router and turned the debug for IP packets on, we would have received a warning about the destination port being unreachable. Unfortunately this kind of reporting can not be done in the application environment with this version of the onePK API, the user not being able to know if their packets reached the destination or not.



```
Thu Jan 01 02:40:33.063 PST: src/main/onep_dpss_engine.c:395: Problem sending contro
l packet to element, errno=0 (Success)
Thu Jan 01 02:40:33.063 PST: src/main/onep_dpss_engine.c:1788: Send of flow timeout
notification for fid 1 to platform (10.0.2.17) failed

Thu Jan 01 02:40:33.063 PST : CFT deleting flow 1 Thu Jan 01 02:40:33.063 PST : <10.
10.20.110:2005,Thu Jan 01 02:40:33.063 PST : 10.10.20.120:2005,Thu Jan 01 02:40:33.0
63 PST : 17,0>1
Thu Jan 01 02:40:33.063 PST :
CFT Ager: CFT [0x13a7f30]: aged out flow 1
```

Figure 4.7: dpss_mp main process, error for the previous UDP flow delivery

We could try and measure the speed of transmission or the rate of successful transmissions for this application, but since we are using a virtual machine, with simulated routers the results of the experiment will not give us any insight into the capabilities of the program. We did have a testing scenario where we were generating an increasingly higher number of packets, but the application was successful during the test – the packets were registered as transmitted in Wireshark. There is a limit – concerning the well functioning of the router – around 45k packets per second, but inside a virtual machine we were not able to reach it.

Chapter 5

The Limits of the onePK API

During the development of our application we encountered some restrictions that the onePK API presents. In this section we are going to recount those limits and why adding the functionality that will solve them is important.

The first issue encountered was the fact that the virtual machine we were developing on and that has the latest version of the API is capable to simulate only one type of router, the ISR G2 router. Even though the router can support all the service sets that the API enlists, it would have been nice to have at least one other type of router supported by the virtual machine. It would have been a good exercise to see how different devices that run the onePK API are communicating between them. On the same note, there is no way to check what is the onePK API version that is running on the equipment. This kind of functionality is critical in networks where will be running multiple different versions of the API.

Another fact that we wished would have been different is the connection to a network element part. Connecting to the network element claims to be a secured operation. The API can support on the paper TLS and socket connection, in reality it only supports TLS connection. This type of connection is probably what we need if we are deploying our application on a different blade than the one where the network element we are going to connect to resides. But, for programs where the application is on the same machine/blade as the network it is going to connect to, this kind of security is an overhead. If an eventual attacker gets access to your machine and can tamper with your software then the TLS connection between the said software and the network element will not help. A simple socket connection would have been preferred for our application, taking in consideration that we were running on a virtual machine and the purpose of our application was to show the capabilities of the API regarding the automatic topology discovery and packet generation.

Besides the API's limits presented so far, there are others that affect the topology discovery segment of the application as well as the packet generation one. When it came to topology discovery the onePK API claimed the topology discovery can be done automatically and the user will see the topology as seen by one of the routing protocols. Unfortunately the only routing protocol that is supported at the moment is Cisco Discovery Protocol, a level two protocol. There are no level three protocols that can be used to discover the topology automatically and the biggest drawback in using CDP is the fact that it only sees the network as fas as one hop away. So, in order to be able to fully map a network that has the diameter bigger than two, we need to call the CDP topology discovery algorithm in a Breadth First Search fashion. This will take up more time and when new edges will appear or disappear from the graph the update will propagate more slowly inside the network than it will propagate if it would be using OSPF for example.

There would have been another solution for the topology discovery - using SNMP protocol. But SNMP discovery would be analogous to doing this through onePK. With SNMP, we would poll the CISCO-CDP-MIB (for example) to get the neighbors. With onePK, we can just use the API to do the same thing.

The automatic topology discovery has another limit, we can not return the MAC address of the neighbouring interfaces. The only way to discover the neighbouring interface address using onePK is by creating a new session connection to this element and then getting the information for all the interfaces of this element.

In the packet generator section of our program we were faced with limits of the API as well as with incomplete functionalities. The service set we had to use for packet generation is available just in the C programming language version of the API, and our application is a Java application. So we had to use Java Native Interface to be able to call native C function inside the Java Virtual Machine.

We are able to generate packets inside the network and have a confirmation that we generated those packets, but we can not get a clear confirmation/feedback from the destination of those packets. For example we can generate ICMP packets from a source router and the user can have a visual feedback that the packets have been generated by the native C library inside the graphical user interface. But we can not have a real confirmation that the packets have been received by the destination. We can register a callback function on the node we connected with the application to the network and wait for echo-reply packets, but we will not know to which sequence number do those packets belong to. For confirmation of reception for UDP packets we would have to create another session and connect to the destination router and register a callback function there, that will count all the UDP packets received.

Another incomplete functionality for the Datapath Service Set is the fact that ICMP packets appear with payload null if they are traced by a networking sniffing program, like Wireshark. The packets are delivered to the target router but their payload is null.

In the light of what we presented so far, we consider that all those restrictions make the use of the API not appropriately yet inside a real life network. Not if we hope to do everything from a central controller, without having to log in on physical devices and get some of the information there.

Chapter 6

Conclusion

In this thesis, we have presented another approach on developing an application that is consistent with Software Defined Networking principles. Our intention was to show that even though there are a lot of limits and restrictions imposed by the API we used in the development of the application the user can still obtain some information about the state of the network.

We chose to show as complete a perspective as possible on the use of the API, creating a program that can let the user to automatically discover the network topology. After that, the information obtain in this stage can be used for generating traffic between the available routers. That way the scope of the application is a broader one, and tries to show the implementation of a generic use case (connecting to a network, which topology you can automatically discover and can then generate traffic inside the said network) with SDN methods.

In our development of the use cases and testing of the API we have encountered a series of restrictions that onePK imposed on us. There were restrictions imposed for the automatic topology discovery, we could not use any other protocol than CDP - only level two discovery. Also, we could only return, from the topology discovery subprogram, the IP addresses, not the MAC addresses. There is no viable way to find the MAC address of an interface - other than creating a new session and connecting on each device to get its MAC addresses. This is a major drawback, since we need the MAC addresses in our packet generation program.

As a consequence of not being able to get the MAC address from the automatic topology discovery, the packet generation part of our program suffered. We had to complete the level two header of our packets with MAC addresses taken by hand from the destination interface. Besides that, the user can not get a real feedback inside the graphical user interface about the packets sent. For example if he sent 10 ICMP packets he might be able to get back 9 or 10 echo-reply packets, but he will not know which one of the packets got lost or to which sequence number do those packets belong to.

Taking in consideration those bounds the API imposed on us we consider that our project was able to show what the development of such a program means. We did not aim to offer you a universal way of doing networking. We wanted to present an alternative one, that even though it has a long way to come - as we show by all those restrictions the API imposes - it might become someday something we will consider to deploy in real life complex networks.

As a final thought, we believe that in order to be able to evolve in the networking domain abstractions must be created. Those abstractions can help us create applications, that will help change the way networking is seen by the outsiders - as a heavy, sometimes boring task, that most people can not even get their heads around. These abstractions are represented by the use of an API that facilitates the creation of applications ready to control the data flow and hopefully proving James Hamilton from Amazon wrong when he said “The network is in my way”. But there is still a long road to ahead of us, before we can do that.

Bibliography

[1] 5th Edition Andrew Tannenbaum. Computer networks, September 2010.

[2] S. Bhattacharjee. An architecture for active networks. in proceedings of high-performance networking, 1997.

[3] M. Caesar. Design and implementation of a routing control platform. in proceedings of the 2nd usenix symposium on networked systems design and implementation (nsdi), 2005.

[4] Martin Casado. Ethane: Taking control of the enterprise. ACM SIGCOMM Computer Communication Review Volume 37 Issue 4, October 2007.

[5] Cisco. Cisco open network environment: Bring the network closer to applications, white paper.

[6] Cisco. Lldp-med and cisco discovery protocol, white paper, 2006.

[7] Cisco. Cisco onepk java api reference. https://developer.cisco.com/media/onePKJavaAPI-v1-1-0/index.html, Last visited on the 1st of July 2014, July 2014.

[8] Cisco. Cisco onepk java api reference. https://developer.cisco.com/media/onePKJavaAPI-v1-1-0/index.html, Last visited on the 1st of July 2014, July 2014.

[9] Cisco. Emulator user guide. https://developer.cisco.com/media/onepk_getting_started_guide/GUID-0A9F0D33-1027-442C-A516-D21F9B0FA3BF.html, Last visited on the 1st of July 2014, July 2014.

[10] Cisco. Qos - packet marking. http://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-packet-marking/10103-dscpvalues.html, Last visited on the 1st of July 2014, July 2014.

[11] Nick Feamster. The road to sdn, December 2013.

[12] International Organization for Standardization. Rfc 768, user datagram protocol, internet standard. http://tools.ietf.org/html/rfc768, August 1980.

[13] International Organization for Standardization. Rfc 791, internet protocol darpa internet program protocol specification, September 1981.

[14] International Organization for Standardization. Rfc 792, internet control message protocol, darpa internet program protocol specification. http://tools.ietf.org/html/rfc792, September 1981.

[15] International Organization for Standardization. Rfc1701, generic routing encapsulation, gre, October 1994.

[16] Gigacom. Openflow: a technology on the move. https://gigaom.com/2011/07/24/openflow-a-technology-on-the-move/, Last visited on the 1st of July 2014.

[17] Evangelos Haleplidis. Software-defined networking: Experimenting with the control to forwarding plane interface, October 2012.

[18] Sheng Liang. The java native interface: Programmer's guide and specification, June 1999.

[19] Marc Loy. Java swing (2 ed.), 2012.

[20] Nick McKeown. Openflow: Enabling innovation in campus networks, March 2008.

[21] Nick McKeown. How sdn will shape networking. Open Networking Summit, October 2011.

[22] Nick McKeown. Protocol independence, October 2013.

[23] Oracle. Jsse reference guide. http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERef-Guide.html, Last visited on the 1st of July 2014, July 2014.

[24] Matthew Palmer. vswitch the new battleground for network virtualization. http://www.sdncentral.com/technology/vswitch-the-new-battleground-what-every-datacenter-operator-must-know/2012/07/, March 2012.

[25] Lucie Smith. Free pool of ipv4 address space depleted, February 2011.

[26] The Internet Society. Rfc 2460, internet protocol, version 6 (ipv6) specification, 1998.

[27] E. Rescorla T. Dierks. The transport layer security (tls) protocol, version 1.2, August 2008.

[28] D. Wetherall. Ants: a toolkit for building and dynamically deploying network protocols. in proceedings of ieee openarch, December 1998.

[29] L. Yang. Rfc 3746, forwarding and control element separation (forces) framework. internet engineering task force. https://www.rfc-editor.org/rfc/rfc3746.txt, 2004.