Master Thesis

TITLE: Distributed Traffic Matrix Measurement in OpenFlow Enabled Networks

MASTER DEGREE: Master in Telematics Engineering

AUTHOR: Arman Keyoumarsi

DIRECTOR: David Rincón Rivera

DATE: July 23, 2014

Departament d'Enginyeria Telemàtica

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ABSTRACT

The current network architecture is constrained. The development of new protocols to cope with new demands in computer networks such as real time streaming and voice over IP is not an easy task. Other than developing new protocols, computer networks should be able to rapidly deploy changes across networking equipment. Software-Defined Networking promises to simplify network management tasks and enable rapid developments of new protocols. SDN separates the control plane from the data planes in the switches, bringing a level of abstraction needed to manage sets of switches from a central location. OpenFlow has become a de facto standard for communication between control plane and data planes.

One key information for network planning and management is the traffic matrix (TM). TM is defined as the volume of traffic transported by the network during a period of time from any ingress node to any egress node in a network. OpenFlow-based networks in particular must be able to continuously monitor performance metrics, in order to quickly adapt forwarding rules to utilize network resources in the best possible manner. However, current solutions for monitoring either require special monitoring equipment or impose significant measurement overhead.

In this thesis, we propose a distributed way of measuring traffic, where every ingress node in the network is responsible for measuring a portion of the traffic matrix. We only used a set of features already available in the latest version of OpenFlow without any need of modification in the platform. Therefore our framework can be deployed on any OpenFlow (version 1.3 or higher) enabled network. The effectiveness of our solution is demonstrated through series of emulation in Mininet and in a test-bed with low-end OpenFlow switches.

Table of Contents

CHAPTER 1. INTRODUCTION

One of the conundrums in today's networking is its complexity [1]. Even though computation and storage have been virtualized, leading to a more flexible and manageable infrastructure, the concept of virtual networks is still fairly new and hard to implement, because of the complexity in today's network structure. Networks used to be simple at core (Ethernet/IP protocols are straightforward and easy to manage [2]) but with the introduction of new services like voice over IP and real time streaming, the need for new control requirements, like Access Control Lists (ACLs), Virtual Local Area Networks (VLANs), Traffic Engineering (TE), and Quality of Service (QoS) led to the complexity of current networks.

Furthermore, to keep up with new demands and services, new protocols and control mechanisms need to be developed. The creation and introduction of new protocols is not an easy task, since they have to co-exist with legacy mechanisms. In addition, the process of testing and approval by the equipment vendors may take years and considerable investment.

So why are computer networks so complex, as of today? The lack of abstraction in the networking industry has kept it behind, compared to other fields. For example, looking at computer programming, networking can be compared to the phase when Assembly language was used, when there were no abstractions and the programmer had to deal with low-level details, which made the process of writing applications fairly complex and time consuming. As Donald Norman[1] said,

"The ability to master complexity is not same as the ability to extract simplicity".

Software defined networking [3] was introduced in 2008, as a concept evolving form the work done at UC Berkeley and Stanford University, by bringing abstraction in computer networking and solving some of the issues mentioned earlier. The concept of SDN focuses on the separation of Control and Data planes, bringing long overdue abstraction needed to create, develop and deploy network applications which are hardware independent and managed from a central point. The vantage point of the centralized SDN controller allows to globally optimize the network behavior with any goal the developer can imagine, such as load balancing, maximizing reliability, minimizing delay or energy consumption, to name a few possibilities. In order for such vision to become realizable, standard interfaces between the controller and the devices must be developed. OpenFlow [4] is one of the protocols that follow the SDN architecture, providing easy-to-use APIs to identify, configure and manage networks of switches from the controller.

To design, develop, manage and test a computer network, good network engineering is needed. Specifically, one key piece of information needed during the network planning and operation is the Traffic Matrix (TM), or Demand Matrix [5]. The TM provides information about the volume of

[1] Donald Arthur "Don" Norman (born December 25, 1935) is an academic in the field of cognitive science, design and usability engineering and a co-founder and consultant with the Nielsen Norman Group. He is the author of the books The Design of Everyday Things and Living with Complexity.

traffic, offered by the end users, that is sent from any ingress node to every egress node of the network during a certain period of time. Knowing how the traffic flows in the network can help designing a better and more efficient network, and is essential to validate the performance of new protocols.

Today, the traffic matrix can be obtained through various methods, either by directly measuring and collecting the packet headers and timestamps, or by collecting aggregates of flows that ingress and egress the network. There are solutions like NetFlow [6] or sFlow [7] that are able to capture the flow statistics, but deploying these solutions needs extra investments, can be fairly complex (for example, the correlation of routing and traffic information is not trivial), in some cases vendor dependent, and can cause unnecessary performance degradation in the network. Monitoring flows in thousands per second in the edge nodes can cause extra CPU load and, besides, collecting all the information needs additional burden on the network and extra equipment to do so [8].

Openflow has introduced new possibilities for obtaining the TM. For the first time a central location has a complete view of the network and can rapidly configure and deploy changes across the network. OpenFlow has gone through various versions and improvements, and new features were added with each version, being some of them related to traffic measurement, such as per-flow counters and meters. Nonetheless OpenFlow is not perfect, and there are still limitations and obstacles in deploying it, such as optimization of memory usage in the switches.

The goal of this thesis is to study how to use the new features introduced in the latest versions of Openflow to obtain TMs accurately and with the least possible overhead, both in terms of operation and deployment. We have developed a framework that evenly distributes tasks for measuring traffic across the edge switches in the network. A central controller will configure all the edge switches to measure their incoming traffic to every egress point in the network, effectively calculating a row of the TM. By doing so we have been able to reduce the processing power needed in the controller, and reduce the load on the switches. Our framework is designed using the current set of features available in OpenFlow 1.3 and above, and therefore making it possible to be deployed in any current OpenFlow enabled network without the need to modify neither the switches nor the controller. Furthermore, we have evaluated the effect of deploying the framework in a test-bed with real low-end networking equipment (OpenWRT [9] routers[2] with OpenFlow capabilities), demonstrating upmost accuracy and low cost of operation. The only concern that may arise is the extra memory needed to allocate rules for measuring the traffic in the switch, which in the case of networks with tons of distinct flows may cause in degradation of the switch performance; but to the best of our knowledge, this is a tradeoff that any TM measurement technique has to face.

[2] OpenWrt is an operating system / embedded operating system based on the Linux kernel, and primarily used on low-end, embedded devices such as small switches, routers, or other network devices.

The remainder of this document is organized as follows:

- Chapter 2 introduces Software Define networking, its history, and its fundamental differences with the current network architecture. The OpenFlow protocol is also described with some detail.

- Chapter 3 defines the concept of traffic matrix, its relevance, the current tools available to obtain TM, as well as the limitations and shortcomings of these monitoring tools.

- Chapter 4 presents the main concept behind the thesis: how to obtain TMs in OpenFlow enabled networks. The related work done in the field by other authors is presented, compared with our approach, and discussed.

- Chapter 5 presents the result of the testbed-based evaluation, where the framework was run on real networking equipment to test and quantify the performance of our approach.

- Chapter 6 presents the conclusion and future lines of development. We will discuss the achieved goals, and discuss the areas where our framework can improve.

CHAPTER 2. SOFTWARE-DEFINED NETWORKS AND OPENFLOW

Before understanding SDN and its benefits, one should look at the current network architecture, its implementation and shortcomings. This chapter will describe the current state of networking and will ease into SDN architecture, its concept, and benefits. Finally, the OpenFlow protocol will be described in detail, emphasizing its features related to the measurement of traffic matrices.

Let’s begin by discussing the fundamental components and behaviors of control and data planes, why they differ and how they might be implemented.

2.1 Control Planes

The control planes are the brain of networking equipment (such as routers or switches). Basically, the control plane decides how and where the packets are forwarded or processed [10], and usually is responsible of learning the network topology and populating routing tables. For example, the control plane of a router or layer 3 switch focuses on what neighbors the device has, how it should interact with each neighbor, what Routing Information Base (RIB) and Label Information Base (LIB) should be populated in the FIB (Forwarding Information Base) and which one would be used later by data plane to forward the incoming packets to the correct egress port [11-13].

2.2 Data Planes

Data planes (also known as forwarding plane) are the hardware and software parts of a device where, upon receiving a packet from an ingress port, a look up is performed in the routing table in order to find the egress port to which the packet has to be forwarded. The routing table is previously populated by the control plane. No complex decisions are taken in the data plane, rather following a set of rules defined by the control plane, and executing them as fast as possible. The performance of the data plane is more hardware-dependent than in the case of the control plane. For example, the use of TCAM[3] (Ternary Content Addressable Memory) [14] will impact the performance of data planes greatly.

Figure 1 shows a typical network architecture, where control and data planes reside in the same hardware. In a device with multiple CPU architecture, the central CPU is typically responsible of implementing control plane operations, while the line card CPUs execute the operations related to the high-speed data planes.

[3] TCAMs are a special type of computer memory used in certain very-high-speed searching applications [15]. Given the tight timing requirements of the longest-prefix matching algorithm executed every time a packet is forwarded, TCAMs are a key technology used in data planes of modern routers and switches.

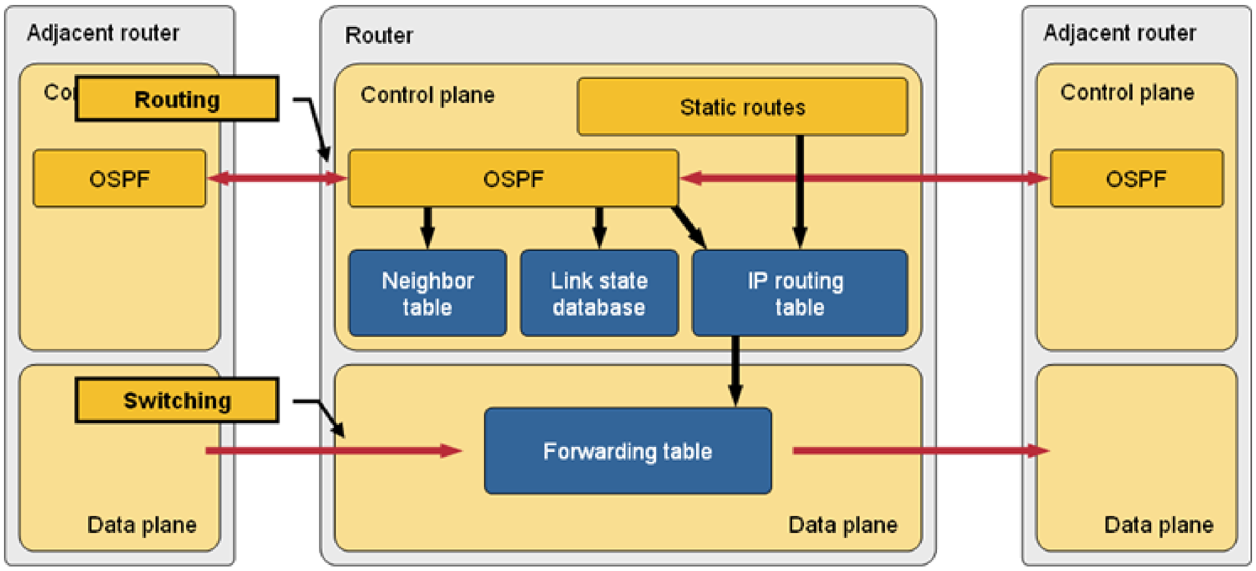


Figure 1: Control and data planes [16]

2.3 Distributed Control and Data Planes

The current architecture of today's networks is based on distributed control and date planes, which means the networking equipment in use today has its own proprietary control and data planes embedded into the hardware. Each vendor has his own control planes (operating system) which can operate and act differently. A router's control plane may as well be standalone from its neighbor as if each one is in a separate administrative domain. While link state flooding protocols have administrative awareness through concepts such as areas in IS-IS and OSPF, or Autonomous Systems (AS) in BGP [17], policy application is pretty elementary at best with such a high level of distribution and low levels of abstraction.

In this model, the individual routers from the same or different vendors participate to distribute reachability information in order to develop a localized view of a consistent, loop free network. Figure 2 shows a typical network architecture, where the control plane (OSPF, RIP, BGP) will populate the RIB table [18], which is optimized for fast lookup of destination addresses, and subsequently will be used by the FIB in data planes. This process depends on the type of network environment. For example, in enterprise networks only a subset of the most frequently used routes are cached from RIB to FIB, while routers used for accessing the entire Internet, however, experience severe performance degradation in refreshing a small cache, and various implementations have moved to having FIBs in one-to-one correspondence with the RIB [19].

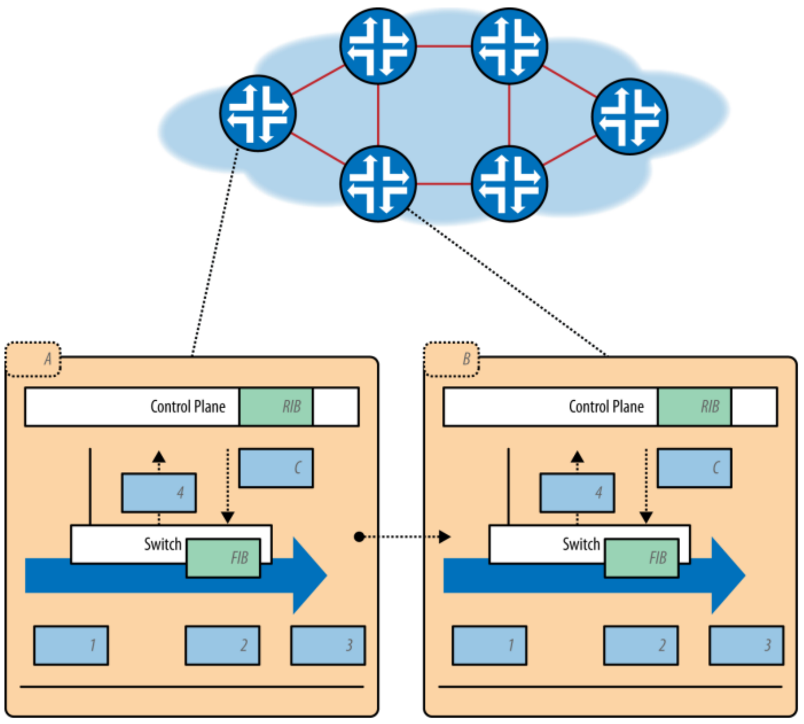


Figure 2: Control and data planes of a typical network [19]

2.3.1 Limitations of the current networking model

The current market requirements demand more than what the current network technologies can offer. In a time when IT departments are trying to squeeze the most out of their networks using device-level management tools and manual processes, while carriers are struggling to answer to constant increasing of demand for more bandwidth, the continuous demand for additional network capacity in the data centers, all are constrained by the limitation of current networks, which include:

- **Complexity:** There are many network protocols used in today's network. To meet technical needs over the past decade, many protocols had to evolve and very few had to be introduced. The problem is that in order to create a new protocol or improve an existing one, months or even years have to be dedicated for various processes of designing, testing, implementing and adapting by the industries. There is no easy way to test a new protocol today because creating a large enough testbed requires substantial investment. Besides, much equipment is using proprietary software and hardware, which is closed source by nature. Hence the nature of networks has become very static.

- **Scale**: As demand grows, so does the cost and complexity of adding hundreds or thousands of network devices to cope with it. Each time a new device is added to the network, it has to be configured and managed by the IT department, which makes the process complex, costly and time consuming.

- **Cost**: Other than adding new devices to the network to cope with demand which claim its own cost, every device in the network has its own life cycle. Network operators have to follow a hardware upgrade path, to solve the scale issue or to process related problems. Since the current model mostly relies on integration of control and data planes into the same hardware, when there is a need to upgrade forwarding cards, operators are forced to upgrade both planes. Even though vendors have tried to introduce several solutions by separating the control and data planes, hardware dependencies from each other, or providing the ability of adding more forwarding cards by means of modules, in many cases complete change of equipment is still required [19].

- **Policy**: To apply network-wide policies, the IT department may have to configure hundreds of devices. For example, creating a new VLAN (Virtual LAN) often takes hours preparing the configuration for all the devices and has to be done in the shortest possible time, in order to avoid any downtime that may occur by misconfiguration or adjustment in the network. Again, this is due to the nature of the current model which heavily relies on a distributed control plane model [20].

- **Pre-emptive market:** Today, when carriers or enterprises want to deploy new capabilities or services, they are limited by the vendor's product cycle and compatibility with the current equipment, and it often happens that devices from different vendors cannot talk to each other. For example, in order to take advantage of a specific protocol like Interior Gateway Routing Protocol (IGRP), which is developed by Cisco, all the routes in the network have to be from Cisco [21].

2.4 Software-Defined Networking

With the introduction of the software defined networking (SDN) concept, network operators hope to address some of the aforementioned shortcomings. Two of the most important characteristics of SDN follow. First, SDN separates the control plane from the data plane, and secondly SDN can consolidate the control plane in a way in which a single software control program can control multiple data plane elements.

Although SDN has gained significant tractions, the idea of programmable networks span out across the past 20 years. The history can be divided into three stages:

1. Active networks (from the mid-1990s to the early 2000s), which introduced programmable functions in the network, leading to greater innovation.

2. Control and data planes separation (from around 2001 to 2007), which developed open interfaces between the control and data planes.

3. The OpenFlow API and network operating systems (from 2007 to around 2010), which represented the first widespread adoption of an open interface and developed ways to make control- and data-plane separation scalable and practical [22].

It is very important to clarify that SDN as a whole is a concept, not a protocol. For example, a slightly different view of SDN is what some people refer to as software-driven networks [19], which highlights a different philosophy. In the software-driven [19] approach, one sees OpenFlow and that architecture as a distinct subset of functionality. Rather than viewing the network as being comprised of logically centralized control planes with brainless network devices, one sees the world as more of a hybrid of the old and the new. The history of SDN and software driven networks are out of scope of this thesis and were mention merely as background knowledge - here the main focus will be on centralized or semi centralized control plane and, in particular, the OpenFlow protocol.

There are other similar concepts to SDN, like Routing Control Platform [23] (RCP) and Network Configuration Protocol [24] (Netconf). For example, RCP uses Border Gateway Protocol (BGP) as a control channel to control forwarding decisions of routers inside an autonomous system (AS). It represents an autonomous system as a single logical entity and compute routes for all routers inside an AS. RCP is an early example of control and data plane separation.

2.4.1 Architecture

The SDN architecture relies on the idea that network intelligence is (logically) centralized in software-based SDN controllers, which maintain a global view of the network. As a result, the network appears to the applications and policy engines as a single, logical switch. With the help of SDN, enterprises and network operators gain vendor-independent control over the entire network from a single logical point, which greatly simplifies the network design and operation. [20]. Figure 3 shows an overview of software defined networking architecture. The following list defines and explains the architectural components of SDN:

- **SDN Application (SDN App):** SDN applications are programs written using higher level programming languages, depending on controller platform, which can directly communicate their network requirements and desired network behavior to the SDN controller via the so-called Northbound Interfaces (to be defined later). The applications are introducing even higher level of abstracted network control, they can be easily created, tested and modified without affecting the network operation. HP has even opened up a SDN app store which provides a marketplace for SDN innovations along with all the tools needed to create, test and validate SDN apps [25, 26].

- **SDN Controller:** The SDN controller [27] is a platform which facilitates the requirements for the application layer to communicate with the SDN datapaths. It also provides an abstract view of the network. An SDN Controller consists of one or more NBI (SDN Northbound Interfaces) agents, the SDN Control Logic, and the CDPI driver (Control to Data-Plane Interface). Its definition as a logically centralized entity neither prescribes nor precludes implementation details such as the federation of multiple controllers, the hierarchical connection of controllers, communication interfaces between controllers, nor virtualization or slicing of network resources [28].

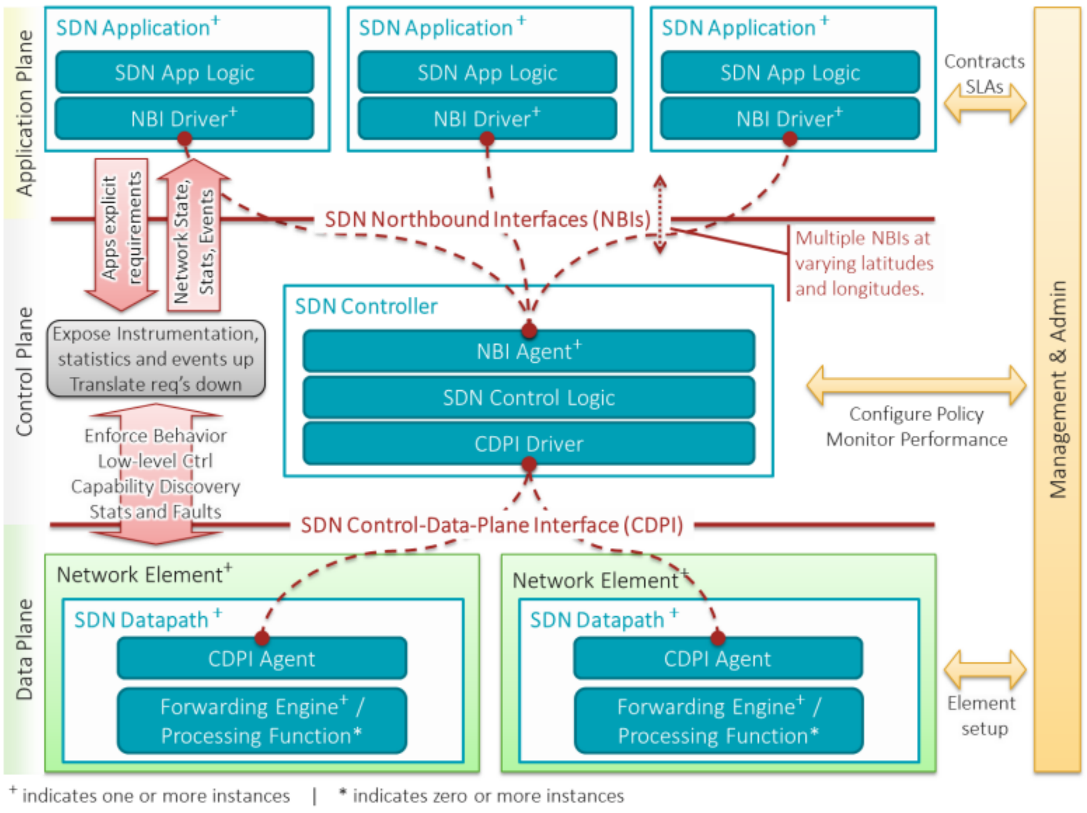


Figure 3: Overview of the Software-Defined Networking Architecture [27]

- **SDN Datapath:** The SDN datapath is a platform which enables network devices to advertise their forwarding and data processing capabilities to the controller. It is in charge of traffic handling, though the decision making has been moved to the application layer. The datapath will forward the incoming traffic according to its forwarding engine (flow table in OpenFlow). This facilitates the vendor-independent operation of network devices.

- **SDN Control to Data-Plane Interface (CDPI):** The SDN CDPI is the interface defined between an SDN Controller and an SDN Datapath [28], which provides at least:
  - Programmatic control of all forwarding operations
  - Advertisement of capabilities
  - Statistics reporting
  - Event notification.

- **SDN Northbound Interfaces (NBI**): SDN NBIs are interfaces between SDN applications and SDN controllers, and typically provide abstract network views and enable direct expression of network behavior and requirements. This may occur at any level of abstraction (latitude) and across different sets of functionality (longitude) [28].

2.5 OpenFlow Protocol

The OpenFlow protocol is the first standard communication interface defined as CPDI or southbound interface. SDN and OpenFlow are often used (incorrectly) interchangeably, leading to much confusion. Whereas SDN is an abstract concept, OpenFlow is an over-the-wire protocol like IP and TCP with explicit specifications and behaviors. Cisco's One Platform Kit (OnePK [29]), for instance, is another approach. OpenFlow implies openness or, being more specific, open source.

OpenFlow uses the concept of flows to identify network traffic based on pre-defined match rules that can be statically or dynamically programmed by the SDN control software. Because Openflow works on a per-flow basis it provides extremely granular control, enabling the network to respond to real-time changes at the application, user, and session levels. Since the fundamental of Traffic Matrix measurement is based on aggregation of flows from an ingress point to an egress point, OpenFlow provides a platform for the measurement of TMs, but as we will see, only the newest versions of OF seem to be TM-aware.

2.5.1 Evolution of OpenFlow Protocol

It has gone through many iterations, but the first version is originated out of a research project at Stanford University back in 2008 and later led to OpenFlow Consortium (OpenFlow Foundation). At the beginning of this thesis the latest OpenFlow version was and currently is 1.4. Although the prior version (1.3) has been out for few years now, the most implemented version until very recently was OpenFlow 1.0. Even now there are not many hardware solutions based on Openflow 1.3 and 1.4 and they are mostly developed in software switches. Only recently the adoption has increased its pace and currently there are many solutions widely available based on version 1.3 and 1.4. A brief history of Openflow can be summarized as follows [30]:

- OpenFlow 1.0 – Dec 2009
    - First widely implemented/deployed version
    - Single flow table, fixed 12 tuples (match fields for defining a flow)

- OpenFlow 1.1 – Feb 2011
    - Not popular - incompatible with 1.0
    - Multi-flow table, groups (Ability to perform group of action on a packet, like multicasting), full VLAN and MPLS support

- OpenFlow 1.2 – Dec 2011
    - First Open Networking Foundation (ONF) release – fixes 1.1
    - More flexibility.
    - Flexible flow match, flexible rewriting of packet header, IPv6.

- OpenFlow 1.3 – Apr 2012
    - Long term release : 1.3.1, 1.3.2, 1.3.3 (Mostly bug fixes)
    - Flexible measurement capabilities, Meters, Provider Backbone Bridges (PBB; known as "mac-in-mac"), event filters

- OpenFlow 1.4 – Aug 2013
    - Bundled actions, management of optical ports, flow monitoring (Keep track of changes to the flow tables, used in multi-controller setup), eviction (Ability of the switch to remove flow entries to reclaim resources)

Since the work described in this Thesis is based on OpenFlow 1.3 and 1.4, the following section will mainly focus on these specifications, describing the main components and key areas where it helped achieving our final goal.

2.5.2 Main Components of the Openflow Protocol

Figure 4 shows main components of an OpenFlow switch. The OpenFlow protocol is implemented on both sides of the interface between network infrastructure devices and the SDN control software, which will be covered in more detail [31, 32].

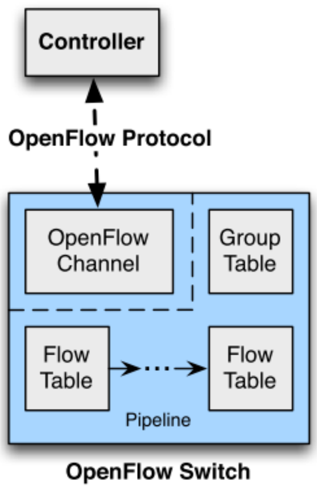


Figure 4: Main components of an OpenFlow switch. [32]

2.5.2.1 OpenFlow Channel

The OpenFlow channel is responsible for the connection between controllers and switches. The controller can connect using this interface to many switches, and since version 1.2 switches can connect to multiple controllers as well. The channel can be setup in a secure and encrypted way using TLS (Transport Layer Security) or simply run over TCP. The OF channel support three types of messages:

- **Controller-to-switch:** This type of message, as the name implies, is used by the controller to do various tasks on the switches, and may or may not require a response from the switch. Some of the messages are:
    - **Features:** used by the controller to request the identity and the basic capabilities of a switch.
    - **Modify-State:** used to add, delete and modify flow/group entries in the OpenFlow tables and to set switch port properties.
    - **Read-State:** used to collect information from the switches.
    - **Packet-out:** used by the controller to send packets out of a specified port on the switch and for forwarding packets received via Packet-in messages. They must contain the list of actions to be applied (Action Set). An empty action set drops the packet.
    - **Role-Request:** can be used to set the role of the controller or query its current state, and can be very useful when the switch connects to multiple controllers.
    - **Asynchronous-Configuration:** used by the controller to set an additional filter on the switch for asynchronous messages that it wants to receive on its OpenFlow channel, or to query the configuration of that filter.
- **Asynchronous:** they are sent without any initiation from the controller. For example, when the switch receives a new packet, it will send a Packet-in message to the controller in order to ask what action has to be applied on the packet.
- **Symmetric:** these types of messages are sent without any previous request, in either direction. For example, Hello (used in both the switch and controller), Echo (used in both, controller and the switch) and Error (used in switch) messages are categorized under symmetric messages.

Figure 5 describes an example of the initial connection between a switch and controller. The controller IP address and TCP port are preconfigured in the switch by the administrator (the default transport port was changed to 6653 in version 1.4). The switch will use the symmetric message “Hello” to check if the controller is reachable. Once the controller receives the “Hello” message, it will reply with another Hello, followed by a “Feature” message, asking the switch to advertise its capabilities. Upon receiving the feature reply message from the switch, which contains the switch Datapath ID (used to identify switches in the network), the controller will send the initial configuration parameters using the Set-config message, installing flow table and table-miss[4] flow entries.

[4] Table Miss defines an action that the switch has to take upon receiving an unknown flow, whether to drop it or send it to the controller.

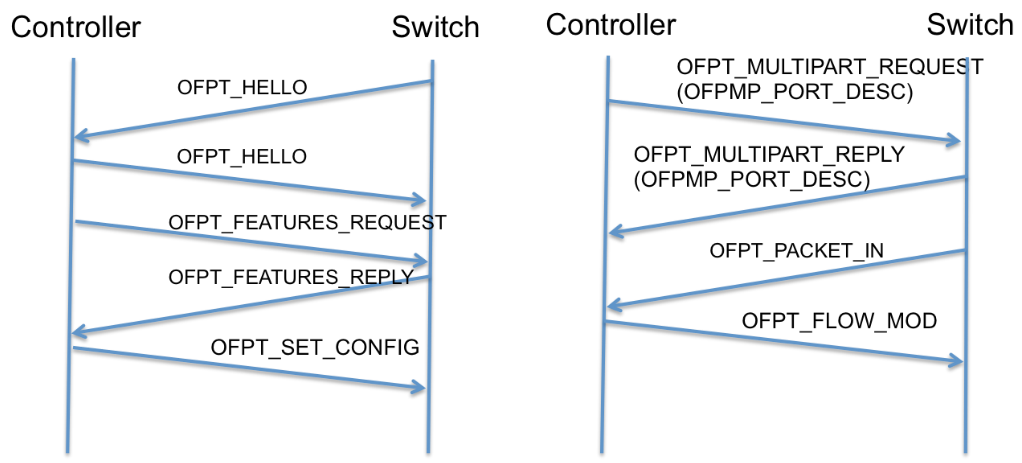


Figure 5: OpenFlow Connection Setup [33]

2.5.2.2 OpenFlow Tables

A key element in an OpenFlow switch are its tables. OpenFlow 1.0 only supported a single table which deemed not sufficient; hence multiple tables were introduced in version 1.1.0 and fully used by version 1.3. OpenFlow supports three types of tables; flow tables, group tables and meter tables – the latter proved to play an essential role towards the goal of this thesis.

2.5.2.3 Flow Tables

Flow tables contain a set of flow entries set up by controller. Upon receiving a packet, the switch will perform a table lookup to see if a flow matches a flow entry in the table and, depending on the presence of pipeline processing, may perform table lookups in other flow tables. Flow entries in the flow tables contain the following:

- **Match fields:** These consist of any number of elements that can define a flow in packet headers. Some of the important match fields are shown in Table 1. The combinations of fields bring the flexibility needed to perform specific tasks on different set of flows.

- **Priority:** can be relevant during the flow entry matching - the flow entry with the highest priority will be selected.

Field name	Description	Field name	Description
in_port	Switch input port	ipv6_src	IPv6 source address
eth_dst	Ethernet destination address	ipv6_dst	IPv6 destination address
eth_src	Ethernet source address	vlan_vid	VLAN id
ip_src	IPv4 source address	mpls_label	MPLS label
ip_dst	IPv4 destination address	meta	Metadata passed between tables
tcp_src	TCP source port	udp_src	UDP source port
tcp_dst	TCP destiny port	udp_dst	UDP destiny port

Table 1 : Packet Header Match Fields

- **Counters:** updated when packets are matched. Since version 1.0, counters have evolved a lot. Table 2 shows the full list of counters in OpenFlow 1.0 and 1.3/1.4. Please note that most of these counters are optional, which means the OpenFlow switch does not necessarily have to support them. Only the counters marked "Required" have to be supported by the OpenFlow switch

  Depending on the instruction of flow entries and on the number of counters available to the switch, each time a flow is matched its corresponding counter will be updated. As shown in tables 2 a lot has been added in terms of counters since Openflow 1.0, like Group counters which updates when a flow entry is point to multipoint (group). Groups enable Openflow to perform additional forwarding tasks on the packets (e.g. used for multicast or broadcast).

  Another addition to OpenFlow counters and perhaps the most significant one regarding traffic monitoring is the concept of Meter counters, which enable OpenFlow to measure the rate of all the flows that point to a meter.

- **Instructions:** each time a flow is matched to a flow entry in the table, the instructions that were defined for that flow entry will be executed. For example, they can instruct flow to go to the next flow table or meter table, or write actions in the action set (flow entries with empty action set will drop the packet). If the instruction contains actions to be applied, all the actions in the action set will be executed and the packet will be forwarded.

- **Timeouts:** define the maximum amount of time or idle time before flow entries expire. (Idle time equal to zero means flow entry will never get expired). It is important to note that flow entries can be removed before their idle time by the controller.

- **Cookie:** opaque data value chosen by the controller. May be used by the controller to filter flow statistics, flow modification and flow deletion. Not used when processing packets.

Counter	Bits
Per Table	
Active Entries	32
Packet Lookups	64
Packet Matches	64
Per Flow	
Received Packets	64
Received Bytes	64
Duration (seconds)	32
Duration (nanoseconds)	32
Per Port	
Received Packets	64
Transmitted Packets	64
Received Bytes	64
Transmitted Bytes	64
Receive Drops	64
Transmit Drops	64
Receive Errors	64
Transmit Errors	64
Receive Frame Alignment Errors	64
Receive Overrun Errors	64
Receive CRC Errors	64
Collisions	64
Per Queue	
Transmit Packets	64
Transmit Bytes	64
Transmit Overrun Errors	64

Counter	**Bits**	
Per Flow Table		
Reference Count (active entries)	32	*Required*
Packet Lookups	64	*Optional*
Packet Matches	64	*Optional*
Per Flow Entry		
Received Packets	64	*Optional*
Received Bytes	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Port		
Received Packets	64	*Required*
Transmitted Packets	64	*Required*
Received Bytes	64	*Optional*
Transmitted Bytes	64	*Optional*
Receive Drops	64	*Optional*
Transmit Drops	64	*Optional*
Receive Errors	64	*Optional*
Transmit Errors	64	*Optional*
Receive Frame Alignment Errors	64	*Optional*
Receive Overrun Errors	64	*Optional*
Receive CRC Errors	64	*Optional*
Collisions	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Queue		
Transmit Packets	64	*Required*
Transmit Bytes	64	*Optional*
Transmit Overrun Errors	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group		
Reference Count (flow entries)	32	*Optional*
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Group Bucket		
Packet Count	64	*Optional*
Byte Count	64	*Optional*
Per Meter		
Flow Count	32	*Optional*
Input Packet Count	64	*Optional*
Input Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Meter Band		
In Band Packet Count	64	*Optional*
In Band Byte Count	64	*Optional*

Table 2: Left: List of Counters in OpenFlow 1.0. Right: List of counters in Openflow 1.3 and 1.4 [4, 31, 32]

2.5.2.4 Matching

When a packet arrives to the OpenFlow switch, the switch will perform a table lookup in the first flow table as shown in Figure 6. Packet match fields are read from the packet and will be used to match against flow entries, and depending on the instruction associated with it, various operations can be applied on the packet. For example, the instruction can send a packet to the next flow table to match against different entries while writing actions to the action set and updating the metadata. This process continues until the action set contains an output action hence forwarding the packet to its destination. Each table must support a table-miss flow entry to process table misses.

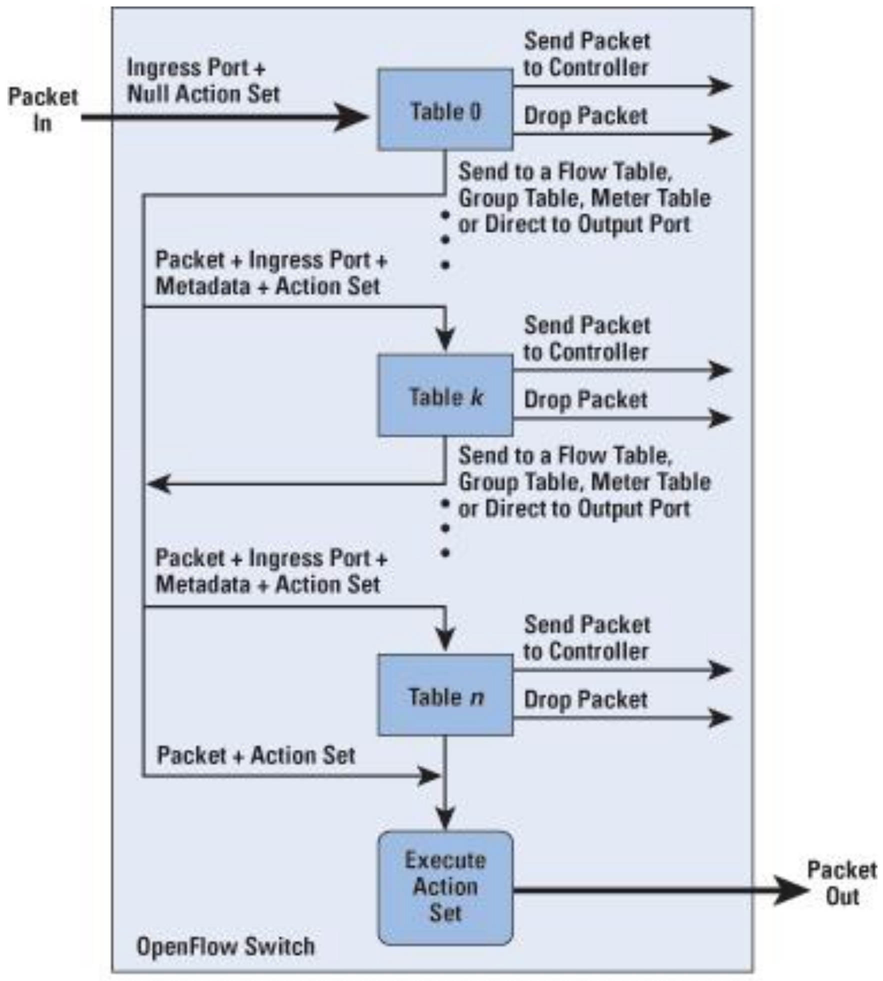

Packet In
Ingress Port + Null Action Set
Table 0
Send Packet to Controller
Drop Packet
Send to a Flow Table, Group Table, Meter Table or Direct to Output Port
Packet + Ingress Port + Metadata + Action Set
Table *k*
Send Packet to Controller
Drop Packet
Send to a Flow Table, Group Table, Meter Table or Direct to Output Port
Packet + Ingress Port + Metadata + Action Set
Table *n*
Send Packet to Controller
Drop Packet
Packet + Action Set
Execute Action Set
Packet Out
OpenFlow Switch

Figure 6 : Packet Flow through OpenFlow-Compliant Switch [34]

2.5.3 OpenFlow Operation

Figure 7 shows the process of packet flow through OpenFlow switch. Let's assume that host 1 wants to send data to host 2. The controller can configure the switch in three different ways [35]:

- **Reactive flow instantiation:** When a packet arrives from host 1, the switch performs a lookup in the flow tables. Since no flow entries have been set by the controller, the switch creates an OFP packet-in packet and sends it off to the controller asking for instructions. The controller will send a packet-out packet instructing the switch to forward the packets to the port where host 2 is connected. This model of instantiation is called Reactive flow. The downfall to this model is the latency created for the packets to be sent and received by the controller, together with the high burden of messages and CPU processing associated with the detection of every new flow.

- **Proactive flow instantiation:** In this mode rather than reacting to a packet, the controller can populate the flow tables ahead of time, matching all the flows from host 1 to be forwarded to host 2. Proactive flow tables eliminate any latency induced by consulting a controller on every flow. But the downfall to this model is that there is no flexibility.

- **Hybrid flow instantiation:** A combination of both approaches would leverage the flexibility of reactive instantiation for particular sets and a granular traffic control of Proactive instantiation, still preserving the low-latency forwarding for the rest of the traffic. For example, when the first packet arrives from host 1 to the switch, it will be forwarded to controller, and the controller will insert a flow entry with finite idle-time directing the packets to host 2. When the idle-time expires, the switch will remove the flow entries from the table and the next packet will be forwarded to the controller. Hybrid flow instantiation is the most common practice of deploying network in OpenFlow.

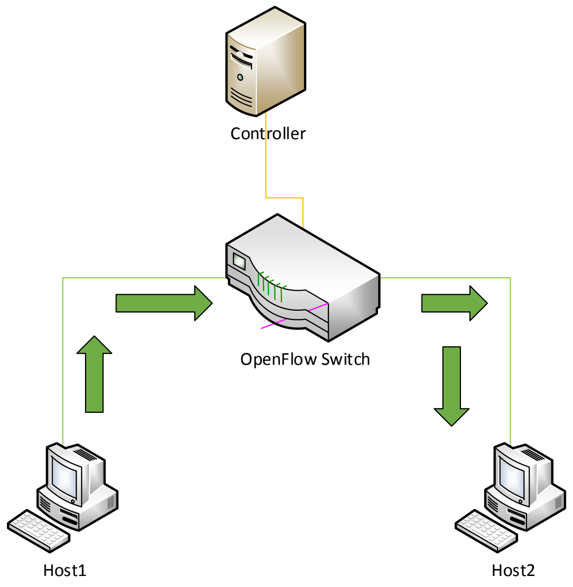


Figure 7: Packet Flow through OpenFlow-Compliant Switch

2.5.4 Multiple Controllers

OpenFlow version 1.2 introduced support for multiple controllers. Having multiple controllers improves reliability, as the switch can continue to operate in OpenFlow mode if one controller or controller connection fails. Other than reliability it can be used as load balancing for the controllers. For example, Figure 8 shows a multiple controller setup where controller 1 is used to only manage switches 1, 2, 3 and 6, and controller 2 and 3 are used as fail over for switch 4 and 5. When OpenFlow operation is initiated, the switch must connect to all the controllers it is configured with and try to maintain connectivity with all of them concurrently. There are three roles defined for the controllers: EQUAL, MASTER and SLAVE.

By default all the controllers are assigned as EQUAL, which means they have full access to the switch. Only one controller can be set to the MASTER role. This is to ensure the correct transition of authority between multiple controllers. When a controller changes its role to MASTER, the switch will change other controller role to SLAVE. The controller can also request its role to be changed to SLAVE; in this role the controller has only read-only access to the switch.

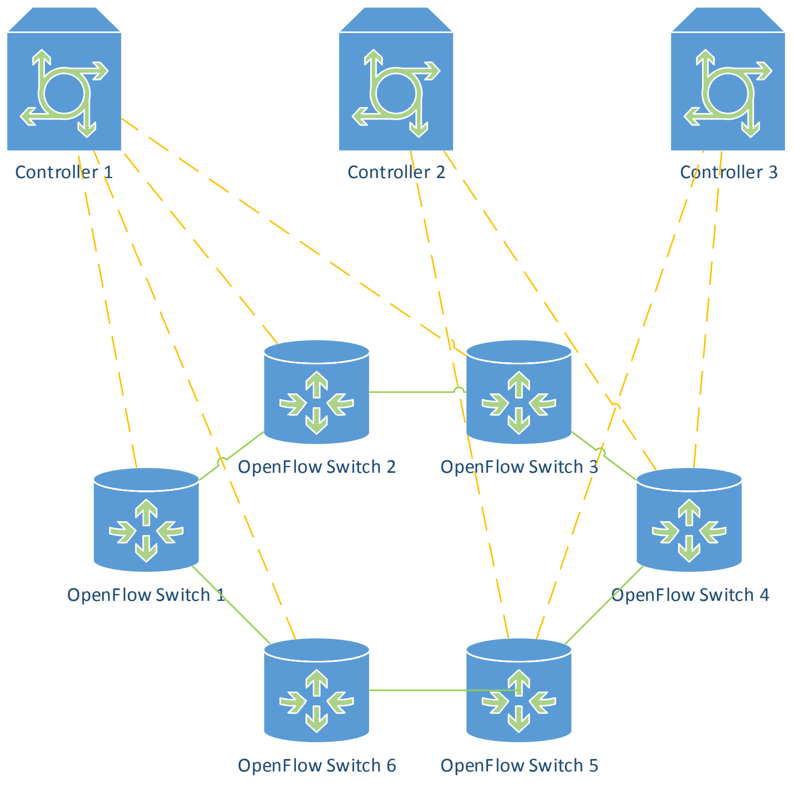


Figure 8: Multiple controller setup

2.6 Current Limitations and Future of SDN

SDN, as mentioned in Section 2.4, abstracts network functions by separating the control planes and data planes, moving the control plane software into a separate hardware. This makes it possible to add new functions to the network, but these functions are still constrained by the capabilities of the Data planes [36]. The current data plane model is based on the Match/Action approach, where a packet header is matched against a set of fields and performs an action according to set of rules. Openflow in particular has grown from 12 to 41 match fields since its introduction [37], but to create and implement new functions, one might need more freedom in defining new fields and actions. Furthermore, current hardware switches are only allowing a limited selection of packet processing actions.

SDN in its current form does not provide enough abstraction between the control and data planes. It is not easy to modify the packet format. Furthermore, writing programs for the data plane is not an easy task. What we need is a high-level language which should be protocol-independent, so one should be able to configure a packet parser and define a set of typed match action table independently of any current control plane protocol. It also should be target-independent, in the sense that provides the ability to write a program that process packet without having deep, intricate knowledge of the switch details. The language would essentially rely on a compiler to configure the target switch [37].

The current models being developed to address these issues are explained in detail in the following paragraphs.

2.6.1 Programmable Data Planes

To overcome the limitations of Data planes mentioned earlier and bring the freedom needed to create and implement new packet functions in the network, two different approaches can be taken, one by further abstracting the data planes operation into the software and make it less hardware dependent, and the second one by rethinking the current switch chip design to make it more flexible and programmable.

- **Click [38]**: The Click modular router is a customizable data plane implemented in software. It consists of elements (building blocks) in which each element provides a unique function like packet switching, lookup and classification. A programmer can take these elements and put them together into a packet processing pipeline, giving it the flexibility of building various complex functions on the fly in the data plane itself. Figure 9 shows it is possible to stitch together click elements to build an entire IP router [39, 40] or even new elements can be written to introduce new functions to the router. Various performance experiments show that Click is still about 90% as fast as the base Linux system for packet forwarding.

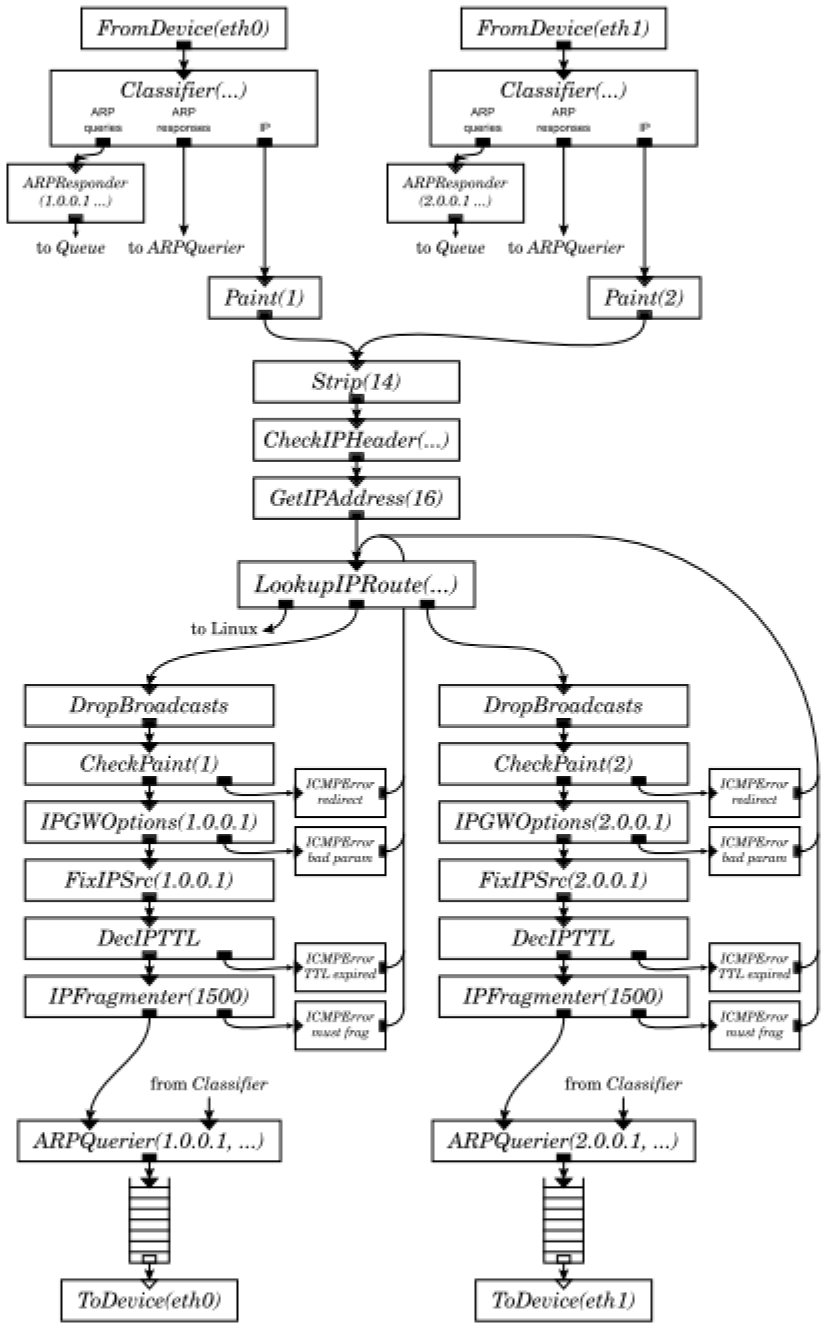


Figure 9: Click IP Router [38]

- **RouteBricks [41]:** implementing Data planes in software instances like Click may not be fast enough for production level deployment. One way of overcoming this obstacle is to make software faster, and this is what RouteBricks tries to accomplish. It uses clusters of servers to achieve fast forwarding rate in software. Figure 10 shows the design principal of RouteBricks, where each server hosts a line card emulating a port of a traditional router.

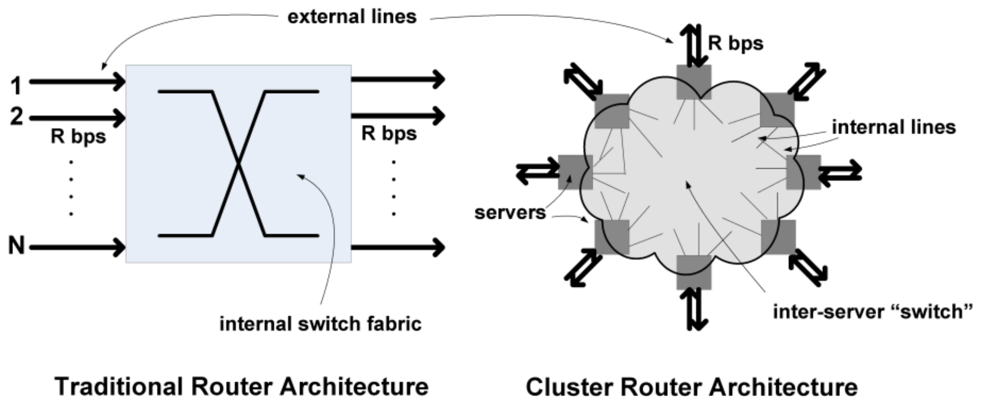


Figure 10: High-level view of a traditional router and a server [41]

- **OpenFlow Chip [36]:** Switch chips generally work much faster comparing to switching performed by CPUs. To achieve the speed needed to operate at 1 Tb/s and to make programmable data planes scalable, we need parallelism of dedicated hardware. RMT (Reconfigurable match tables) is a new RISC[5]-inspired pipelined architecture for switching chip that allows processing to effectively ride Moore's law. In other words, as chips get faster and faster, one can process packets at higher rates yet composing these instructions to perform fairly complex operations. It allows a set of pipeline stages each with tables of arbitrary depth and width. Figure 11 shows the RMT model, which enables new possibilities in terms of adding new fields, actions and queues while maintaining compatibility with current protocols.

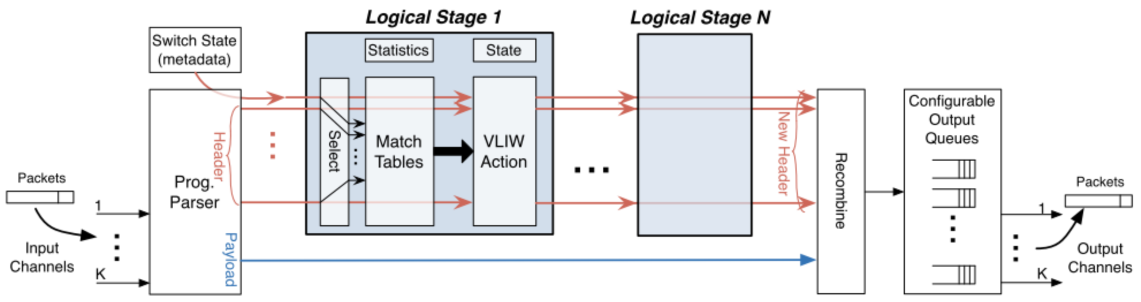


Figure 11: RMT model as a sequence of logical Match-Action stages [36]

[5] Reduced Instruction Set Computing [42], or RISC, is a CPU design strategy based on the insight that simplified instruction set (as opposed to a complex set) provides higher performance when combined with a microprocessor architecture capable of executing those instructions using fewer microprocessor cycles per instruction.

2.6.2 Programming Protocol-Independent Packet Processors

The Openflow chip design options explained earlier demonstrates that flexible mechanisms for parsing packets and matching header fields are possible using custom ASICs at terabit-per-second speed, but programming these new switch chips is really hard and complex. P4 [37] is a high-level language for programming protocol-independent packet processors, which tries to simplify the process of programming for switches by bringing further abstractions between the controller and the switches. Figure 12 shows the P4 approach working in conjunction with current protocols like Openflow, using its compiler to configure switches telling them how packets have to be processed.

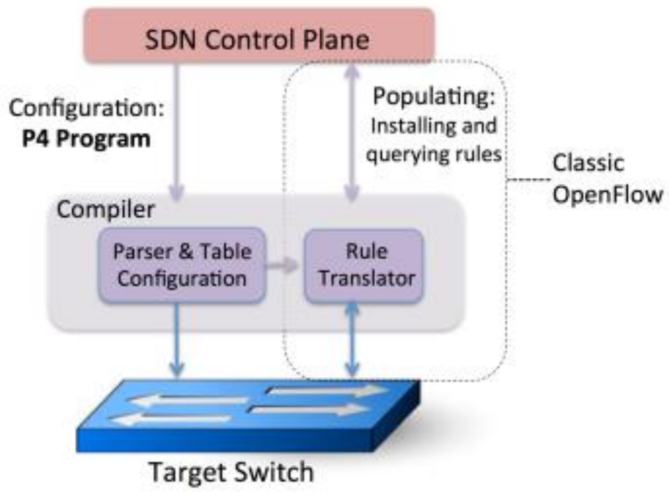


Figure 12: P4 Language [37]

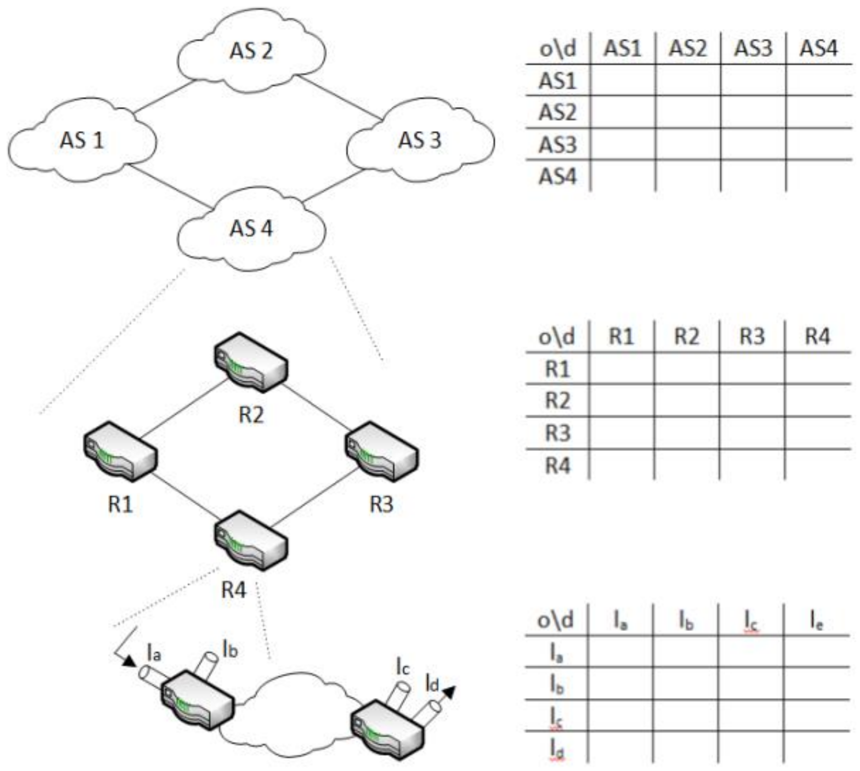
There are three main distinctions between OpenFlow and P4. First, OpenFlow works on a fixed parser, meaning the programs can be written using fixed set of predefined match/actions rules, whereas P4 supports a more robust parser where new headers can be defined. Second, OpenFlow works in series, meaning match/actions are operated in serial mode which limits the functionalities of programs written for OpenFlow, P4 solves this limitation by introducing parallel operational mode where multiple match/action can be performed by the switch simultaneously and this enables developers to create programs that can operate independently of concurrent programs running on the controller. Third, P4 model brings further abstraction to data planes by generalizing how packets are processed in different forwarding device with different chip technologies (e.g., fixed-function switch ASICs, NPUs, reconfigurable switches, software switches, FPGAs).

CHAPTER 3. TRAFFIC MATRICES

This chapter will focus on introducing the concept of Traffic Matrix (TM): why it is needed, the challenges in TM measurement, and the state of the art.

3.1 Definition

The Traffic Matrix is the volume of traffic transported by a network during a period of time from any ingress point to any egress point, i.e., from the point that connects to the device that generates a packet, to the point that connects to the device that receives it. There are several levels of aggregation of the TM, which can vary from ports (interfaces) of a single router, devices (switches or routers), to points of presence (PoPs) or entire Autonomous Systems (AS). Figure 13 shows an example of different aggregation levels, where diagonal entries in the table represent inbound traffic that doesn't cross the network.



o\d	AS1	AS2	AS3	AS4
AS1				
AS2				
AS3				
AS4				

o\d	R1	R2	R3	R4
R1				
R2				
R3				
R4				

o\d	I_a	I_b	I_c	I_e
I_a				
I_b				
I_c				
I_d				

Figure 13: TM aggregation levels [43]

3.2 Relevance of TMs

One could imagine working in a dark without TM. Knowing the network topology or routing information does not exactly puts emphasis on what is going on in the network. We are moving away from days of best effort network delivery, with the introduction of streaming content such as video, high definition television and Voice over IP (VoIP); excessive packet drops and delays would produce highly noticeable artefacts in these services hence making TM measurements more and more critical. Besides, by knowing what is going on in the network, TM can be used for various enhancements in network planning and optimization.

One of the benefits of knowing how data travels inside a network is that it can be used for capacity planning to ensure there is adequate bandwidth for users in the present and future, but at minimal cost. Network engineering tasks that need the TM as input include routing optimization [44], organizing traffic flows in the network, as well as modelling routing of large networks. This includes functions such as finding the shortest paths for flows but also, importantly, load balancing to ensure links remain uncongested. In all these cases, the traffic matrix is a key input to perform the tasks effectively and efficiently.

Detecting anomalies is another area where TM can be handy. It can happen due to events like network failures, infected hosts in a network, denial of service attacks (DoS) and compromised network infrastructures. Detecting anomalies in a timely fashion is essential to avoid any further damage and TM can help achieving that.

In the event of network failure, one should have a backup plan. This is where reliability analyze comes into play. Not only a redundant path should exist in case of failure but the path should be able to carry the sufficient capacity for the traffic. TM can help in planning reliability and load balancing in the network.

As demand for new services grows in today's network so does the need for new protocols to address them, designing of new protocols require realistic measurements or models of such traffic since one of the ways to evaluate a network protocol is depending on the amount of traffic it can carry. Having a good TM model (for example, from measurements in operational networks) one can create artificially synthesized traffic to test these protocols.

3.3 Measurement of TMs

There are several well-known strategies for collecting traffic measurement, one being "packet trace" which collects packets headers and timestamps to construct the TM. Even though it sounds perfect, actually, collecting packet headers is expensive due to the need of dedicated hardware and the huge amount of storage required. For example, over a terabyte of data per hour on links with 2.5 Gbps capacity is needed to construct the TM. The other method, and perhaps the most common one, is flow-level aggregation, where flows are defined by keys. For identifying a flow the keys are usually composed by 5 tuples, IP source and destination, TCP/UDP source and destination port numbers and transport protocol. This method is in line with flow characteristics of OpenFlow, where the flow can be defined by combining match keys stated in Table 1. This makes OpenFlow an ideal platform for Traffic Matrix Measurement. [44]

A less costly alternative to direct measurement is the indirect method, using easily obtainable link counts from the Simple Network Management Protocol (SNMP). SNMP, being an IETF standard, is available in almost every device in the network. Every router or switch maintains a cyclic counter of the number of bytes transmitted and received on each of its interfaces and records it in a Management Information Base (MIB). All what is needed to collect this information from the MIB is a SNMP puller that periodically records this data through an interface, typically a UDP port. [45]. Later, this information is combined with a model such as the Gravity Model [46].

Although a TM derived from SNMP data may imply lower computational cost, they are highly susceptible to error. Since data is usually transmitted via UDP, it may contain errors due to the polling procedure; also the sampling interval might be too high for the purpose of measuring the TM and might be inaccurate due to poor SNMP agent implementations, high loads or network delays. In order to obtain accurate TM data, more information is needed. For example as shown in Figure 14, since link loads are aggregate of all flows and do not include origin and destination, more information is needed to carry out a procedure of indirect inference of the TM from the SNMP link loads. Finally, the need of a model to obtain the estimation of the TM introduces further inaccuracy.

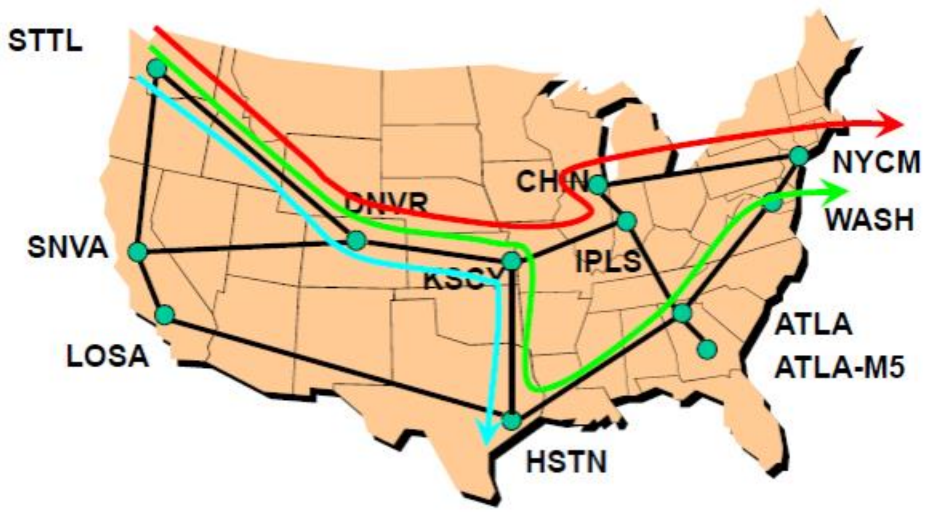


Figure 14. End-to-end flows example in Abilene network [47]

Therefore, the indirect method has a lot of drawbacks. One of the most adopted applications that uses direct measurement is NetFlow, which is described in details below.

3.3.1 NetFlow

NetFlow [6] is a technology developed by Cisco that monitors and records all traffic passing through the supported NetFlow router/switch. It has become a standard and the majority of network equipment manufacturers implement it in their devices. It consists of three main components, as shown in Figure 15:

- **NetFlow probe (exporter)**: It monitors, for each packet, the source and destination IP address, source and destination TCP/User Datagram Protocol (UDP) ports, type of service (ToS), packet and byte counts, start and end timestamps, input and output interface numbers, TCP flags and encapsulated protocol (TCP/UDP), among other information, to create flow records and transmit that data to the NetFlow collector.

- **NetFlow collector:** collects the records sent from the exporter, organizes those flow records into an easy-to-read format, stores them in a local database, and forwards the records to an analyzer [6].

- **NetFlow analyzer:** analyzes the NetFlow records for information of interest, which may include bandwidth usage, policy adherence, and forensic research. A screenshot of a network analyzer is shown in Figure 16.

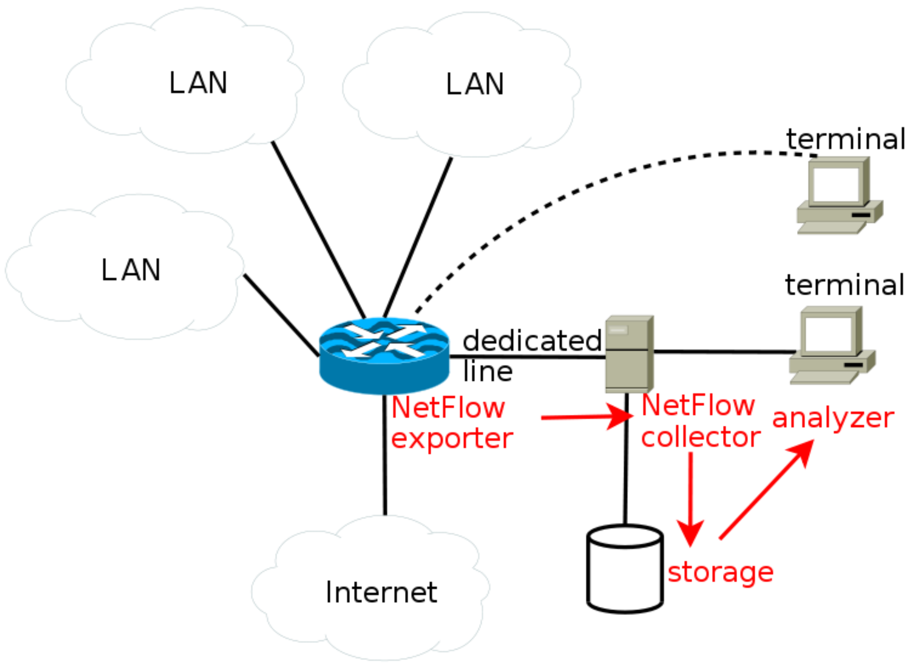
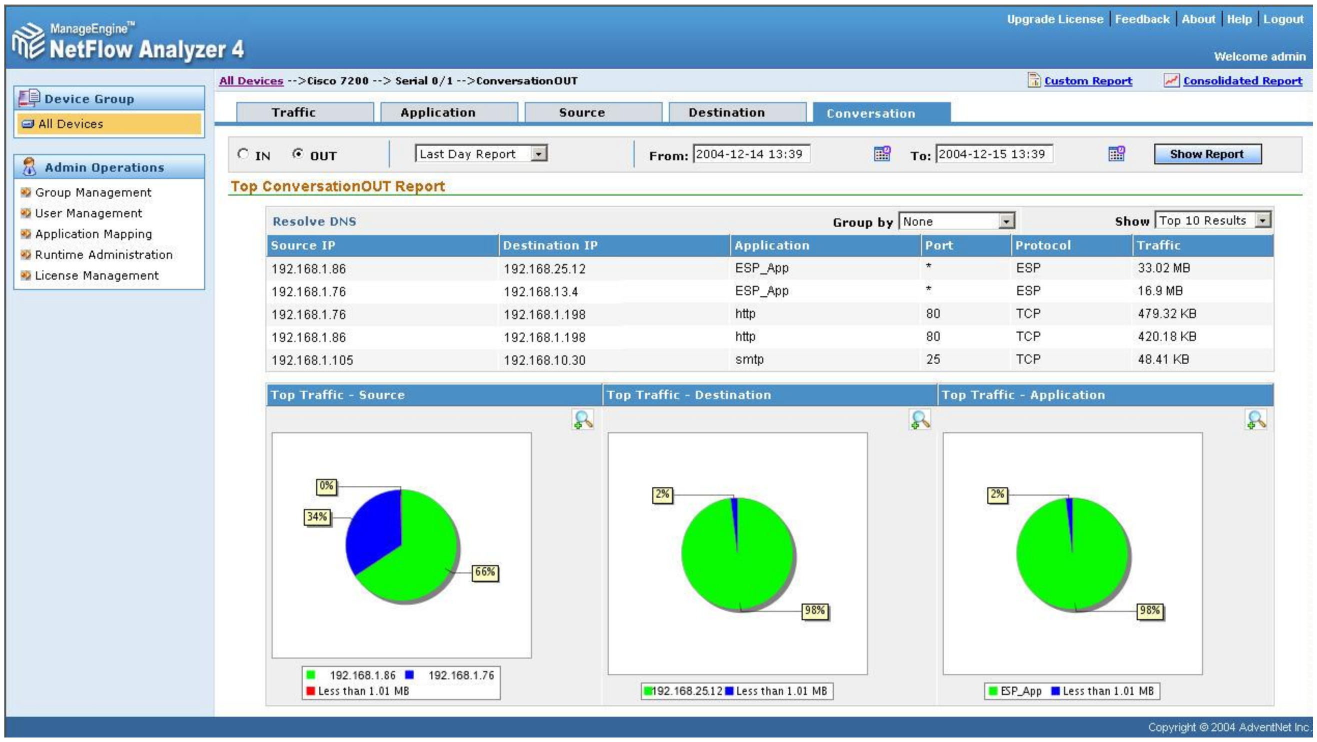


Figure 15 : NetFlow Architecture [48]



IN [✓ OUT]   Last Day Report   From: 2004-12-14 13:39   To: 2004-12-15 13:39

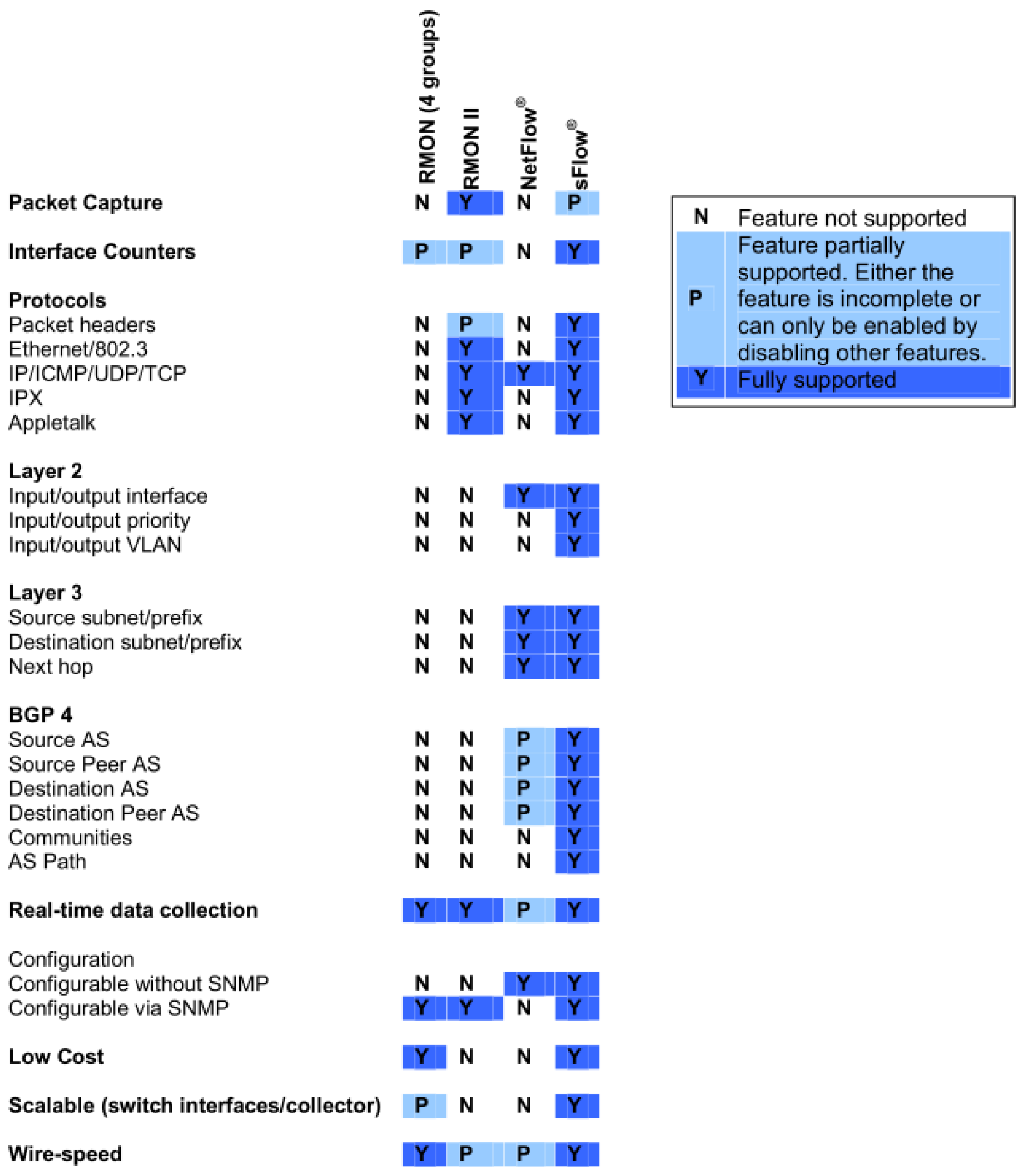
Group by: None   Show: Top 10 Results

Source IP	Destination IP	Application	Port	Protocol	Traffic
192.168.1.86	192.168.25.12	ESP_App	*	ESP	33.02 MB
192.168.1.76	192.168.13.4	ESP_App	*	ESP	16.9 MB
192.168.1.76	192.168.1.198	http	80	TCP	479.32 KB
192.168.1.86	192.168.1.198	http	80	TCP	420.18 KB
192.168.1.105	192.168.10.30	smtp	25	TCP	48.41 KB

Figure 16: Example of NetFlow analyzer interface [49]

Although NetFlow is widely in use today, in order to reduce the number of flow recorded at a router, NetFlow uses packet sampling which may make the final result not so accurate. Plus, the cost of deploying NetFlow in a network is high and in many cases not feasible for small enterprise to adhere. Aside from inaccurate data and cost, sending NetFlow can add too much overhead to already over-loaded routers and switches hence stopping engineers from enabling NetFlow on their network.

There are other solutions that work similar to NetFlow, sFlow is one of them and provides the ability to continuously monitor application level traffic flows at wire speed on all interfaces simultaneously [7]. It provides more features and sampling but has the same disadvantages as NetFlow. Other similar technologies are shown in Figure 17.



	RMON (4 groups)	RMON II	NetFlow®	sFlow®
Packet Capture	N	Y	N	P
Interface Counters	P	P	N	Y
Protocols				
Packet headers	N	P	N	Y
Ethernet/802.3	N	Y	N	Y
IP/ICMP/UDP/TCP	N	Y	Y	Y
IPX	N	Y	N	Y
Appletalk	N	Y	N	Y
Layer 2				
Input/output interface	N	N	Y	Y
Input/output priority	N	N	N	Y
Input/output VLAN	N	N	N	Y
Layer 3				
Source subnet/prefix	N	N	Y	Y
Destination subnet/prefix	N	N	Y	Y
Next hop	N	N	Y	Y
BGP 4				
Source AS	N	N	P	Y
Source Peer AS	N	N	P	Y
Destination AS	N	N	P	Y
Destination Peer AS	N	N	P	Y
Communities	N	N	N	Y
AS Path	N	N	N	Y
Real-time data collection	Y	Y	P	Y
Configuration				
Configurable without SNMP	N	N	Y	Y
Configurable via SNMP	Y	Y	N	Y
Low Cost	Y	N	N	Y
Scalable (switch interfaces/collector)	P	N	N	Y
Wire-speed	Y	P	P	Y

N	Feature not supported
P	Feature partially supported. Either the feature is incomplete or can only be enabled by disabling other features.
Y	Fully supported

Figure 17 SFlow Overview [7]

CHAPTER 4. PROBLEM STATEMENT AND PROPOSED SOLUTION

This chapter will present the state of the art in obtaining the TM with Openflow, their limitations, and possible solutions towards the goal of measuring Traffic Matrices. Later we will present our design for deploying a distributed traffic measurement technique and different ways it can be implemented. At the end of this chapter a complete analysis of our framework will be presented.

4.1 Objective

Monitoring is a crucial part of network management. The management applications need constant statistics on network resources at different levels of aggregation (flow, packet and interface) [50] in order to manage network efficiently and change network behavior according to the resources available in the network. As described in Section 2.4, SDN introduces new possibilities towards obtaining the TM, specifically the fact that there is a controller which has a complete view of the network. In order to quickly adapt forwarding rules in response to workload in the network, SDN must be able to continuously monitor network resources [51].

As stated in Section 3.2.1 there exist already solutions for conventional networks to obtain TM, like NetFlow, sFlow and SNMP link data, which come with their limitations and drawbacks. Our goal is to reduce or eliminate those limitations in the new Openflow-enabled network scenario.

Since the wide adoption of OpenFlow protocol and its embracement by the industry made it a de facto and synonymous with SDN, it is chosen as our platform. The objective is to enable TM measurement with minimal burden in Software-defined Networks where Openflow protocol is implemented, taking advantage of the most recent features introduced in the latest versions of Openflow. There are factors needed to be taken into consideration when designing monitoring tools for SDN: 1) there should be a balance between accuracy and overhead, and 2) the monitoring application shouldn't interfere with other application running on top of SDN, such as firewalls, load balancers, or forwarding, among others.

Other authors have proposed monitoring solutions based on SDN and OpenFlow that facilitate network monitoring in SDN (PayLess, OpenTM, FlowSense, and OpenSketch, to name a few), but almost all of these tools are based on earlier versions of OpenFlow. We will discuss their features and their limitations in the following section.

4.2 Related Works

4.2.1 OpenTM

OpenTM [52] is an application that runs in the controller and periodically query switches along the path to obtain the values of the per-flow counters (bytes and packet per time unit). It then uses routing information to construct the TM by aggregating the flows that originated from the same source and egress at the same destination. The total number of queries generated by OpenTM during each interval is bounded by the number of active flows in the network. To minimize switch and network load during TM operation, OpenTM keeps track of switch loads and will query the switch with least overhead using the number of outstanding queries[6] on each switch as the load metric.

In order to determine which switch to query, different switch querying strategies are proposed: (a) querying the last switch in the path (the egress point), (b) querying switches on the flow path uniformly at random, (c) round-robin querying, (d) non-uniform random querying that tends to query switches closer to the destination with a higher probability and (e) querying the least loaded switch.

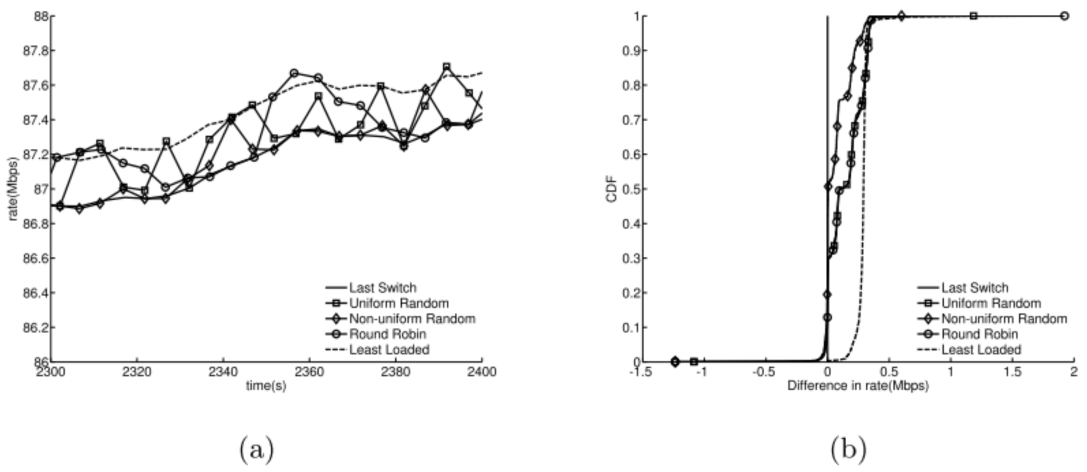


Figure 18: OpenTM Evaluation [52]

Figure 18(a) shows the average throughput versus time for traffic between two hosts and figure 18(b) shows the cumulative distribution function (CDF) of differences of rate between each querying strategy. From Figure 18 one can conclude that each method comes with a tradeoff. For example, the last switch method results in highest accuracy but with the highest load, where least loaded strategy results in least accuracy since the first switch in the path is usually the least loaded.

OpenTM promises high accuracy with low overhead but in reality it falls short to its promises. Since it is a query-based monitoring tool, it needs to constantly query switches along the path. Therefore, by measuring network-wide traffic matrix by periodically polling one switch on each flow’s path for collecting flow level statistics, it is causing a significant overhead. On the other hand, using a polling method that randomly selects some switches may affect accuracy if the switch is not carefully chosen. Finally, computing the TM data may result in significant CPU

[6] The shortest time it takes to query a switch and receive a reply from it is considered as its load metric.

overhead of the controller, since it needs to analyze all the flow to determine the ingress and egress points and construct the TM table.

4.2.2 FlowSense

FlowSense [53] takes a push-based approach by using the Packet-In and Flow-Removed messages received by the controller to indirectly compute the utilization of links between switches. This approach enables FlowSense to monitor OpenFlow networks with zero measurement cost. Instead of querying each switch for the flow stats, FlowSense will use the information available from Packet-In messages (which is received by the controller when a new flow arrives to the switch) and flow-Removed message (when an existing flow removed by the switch either due to Flow expiration or direct removal by the controller) to approximately measure the aggregate of flows that ingress and egress the network during a time interval. Figure 19 shows the basic implementation of FlowSense. The parser module captures the control traffic and sends it to the monitor, where link utilization is estimated.

Although this approach has zero cost of monitoring, the estimation of the TM is limited due to the characteristics of traffic flows. For example, for TM estimation may have inaccurate results due to flows with long idle-timeout exceeding the interval time, therefore stats from these flows are never calculated. In the case of Figure 19, to calculate the TM at “t6” the flow f2 can be incorporated to the TM only after time “t6”. Moreover, there might exist flows with zero idle-timeout, which means that a certain flow entries in the switch will never get expired. This method, as described in Section 2.4.1, is used to reduce controller load.

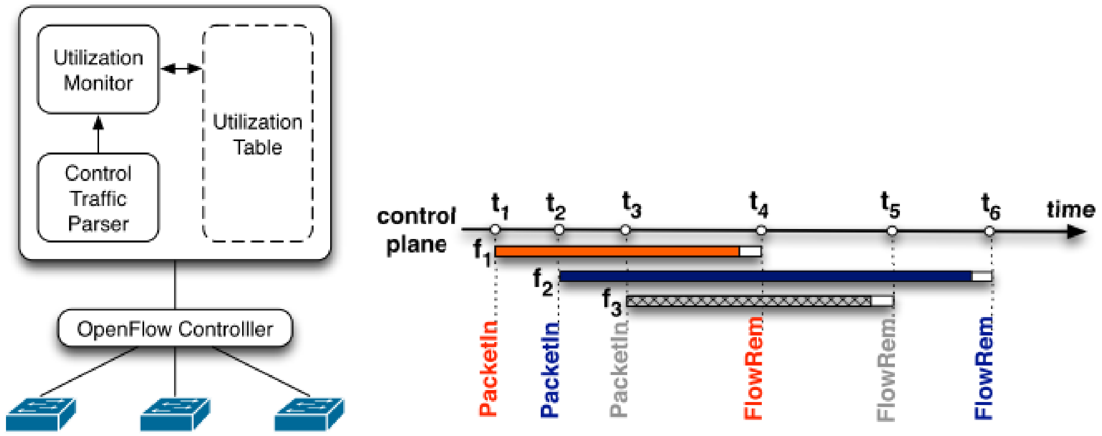
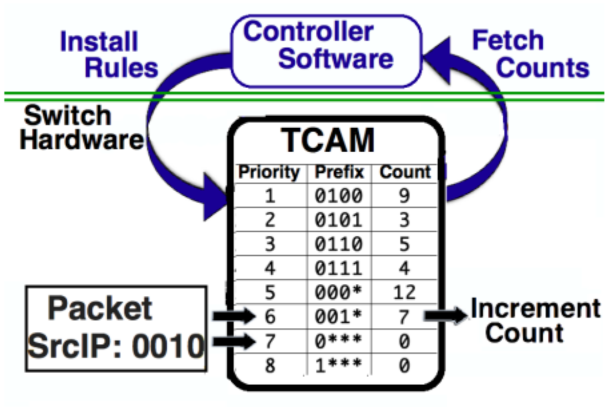


Figure 19: source FlowSense [53]

4.2.3 Commodity-Switch

Online Measurement of Large Traffic Aggregates on Commodity Switches [54] takes advantages of wildcards rules that match on bits in the packet header to update counters on the switch. Figure 20 shows the framework of this approach, where during packet processing, the switch identifies the matching rules, picks the rule with the highest priority, and updates the associated counter.

While the framework is designed to reduce overall overhead of estimating TM while being able to sufficiently measure and identify large traffic and Denial of Service (DDoS), it sacrifices accuracy to do so. For example, an algorithm can detect large traffic aggregates by iteratively adjusting the wildcard rules—leading to a short delay in detecting the traffic, while producing intermediate useful results at a somewhat coarser level of aggregation.



Priority	Prefix	Count
1	0100	9
2	0101	3
3	0110	5
4	0111	4
5	000*	12
6	001*	7
7	0***	0
8	1***	0

Figure 20 Illustration of the measurement framework in Commodity-switch [54]

4.2.4 PayLess

Payless [50], like OpenTM, is a query-based monitoring framework which provides a RESTful API for flow statistics collection at different level of aggregations. The PayLess monitoring framework seats on top of an OpenFlow controller's Northbound API By providing a set of well-defined API, it lets different applications perform monitoring task without worrying the underlying technology. Figure 21 illustrates the Payless scenario, where different components work together to seamlessly implement user applications in the network. For example, the job of the interpreter is to translate high-level primitives expressed by the applications to flow-level primitives, while the Scheduler determines which type of statistics to poll, based on the nature of the request received from an application.

PayLess provides a high-level RESTful API, which can be accessed by any programming language. Therefore, it is very easy for different network applications to develop their own monitoring applications and accesses the collected data from the PayLess data stored at different aggregation levels.

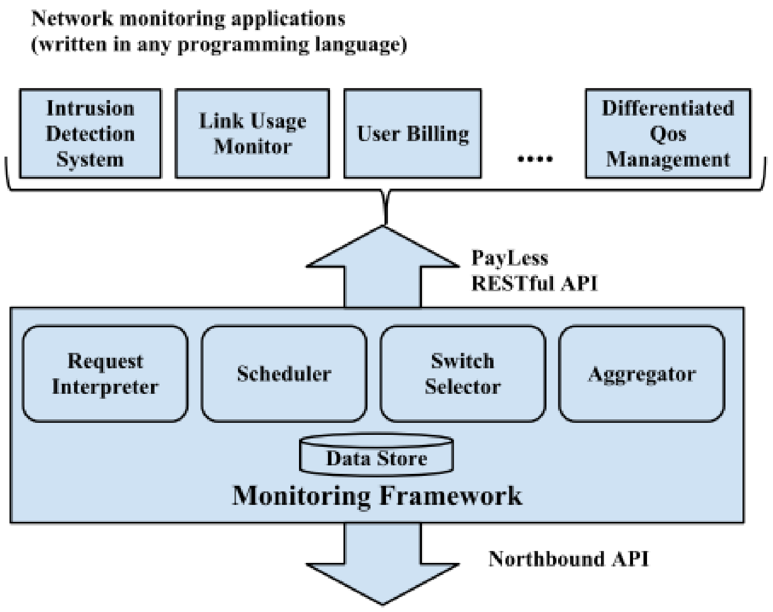


Figure 21: PayLess Network Monitoring Framework [50]

Even though PayLess provides an abstract for other applications to perform monitoring operation on the network, since the framework is based on periodic query of the switches, there are still tradeoffs to be considered. Figure 22 shows the relation between overhead and measurement error. The result shows that monitoring data is very accurate for short querying intervals, but achieving this accuracy results in extremely high message overhead, and by increasing the polling period the accuracy decreases considerably.

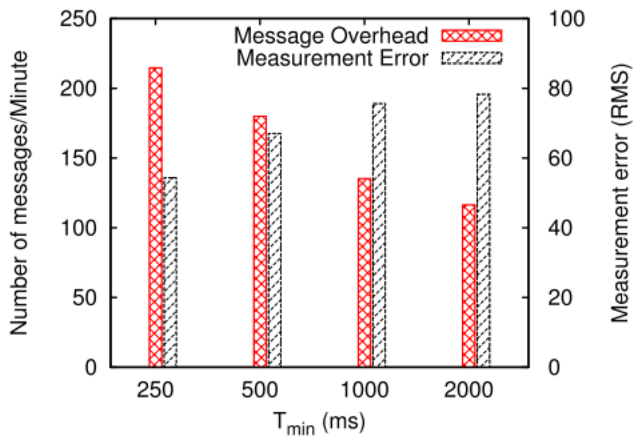


Figure 22: Overhead and measurement error [50]

Figure 23 shows the actual link utilization in a testbed. The accuracy of the result in different time intervals is shown: the longer the interval time, the lower the accuracy. This inaccuracy is due to flows with short lifespan. Since upon expiration of the flow entry in the switch their associated statistics are also removed, Payless cannot incorporate these statistics in the final result, resulting in considerable gap between the actual and the estimated traffic volumes.

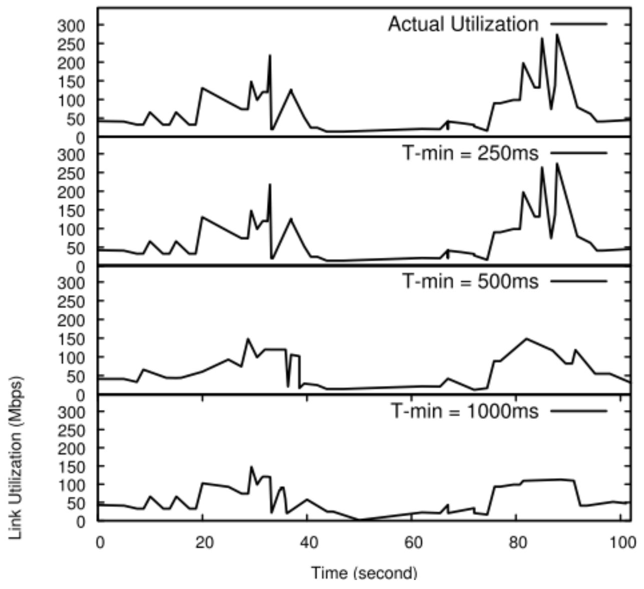


Figure 23: Effect of polling time on Accuracy [50]

4.2.5 Distributed TM measurement

In Distributed TM measurement [43], each switch calculates a portion of traffic matrix and sends it to the controller. The controller therefore uses the information received from the switch to construct the TM. Distributed TM achieves it by using a set of global counters defined in each switch. Specifically, each switch has a global counter (bytes or packets per measurement interval) for each one of the egress points of the network. In this sense, each ingress switch is able to compute one row of the TM, since it is monitoring all the flows that ingress at that switch and egress at any of the other N-1 switches.

During the arrival of new flows to the controller, as packet-out messages are constructed, each flow is assigned to a certain global counter, and this information is told to the ingress switch. This is done by the controller, since it has the complete view of the network and therefore is the only one that can assign each flow with the same egress point to a unique global counter. The basic concept behind Distributed TM measurement is shown in Figure 24. The controller synchronizes the switches with a new OF message, setting the interval where each switch has to send its data to the controller.

The main advantages using Distributed TM measurement is that the CPU load of the controller is minimized, since each switch performs a part of the measurement task. Also the amount of messages needed to construct the TM is reduced since the controller does not query all the switches - each switch sends its data to the controller according to the timer set by the controller. One minor problem with this approach is that in order to synchronize all the switches in the network either a periodical resynchronization message or NTP (Network Time Protocol) is needed. Finally, both the Openflow 1.0 messages and the switch functionality had to be modified, which made the approach interesting but difficult to implement in commercial equipment.

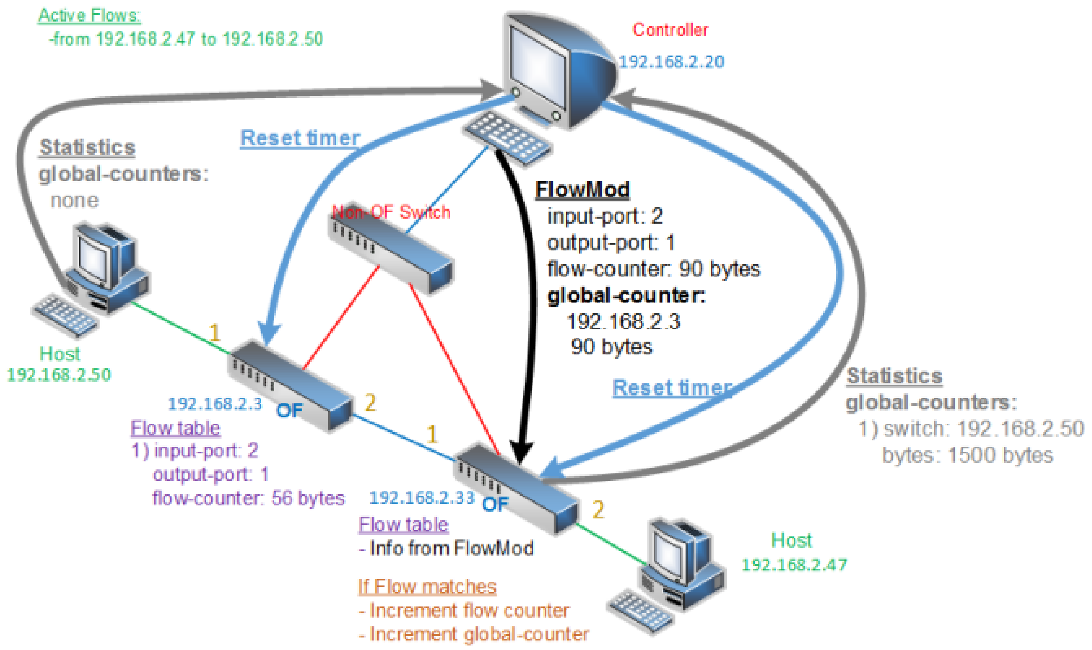


Figure 24: Distributed measurement of the Traffic Matrix model [43]

4.3 Proposed solution

Even though there are already methods to measure TM in OpenFlow networks, they either sacrifice accuracy for cost or vice versa. Distributed TM tried to tackle some of these problems, but to implement it there are modifications needed on both OpenFlow protocol, controller and switches, hence make it incompatible with current implementations of Openflow.

Our goal is to implement Distributed TM using the existing features in OpenFlow protocol without any modification neither to the protocol nor the switch. After throw investigation on OpenFlow features, with the introduction of OpenFlow 1.3 and 1.4, several new features were found that can be exploited to obtain TM with minimal cost. They are described as follows:

4.3.1 Meter

Meter was introduced in OpenFlow 1.3 to enable the controller to measure and rate traffic and implement various simple QoS operations. Meters are directly attached to flow entries, are identified by a meter ID, and there can be multiple meters in the same table as long as they are associated to different sets of flow (for using the same set of flows associated to multiple meters, they should be in successive flow tables). Meters consist of three main components:

- **Meter identifier:** It is a 32 bit unsigned integer which is used to identify the meter. The controller can use the meter ID to properly assign a set of flows with same ingress and egress to a unique meter.

- **Meter bands:** Each meter may have one or more meter bands. Each band specifies the rate and the way packets should be processed. If the current rate measured by the meter is lower than any specified meter band rate, no meter band is applied. They can be useful in TM measurement in order to determine if any packet shaping has been applied to a set of flows.

- **Counters:** updated when packets are processed by a meter. They are the most important feature of meters, and can be used to obtain statistics of aggregates of flows such as the amount of packets or bytes that have been processed by the switch. They record the duration of the meter since being active, and therefore can be used to obtain the exact amount of traffic being processed during a period of time. Table 3 shows the complete list of counters available in the meter tables.

Per Meter		
Flow Count	32	*Optional*
Input Packet Count	64	*Optional*
Input Byte Count	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Meter Band		
In Band Packet Count	64	*Optional*
In Band Byte Count	64	*Optional*

Table 3: Meter Table counters

After learning of the existence of the meter feature in OpenFlow, the whole solution has been based on it, but it should be noted that the meter features are optional in OpenFlow specification and all the switches that implement OpenFlow might not necessary have meter features. For example OVS (Open Virtual Switch) [55] which is one of the most complete software based switch solutions that supports Openflow 1.3 does not supports meters.

4.3.2 Bundle Messages

Bundles are set of actions that are to be performed together. They can be prepared and pre-validated on each switch, and be applied at the same time. They are introduced in OpenFlow 1.4, with two main purposes: the first one is to apply a set of changes on a single switch in a way that all changes are applied together or none of them is applied, and the second one, and perhaps the most important one towards our goal [32], is to better synchronize changes across a set of OpenFlow switches. This second feature can be used to add meters and receive meters stats from all the switches at the same time, overcoming the synchronization issues.

4.3.3 Auxiliary Connections

Prior to Openflow 1.3 the channel between the switch and the controller was exclusively made out of a single TCP connection. With the introduction of auxiliary connections, switches are able to have multiple parallel Openflow channels to the controller. Since they have parallel connection, the chance of congestion is reduced, therefore overall switch processing performance is improved and eliminate any concern regarding the extra packets created by the TM operation.

4.3.4 Multiple Controller

The Multiple Controller scenario has already been discussed in Section 2.5.4. It can be used to assign separate controllers solely in charge of collecting meter stats from the switches, which will reduce the load on the main controller. Also, in comparison to NetFlow, this will enable us to combine both Collector and Analyzer functions into a single machine. In terms of inter-domain networks this feature can be used to assign a neutral controller to both networks to obtain edge router information without involving any unnecessary security risk.

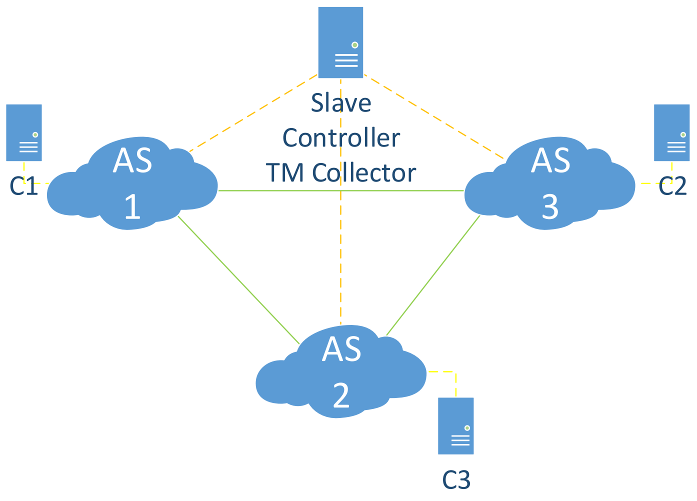


Figure 25: Multiple Controller Setup

Figure 25 shows three connected AS (autonomous system), where each AS is managed by their respective controllers, together with a neutral controller which can collect all the meters stats from each AS and construct the TM, making essentially the combination of collector and analyzer in NetFlow setup.

4.4 Design of the solution

Traffic measurements in SDN, as discussed in Section 4.2, are mostly focused on query-based schemes, meaning that, according to different algorithms, selected switches are queried to obtain their flow statistics during a time interval. These tools are using the generalized network view and routing information provided by the controller to aggregate flow statistics and construct the TM. There are tradeoffs in query-based scheme; to achieve high accuracy the time interval should be small resulting in high overhead in the switch. Requesting statistics for individual flows or even the aggregate of flows create large amount of extra traffic between the controller and the switch and directly affects the performance of switches, since they have to aggregate these statistics on the fly and send them to the controller.

DTM (Distributed Traffic Measurement) is based on the distribution of measurement tasks between all the ingress nodes in the network. The meter feature of Openflow 1.3 is used to assign a meter in each ingress node to keep track of all the flows that exit towards each one of the egress nodes in the network. Using the routing information available to the controller or through user defined policies, DTM will create aggregate flow rules with meter instructions and dynamically updated OpenFlow switches. Each time a packet passes through the ingress node that matches these rules, the meter corresponding to that rule is incremented. Then a separate monitoring application periodically pulls meters status from all the edge nodes to construct the TM table. Each node calculates a row of traffic matrix. By using this approach we have been able to reduce the overall overhead and cost of TM operations.

Assigning individual flows to a Meter identifier is not a feasible solution due to the limited number of rules that switches can store in their memory. Modern router and switches can handle between 64k to 512k entries in their memory [50]. According to [56, 57] the number of flows from a network of 5.000 to 8.000 hosts is around 1.000 to 10.000 distinct flows at any given time. Let’s suppose there are four ingress-egress (IE) nodes in the network ($4^2 = 16$ IE pairs). If for each unique flow we create a rule that points to a meter, up to 40.000 extra flow entries are needed to monitor these flows, which is simply not feasible. Our solution uses dynamic unfolding to aggregate all the flows with the same IE into a single rule. This way the 40,000 extra rules will be reduced to total of 16 rules for all the IE pairs.

To compute the number of meter identifiers that each ingress node needs to have in order to measure the the TM at the node or PoP level, J, if we assume that N is the number of nodes, and since the number of egress points is $N - 1$, we will have:

$$J = (N - 1) + 1 = N, \qquad (1)$$

where the extra meter is used to measure the traffic that ingress and egress in the same node (diagonal entries in TM table, corresponding to traffic that is exchanged, for example, by two LANs connected to the same node or PoP, and that does not enter the core network). At the node or PoP level each node will calculate a row of traffic matrix. Figure 26 shows a simple scenario where for an ingress point in a network, there are two egress points. To measure the aggregate of all the traffic in the network we only need three extra flow entries with meter instruction, each defining a pair of ingress-egress.

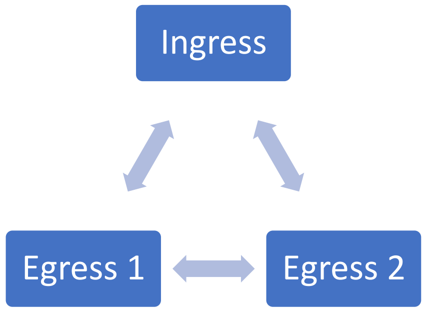


Figure 26: Network of three nodes, with one ingress and two egress points.

To measure the TM at the interface level, a different approach needs to be taken. Since the meters are inserted at each switch, the number of interfaces available in them becomes a factor in defining meters. As described in section 2.5.2, during the initial connection switches will advertise their capabilities to the controller. Therefore the controller is aware of the number of interfaces in each switch. To compute the number of meter identifiers that each ingress node needs to have in order to measure the traffic matrix at the interface level, J, if we assume that L_t is the total number of interfaces in the network (the TM will be a matrix of dimension $(L_t * L_t)$, and L_i is the number of edge interfaces connected to the node i, we will have:

$$J = (L_i * (L_t - L_i)) + L_i * (L_i - 1) = L_i L_t - L_i^2 + L_i^2 - L_i = L_i L_t - L_i = L_i(L_t - 1), \qquad (2)$$

where $(L_i * (L_t - L_i))$ is to calculate the number of meters for IE pairs going outbound and $(L_i * (L_i - 1))$ is to calculate number of meters for IE pairs going inbound (i.e. the traffic between the interfaces connected to the same node). Opposite to the case where the TM is measured at the node level, depending on the number of edge interfaces each node will calculate more than one row of traffic.

Figure 27 shows a simple scenario where the TM is measured at interface level and each switch has a different number of edge interfaces. Depending on the number of edge interfaces, each switch measure a different number of rows in the table, as shown in Table 4. According to (2) for S1 we need 24 meter IDs $(3 * (9 - 1))$, for S2 we need 16 meter IDs $(2 * (9 - 1))$ and for S3 we need 32 meter IDs $(4 * (9 - 1))$. There is no inbound traffic in an interface connected to a node. Therefore to reduce the number of meters we have omitted the meters IDs responsible for diagonal entries in the TM table, which in this case will be always zero – hosts do not generate traffic towards themselves through the switches (they can do it through the localhost interface).

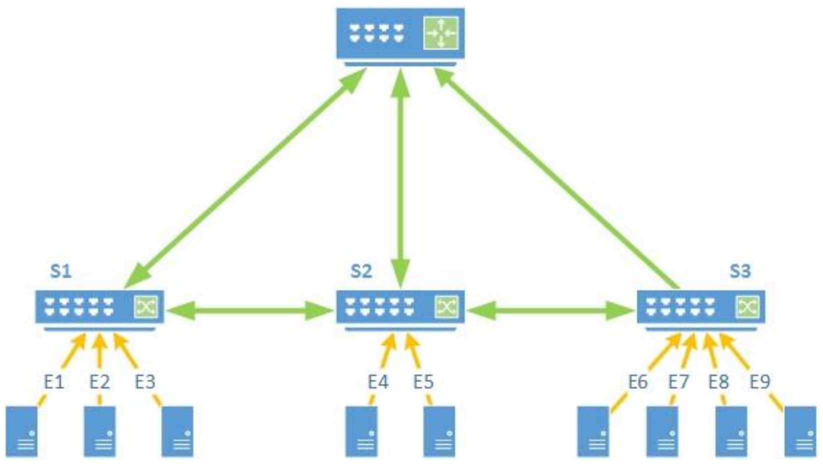


Figure 27: Example of TM measurement at interface level

TM		E1	E2	E3	E4	E5	E6	E7	E8	E9
S 1	E1	0								
	E2		0							
	E3			0						
S 2	E4				0					
	E5					0				
S 3	E6						0			
	E7							0		
	E8								0	
	E9									0

Table 4: Traffic Matrix Table for 9 edge interfaces

Using DTM we have been able to maintain 100% accuracy while keeping the overhead to minimum. Querying the switches may result in more packets exchanging between controller and the switches, since all the switches are involved in traffic measurement, but the message size are considerably smaller than with the query-based scheme. Also, increasing the query time interval will not affect the accuracy of the final result, since the rate of all flows (even the ones with short lifespan) are collected by the meter. The most important part is that the task for traffic measurement is equally divided between all the edge nodes, reducing the effect of TM measurement on the network to minimum.

4.5 Implementation details

Figure 29 shows a simple topology where there are three edge routers and one controller. Given two routers A and B, the traffic between these edge routers T is defined as the total amount of traffic that is entering the network A and exits at network B. Since the controller has the complete view of the network topology and routing information it can identify each pair of ingress and egress in the network. The controller will use the switches' Datapath ID to assign meter ID and insert meters in each switch, each defining a pair of ingress-egress.

There are different ways of adding flow entries with the meter instruction:

- It can be tightly integrated with the main program in charge of forwarding decisions so that with any new flow arriving to the controller that meets the condition of TM, the meter instruction can be added directly to the Packet-out message carrying flow entries together with the output action. Even though this method eliminates any extra rule needed to measure the traffic, in reality it is very complex and network-dependent. For each network the application has to be written from scratch, which is not feasible and scalable, therefore it is not recommended.

- The flow entries with meter instruction can be added by a separate application on the controller without any "apply action". For example, Figure 28 shows a multiple setup in OpenFlow switch, where a table can be reserved only for flow entries with meter instruction. A separate application can add these flow entries with a "goto action" (goto actions specifies the next table in pipeline processing in OpenFlow) where meters will be applied to flows that match the conditions without any interruption of the operation of the action set in the pipeline. Using this approach may result in extra rules in the switch flow table, but aggregating the flows with same IE will significantly reduce the number of rules needed to measure traffic. For example, in a network with N nodes and M flows with the same ingress-egress points, assuming we can aggregate M by a factor of $1/K$, where K is the aggregation level, then only $N * (M/K)$ extra entries are added in the switch flow table.

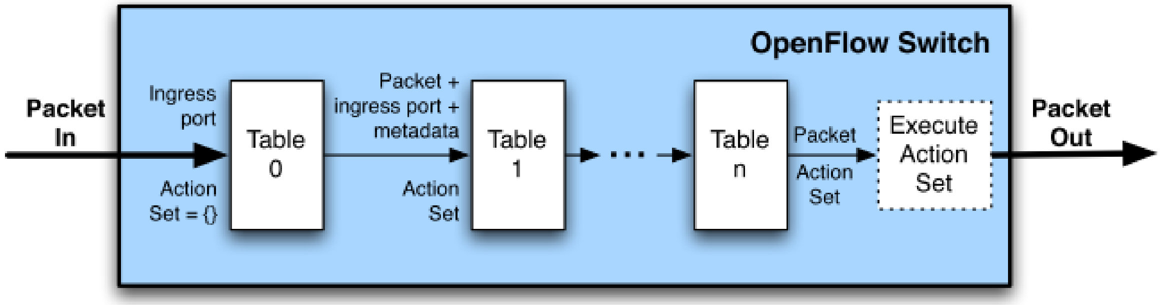


Figure 28: Multiple table in OpenFlow [32]

For example, in Figure 29 the controller has assigned three meters to OFS 1 (OpenFlow Switch with Datapath ID 1). Meter 11 defines the inbound traffic generated from hosts connected to OFS1. Meter 12 defines traffic ingress the OFS1 and egress the OFS2. Meter 13 defines traffic ingress the OFS1 and egress OFS3. Therefore each switch will calculate a row of traffic matrix.

Sometimes, knowing the total traffic between each pair is not enough. Furthermore, in order to improve traffic monitoring, we can extend meter assignments to different types of flows for each pair of ingress and egress using the match fields in flow entries. For example, TCP port 80 is assigned to HTTP traffic [58], and the controller can add a new meter 1280 and assign all the flows with source IP address 10.10.10.0/24, destination IP address 10.10.20.0/24 and destination TCP port 80 to that meter. Recall that the meter ID is a 32 bit unsigned integer and to simplify the explanation we have used the above numbering scheme.

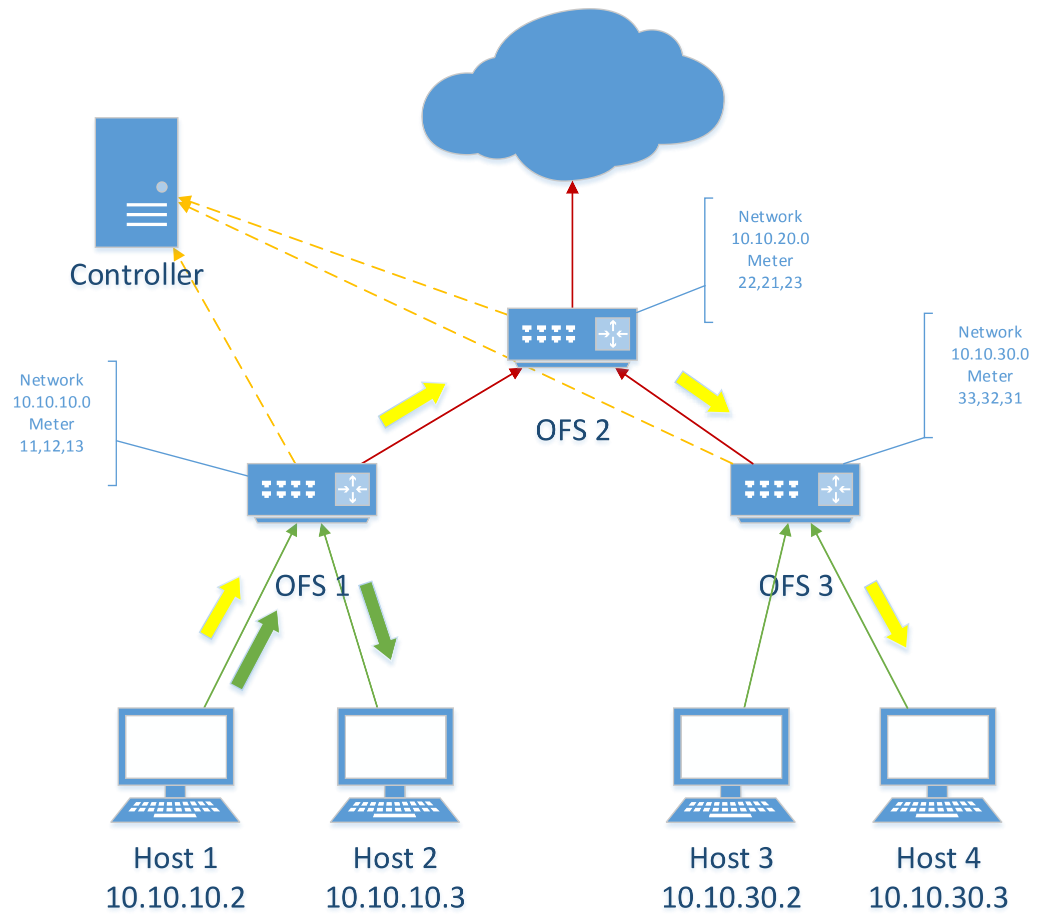


Figure 29: Simple Topology Setup

Furthermore in the case of Data Centers (DC), where being able to scale with demand is a necessity, TM measurement also needs to be elastic. Within our framework, upon the addition of new switches, the controller can dynamically update all the edge switches to incorporate the new changes.

4.6 Analysis

Our framework brings new possibilities in terms of TM measurement. The measurement of traffic has been evenly distributed between all the edge switches, hence reducing load on the controller and the edge switches and minimizing cost while maintaining the 100% accuracy of TM. Since meters will measure the aggregate of all the flow with same ingress and egress, the amount of information sent to the controller during query process is small, reducing the network load between controller and the switches.

There are few limitations to our approach. Depending on different types of monitoring needs, if the meter integration is made by a separate application, extra flow entries are needed to measure the rate of traffic. The addition of extra flow entries may reduce processing power of the switch, because the TCAM size is constrained. Modern switches and routers can support in the range of 64k to 512k entries as mentioned in section 4.4. Assuming DTM can aggregate all the flows with the same origin and destination into a single rule: if N is the total number of edge nodes and q is the total number of measurements required by the monitoring system (like aggregate of all the flow and HTTP traffic from an ingress node towards all the egress nodes), the total amount of flow entries with meter instructions M in each ingress node needed to calculate the TM is:

$$M = (N * q). \tag{3}$$

For example, let's assume a large data center, where there are total of 1000 edge node ($N = 1000$) and the operator needs to know both aggregate of all the traffic between each IE pair and monitor 3 different types of traffic (HTTP, HTTPS, TFTP) for each IE pair. In order to deploy DTM, we need to handle more than 4.000 (($N * q$), where q = 4 and N = 1000) flow entry rules in each switch to measure the traffic. The DTM process consumes less than 6.25% of total flow entries a modern switch can accommodate at the minimum. In most cases, this amount does not affect the performance of the switch nor is limiting the capabilities of the network, but is still something that needs to be taken into consideration.

Other than concerns regarding memory usage in the switches, since every edge node is queried during the measurement process, in case of network with large number of ingress egress nodes, the number of packets created may significantly limit the controller performance. In regard to previous example, for each query interval more than 2.000 messages (1000 ingress node = 1000 request message + 1000 reply messages) are exchanged between the controller and the switches. This problem can be solved using extra controller in charge of the query process and auxiliary connection between the switches and the controller as described in section 4.3.3 and 4.3.4, but to do so, additional investments are needed and may increase the total cost of the operation.

Figure 30 shows the relation between the numbers of flow entries needed in each ingress node to the total number of egress depending on the number of measurement requirement by the operator according to (3). In addition, messages generated during each query process are shown by the dotted line. Since most of the networks never exceed 100 pair of ingress-egress (we are assuming here that we are measuring the TM at the node or PoP level of aggregation), the number of messages between each query process is at an acceptable level and it stays constant regardless of number of meters in the switch. But the most concerning part may arise from the number of entries which increases linearly. Assuming we can combine every level of aggregation into a single rule, as the measurement requirement increases, the number of flow entries needed to be written in the switch may become a factor in switch overall performance and capabilities.

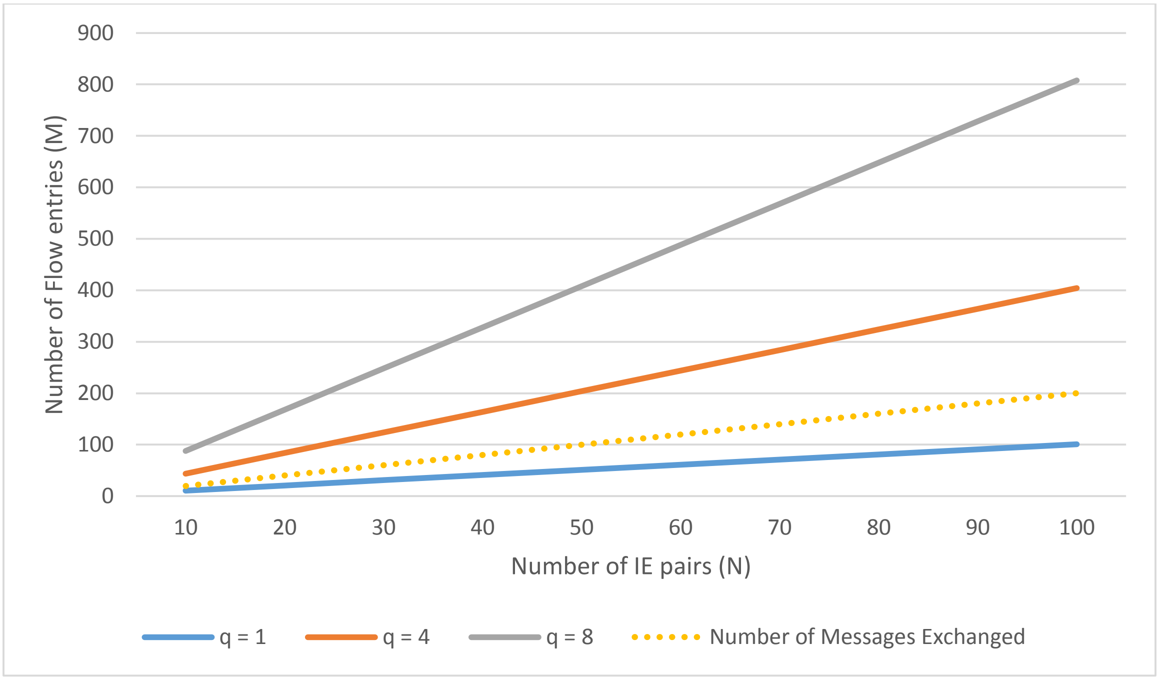


Figure 30 : Relation Between Number of Flow entries, IEs and Types of Measurments

4.7 Comparison with previous works

There are significant differences between DTM and related works regarding traffic measurement in SDN. Even though DTM uses built-in features in OpenFlow, these features are marked as optional, meaning OpenFlow switches are not mandated to support them (i.e., meter). DTM reduces the size of the reporting packets (meter counters are smaller than individual flow counters) during query preprocess but since all the edge switches are involved in monitoring process, the number of packets generated during each query interval is greater than with other methods. Finally, in order to deploy DTM, OpenFlow switches need to be reconfigured and flow entries with meter instructions need to be written in each and every edge switch in the network.

For large networks with many concurrent applications deploying DTM may become too complex[7] to justify its benefits.

In comparison to OpenTM (Section 4.2.1), since each edge switch aggregates flow statistics during the query process, the network load and packet overhead of DTM are smaller. Furthermore, there is no extra processing power needed in the controller to aggregate the flow stats to build the TM, resulting in lower load at the controller.

In comparison to FlowSense (Section 4.2.2), even though DTM includes extra processes like configuring and querying the switches, the limitations in FlowSense have been eliminated. Our approach has a more accurate result while keeping the overhead to minimum unlike FlowSense, where TM measurement is bound to control messages received by the controller (which can result in inaccuracy due to operators installing predetermined decision rules, thus avoiding the Flow Mod messages getting to the controller). Also processing power needed in FlowSense is much greater compare to our approach since it is continuously monitoring packet-in and flow-removed messages and cannot work without the main controller, therefore it is continuously using resources of the controller.

In comparison to Commodity-solution (Section 4.2.3), TM measurement in our approach is more accurate since TM contains all the traffic between each pair rather than only the “heavy hitter” traffic (flows that represent major traffic of the network). The only limitation is that DTM generates more CPU overhead at the switches. The amount of extra messages sent to the controller from the switch is negligible comparing to Commodity-solution (since the latter constantly changes rules in the switches to detect heavy hitters).

In comparison to PayLess (Section 4.2.4), the number of packets generated by DTM during each query interval are greater than PayLess. Regarding accuracy, in Payless the accuracy decreases with the increase in the time between each interval. Our approach can maintain 100% of accuracy regardless of the querying interval. Also the size of the packets and load on the switches and controller are considerably lower than in PayLess, since the load is distributed evenly between all the edge switches and the packet size from meter statistics is relatively smaller comparing to individual or aggregate flow statistics required by Payless.

Our work was built upon Distributed TM measurement (Section 4.2.5), hence it follows the same concept with some enhancement. Global counters have been replaced by meters (which are built-in functions in OpenFlow), there is no more the need for synchronization in the switch since each meter carries the duration of the times it has been alive[8], and additionally multiple controller mechanism can be deployed to further reduce the load on the controller.

[7] Other approaches like OpenTM and PayLess do not need to modify switches flow tables. On the other hand in order to deploy DTM we need to modify flow tables to insert flow rules that point to a specific meter.
[8] Each meter maintains the time it has been active since its insertion by the controller. The controller is responsible to query all the switches in the network. Therefore, there is no more need for synchronization.

CHAPTER 5: DEPLOYMENT, TESTING, AND EVALUATION

In this chapter we will present the approaches taken in order to deploy DTM in a simple scenario where OpenFlow is used. Our goal is to show that measuring traffic in OpenFlow enabled networks is possible using the meter features added in OpenFlow version 1.3. We will explain the ways we tried to deploy the framework, the choice of software and hardware, different approaches towards deploying the framework and final testbed deployments, and the results obtained.

5.1 Choice of Controller

At the beginning of this work, OpenFlow 1.4 was just released and adoption of OpenFlow 1.3 was practically inexistent. Even as of today (July 2014) there are few switch implementations that support OpenFlow 1.3. Until few months ago the only controller that supported OpenFlow 1.3 was Ryu, hence this work has been developed based on the Ryu Controller.

5.1.2 Ryu Controller

According to [59], Ryu is a component-based software defined networking framework. Ryu provides software components with well-defined API that makes it easy for developers to create new network management and control applications. Ryu supports various protocols for managing network devices, such as OpenFlow, Netconf, OF-config, etc. Regarding OpenFlow, Ryu supports fully 1.0, 1.2, 1.3, 1.4 and Nicira Extensions. All of the code is freely available under the Apache 2.0 license.

Ryu is written fully in Python, and therefore it is easy to develop for. It comes with various applications already written to deploy SDN network, like Spanning Tree manager, basic switch application for creating forwarding rules, firewall and routing application. The Ryu architecture is shown in Figure 31. It provides a rich set of APIs allowing developers to create specific application for managing different aspects of a network according to the operator requirement. It can support legacy networks along with SDN implementations, and recently a huge collection of documentation and references has been added that can be used to create specific application. For more details about the operation of Ryu refer to Appendix B1.

The monitoring application created by us in order to collect meter statistics from all the switches in the network uses Ryu API's. Ryu can support multiple controller setup to support fail over and load distribution between multiple controllers. It uses Zookeeper[9] to achieve multiple controller support, as shown in Figure 32.

[9] ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services, developed by Apache [51].

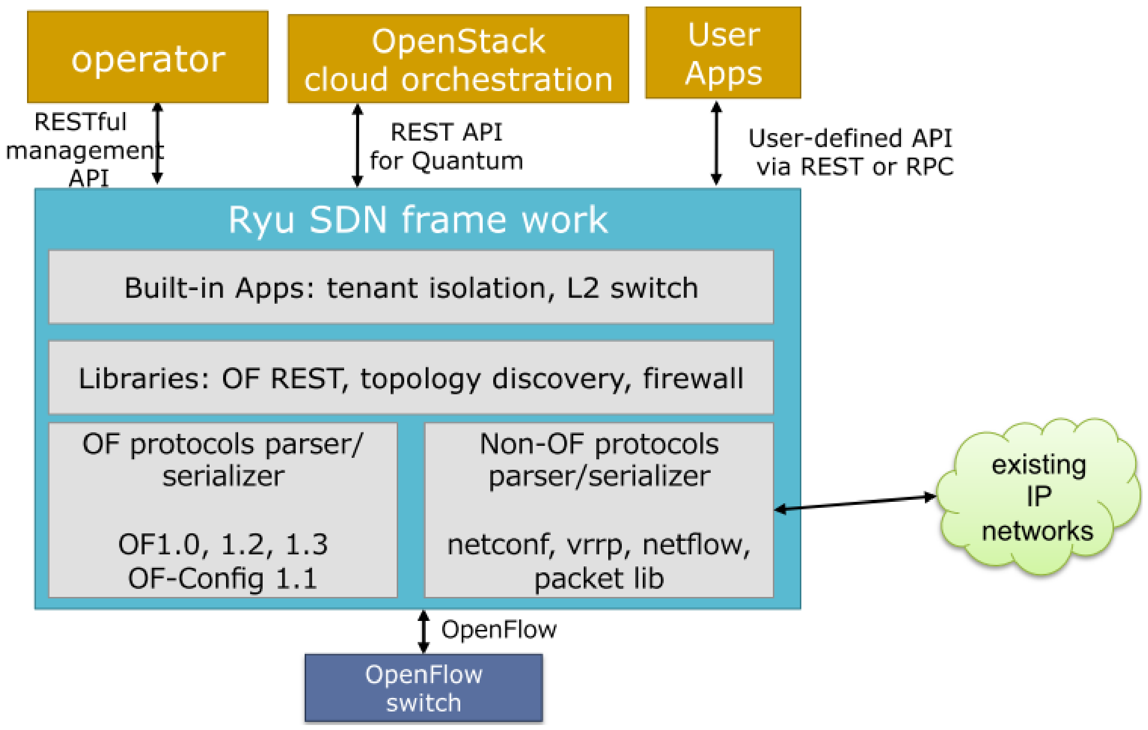

operator
OpenStack cloud orchestration
User Apps
RESTful management API
REST API for Quantum
User-defined API via REST or RPC
Ryu SDN frame work
Built-in Apps: tenant isolation, L2 switch
Libraries: OF REST, topology discovery, firewall
OF protocols parser/ serializer
OF1.0, 1.2, 1.3 OF-Config 1.1
Non-OF protocols parser/serializer
netconf, vrrp, netflow, packet lib
existing IP networks
OpenFlow
OpenFlow switch

Figure 31: Ryu Architecture [59]

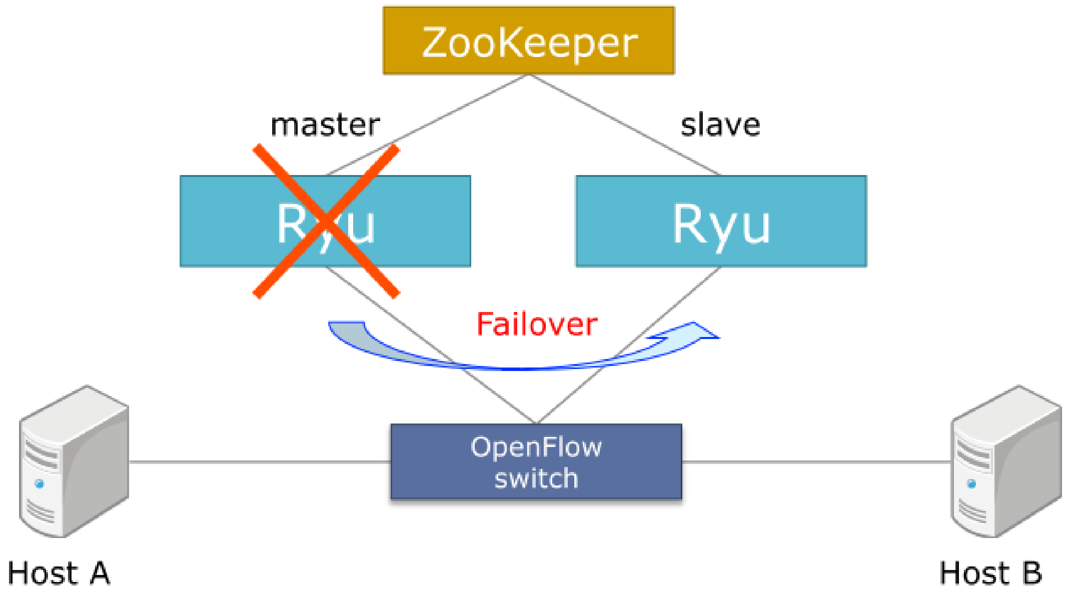


Figure 32: Scenario with multiple Controller in Ryu [59]

5.2 Choice of OpenFlow Switch

In terms of switches which support version 1.3 of OpenFlow the choices are very limited. There is a number of solutions as in software switches that can run either in a PC or that can be ported to real low end devices and higher end devices that support Openflow 1.3. It is also possible to simulate network using network simulators like Mininet.

5.2.1 Mininet

Mininet [60] is a network emulation platform that intends to be the quickest and easiest way to create virtual networks for research, education and system development. By providing an instant virtual network on your laptop that can run real application, switch, and controller code, Mininet makes it easy to get started with Software-Defined Networking and OpenFlow without any additional hardware, while still developing real systems that can be readily deployed on hardware when it is available. Additionally, hardware networks can be replicated in Mininet for development, experimentation or demonstration.

5.2.2 Software Switch

At the beginning of this work there were only two software switches that could support OpenFlow 1.3. Using software switches in conjunction with Mininet can produce a fair estimation of network behavior during research. Two software switches that were used in this work are the following:

- **CPqD Software switch [61]:** is one of the few software switches that supports OpenFlow 1.3 specification. It is built upon Stanford OpenFlow 1.0 reference switch and Ericsson's Traffic Lab OpenFlow 1.1 switch and is intended for fast experimentation purposes. It supports most of OpenFlow 1.3 features and most importantly the meter features. Unfortunately there are a few bugs existing in current CPqD software switch, like not recognizing IP bit masks.

- **Open Virtual Switch [62]:** is one of the most complete and features rich solutions. It can operate both as a soft switch running within the hypervisor and as the control stack for switching silicon. It has been ported to multiple virtualization platforms and switching chipsets. Like CPqD, it supports OpenFlow 1.3 but, unfortunately, even though we can add meters in the switch, adding flow entries with meter instructions are not supported yet by Open Virtual switch.

The developers of Ryu continuously test various OpenFlow version 1.3 switches. In order to see the current number of switch solutions and what kind of action and matches work (or not) please refer to [63].

5.2.3 High End Devices

Some high end devices can support OpenFlow version 1.3. Even though these set of switches are ideal to deploy our test-bed, because of the cost and unavailability we have not been able to use them in our project. Some examples are:

- **Centec V350 [64]:** V350 Series Switch is not only a cost effective high performance switch for SDN/Openflow applications but also, more important, is a complete open SDN platform. Centec provides a complete turnkey solution from switching silicon to standard productized hardware platform as well as system level software integrated with open SDK to help SDN vendors to shorten their time to market and reduce development cost. The openness of the V350 platform enables further customization and differentiation which is critical for SDN vendors to compete against other vendors. Figure 33 shows Centec V350 switch, some of its features are as follow:

    - Supports OpenFlow specification 1.3.1
    - Supports up to 2K TCAM embedded flows with complete match field and stats
    - Supports 32K 12-tuple flows and up to 64K hash-based exact match flows
    - Supports L2 to L4 complete matching fields

Figure 33: Centec V350 OpenFlow Switch [64]

- **HP 2920 Switch Series [65]:** HP is one of the leading companies in terms of supporting SDN and OpenFlow. It has many switches that can support OpenFlow 1.3 and even has its own proprietary OpenFlow 1.3 Controller. Figure 34 shows HP 2920 series switches, some of its features are:

    - supports OpenFlow 1.0 and 1.3 specifications to enable SDN by allowing separation of the data (packet forwarding) and control (routing decision) paths;
    - provides complete support of SNMP; provides full support of industry-standard Management Information Base (MIB) plus private extensions; SNMPv3 supports increased security using encryption;
    - up to four optional 10-Gigabit ports (SFP+ and/or 10GBASE-T);
    - Optional two-port stacking module allows stacking of up to four switch units into a single virtual device.

Figure 34: HP 2920 Series [65]

5.2.4 Low End Devices

During this project there were two low-cost devices available to us: Cisco Linksys WRT54GL [66] shown in Figure 35 and Mikrotik’s RouterBoard 750GL [67] shown in Figure 36. Both devices have 5 LAN ports, but they run at different speeds (1 Gbit/s the Mikrotik, 100 Mbit/s the Linksys). WRT54GL has less RAM, CPU and storage capacity than 750GL.

In order to enable these devices to support OpenFlow 1.3, we used OpenWrt and software switch to build a complete image from source to be ported to these switches. OpenWrt [9] is a Linux distribution for embedded devices. Instead of trying to create a single, static firmware, OpenWrt provides a fully writable file system with package management. This frees you from the application selection and configuration provided by the vendor and allows you to customize the device through the use of packages to suit any application. For a developer, OpenWrt is the framework to build an application without having to build a complete firmware around it.

We were able to build a version of OpenWrt using CPqD OpenFlow 1.3 software, which can turn OpenWrt router into OpenFlow switches. RouterBoard Mikrotik 750GL was chosen to port the OpenWrt image because of its higher RAM and CPU power. For complete installation details refer to appendix C.

Figure 35: Liksys WRT54GL [66]

Figure 36: RouterBoard Mikrotik 750GL [67]

5.3 First experiment

For the first test we wanted to show the traffic measurement at interface level. In our scenario shown in Figure 37, Mininet is used to create a simple topology which consists of one switch and three connected hosts. The switch is based on CPqD software switch running on a PC. The goal is to measure traffic generated from each host towards other hosts in the network, therefore for each ingress point (host) there are two egress points. According to (2), we only need to add 6 meters ($3 * (3 - 3)$) to construct the TM table (measuring TM at the interface level).

We have chosen reactive instantiation as described in 2.5.3; therefore meters and flow entries with meter instruction were added manually using *dpctl*[10] tool. A total of 6 meters are used to measure the traffic in each interface. *Iperf*[11] and ping are used to generate traffic between each host in the network. In DTM we do not need to worry about the accuracy of the result, since

[10] Dpctl is a management utility that enable some control over the OpenFlow switch. With this tool it is possible to add flows to the flow table, query for switch features and status, and change other configurations.

[11] Iperf tool is used. Iperf is a commonly used network testing tool that can create Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data streams and measure the throughput of a network that is carrying them [80].

meters constantly measure the traffic generated from each flow and flow removal does not affect meters.

To construct the TM table, during each query interval the amount obtained from first query can be subtracted from the second query to calculate the traffic during that interval. For example *Iperf* was used to generate UDP traffic at a rate of 100 Mbit/s (Approximately 12.5 Mbytes of data per second) from host 1 towards host 2 and 3 for a duration of 100 seconds. Table 5 shows the actual traffic sent and traffic measured by the meters during 10 second query interval. For complete configuration procedure and test result refer to Appendix A1.

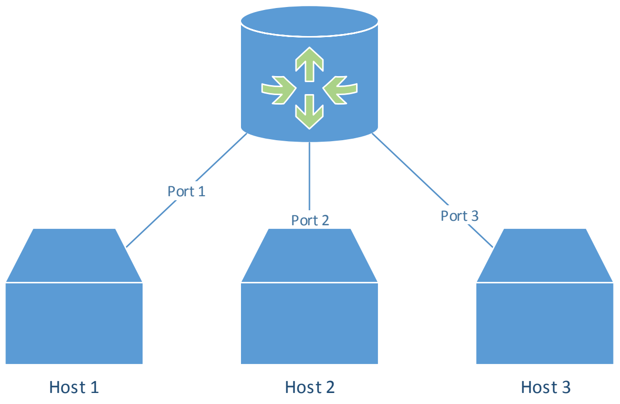


Figure 37: Switch with three connected hosts

Traffic Matrix	Host 1	Host 2	Host 3
Host 1	Actual Traffic : 0 Measured Traffic: 0	Actual Traffic : 124 MBytes Measured Traffic: 124 MBytes	Actual Traffic : 127 Mbytes Measured Traffic: 127 MBytes
Host 2	Actual Traffic : 0 Measured Traffic: 0	Actual Traffic : 0 Measured Traffic: 0	Actual Traffic : 0 Measured Traffic: 0
Host 3	Actual Traffic : 0 Measured Traffic: 0	Actual Traffic : 0 Measured Traffic: 0	Actual Traffic : 0 Measured Traffic: 0

Table 5: TM table during 10 second interval

To further test the switch and monitor performance we created series of UDP traffic at different rates between H1-H2 (10Mbps), H1-H3 (50Mbps), H3-H1 (34Mbps) and H2-H1 (60Mbps). Figure 38 shows the transferred rate between each host during a 100 second period where the switch were queried every 10 seconds. After comparison with the meter rate we conclude the 100% accuracy of our framework.

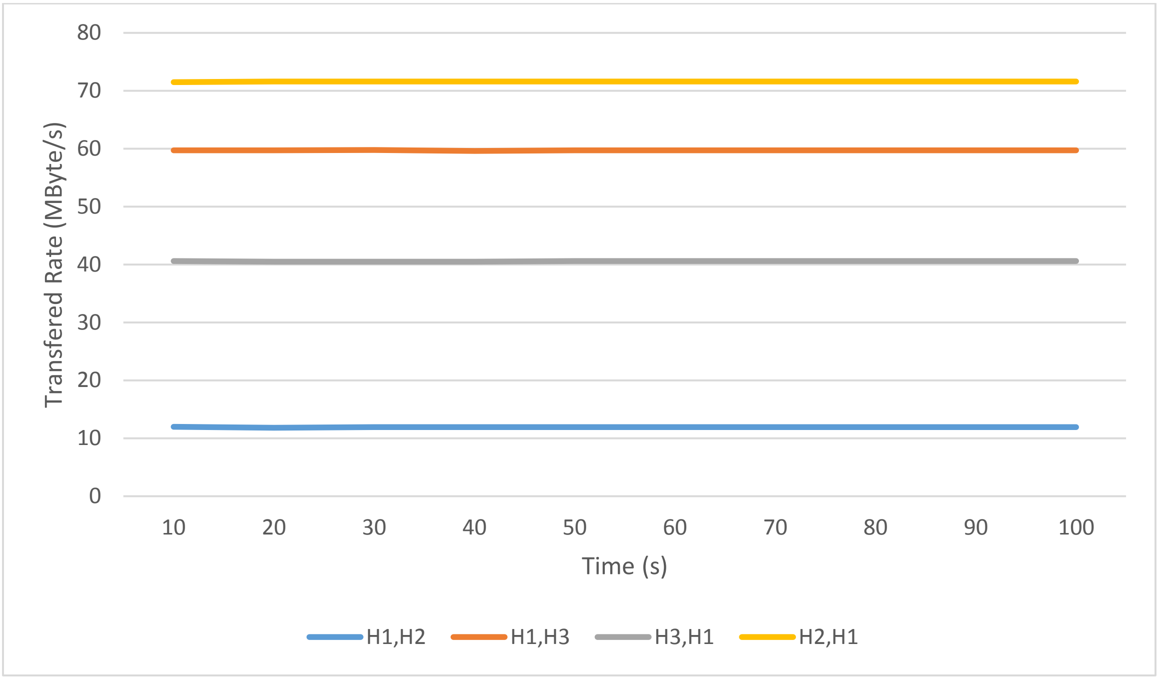


Figure 38: Transfer rate between all the hosts

In comparison to other monitoring tools described in section 4.2, regardless of query interval time, DTM will measure the traffic from any ingress point to any egress point with 100% accuracy. Figure 23 shows the effect of query interval time on measured traffic and its differences with actual traffic in PayLess monitoring tool (described in section 4.2.4). Evidently the monitoring data is very accurate for short interval query of 250ms, but it gradually degrades with higher value of query interval. The number of messages sent to all the switches can be reduced by increasing the interval time, since we do not need to worry about the accuracy of our operation.

5.4 Second experiment

For the second experiment, we wanted to test our framework in real networking equipment. The goal of this experiment is to measure extra traffic generated from querying process and the effect of the monitoring procedure on the performance of the network. The network is setup in reactive mode, meaning that once the rules are written in the switches incoming packets will not be sent to the controller. To measure packet size and network behavior we used latest version of Wireshark [68] which can monitor OpenFlow messages. The list of equipment and topology of the setup are explained in great detail in the next pages.

5.4.1 Components

The following hardware was used in our Traffic Matrix deployment:

- Three Mikrotik RB 750GL, running CPqD OpenFlow 1.3;
- Five Raspberry Pi acting as traffic generators;
- Laptop for managing connection to Raspberry Pi's and also as a sixth traffic generator;
- A hub connecting all the Mikrotik switches to the controller;
- Ubuntu PC machine acting as a Controller.

5.4.2 Hosts

For the Host emulation, five Raspberry Pi and a laptop computer are used. Iperf has been installed on all machines to help evaluating the network performance. The laptop is used to connect through secure shell (*ssh*) to Raspberry Pi's in order to perform network operations, check connectivity between hosts and to see the states of hosts and result of the Iperf. Since the controller is configured in "Out Of Band" mode, there is no access to hosts from the controller, i.e. connection between switches and controller are not part of the network connecting the hosts. For more detail installing Iperf on Raspberry Pi refer to Appendix A2.1

5.4.3 Switches

There are three Mikrotik 750n series acting as Openflow switches, accomplished porting OpenWRT framework build from source alongside CPqD/ofsoftswitch13 on top of it (for detailed installation procedure please refer to Appendix C). Due to a bug in CPqD/ofsoftswitch13 the switches cannot recognize subnet bit mask hence restricting us to use them as router or layer 3 switches. During start up switches will try to connect to the controller using IP address and TCP port manually configured by us. If connection to the controller is lost, switches will retry automatically every 10 seconds. For more detail about switch setup and samples refer to Appendix A 2.3.

5.4.4 Controller

As described in section 5.1.2, because of Ryu's early adoption of OpenFlow 1.3, we have written our monitoring application based on it. Ryu controller provides southbound APIs for developer to create application for SDN enabled networks. We have used some of Ryu's built-in application to configure switches forwarding rules and a separate script to add flow entries with meter instruction.

After initial connection between switches and controller, forwarding application will install table-miss entries in all the switches. During first packet-in message sent from switch to controller, the forwarding application will create a MAC address table and configure switches as layer 2 MAC learning. For configuration command refer to Appendix A2.4 and for how Switching Hub application works refer to Appendix B2.

5.4.5 Scenario

Figure 39 illustrates the topology and all the IP addresses associated with each switch and host. The yellow boxes next to each link shows the actual port the hosts are connected to and the logical port assigned to them by the OpenFlow switch. As aforementioned, CPqD does not support IP address match with bitmask such as ip_src=10.0.0.1/8 therefore assigning flow entries with IP characteristics causes the switch to crash, and therefore Ethernet addresses are used instead. In a proper OpenFlow 1.3 or 1.4 switch implementations flows can have different characteristics. For example only ARP traffic or flows from a specific range of IP address with a specific TCP or UDP port can be assigned to a meter.

In the topology, switches are connected linearly as shown by the red line. The goal is to measure all the traffic from each ingress point (all the connected hosts) to each egress point (other switches) and the inbound traffic flowing between hosts connected to the same switch. Each switch has two hosts connected to it, in order to generate inbound traffic and outbound traffic towards others switches.

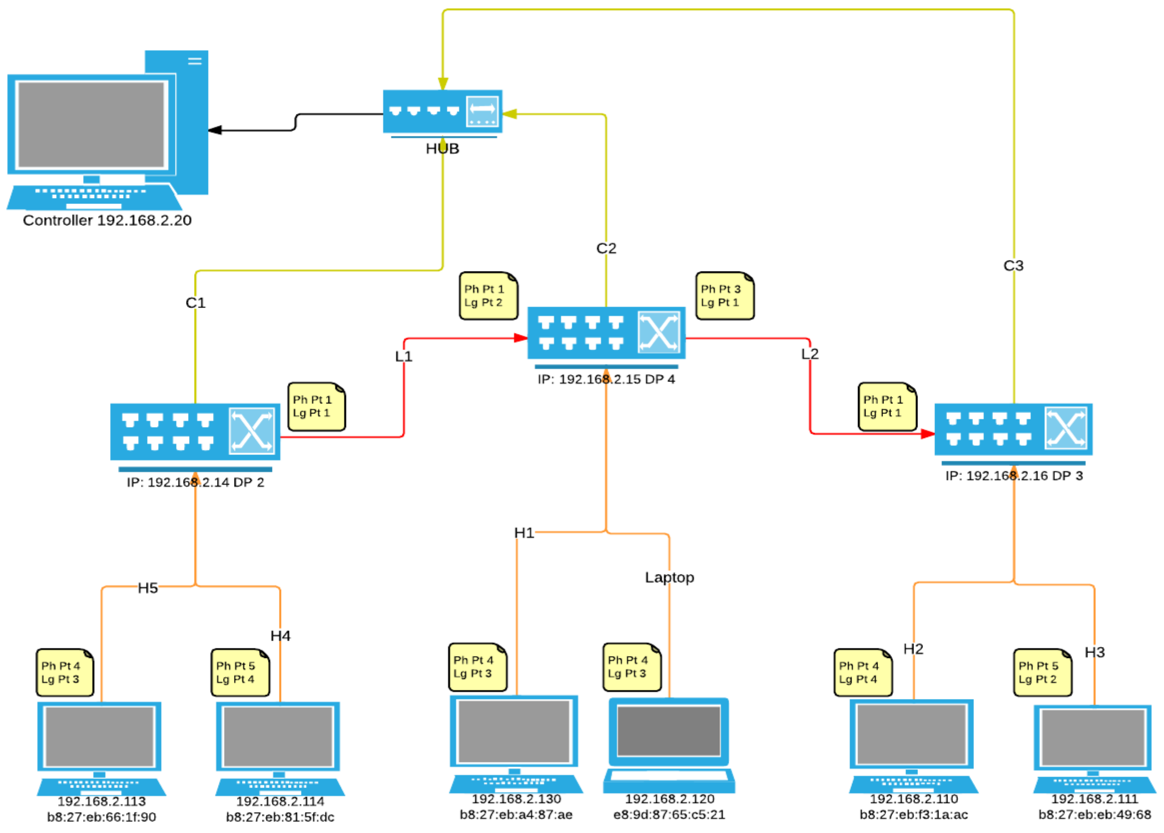


Figure 39: Testbed Topology

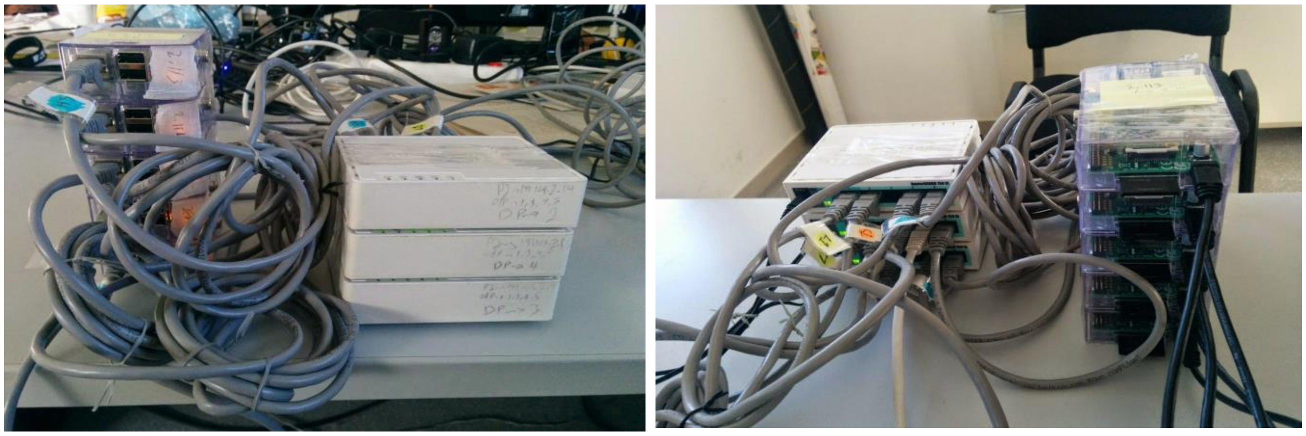


Figure 40: Testbed in the Lab

5.5.5 Testing network bandwidth capabilities with and without presence of Meter

For assessing the network performance, we ran series of tests using the Iperf tool, one without any meter instruction and one with meter instruction. To test the network performance we generated series of UDP and TCP traffic using Iperf and comparing the obtained results in both approaches. The network is not optimized since the performance of the meter is our center of focus rather than the network itself. Even though the Mikrotik routers are having 1 Gb/s Ethernet connection, Raspberry Pi's connected to them are limited only to 100 Mb/s. Also read/write speed of the memory card used in Raspberry Pi's is slow and, combined with weak CPU power, the expected bandwidth is not very high.

For the first test switches are configured as layer 2 MAC learning switches without any meter instructions. Iperf is used to measure the bandwidth capability of the hosts connected to the network. A total of three hosts were chosen to participate in the test, two closest hosts and two furthest away i.e. H2, H3 and H2, H5. Series of TCP and UDP traffic were generated towards each direction, to be sure test was repeated at least 10 times. The average throughput of the network from obtained test result for both directions (H2, H3 & H2, H5) was around 35 Mb/s with a jitter[12] of 0.150ms.

For the second test, meters and flow entries were added without modifying the configuration of the Mikrotik switches, meaning extra flow entries pointing to the meter IDs. The same procedure was repeated as for the first test. Predictably, we got the same average throughput of 35 Mb/s and Jitter of 0.150ms. Comparing both results shows that in small to medium scale DTM won't affect network performance since the number of extra flow entries is reasonably small. For complete configuration commands and sample of test result refer to Appendix A2.5.

[12] Jitter is the variation in the time between packets arriving, caused by network congestion, timing drift, or route changes.

5.6 How to obtain the Traffic Matrix, step by step

We now present in detail the series of operations needed to obtain the TM, from the time OpenFlow switches connected to the controller to the time Meter statistics are collected from the switches.

- Two apps run on top of Ryu controller. The first one (Simple_Switch13.py) is responsible for configuring all the switches to act like a learning layer 2 switches and insert table-miss entries during initial switch configuration.
- The second application (trafficmatrix.py) is responsible for collecting meters status from all the switches and displaying them in the terminal. Only meter ID, duration of the meter and total number of bytes that have been processed by the meters are extracted from the "OFPMeterStatsReply" message. The application also saves the JSON format of all the information received by "OFPMeterStatsReply" in a separate file so it can be used for further analysis.
- Inserting flow entries with meter instruction and meter IDs can be either done by a separate program, manually or can be integrated into the main application in charge of forwarding.
- After adding meter instructions, each time the flow passes through the switch and matches a flow entry with the meter instruction, the meter ID associated to that entry is updated.
- The Traffic Matrix app sends OFPMeterStatsRequest every 3 seconds (adjustable to any amount) to all of the switches in the network and collects OFPMeterStatsReply from the switches, extracts the necessary values and prints them on the terminal. A sample of the result from terminal and JSON output are shown in Tables 6 and 7.

```
['DP=2 MeterID=4 Byte=41576 duration=428258', 'DP=2 MeterID=2 Byte=83835364 duration=428258', 'DP=2
MeterID=3 Byte=80986626 duration=428258']

['DP=3 MeterID=4 Byte=40524 duration=428257', 'DP=3 MeterID=2 Byte=80957644 duration=428257', 'DP=3
MeterID=3 Byte=0 duration=428257']

['DP=4 MeterID=4 Byte=36478 duration=428258', 'DP=4 MeterID=2 Byte=19892 duration=428806', 'DP=4
MeterID=3 Byte=28208 duration=428806']
```

Table 6 : Meter Statistics shown on the terminal

```
{
  "OFPMeterStatsReply": {
    "body": [
      {
        "OFPMeterStats": {
          "band_stats": [],
          "byte_in_count": 732,
          "duration_nsec": 311000,
          "duration_sec": 426959,
          "flow_count": 2,
          "len": 40,
          "meter_id": 4,
          "packet_in_count": 8
        }
      },
      {
        "OFPMeterStats": {
          "band_stats": [],
          "byte_in_count": 0,
          "duration_nsec": 320000,
          "duration_sec": 426959,
          "flow_count": 2,
          "len": 40,
          "meter_id": 2,
          "packet_in_count": 0
        }
      },
      {
        "OFPMeterStats": {
          "band_stats": [],
          "byte_in_count": 0,
          "duration_nsec": 316000,
          "duration_sec": 426959,
          "flow_count": 4,
          "len": 40,
          "meter_id": 3,
          "packet_in_count": 0
        }
      }
    ],
    "flags": 0,
    "type": 9
  }
}
```

Table 7: Sample of OFPMeterStatsReply from the JSON file

The JSON output saves more information about the meter stats collected from the switches; for example the number of flows associated to each meter at the requested time, number of packet_in_count processed by the switch, duration in nanoseconds and band status in case any packet shaping has been applied through the meter bands.

A Traffic Matrix table is illustrated in Table 8.

Traffic Matrix	Switch DP 2	Switch DP 3	Switch DP 4
Switch DP 2	Meter 2	Meter 3	Meter 4
Switch DP 3	Meter 2	Meter 3	Meter 4
Switch DP 4	Meter 2	Meter 3	Meter 4

Table 8 : Meter Stats from each switch represents a row of a Traffic Matrix table

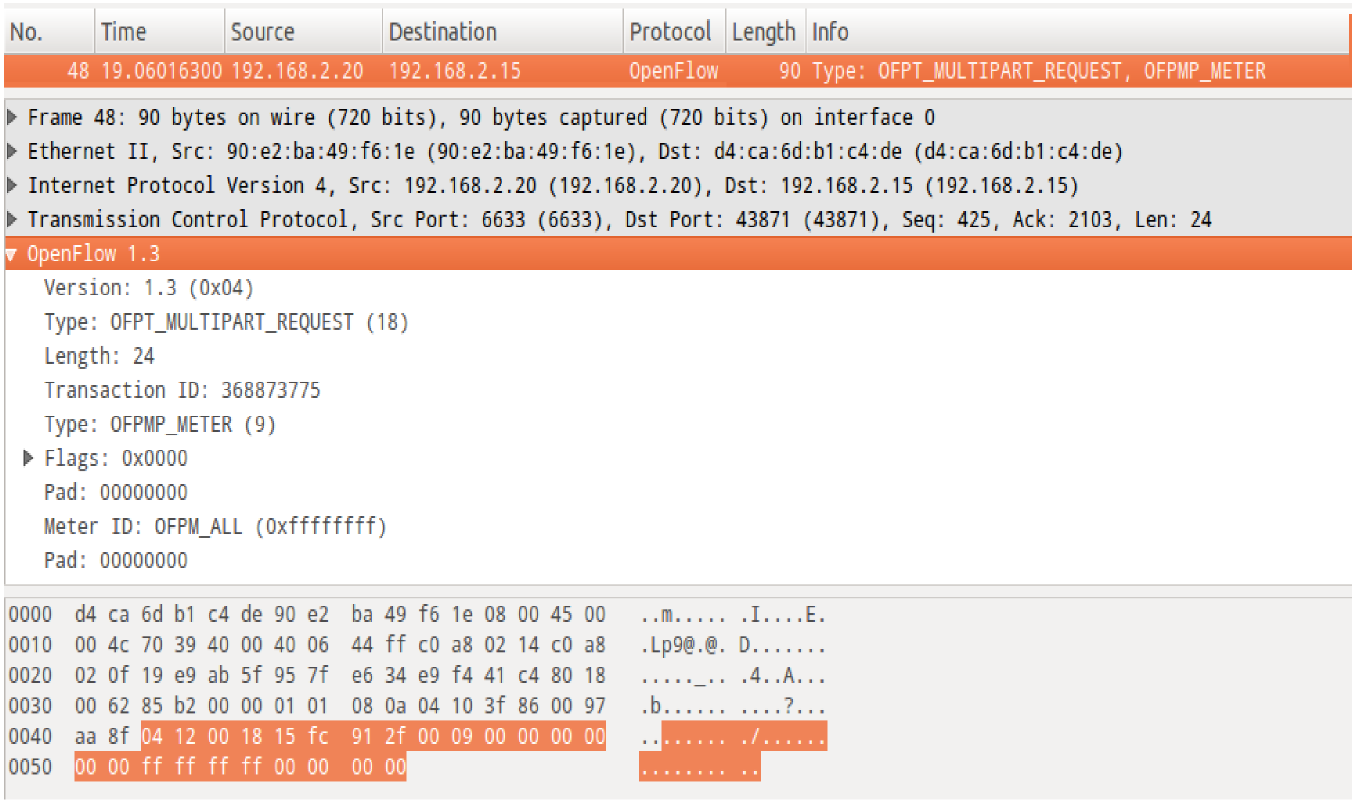
5.7 Impact of Traffic Matrix measurement on network load

Before explaining the network overload caused by Traffic matrix, it is necessary to explain that, in this context, by network we mean the Openflow channels between switches and the controller. Since the setup is usually out of band, they are completely separated from the network that connects the host together, hence there is no extra packet created in the inner network. Also there are three ways switches can be configured by the controller: Reactive, Proactive and Hybrid. In reactive mode every packet received by the switches is sent to the controller and depending on the decision taken by the controller, they will be forwarded. In proactive mode (the method used in our test-bed) flow table and flow entries are created during initial receive of unknown flows through packet_in messages from the switch and for the rest of the packet, they will be forwarded directly by the switch using a flow table configured by the controller in the packet_out message. In Hybrid mode, the combination of both reactive and proactive methods is used. The latter method (Hybrid) is the most used case since we have less delay in packet forwarding while maintaining visibility and security in the network.

In DTM, regardless of SDN implementation, we are interested in the amount of extra packet caused by DTM itself. There are two types of extra packets created, OFPMeterStatsRequest from the server and OFPMeterStatsReply from the switch.

OFPMeterStatsRequest shown in Figure 41 is used by the controller to ask for the status of all the meters in the switch regardless of the amount of its meters, and its packet length is 90 bytes.

- Analysis of OFPMeterStatsRequest in detail:
    - Ethernet header + IP header: 14 bytes + 20 bytes
    - TCP header: 20 bytes + 12 of Options
    - Total OF message: 24 bytes

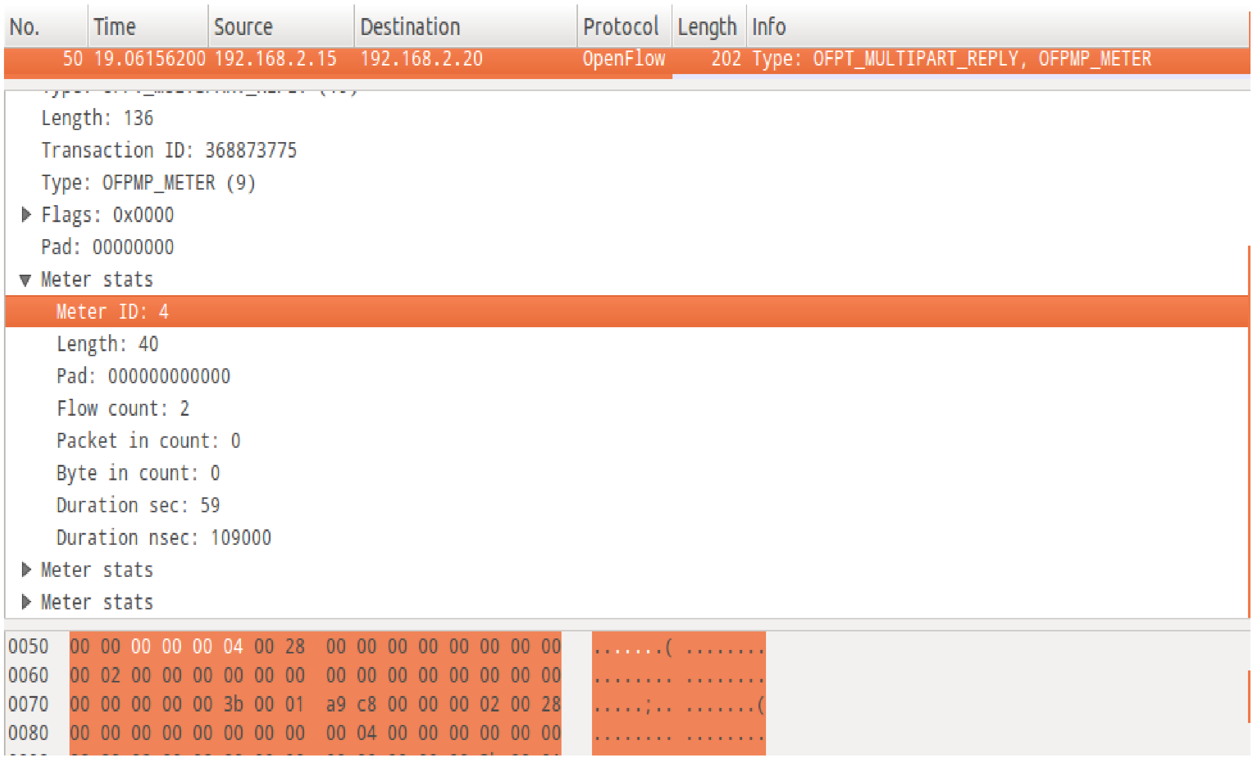


```
No.  Time           Source         Destination    Protocol  Length  Info
  48 19.06016300 192.168.2.20   192.168.2.15     OpenFlow      90 Type: OFPT_MULTIPART_REQUEST, OFPMP_METER

▶ Frame 48: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface 0
▶ Ethernet II, Src: 90:e2:ba:49:f6:1e (90:e2:ba:49:f6:1e), Dst: d4:ca:6d:b1:c4:de (d4:ca:6d:b1:c4:de)
▶ Internet Protocol Version 4, Src: 192.168.2.20 (192.168.2.20), Dst: 192.168.2.15 (192.168.2.15)
▶ Transmission Control Protocol, Src Port: 6633 (6633), Dst Port: 43871 (43871), Seq: 425, Ack: 2103, Len: 24
▼ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_MULTIPART_REQUEST (18)
    Length: 24
    Transaction ID: 368873775
    Type: OFPMP_METER (9)
  ▶ Flags: 0x0000
    Pad: 00000000
    Meter ID: OFPM_ALL (0xffffffff)
    Pad: 00000000

0000  d4 ca 6d b1 c4 de 90 e2  ba 49 f6 1e 08 00 45 00   ..m..... .I....E.
0010  00 4c 70 39 40 00 40 06  44 ff c0 a8 02 14 c0 a8   .Lp9@.@. D.......
0020  02 0f 19 e9 ab 5f 95 7f  e6 34 e9 f4 41 c4 80 18   ....._.. .4..A...
0030  00 62 85 b2 00 00 01 01  08 0a 04 10 3f 86 00 97   .b...... ....?...
0040  aa 8f 04 12 00 18 15 fc  91 2f 00 09 00 00 00 00   ........ ./......
0050  00 00 ff ff ff ff 00 00  00 00                     ........ ..
```

Figure 41: Meter Request Message

The OFPMeterStatsReply, shown in Figure 42, is sent along with the information of requested meters to the controller. The size varies, depending to the number of meters in the message. Here, since each switch only stores three meters, the size of the packet is 202 bytes.

- Analysis of OFPMeterStatsReply in details:
    - Ethernet header + IP header: 14 bytes + 20 bytes
    - TCP header: 20 bytes + 12 of Options
    - Total OF message: 40 bytes for each meter data + 16 bytes Header, here we have three meters hence in total the length is 136 bytes.



```
No.  Time         Source        Destination   Protocol  Length Info
  50 19.06156200 192.168.2.15  192.168.2.20  OpenFlow     202 Type: OFPT_MULTIPART_REPLY, OFPMP_METER
```

```
  Length: 136
  Transaction ID: 368873775
  Type: OFPMP_METER (9)
▸ Flags: 0x0000
  Pad: 00000000
▾ Meter stats
    Meter ID: 4
    Length: 40
    Pad: 000000000000
    Flow count: 2
    Packet in count: 0
    Byte in count: 0
    Duration sec: 59
    Duration nsec: 109000
▸ Meter stats
▸ Meter stats
```

```
0050  00 00 00 00 00 04 00 28  00 00 00 00 00 00 00 00   .......( ........
0060  00 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0070  00 00 00 00 00 3b 00 01  a9 c8 00 00 00 02 00 28   .....;.. .......(
0080  00 00 00 00 00 00 00 00  00 04 00 00 00 00 00 00   ........ ........
```

Figure 42: Meter Stats Reply

In addition to messages generated by controller and the switch, since the connection is established in TCP, ACK messages are sent by each controller and the switch hence adding 66 bytes for each ACK message.

- Ethernet header + IP header: 14 bytes + 20 bytes
- TCP header: 20 bytes + 12 of Options

Since in our approach the Traffic Matrix application only sends stats requested to the edge switches, the load is proportional to the number of rows or columns in the traffic matrix table. For example, in a traffic matrix table with three rows there are: three switches, three meter status in each switches and 6 ACK message each time the switches are queried.

To obtain the extra traffic generated by the process of obtaining traffic measurement, the following formula can be used.

$$\big((r*2)*Ack\big) + (r*Req) + \Big(r*\big((r*Of)+Rep\big)\Big) = r\,[\,2Ack + Req + Rep + (r*Of)] \quad (4)$$

Where:

- r is the number of rows in the TM table
- Ack is the size of the ACK packet
- Req is the size of the OFPMeterStatsRequest packet

- Of is the size of the meter stats for each meter presented in the switch
- Rep is the overhead of OFPMeterStatsReply packet send by the switch

To conclude, every time the controller queries the switches to obtain their meter stats, series of extra packets are created. In case of small to medium network, the extra traffic and packet generated can be acceptable. As the number of participant switches grows, the extra traffic may become a concern and a factor in the overall network performance. From the formula above, the total traffic generated during each query process for three switches in the network is:

$$((\mathrm{r} * 2) * 66) + (\mathrm{r} * 90) + (\mathrm{r} * ((\mathrm{r} * 40) + 82))$$

$$(6 \times 66) + (3 \times 90) + (3 \times 202) = 1272 \text{ bytes}$$

where r is the number of rows in the TM. Table 9 shows the transmitted bytes during each query process. There is a quadratic growth due to the number of meters each switch has to update. Comparing to previous work explained in section 4.2.5, we have been able to reduce the overhead present in Distributed TM because there is no need for synchronization messages to be sent.

Each query process	**10 Switches**	**100 Switches**	**1000 switches**
Total Overhead	6.8 KB	420.3 KB	39.3 MB

Table 9: Comparison between our approach and the previous work in terms of total message overhead

CHAPTER 6. CONCLUSIONS

6.1 Achievements

Distributed Traffic Measurement (DTM) estimates the TM with upmost accuracy and minimal computing overhead. The separation of Control and Data planes in SDN has given us a flexibility to dynamically assign meters in edge switches to measure aggregate of all flows between each pair of ingress and egress in the network. Our framework provides the controller the ability to dynamically configure each edge router to measure and aggregate a portion of the TM.

The framework has been design so that no further modifications are needed in the switches that are running OpenFlow 1.3 or higher as long as they support the meter features. In addition, our framework leave rooms for further enhancement, like using multiple controller setup to take the load off the master controller. DTM also enables network engineers to measure traffic in more detail using match fields of the packet header defined in OpenFlow. These match fields enable us to create meters rule for specific type of traffic like HTTP, for example.

Our evaluation of the framework has shown that the amount of extra packets created from the TM operation is proportional to the number of edge nodes in the network and the resulted amount is negligible. For example, approximately only 1 Kbyte of information are generated during each time a three edge network is queried by the controller. The meters only carries the aggregate statistics of all the flows associate to it, which considerably reduces the message size sent to the controller.

There are still a few weaknesses in our framework. Depending to the implementation, separate rules are used to create flow entries with meter instruction. The extra memory needed to incorporate these entries in the switch may reduce the switch processing power. One solution is to integrate TM measurement with the main application in the controller where meters are directly added to flow entries with output actions. This process will eliminate any extra flow entries needed to measure the traffic but may become too complex to implement. Also, the integration of meters with flow entries that have actions in their action set will cause them to be removed in case of removal of meter itself. This may interrupt the normal operation of the network, since the controller has to reconfigure the flow tables.

Finally, when compared to related works done in the same field (Section 4.2), our framework significantly reduces the cost of the TM measurement while preserving the accuracy of the final result. The goal is to create a balance between cost needed to obtain TM and the accuracy of the final result. For example, compared to other approaches like FlowSense, where TM measurement is bound to control messages received by the controller, or the Commodity-solution, where accuracy has been scarified for the cost of the operation, our framework can produce more accurate result while keeping the cost to minimum.

6.2 Future lines of development

SDN is rapidly evolving. Since the beginning of this work, the number of controller and switches implementing OpenFlow has doubled. There are many ongoing research efforts about how to improve SDN as a concept and how to further abstract network software from hardware.

In regards to our framework, one of the objectives is to reduce the memory usage in the switch devoted to the measurement of the TM. This can be achieved using intelligent algorithms to aggregate flow rules for measuring traffic [69] or integrating the TM process with the main application on the controller in charge of decision making (not recommended as explained in section 4.5).

A second objective is to make the DTM process independent of other applications in the network. There is still a need for coordination between different applications running on top of controller. Even if a separate flow table is reserved for the meter instructions, other applications should be aware of it. One of the solutions to this problem is the use of modular programming languages. It enables network programmers and operators to write succinct modular network applications by providing powerful abstractions. One example of these modular programming languages is Pyretic [70].

There are other areas of improvement. For example, Payless provides a rich set of APIs for developers and operator to monitor network resource. Combining the efficiency of DTM and the flexibility of Payless can provide interesting advances on network monitoring. Another example can be combining query-based monitoring tools with DTM. Different type of monitoring can be divided between DTM and query-based tools. For example DTM best serves when monitoring aggregates of all flows with the same origin and destination, but monitoring specific flows like heavy hitters may introduce many extra flow entries.

Unfortunately, in today's hardware being deployed there are still constrains that limit what we are capable of doing. Openflow is hardware dependent because of these constrains, therefore the set of functions that can be performed on the packets are pretty limited. Recent development in modular data planes (Section 2.6.1) enables SDN to create flow rules more efficiently and can be used in DTM to define broader set of rules and improve overall performance of the framework.

REFERENCES:

1. Network Complexity Conundrum, By Russ White
   Available at: http://www.networkcomputing.com/informationweek-home/the-network-complexity-conundrum/a/d-id/1269333
2. Industrial Ethernet: A Control Engineer's Guide
   Available at:
   http://www.cisco.com/web/strategy/docs/manufacturing/industrial_ethernet.pdf
3. Casado, Martin, et al. "Ethane: Taking control of the enterprise." ACM SIGCOMM Computer Communication Review 37.4 (2007): 1-12.
4. Pfaff, Ben, B. LANTZ, and B. HELLER. "OpenFlow switch specification, version 1.3. 0." Open Networking Foundation (2012).
5. Mahoney, Suzanne M., and Kathryn B. Laskey. "Network engineering for complex belief networks." Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence. Morgan Kaufmann Publishers Inc., 1996.
6. What is Cisco NetFlow? How does it work?
   Available at: http://www.plixer.com/blog/scrutinizer/what-is-netflow-how-does-it-work/
7. Traffic Monitoring using sFlow
   Available at: http://www.sflow.org/sFlowOverview.pdf
8. Jain, Navendu, et al. "Network Imprecision: A New Consistency Metric for Scalable Monitoring." OSDI. 2008.
9. OpenWrt home webpage
   Available at: https://openwrt.org/
10. The Control Plane, Data Plane and Forwarding Plane in Networks, By Brent Salisbury
    Available at: http://networkstatic.net/the-control-plane-data-plane-and-forwarding-plane-in-networks/
11. Yang, Lily, et al. Forwarding and control element separation (ForCES) framework. RFC 3746, April, 2004.
12. Headquarters, Americas. "IP Routing: Protocol-Independent Configuration Guide, Cisco IOS Release 15.0 M." (2011).
13. Dugal, D., and C. Pignataro. R. Dunn," Protecting the Router Control Plane. RFC 6192, March, 2011.
14. Doeringer, Willibald, Günter Karjoth, and Mehdi Nassehi. "Routing on longest-matching prefixes." IEEE/ACM Transactions on Networking (TON) 4.1 (1996): 86-97.

15. Content-addressable memory
Available at: http://en.wikipedia.org/wiki/Content-addressable_memory#Ternary_CAMs
16. Control and Data plane. By Ivan Pepelnjak
Available at: http://wiki.nil.com/Control_and_Data_plane
17. Fu, Jing, and Jennifer Rexford. "Efficient IP-address lookup with a shared forwarding table for multiple virtual routers." Proceedings of the 2008 ACM CoNEXT Conference. ACM, 2008.
18. RIBS and FIBS (Aka IP Routing Table and CEF Table). By, Ivan Pepelnjak
Available at: http://blog.ipspace.net/2010/09/ribs-and-fibs.html
19. Nadeau, Thomas D., and Ken Gray. SDN: Software Defined Networks. O'Reilly Media, Inc., 2013.
20. Foundation, Open Networking. "Software-defined networking: The new norm for networks." ONF White Paper (2012).
21. Rutgers, Charles L. Hedrick. "An introduction to IGRP."
Available at:
http://web.diegm.uniud.it/Utenti/pierluca/public_html/teaching/reti_di_calcolatori_II/documentazione_tecnica/igrp/CISCO_IGRP.pdf
22. Feamster, Nick, Jennifer Rexford, and Ellen Zegura. "The road to SDN." Queue 11.12 (2013): 20.
23. Caesar, Matthew, et al. "Design and implementation of a routing control platform." Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2. USENIX Association, 2005.
24. Bjorklund, Martin. "YANG-A data modeling language for the Network Configuration Protocol (NETCONF)." (2010).
25. HP SDN Application Store and Open SDN Ecosystem
Available at: http://www.opennetsummit.org/pdf/2014/sdn-idol/HP-SDN-Idol-Proposal-1.pdf
26. HP open Ecosystem breaks down barriers to SDN
Available at: http://www.itvarnews.in/2013/10/15/hp-open-ecosystem-breaks-down-barriers-to-sdn/
27. SDN Architecture Overview
Available at: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf

28. Salim, J. Hadi, D. Meyer, and O. Koufopavlou. "SDNRG E. Haleplidis Internet-Draft S. Denazis Intended status: Informational University of Patras Expires: September 4, 2014 K. Pentikousis EICT." (2014).
29. Cisco's One Platform Kit (onePK)
Available at: http://www.cisco.com/c/en/us/products/ios-nx-os-software/onepk.html
30. The history of OpenFlow
Available at: http://www.computerweekly.com/feature/The-history-of-OpenFlow
31. Pfaff, B. "OpenFlow Switch Specifications 1.0. 0." (2009).
32. Specification-Version, OpenFlow Switch. "1.4. 0." (2013).
33. OpenFlow version 1.3 tutorial
Available at: http://sdnhub.org/tutorials/openflow-1-3/
34. Software-Defined Networks and OpenFlow, by William Stallings
Available at: http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_16-1/161_sdn.html
35. OpenFlow: Proactive vs Reactive
Available at: http://networkstatic.net/openflow-proactive-vs-reactive-flows/
36. Bosshart, Pat, et al. "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN." Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM. ACM, 2013.
37. Bosshart, Pat, et al. "Programming protocol-independent packet processors." arXiv preprint arXiv:1312.1719 (2013).
38. Click Modular Router Project
Available at: http://www.read.cs.ucla.edu/click/click
39. Layered Protocols in Click
available at: http://www.pats.ua.ac.be/software/click/click-2.0/layeredprotocols.pdf
40. Kohler, Eddie, et al. "The click modular router project." Internet, May (2006).
41. Dobrescu, Mihai, et al. "RouteBricks: exploiting parallelism to scale software routers." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009.
42. Patterson, David A. "Reduced instruction set computers." Communications of the ACM 28.1 (1985): 8-21.
43. Solé Orrit, Adrià. "Distributed Traffic Matrix Measurement in Software-Defined Networks." Master Thesis, Escola d'Enginyeria de Telecomunicacions I Aeronàutica de Castelldefels (EETAC), UPC. November 2013.

44. Paul Tune and Matthew Roughan. "Internet traffic matrices: A primer. (2013).
45. Zhang, Yin, et al. "Fast accurate computation of large-scale IP traffic matrices from link loads." ACM SIGMETRICS Performance Evaluation Review. Vol. 31. No. 1. ACM, 2003.
46. Roughan, Matthew, et al. "Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning." Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment. ACM, 2002.APA
47. Rincón D. Network topology and traffic [slides]. Lecture Notes, Course on Networks and Applications Design. EETAC (UPC, Castelldefels): 93 slides. November of 2012.
48. Netflow Architecture
Available at: http://commons.wikimedia.org/wiki/File:Netflow_architecture_en.svg
49. ManageEngine NetFlow Analyzer(NFA) – A Trip Down the Memory Lane, by Vithya
Available at: https://blogs.manageengine.com/product-blog/netflowanalyzer/2010/11/02/manageengine-netflow-analyzer-nfa-a-trip-down-the-memory-lane.html
50. S.R. Chowdhury, M.F. Bari, R. Ahmed, R. Boutaba, Payless: a low cost network monitoring framework for software defined networks, in: Proceedings of the 14th IEEE/IFIP Network Operations and Management Symposium, NOMS'14, May 2014.
51. Zookeeper
Available at: http://zookeeper.apache.org/
52. Tootoonchian, Amin, Monia Ghobadi, and Yashar Ganjali. "OpenTM: traffic matrix estimator for OpenFlow networks." Passive and Active Measurement. Springer Berlin Heidelberg, 2010.
53. Yu, Curtis, et al. "Flowsense: monitoring network utilization with zero measurement cost." Passive and Active Measurement. Springer Berlin Heidelberg, 2013.
54. Jose, Lavanya, Minlan Yu, and Jennifer Rexford. "Online measurement of large traffic aggregates on commodity switches." Proc. of the USENIX HotICE workshop. 2011.
55. Open Virtual Switch
Available at: http://openvswitch.org/
56. Pang, R., Allman, M., Bennett, M., Lee, J., Paxson, V., Tierney, B.: A first look at modern enterprise traffic. In: Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement, Berkeley, CA, USA (2005) 2–2
57. Naous, J., Erickson, D., Covington, G.A., Appenzeller, G., McKeown, N.: Implementing an OpenFlow switch on the NetFPGA platform. In Franklin, M.A., Panda, D.K., Stiliadis, D., eds.: ANCS, ACM (2008) 1–9

58. List of TCP and UDP port numbers

Available at: http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

59. Ryu SDN Framework

Available at: http://osrg.github.io/ryu/

60. Mininet Project

Available at: http://mininet.org/

61. CPqD Software switch

Available at: https://github.com/CPqD/ofsoftswitch13

62. Virtual switching with Open vSwitch, By Ralf Spenneberg

Available at: http://www.admin-magazine.com/CloudAge/Articles/Virtual-switching-with-Open-vSwitch

63. List of OpenFlow Switches compatible with Ryu Controller

Available at: http://osrg.github.io/ryu/certification.html#ofs

64. Centec V350 OpenFlow Switch

Available at: http://www.centecnetworks.com/en/SolutionList.asp?ID=43

65. HP 2920 switch

Available at: http://h17007.www1.hp.com/us/en/networking/products/switches/HP_2920_Switch_Series/index.aspx

66. Linksys WRT54GL

Available at: http://www.linksys.com/es-eu/products/routers/WRT54GL

67. RB750GL

Available at: http://routerboard.com/RB750GL

68. Install Wireshark on Linux for OpenFlow capture

Available at: http://networkstatic.net/installing-wireshark-on-linux-for-openflow-packet-captures/

69. Malboubi, Mehdi, et al. "Intelligent SDN based Traffic (de) Aggregation and Measurement Paradigm (iSTAMP)."

70. Reich, Joshua, et al. "Modular SDN Programming with Pyretic." USENIX; login 38.5 (2013): 128-134.

71. Installing Mininet from Source

Available at: https://github.com/mininet/mininet

72. Raspberry Pi

Available at: Availabe at: http://www.raspberrypi.org/

73. Getting Started with Ryu
Available at: http://ryu.readthedocs.org/en/latest/getting_started.html#what-s-ryu
74. Ryu OpenFlow Tutorial
Available at: https://github.com/osrg/ryu/wiki/OpenFlow_Tutorial
75. RYU SDN framework using OpenFlow 1.3
Available at: http://osrg.github.io/ryu/resources.html#documentation
76. Ryu OpenFlow version 1.3 Reference
Available at: http://ryu.readthedocs.org/en/latest/ofproto_v1_3_ref.html
77. Installing OpenWRT on a RouterBOARD 750GL
Available at: http://www.ericconrad.com/2012/05/installing-openwrt-on-routerboard-750gl.html
78. OpenWRT on Mikrotik Routerboard 411/750
Available at: OpenWRT on Mikrotik Routerboard 411/750
79. OpenFlow, OpenWRT and bro-ids on rb450g
Available at: http://chabbir.blogspot.com.es/2013/01/openflow-openwrt-and-bro-ids-on-rb450g.html
80. Iperf, Network Performance measurement
Available at: https://iperf.fr/

ACRONYMS AND ABBREVIATIONS TABLE

Acronym	Term	Acronym	Term
ACLs	Access Control Lists	ASICs	Application-Specific Integrated Circuit
AS	Autonomous System	CDF	Cumulative Distribution Function
BGP	Boarder Gateway Protocol	DDos	Distributed Denial of Service
CDPI	Control to Data-Plane Inteface	DoS	Denial of Service
DTM	Distributed Traffic Matrix	FPGA	Field-Programmable Gate Array
FIB	Forwarding Information Base	IE	Ingress-Egress
IGRP	Interior Gateway Routing Protocol	MIB	Management Information Base
LIB	Label Information Base	NPU	Network Processing Unit
NBI	Northbound Interface	NTP	Network Time Protocol
Netconf	Network configuration Protocol	ONF	Open Networking Foundation
OSPF	Open Shortest Path First	OVS	Open Virtual Switch
QoS	Quality of Service	PBB	Provider Backbone Bridge
RCP	Routing Control Protocol	PoPs	Point of Presence
RIB	Routing Information Base	RMT	Reconfigurable Match Table
RIP	Routing Information Protocol	SNMP	Simple Network Management Protocol
SDN	Software Define Networks	TCP	Transmission Control Protocol
TCAM	Ternary Content Addressable Memory	TLS	Transport Layer Protocol
TE	Traffic Engineering	ToS	Type of Service
TM	Traffic Matrix	UDP	User Datagram Protocol
VLANs	Virtual Local Area Networks	VoIP	Voice over IP

APPENDIX

A. Experiments Configuration and Test results

A1. Detailed Configuration and test result for the first experiment

First, Mininet was installed from source on an Ubuntu machine. For complete installation guide refer to [71]. To utilize Meter features in OpenFlow we needed a software switch that can support them. Therefore CPqD was built from source on the same machine. For complete guide regarding the installation process refer to [61]. To create a simple topology consisting of one switch and three connected host the following command is used:

```
sudo mn --topo single,3 --mac --switch user --controller remote
```

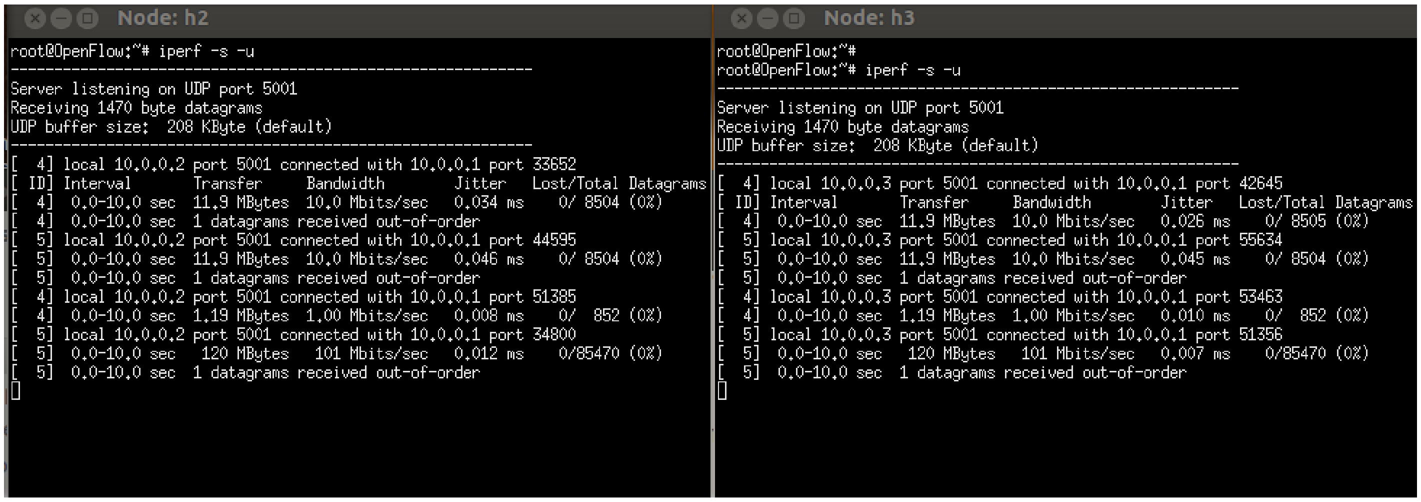
Note that "—switch user" will tell Mininet to use CPqD software switch instead of built-in switch. Ryu controller with Simple switch is used to configure the forwarding rules in the switch for detailed explanation regarding how simple switch works refer to (appendix b2). After initial configuration of the switch we used *dpctl* tool to add Meters ID and flow entries with the meter instruction. The commands for adding Meters ID are as follow:

```
sudo dpctl  unix:/tmp/s1 meter-mod cmd=add,flags=1,meter=12
sudo dpctl  unix:/tmp/s1 meter-mod cmd=add,flags=1,meter=13
sudo dpctl  unix:/tmp/s1 meter-mod cmd=add,flags=1,meter=21
sudo dpctl  unix:/tmp/s1 meter-mod cmd=add,flags=1,meter=23
sudo dpctl  unix:/tmp/s1 meter-mod cmd=add,flags=1,meter=32
```

The purpose of the experiment is to show the measurement process of Meter implementation, to add flow entries with meter instruction we used interface port as an ingress point and destination MAC address as egress point. For each interface there are two possible paths with distinct MAC addresses. The flow entries with meter pointer are as follow:

```
sudo dpctl  unix:/tmp/s1 flow-mod table=0,cmd=add,prio=2
in_port=1,eth_dst=00:00:00:00:00:03 meter:13 apply:output=3
sudo dpctl  unix:/tmp/s1 flow-mod table=0,cmd=add,prio=2
in_port=1,eth_dst=00:00:00:00:00:02 meter:12 apply:output=2
sudo dpctl  unix:/tmp/s1 flow-mod table=0,cmd=add,prio=2
in_port=2,eth_dst=00:00:00:00:00:01 meter:21 apply:output=1
sudo dpctl  unix:/tmp/s1 flow-mod table=0,cmd=add,prio=2
in_port=2,eth_dst=00:00:00:00:00:03 meter:23 apply:output=3
sudo dpctl  unix:/tmp/s1 flow-mod table=0,cmd=add,prio=2
in_port=3,eth_dst=00:00:00:00:00:02 meter:32 apply:output=2
sudo dpctl  unix:/tmp/s1 flow-mod table=0,cmd=add,prio=2
in_port=3,eth_dst=00:00:00:00:00:01 meter:31 apply:output=1
```

To see the effect of meter instruction on the switch performance we ran the test twice, once without any meter presence and once with meter instruction. Iperf was used to generate series of UDP traffic at 10Mbits/sec from host 1 towards host 2 and 3. Figure 43 shows a sample of result obtained from host 2 and 3. The first result is without meter presence. Each host received a total of 11.9Mbytes with bandwidth rate of 10 Mbps. The second result with same procedure where meter instruction are used shows that meters do not affect network performance and we got same performance.



```
root@OpenFlow:~# iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 33652
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  4]  0.0-10.0 sec  11.9 MBytes  10.0 Mbits/sec   0.034 ms    0/ 8504 (0%)
[  4]  0.0-10.0 sec  1 datagrams received out-of-order
[  5] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 44595
[  5]  0.0-10.0 sec  11.9 MBytes  10.0 Mbits/sec   0.046 ms    0/ 8504 (0%)
[  5]  0.0-10.0 sec  1 datagrams received out-of-order
[  4] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 51385
[  4]  0.0-10.0 sec  1.19 MBytes  1.00 Mbits/sec   0.008 ms    0/  852 (0%)
[  5] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 34800
[  5]  0.0-10.0 sec   120 MBytes   101 Mbits/sec   0.012 ms    0/85470 (0%)
[  5]  0.0-10.0 sec  1 datagrams received out-of-order
```

```
root@OpenFlow:~#
root@OpenFlow:~# iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.3 port 5001 connected with 10.0.0.1 port 42645
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  4]  0.0-10.0 sec  11.9 MBytes  10.0 Mbits/sec   0.026 ms    0/ 8505 (0%)
[  5] local 10.0.0.3 port 5001 connected with 10.0.0.1 port 55634
[  5]  0.0-10.0 sec  11.9 MBytes  10.0 Mbits/sec   0.045 ms    0/ 8504 (0%)
[  5]  0.0-10.0 sec  1 datagrams received out-of-order
[  4] local 10.0.0.3 port 5001 connected with 10.0.0.1 port 53463
[  4]  0.0-10.0 sec  1.19 MBytes  1.00 Mbits/sec   0.010 ms    0/  852 (0%)
[  5] local 10.0.0.3 port 5001 connected with 10.0.0.1 port 51356
[  5]  0.0-10.0 sec   120 MBytes   101 Mbits/sec   0.007 ms    0/85470 (0%)
[  5]  0.0-10.0 sec  1 datagrams received out-of-order
```

Figure 43: Host 2 & 3 Iperf result

The second application (refer to appendix B3) created by us is used to query switches for meter status in 10 seconds intervals. The sample of one interval is shown below:

```
['DP=1 MeterID=32 Byte=0 duration=296', 'DP=1 MeterID=12 Byte=12582912 duration=298', 'DP=1 MeterID=21 Byte=4942 duration=296', 'DP=1 MeterID=13 Byte=25165824 duration=296', 'DP=1 MeterID=31 Byte=6300 duration=296', 'DP=1 MeterID=23 Byte=0 duration=296']

['DP=1 MeterID=32 Byte=0 duration=306', 'DP=1 MeterID=12 Byte=125829120 duration=308', 'DP=1 MeterID=21 Byte=4942 duration=306', 'DP=1 MeterID=13 Byte=125829120 duration=306', 'DP=1 MeterID=31 Byte=7854 duration=306', 'DP=1 MeterID=23 Byte=0 duration=306']
```

The duration of the meter is the time the meter has been active in an OpenFlow switch. From the result above we can see the 10 second difference between these durations. Our monitoring application will output a complete data from each query to a JSON file so it can be used by other analyzing application. A sample of the JSON output at the end of the query interval is shown below:

```
"OFPMeterStatsReply": {
    "body": [
      {
        "OFPMeterStats": {
          "band_stats": [],
          "byte_in_count": 0,
          "duration_nsec": 955000,
          "duration_sec": 165,
          "flow_count": 1,
          "len": 40,
          "meter_id": 32,
          "packet_in_count": 0
        }
      },
      {
        "OFPMeterStats": {
          "band_stats": [],
          "byte_in_count": 123881548,
          "duration_nsec": 102000,
          "duration_sec": 168,
          "flow_count": 1,
          "len": 40,
          "meter_id": 12,
          "packet_in_count": 81938
        }
      },
      {
        "OFPMeterStats": {
          "band_stats": [],
          "byte_in_count": 3430,
          "duration_nsec": 984000,
          "duration_sec": 165,
          "flow_count": 1,
          "len": 40,
          "meter_id": 21,
          "packet_in_count": 9
        }
      },
      {
        "OFPMeterStats": {
          "band_stats": [],
          "byte_in_count": 14150892,
          "duration_nsec": 1000,
          "duration_sec": 166,
          "flow_count": 1,
          "len": 40,
          "meter_id": 13,
            "packet_in_count": 9361
          }
        },
        {
          "OFPMeterStats": {
            "band_stats": [],
            "byte_in_count": 3150,
            "duration_nsec": 951000,
            "duration_sec": 165,
            "flow_count": 1,
            "len": 40,
            "meter_id": 31,
            "packet_in_count": 5
          }
        },
        {
          "OFPMeterStats": {
            "band_stats": [],
            "byte_in_count": 0,
            "duration_nsec": 967000,
            "duration_sec": 165,
            "flow_count": 1,
            "len": 40,
            "meter_id": 23,
            "packet_in_count": 0
          }
        }
      ],
      "flags": 0,
      "type": 9
  }
}{
  "OFPMeterStatsReply": {
    "body": [
        {
          "OFPMeterStats": {
            "band_stats": [],
            "byte_in_count": 0,
            "duration_nsec": 960000,
            "duration_sec": 175,
            "flow_count": 1,
            "len": 40,
            "meter_id": 32,
            "packet_in_count": 0
          }
        },
        {
          "OFPMeterStats": {
            "band_stats": [],
            "byte_in_count": 143384836,
            "duration_nsec": 107000,
            "duration_sec": 178,
            "flow_count": 1,
            "len": 40,
            "meter_id": 12,
            "packet_in_count": 94837
          }
        },
        {
          "OFPMeterStats": {
            "band_stats": [],
            "byte_in_count": 4942,
            "duration_nsec": 989000,
            "duration_sec": 175,
            "flow_count": 1,
            "len": 40,
            "meter_id": 21,
            "packet_in_count": 10
          }
        },
        {
          "OFPMeterStats": {
            "band_stats": [],
            "byte_in_count": 24872484,
            "duration_nsec": 6000,
            "duration_sec": 176,
            "flow_count": 1,
            "len": 40,
            "meter_id": 13,
            "packet_in_count": 16452
          }
        },
        {
          "OFPMeterStats": {
            "band_stats": [],
            "byte_in_count": 3150,
            "duration_nsec": 956000,
            "duration_sec": 175,
            "flow_count": 1,
            "len": 40,
            "meter_id": 31,
            "packet_in_count": 5
          }
        },
        {
          "OFPMeterStats": {
            "band_stats": [],
            "byte_in_count": 0,
            "duration_nsec": 972000,
            "duration_sec": 175,
            "flow_count": 1,
            "len": 40,
            "meter_id": 23,
            "packet_in_count": 0
          }
        }
      ],
      "flags": 0,
      "type": 9
    }
```

A2. Detailed Configuration and test result for the second experiment

A2.1 Raspberry Pi

The Raspberry Pi [72] shown in Figure 44 is a credit-card sized computer that plugs into your TV and a keyboard. It is a capable little computer which can be used in electronics projects, and for many of the things that your desktop PC does, like spreadsheets, word-processing and games. It also plays high-definition video. We want to see it being used by kids all over the world to learn how computers work, how to manipulate the electronic world around them, and how to program. It comes in different models, the model used in the test-bed is Model B and include following specifications:

- 700 MHz ARM version 6 processor
- 512 MB of RAM
- onboard 10/100 Ethernet RJ45 jack
- Dual USB Connector
- SD, MMC, SDIO card slot for onboard storage
- Linux operating system

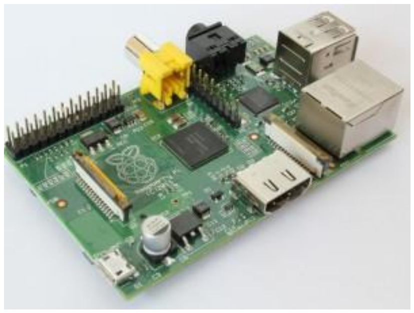


Figure 40: Raspberry Pi Model B

A2.2 Iperf setup on Raspberry Pi’s

Since the Raspberry Pi’s are running on ARM version of Debian Linux, to install Iperf on Raspberry Pi’s follow the below procedure:

- First download the ARM specific version from the following repository
  - https://packages.debian.org/sid/armel/iperf/download
- Unpack the package using the following command
  - sudo dpkg -x iperf_2.0.5-3_armel.deb /home/pi/
- And finally copy the Iperf folder into the “/usr/bin” location
  - sudo cp iperf /usr/bin/

After installing Iperf on all machines we have triggered all Raspberry Pi’s to act as both Iperf servers and clients.

A2.3 Mikrotik switch setup and connection to Controller

As described in Section 5.2, Mikrotik switches we chosen because of their higher capabilities. To start the OpenFlow protocol in the switches, when they boot for the first time, the following command needs to be executed.

- /etc/init.d/openflow restart

A sample of result shown in terminal after executing the above command is demonstrated below:

```
root@Of12214:~# /etc/init.d/openflow restart
eth0.1,eth0.3,eth0.4,eth0.5
Configuring OpenFlow switch for out-of-band control
Jan 05 22:16:26|00001|dp_ports|ERR|eth0.3 device has assigned IPv6 address fe80::d6ca:6dff:feb5:6a6c
Jan 05 22:16:26|00002|dp_ports|ERR|eth0.4 device has assigned IPv6 address fe80::d6ca:6dff:feb5:6a6c
Jan 05 22:16:26|00003|dp_ports|ERR|eth0.5 device has assigned IP address 147.83.113.200
Jan 05 22:16:26|00004|dp_ports|ERR|eth0.5 device has assigned IPv6 address fe80::d6ca:6dff:feb5:6a6c
No need for further configuration for out-of-band control
Jan 05 22:16:29|00001|vlog|INFO|opened log file /tmp/log/ofdatapath.log
root@Of12214:~# Jan 05 22:16:29|00002|secchan|INFO|OpenFlow reference implementation version 1.3.0
Jan 05 22:16:29|00003|secchan|INFO|OpenFlow protocol version 0x04
Jan 05 22:16:29|00004|secchan|WARN|new management connection will receive asynchronous messages
Jan 05 22:16:29|00005|rconn|INFO|tcp:127.0.0.1:6634: connecting...
Jan 05 22:16:29|00006|rconn|INFO|tcp:192.168.2.20:6633: connecting...
Jan 05 22:16:29|00007|rconn|INFO|tcp:192.168.2.20:6633: connection failed (Connection refused)
Jan 05 22:16:29|00008|rconn|WARN|tcp:192.168.2.20:6633: connection dropped (Connection refused)
Jan 05 22:16:29|00009|rconn|INFO|tcp:127.0.0.1:6634: connected
Jan 05 22:16:29|00010|port_watcher|INFO|Datapath id is 000000000002
Jan 05 22:16:30|00011|rconn|INFO|tcp:192.168.2.20:6633: connecting...
```

From the sample above, upon starting of the OpenFlow protocol on the switch, all the configuration instruction are read from a file located at “/etc/config/openflow”, which contains physical ports that OpenFlow can use, Data path (DP), IP address and port number of the controller. Since the controller is not yet running, the connection is being refused, and the switch will automatically try to reconnect to the controller every 3 second until it establishes the connection.

A2.4 Ryu initial setup and connection sample

Two applications are running on top of Ryu framework, one for configuring the switches to allow connectivity between themselves and host machines and another application developed by us to get the meters information from all the switches. For starting the Controller with the two applications mentioned above, the following command is executed in the terminal:

```
ryu-manager --verbose ryu.app.simple_switch_13.py app/trafficmatrix.py
```

A sample of the result shown in terminal after executing the above command, is demonstrated below:

```
user:~$ ryu-manager --verbose ryu.app.simple_switch_13.py app/trafficmatrix.py
loading app ryu.app.simple_switch_13.py
loading app app/trafficmatrix.py
loading app ryu.controller.ofp_handler
loading app ryu.controller.ofp_handler
instantiating app ryu.app.simple_switch_13.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app app/trafficmatrix.py of SimpleMonitor
BRICK SimpleMonitor
  CONSUMES EventOFPStateChange
  CONSUMES EventOFPMeterStatsReply
BRICK SimpleSwitch13
  CONSUMES EventOFPPacketIn
  CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
  PROVIDES EventOFPStateChange TO {'SimpleMonitor': set(['main', 'dead'])}
  PROVIDES EventOFPPacketIn TO {'SimpleSwitch13': set(['main'])}
  PROVIDES EventOFPMeterStatsReply TO {'SimpleMonitor': set(['main'])}
  PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitch13': set(['config'])}
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPHello
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPSwitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x2607a90> address:('192.168.2.16', 47829)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x2607e10>
move onto config mode
EVENT ofp_event->SimpleSwitch13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x75e80482
OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=3,n_buffers=256,n_tables=64)
move onto main mode
EVENT ofp_event->SimpleMonitor EventOFPStateChange
register datapath: 0000000000000003
EVENT ofp_event->SimpleMonitor EventOFPMeterStatsReply
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 3 d4:ca:6d:b1:c5:15 ff:ff:ff:ff:ff:ff 2
connected socket:<eventlet.greenio.GreenSocket object at 0x2607b50> address:('192.168.2.15', 34467)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x260e7d0>
```

```
move onto config mode
EVENT ofp_event->SimpleSwitch13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x9a7cfeb4
OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=4,n_buffers=256,n_tables=64)
move onto main mode
EVENT ofp_event->SimpleMonitor EventOFPStateChange
register datapath: 0000000000000004
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 4 d4:ca:6d:b1:c4:de ff:ff:ff:ff:ff:ff 4
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 3 d4:ca:6d:b1:c4:de ff:ff:ff:ff:ff:ff 1
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 3 d4:ca:6d:b1:c5:15 ff:ff:ff:ff:ff:ff 2
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 4 d4:ca:6d:b1:c5:15 ff:ff:ff:ff:ff:ff 1
connected socket:<eventlet.greenio.GreenSocket object at 0x260e690> address:('192.168.2.14', 42041)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x260ec10>
move onto config mode
EVENT ofp_event->SimpleSwitch13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x7a6841bf
OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=2,n_buffers=256,n_tables=64)
move onto main mode
EVENT ofp_event->SimpleMonitor EventOFPStateChange
register datapath: 0000000000000002
```

A2.5 Meter Configuration command and sample result

To assess the bandwidth, two hosts are configured as an Iperf server and one host as a client using the following commands:

- client = 192.168.2.110
- server = 192.168.2.111
- server= 192.168.0.113
- user@server# iperf –s –u –i 1
- root@raspi# iperf -c 192.168.2.113 –u –b 50m
- root@raspi# iperf -c 192.168.2.113 –u –b 50m

Series of UDP traffic were sent from Host 2 towards Host 3 and Host 5. After running the test for at least 10 times on each host, the average bandwidth obtained was 35 Mb/s in each route, sample of Iperf test result are shown below.

Host2 to Host3

```
pi@raspberrypi:~$ iperf -s -u -i 1
WARNING: interval too small, increasing from 0.00 to 0.5 seconds.
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
[  3] local 192.168.2.111 port 5001 connected with 192.168.2.110 port 42395
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0- 0.5 sec  2.11 MBytes  35.4 Mbits/sec   0.145 ms  522/ 2027 (26%)
[  3]  0.5- 1.0 sec  2.10 MBytes  35.3 Mbits/sec   0.136 ms  622/ 2122 (29%)
[  3]  1.0- 1.5 sec  2.12 MBytes  35.6 Mbits/sec   0.170 ms  609/ 2124 (29%)
[  3]  1.5- 2.0 sec  2.10 MBytes  35.3 Mbits/sec   0.170 ms  593/ 2093 (28%)
[  3]  2.0- 2.5 sec  2.14 MBytes  35.8 Mbits/sec   0.121 ms  634/ 2158 (29%)
[  3]  2.5- 3.0 sec  2.10 MBytes  35.3 Mbits/sec   0.111 ms  613/ 2114 (29%)
[  3]  3.0- 3.5 sec  2.14 MBytes  35.9 Mbits/sec   0.184 ms  610/ 2136 (29%)
[  3]  3.5- 4.0 sec  2.14 MBytes  35.9 Mbits/sec   0.125 ms  585/ 2113 (28%)
[  3]  4.0- 4.5 sec  2.13 MBytes  35.8 Mbits/sec   0.162 ms  612/ 2134 (29%)
[  3]  4.5- 5.0 sec  2.11 MBytes  35.4 Mbits/sec   0.195 ms  616/ 2123 (29%)
[  3]  5.0- 5.5 sec  2.14 MBytes  36.0 Mbits/sec   0.098 ms  598/ 2127 (28%)
[  3]  5.5- 6.0 sec  2.14 MBytes  36.0 Mbits/sec   0.122 ms  592/ 2122 (28%)
[  3]  6.0- 6.5 sec  2.14 MBytes  36.0 Mbits/sec   0.120 ms  599/ 2128 (28%)
[  3]  6.5- 7.0 sec  2.08 MBytes  34.9 Mbits/sec   0.200 ms  603/ 2085 (29%)
[  3]  7.0- 7.5 sec  2.13 MBytes  35.8 Mbits/sec   0.128 ms  645/ 2165 (30%)
[  3]  7.5- 8.0 sec  2.13 MBytes  35.8 Mbits/sec   0.127 ms  589/ 2109 (28%)
[  3]  8.0- 8.5 sec  2.13 MBytes  35.7 Mbits/sec   0.177 ms  626/ 2143 (29%)
[  3]  8.5- 9.0 sec  2.12 MBytes  35.6 Mbits/sec   0.113 ms  591/ 2106 (28%)
[  3]  9.0- 9.5 sec  2.10 MBytes  35.3 Mbits/sec   0.129 ms  643/ 2143 (30%)
[  3]  9.5-10.0 sec  2.15 MBytes  36.1 Mbits/sec   0.117 ms  589/ 2123 (28%)
[  3]  0.0-10.0 sec  42.6 MBytes  35.6 Mbits/sec   0.160 ms 12118/42493 (29%)
```

Host2 to Host5

```
pi@raspberrypi:~$ iperf -s -u -i 1
WARNING: interval too small, increasing from 0.00 to 0.5 seconds.
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
[  3] local 192.168.2.113 port 5001 connected with 192.168.2.110 port 59471
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0- 0.5 sec  2.06 MBytes  34.5 Mbits/sec   0.185 ms  487/ 1953 (25%)
[  3]  0.5- 1.0 sec  2.08 MBytes  34.8 Mbits/sec   0.094 ms  645/ 2126 (30%)
[  3]  1.0- 1.5 sec  2.04 MBytes  34.2 Mbits/sec   0.123 ms  635/ 2090 (30%)
[  3]  1.5- 2.0 sec  2.06 MBytes  34.5 Mbits/sec   0.109 ms  652/ 2120 (31%)
[  3]  2.0- 2.5 sec  2.08 MBytes  35.0 Mbits/sec   0.132 ms  644/ 2130 (30%)
[  3]  2.5- 3.0 sec  2.06 MBytes  34.6 Mbits/sec   0.180 ms  681/ 2151 (32%)
[  3]  3.0- 3.5 sec  2.04 MBytes  34.2 Mbits/sec   0.127 ms  647/ 2099 (31%)
[  3]  3.5- 4.0 sec  2.06 MBytes  34.5 Mbits/sec   0.107 ms  668/ 2136 (31%)
[  3]  4.0- 4.5 sec  2.07 MBytes  34.8 Mbits/sec   0.131 ms  638/ 2118 (30%)
[  3]  4.5- 5.0 sec  2.04 MBytes  34.2 Mbits/sec   0.139 ms  716/ 2169 (33%)
[  3]  5.0- 5.5 sec  2.04 MBytes  34.3 Mbits/sec   0.143 ms  612/ 2069 (30%)
[  3]  5.5- 6.0 sec  2.06 MBytes  34.6 Mbits/sec   0.125 ms  669/ 2138 (31%)
[  3]  6.0- 6.5 sec  2.07 MBytes  34.7 Mbits/sec   0.106 ms  647/ 2124 (30%)
[  3]  6.5- 7.0 sec  2.07 MBytes  34.7 Mbits/sec   0.114 ms  654/ 2128 (31%)
[  3]  7.0- 7.5 sec  2.06 MBytes  34.6 Mbits/sec   0.118 ms  641/ 2112 (30%)
[  3]  7.5- 8.0 sec  2.03 MBytes  34.0 Mbits/sec   0.174 ms  699/ 2144 (33%)
[  3]  8.0- 8.5 sec  2.07 MBytes  34.7 Mbits/sec   0.140 ms  641/ 2116 (30%)
[  3]  8.5- 9.0 sec  2.05 MBytes  34.4 Mbits/sec   0.147 ms  669/ 2131 (31%)
[  3]  9.0- 9.5 sec  2.08 MBytes  34.8 Mbits/sec   0.128 ms  640/ 2121 (30%)
[  3]  9.5-10.0 sec  1.99 MBytes  33.4 Mbits/sec   0.159 ms  712/ 2132 (33%)
[  3]  0.0-10.0 sec  41.3 MBytes  34.5 Mbits/sec   0.233 ms 13064/42506 (31%)
[  3]  0.0-10.0 sec  1 datagrams received out-of-order
```

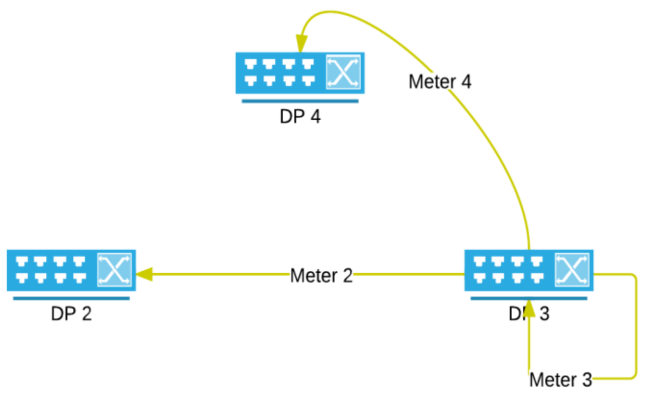
To configure the switches with meter id and adding flow entries with meter instruction, the Dpctl tool is used. Dpctl is a command-line utility that sends basic OpenFlow messages to a Switch. Commands used for each switch are shown below.

Meters ID for Switch with DP 3

```
sudo dpctl  tcp:192.168.2.16:6634 meter-mod cmd=add,flags=1,meter=2
sudo dpctl  tcp:192.168.2.16:6634 meter-mod cmd=add,flags=1,meter=3
sudo dpctl  tcp:192.168.2.16:6634 meter-mod cmd=add,flags=1,meter=4
```

Flows with meter instruction in Switch with DP 3

```
sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:f3:1a:ac,eth_dst=e8:9d:87:65:c5:21 meter:4 apply:output=1
sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:f3:1a:ac,eth_dst=b8:27:eb:a4:87:ae meter:4 apply:output=1
sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:f3:1a:ac,eth_dst=b8:27:eb:66:1f:90 meter:2 apply:output=1
sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:f3:1a:ac,eth_dst=b8:27:eb:81:5f:dc meter:2 apply:output=1

sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:eb:49:68,eth_dst=e8:9d:87:65:c5:21 meter:4 apply:output=1
sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:eb:49:68,eth_dst=b8:27:eb:a4:87:ae meter:4 apply:output=1
sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:eb:49:68,eth_dst=b8:27:eb:66:1f:90 meter:2 apply:output=1
sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:eb:49:68,eth_dst=b8:27:eb:81:5f:dc meter:2 apply:output=1

sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:eb:49:68,eth_dst=b8:27:eb:f3:1a:ac meter:3 apply:output=4
sudo dpctl tcp:192.168.2.16:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:f3:1a:ac,eth_dst=b8:27:eb:eb:49:68 meter:3 apply:output=2
```

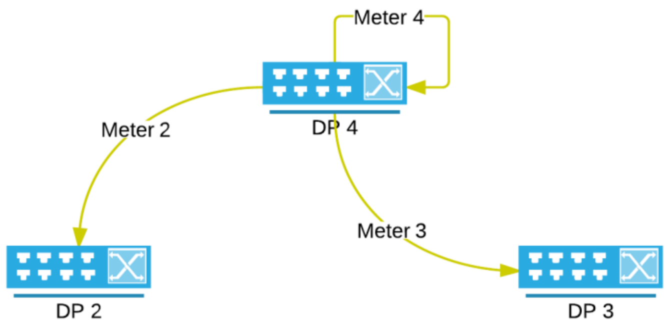


Meters ID for Switch with DP 4

```
sudo dpctl  tcp:192.168.2.15:6634 meter-mod cmd=add,flags=1,meter=2
sudo dpctl  tcp:192.168.2.15:6634 meter-mod cmd=add,flags=1,meter=3
sudo dpctl  tcp:192.168.2.15:6634 meter-mod cmd=add,flags=1,meter=4
```

Flows with meter instruction in Switch with DP 4

```
sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=e8:9d:87:65:c5:21,eth_dst=b8:27:eb:66:1f:90 meter:2 apply:output=2
sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=e8:9d:87:65:c5:21,eth_dst=b8:27:eb:81:5f:dc meter:2 apply:output=2
sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=e8:9d:87:65:c5:21,eth_dst=b8:27:eb:f3:1a:ac meter:3 apply:output=1
sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=e8:9d:87:65:c5:21,eth_dst=b8:27:eb:eb:49:68 meter:3 apply:output=1

sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:a4:87:ae,eth_dst=b8:27:eb:66:1f:90 meter:2 apply:output=2
sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:a4:87:ae,eth_dst=b8:27:eb:81:5f:dc meter:2 apply:output=2
sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:a4:87:ae,eth_dst=b8:27:eb:f3:1a:ac meter:3 apply:output=1
sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:a4:87:ae,eth_dst=b8:27:eb:eb:49:68 meter:3 apply:output=1

sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:a4:87:ae,eth_dst=e8:9d:87:65:c5:21 meter:4 apply:output=3
sudo dpctl tcp:192.168.2.15:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=e8:9d:87:65:c5:21,eth_dst=b8:27:eb:a4:87:ae meter:4 apply:output=4
```

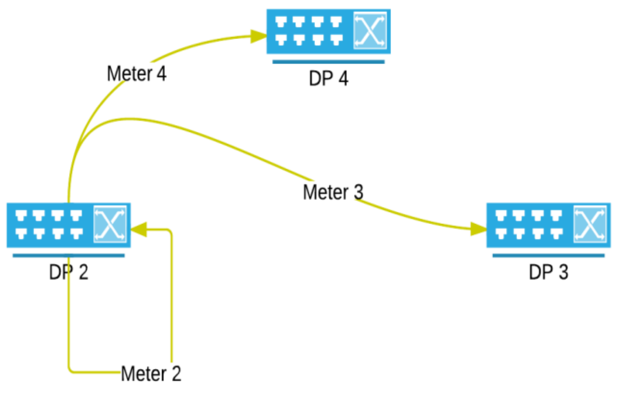


Meters ID for Switch with DP 2

```
sudo dpctl  tcp:192.168.2.14:6634 meter-mod cmd=add,flags=1,meter=2
sudo dpctl  tcp:192.168.2.14:6634 meter-mod cmd=add,flags=1,meter=3
sudo dpctl  tcp:192.168.2.14:6634 meter-mod cmd=add,flags=1,meter=4
```

Flows with meter instruction in Switch with DP 2

```
sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:66:1f:90,eth_dst=e8:9d:87:65:c5:21 meter:4 apply:output=1
#sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:66:1f:90,eth_dst=b8:27:eb:a4:87:ae meter:4 apply:output=1
sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:66:1f:90,eth_dst=b8:27:eb:f3:1a:ac meter:3 apply:output=1
sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:66:1f:90,eth_dst=b8:27:eb:eb:49:68 meter:3 apply:output=1

sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:81:5f:dc,eth_dst=e8:9d:87:65:c5:21 meter:4 apply:output=1
#sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:81:5f:dc,eth_dst=b8:27:eb:a4:87:ae meter:4 apply:output=1
sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:81:5f:dc,eth_dst=b8:27:eb:f3:1a:ac meter:3 apply:output=1
sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:81:5f:dc,eth_dst=b8:27:eb:eb:49:68 meter:3 apply:output=1

sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:81:5f:dc,eth_dst=b8:27:eb:66:1f:90 meter:2 apply:output=3
sudo dpctl tcp:192.168.2.14:6634 flow-mod cmd=add,table=0,prio=2
eth_type=0x800,eth_src=b8:27:eb:66:1f:90,eth_dst=b8:27:eb:81:5f:dc meter:2 apply:output=4
```



Example of how to add a Meter to a switch, and response message from the switch:

```
sudo dpctl  tcp:192.168.2.16:6634 meter-mod cmd=add,flags=1,meter=3

SENDING:
meter_mod{cmd="add", flags="0x1"", meter_id="3"", bands=[]}


OK.
```

Example of adding a flow entry with meter instructions:

```
SENDING:
flow_mod{table="0", cmd="add", cookie="0x0", mask="0x0", idle="0", hard="0", prio="2", buf="none", port="any",
group="any", flags="0x0", match=oxm{eth_dst="b8:27:eb:81:5f:dc", eth_src="b8:27:eb:eb:49:68",
eth_type="0x800"}, insts=[meter{meter="2"}, apply{acts=[out{port="1"}]}]}
```

After setting the Meter in all the switches, the same procedure from the previous test was done to see if the presence of meter has any impact on the network performance. In average the bandwidth obtained from the test was same as before, approximately 35 Mb/s, which clarifies that presence of a meter does not affect network performance and the difference is negligible, A sample of the result from the test is shown below.

H2 to H3

```
[  4] local 192.168.2.111 port 5001 connected with 192.168.2.110 port 39013
[  4]  0.0- 0.5 sec  2.01 MBytes  33.8 Mbits/sec   0.137 ms  583/ 2020 (29%)
[  4]  0.5- 1.0 sec  2.04 MBytes  34.2 Mbits/sec   0.132 ms  670/ 2124 (32%)
[  4]  1.0- 1.5 sec  2.03 MBytes  34.1 Mbits/sec   0.145 ms  675/ 2123 (32%)
[  4]  1.5- 2.0 sec  2.04 MBytes  34.2 Mbits/sec   0.183 ms  671/ 2127 (32%)
[  4]  2.0- 2.5 sec  1.98 MBytes  33.3 Mbits/sec   0.099 ms  710/ 2125 (33%)
[  4]  2.5- 3.0 sec  2.03 MBytes  34.1 Mbits/sec   0.135 ms  674/ 2124 (32%)
[  4]  3.0- 3.5 sec  2.04 MBytes  34.2 Mbits/sec   0.168 ms  672/ 2125 (32%)
[  4]  3.5- 4.0 sec  2.05 MBytes  34.4 Mbits/sec   0.121 ms  664/ 2126 (31%)
[  4]  4.0- 4.5 sec  2.00 MBytes  33.6 Mbits/sec   0.145 ms  698/ 2126 (33%)
[  4]  4.5- 5.0 sec  2.05 MBytes  34.4 Mbits/sec   0.132 ms  664/ 2127 (31%)
[  4]  5.0- 5.5 sec  2.05 MBytes  34.3 Mbits/sec   0.113 ms  668/ 2127 (31%)
[  4]  5.5- 6.0 sec  2.05 MBytes  34.4 Mbits/sec   0.123 ms  664/ 2126 (31%)
[  4]  6.0- 6.5 sec  2.00 MBytes  33.6 Mbits/sec   0.132 ms  697/ 2125 (33%)
[  4]  6.5- 7.0 sec  2.04 MBytes  34.3 Mbits/sec   0.140 ms  667/ 2125 (31%)
[  4]  7.0- 7.5 sec  2.03 MBytes  34.0 Mbits/sec   0.142 ms  680/ 2127 (32%)
[  4]  7.5- 8.0 sec  2.05 MBytes  34.5 Mbits/sec   0.166 ms  661/ 2126 (31%)
[  4]  8.0- 8.5 sec  2.00 MBytes  33.6 Mbits/sec   0.121 ms  698/ 2126 (33%)
[  4]  8.5- 9.0 sec  2.06 MBytes  34.6 Mbits/sec   0.164 ms  655/ 2124 (31%)
[  4]  9.0- 9.5 sec  2.04 MBytes  34.2 Mbits/sec   0.151 ms  671/ 2126 (32%)
[  4]  9.5-10.0 sec  2.04 MBytes  34.2 Mbits/sec   0.139 ms  669/ 2125 (31%)
[  4]  0.0-10.0 sec  40.7 MBytes  34.1 Mbits/sec   0.159 ms 13442/42507 (32%)
[  4]  0.0-10.0 sec  1 datagrams received out-of-order
```

H2 to H5

```
[  4] local 192.168.2.113 port 5001 connected with 192.168.2.110 port 53998
[  4]  0.0- 0.5 sec  1.96 MBytes  32.8 Mbits/sec   0.163 ms  513/ 1908 (27%)
[  4]  0.5- 1.0 sec  1.97 MBytes  33.0 Mbits/sec   0.173 ms  725/ 2129 (34%)
[  4]  1.0- 1.5 sec  1.91 MBytes  32.1 Mbits/sec   0.126 ms  759/ 2122 (36%)
[  4]  1.5- 2.0 sec  1.96 MBytes  32.9 Mbits/sec   0.142 ms  729/ 2126 (34%)
[  4]  2.0- 2.5 sec  1.95 MBytes  32.8 Mbits/sec   0.208 ms  730/ 2123 (34%)
[  4]  2.5- 3.0 sec  1.94 MBytes  32.6 Mbits/sec   0.157 ms  739/ 2124 (35%)
[  4]  3.0- 3.5 sec  1.92 MBytes  32.2 Mbits/sec   0.141 ms  755/ 2123 (36%)
[  4]  3.5- 4.0 sec  1.97 MBytes  33.1 Mbits/sec   0.149 ms  721/ 2127 (34%)
[  4]  4.0- 4.5 sec  1.97 MBytes  33.0 Mbits/sec   0.126 ms  721/ 2123 (34%)
[  4]  4.5- 5.0 sec  1.98 MBytes  33.1 Mbits/sec   0.197 ms  716/ 2125 (34%)
[  4]  5.0- 5.5 sec  1.94 MBytes  32.5 Mbits/sec   0.161 ms  744/ 2125 (35%)
[  4]  5.5- 6.0 sec  1.97 MBytes  33.1 Mbits/sec   0.138 ms  719/ 2125 (34%)
[  4]  6.0- 6.5 sec  1.97 MBytes  33.0 Mbits/sec   0.158 ms  719/ 2124 (34%)
[  4]  6.5- 7.0 sec  1.98 MBytes  33.2 Mbits/sec   0.127 ms  716/ 2127 (34%)
[  4]  7.0- 7.5 sec  1.93 MBytes  32.5 Mbits/sec   0.163 ms  741/ 2121 (35%)
[  4]  7.5- 8.0 sec  1.95 MBytes  32.6 Mbits/sec   0.164 ms  740/ 2128 (35%)
[  4]  8.0- 8.5 sec  1.97 MBytes  33.1 Mbits/sec   0.178 ms  715/ 2121 (34%)
[  4]  8.5- 9.0 sec  1.96 MBytes  32.9 Mbits/sec   0.140 ms  737/ 2135 (35%)
[  4]  9.0- 9.5 sec  1.95 MBytes  32.7 Mbits/sec   0.193 ms  716/ 2105 (34%)
[  4]  9.5-10.0 sec  1.95 MBytes  32.8 Mbits/sec   0.190 ms  742/ 2135 (35%)
[  4]  0.0-10.0 sec  39.3 MBytes  32.8 Mbits/sec   0.195 ms 14477/42485 (34%)
[  4]  0.0-10.0 sec  1 datagrams received out-of-order
```

B. Ryu Controller and Code Implementation

B1. Ryu Controller

Ryu [73] is a component-based software defined networking framework. Ryu provides software components with well-defined API that make it easy for developers to create new network management and control applications. Ryu supports various protocols for managing network devices, such as OpenFlow, Netconf, OF-config, etc. About OpenFlow, Ryu supports fully 1.0, 1.2, 1.3, 1.4 and Nicira Extensions.

Ryu is fully written in Python and therefore it is easy to learn and develop for. To install Ryu on the machine, we can use two different methods. One is to install using “pip” as show below:

```
% pip install ryu
```

The second method is to build from source, using the following commands:

```
% git clone git://github.com/osrg/ryu.git
% cd ryu; python ./setup.py install
```

Ryu [74] heavily utilizes decorator. The decorator is syntax sugar[13] to twist functions/methods. It is used to twist functions/methods. For example.

```
def log_on_entry(method):
  def _method(self):
     print 'on-entry'
     return method(self)
  return _method

class aClass(object)

  @log_on_entry
  def a_method(self):
     print self, 'a_method is called'

a = aClass()
a.a_method()
```

[13] Syntax sugar refer to the fact that, everything can be achieved without decorators but with more lines of code

@log_on_entry is same to:

```
class aClass(object)

   def a_method(self):
       print 'a_method is called'

   a_method = log_on_entry(a_method)
```

The output will be:

```
on-entry <main.aClass object at 0x7fc75eaf4d10> a_method is called.
```

The applications written for this thesis are based on Ryu, and are presented next:

B 2. Ryu Switching Hub

Switching hub [75] has a variety of functions. It learns the MAC address of the host connected to a port and retains it in the MAC address table. Also, when receiving packets addressed to a host already learned, transfers them to the port connected to the host; otherwise performs flooding.

The source code is shown below.

```
# Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# You may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# Limitations under the License.

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

class SimpleSwitch13(app_manager.RyuApp):
  OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

  def __init__(self, *args, **kwargs):
     super(SimpleSwitch13, self).__init__(*args, **kwargs)
     self.mac_to_port = {}

  @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
  def switch_features_handler(self, ev):
     datapath = ev.msg.datapath
     ofproto = datapath.ofproto
     parser = datapath.ofproto_parser

     # install table-miss flow entry
     #
     # We specify NO BUFFER to max_len of the output action due to
     # OVS bug. At this moment, if we specify a lesser number, e.g.,
     # 128, OVS will send Packet-In with invalid buffer_id and
     # truncated packet data. In that case, we cannot output packets
     # correctly.
     match = parser.OFPMatch()
     actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                       ofproto.OFPCML_NO_BUFFER)]
     self.add_flow(datapath, 0, match, actions)

  def add_flow(self, datapath, priority, match, actions):
     ofproto = datapath.ofproto
     parser = datapath.ofproto_parser

     inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                          actions)]

     mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                             match=match, instructions=inst)
     datapath.send_msg(mod)
```

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    #self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```

B 3. TM Measurement code

The TM application will query all the switches in the network, requesting all the meters available in them. The full source code is shown below.

```
import os

from operator import attrgetter

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub
import json

class SimpleMonitor(app_manager.RyuApp):

    def __init__(self, *args, **kwargs):
        super(SimpleMonitor, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)
        #self.num_of_switches = 0
        self.count = 0

    @set_ev_cls(ofp_event.EventOFPStateChange,
                [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
            if not datapath.id in self.datapaths:
                self.logger.debug('register datapath: %016x', datapath.id)
                self.datapaths[datapath.id] = datapath
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('unregister datapath: %016x', datapath.id)
                del self.datapaths[datapath.id]

    """def _monitor(self):
        while True:
            #print len(self.datapaths.values())
            #num_of_switches = len(self.datapaths.values())
            hub.sleep(10)
```

```
                    a = 0
                    while a < 2:
                            print a
                            a += 1
                            for dp in self.datapaths.values():
                            #num_of_switches += 1
                                    self._request_stats(dp)
                            hub.sleep(1)

            #num_of_switches = 0
    """
    def _monitor(self):
            while True:
                    #print len(self.datapaths.values())
                    #num_of_switches = len(self.datapaths.values())
                    for dp in self.datapaths.values():
                            #num_of_switches += 1
                            self._request_stats(dp)
                    hub.sleep(3)

            #num_of_switches = 0

    def _request_stats(self, datapath):
            #self.logger.info('send stats request: %016x', datapath.id)
            ofp = datapath.ofproto
            ofp_parser = datapath.ofproto_parser

            req = ofp_parser.OFPMeterStatsRequest(datapath, 0, ofp.OFPM_ALL)
            datapath.send_msg(req)


    @set_ev_cls(ofp_event.EventOFPMeterStatsReply, MAIN_DISPATCHER)
    def meter_stats_reply_handler(self, ev):

            self.count += 1
            for stat in ev.msg.body:
                    meter.append('DP=%d MeterID=%d Byte=%d duration=%d' %
(ev.msg.datapath.id, stat.meter_id, stat.byte_in_count, stat.duration_sec))

            for stat in ev.msg.body:
                    meters.append('meter_id=%d len=%d flow_count=%d '
                                                'packet_in_count=%d byte_in_count=%d '
                                                'duration_sec=%d duration_nsec=%d '
```

```
                                                        'band_stats=%s' %
                                                        (stat.meter_id, stat.len, stat.flow_count,
                                                         stat.packet_in_count, stat.byte_in_count,
                                                         stat.duration_sec, stat.duration_nsec,
                                                         stat.band_stats))
                        testing.append('id=%d byte=%d' % (stat.meter_id, stat.byte_in_count))
                        mdic.append('%d:%d' % (stat.meter_id, stat.byte_in_count))
                print meter
                print ""
                #self.logger.info('%s', json.dumps(ev.msg.to_jsondict(), ensure_ascii=True,
indent=3, sort_keys=True))
                with open("TMJsonDumps.txt", "a") as f:
                        f.write(json.dumps(ev.msg.to_jsondict(), ensure_ascii=True, indent=3,
sort_keys=True))
```

Fixed-Cycle Processing (Constant number to repeat the query)

Some eventlet (Standard Python Library) wrappers and basic class implementation in ryu.lib.hub are used to create a thread to periodically issue a request to the OpenFlow switch to acquire statistical information.

```
from ryu.lib import hub
```

We have used hub.spawn(), which creates threads. The thread actually created is an eventlet green thread.

```
def _monitor(self):
                while True:
                        #print len(self.datapaths.values())
                        #num_of_switches = len(self.datapaths.values())
                        for dp in self.datapaths.values():
                                #num_of_switches += 1
                                self._request_stats(dp)
                        hub.sleep(3)
```

The hub.sleep() will define the duration of interval the application will query all the switches in the network, and the following class [76] is used to send meter stats requests to all the switches.

```
req = ofp_parser.OFPMeterStatsRequest(datapath, 0, ofp.OFPM_ALL)
                datapath.send_msg(req)
```

In order to handle the replies from the switches, the following class is used where only the desired information are extracted from the replied message.

```
@set_ev_cls(ofp_event.EventOFPMeterStatsReply, MAIN_DISPATCHER)
        def meter_stats_reply_handler(self, ev):

                self.count += 1
                for stat in ev.msg.body:
                        meter.append('DP=%d MeterID=%d Byte=%d duration=%d' %
(ev.msg.datapath.id, stat.meter_id, stat.byte_in_count, stat.duration_sec))
```

For further analysis the application will save the complete replied message from all the switches in a JSON dump, so an analyzer can build a complete TM table. The code is shown in the following example.

```
                with open("TMJsonDumps.txt", "a") as f:
                        f.write(json.dumps(ev.msg.to_jsondict(), ensure_ascii=True, indent=3,
sort_keys=True))
```

C. Openflow 1.3 for OpenWRT in MicroTick Routerboard 750GL

To Implement CPqD Openflow Software switch on a Mikrotik Router board the following procedures are performed [61, 77-79]

First let's install prerequisites:

- sudo apt-get install mini-httpd
- apt-get install build-essential binutils flex bison autoconf gettext texinfo sharutils subversion libncurses5-dev ncurses-term zlib1g-dev gawk
- apt-get install g++ gawk libncurses5-dev git-core
- apt-get install dhcp3-server atftpd apache2
- apt-get install wireshark

Create a folder for keeping Openwrt and OpenFlow Files:

- mkdir /ofwrt13 # do not make the folder at root cause later you are going to have permission issues
- cd /ofwrt13

Downloading the Openwrt Files:

- svn co svn://svn.openwrt.org/openwrt/trunk/
- cd trunk/
- ./scripts/feeds update –a
- ./scripts/feeds install –a

First we compile OpenWRT once and then we are going to add the packages:

- cd trunk/
- make menuconfig
    - Select your platform for Target System (Atheros AR7xxx) # as a precaution
    - Select Subtarget (Mikrotik devices with NAND flash)
    - In Target Images -> select only [ ] ramdisk
- make prereq # good idea to check for prerequisites
- make # we can use make or for speeding up the operation we can use the command below
- ionice -c 3 nice -n 20 make -j 3 #for dual-core CPU put "2"

Now we create the necessary files for adding OpenFlow to OpenWrt image:

- cd ~/ofwrt13
- git clone https://github.com/CPqD/openflow-openwrt.git
- cd ~/ofwrt13/trunk/package/
- ln -s ~/ofwrt13/openflow-openwrt/openflow-1.3/ # be careful about folder name "ofwrt"
- cd ~/ofwrt13/trunk/

- ln -s ~/ofwrt13/openflow-openwrt/openflow-1.3/files
- cd ~/ofwrt13/

Now it is time to create the image:

- cd /trunk
- make menuconfig
  - Select your platform for Target System (Atheros AR7xxx)
  - Select Subtarget (Mikrotik devices with NAND flash)
  - In Target Images -> select only [ ] ramdisk
  - Select tc package under network
  - Select OpenFlow package under network from main menu
  - Select kmod-tun under Kernel Modules->Network Support
  - save and exit
- make kernel_menuconfig
  - Under Networking Support->Networking options->QoS and/or fair queueing select Hierarchical Token Bucket (HTB)
  - Save and Exit
- ionice -c 3 nice -n 20 make -j 3

Now that we have the RamDisk image, we should create a tar.gz as following:

- make menuconfig
  - In Target Images -> select only [ ] .tar.gz
  - save and exit
- ionice -c 3 nice -n 20 make -j 3

Now we should set up our pc interface as follow:

- Edit /etc/network/interfaces, use 192.168.1.X as a static IP address # since defualt openwrt address is 192.168.1.1, avoid using it.
  - It should be something like below.

```
auto lo
iface lo inet loopback
auto eth1
iface eth1 inet static
address 192.168.1.3
netmask 255.255.255.0
```

- Next Edit /etc/default/atftpd, changes in bold:

```
USE_INETD=false
OPTIONS="--bind-address 192.168.1.3 --tftpd-timeout 300 --retry-timeout 5 --mcast-port
1758 --mcast-addr 239.239.239.0-255 --mcast-ttl 1 --maxthread 100 --verbose=5 /srv/tftp"
```

- Next Edit /etc/dhcp/dhcpd.conf, and use the first listed MAC address as the hardware ethernet address:

```
authoritative;
allow booting;
allow bootp;
one-lease-per-client true;

subnet 192.168.1.0 netmask 255.255.255.0 {
 option routers 192.168.1.254;
 option subnet-mask 255.255.255.0;
 option broadcast-address 192.168.1.255;
 ignore client-updates;
}

group {
 host routerboard {
   hardware ethernet d4:ca:6d:b5:6a:53;
   next-server 192.168.1.3;
   fixed-address 192.168.1.99;
   filename "openwrt-ar71xx-mikrotik-vmlinux-initramfs.elf";
 }
}
```

- and copy the file to
  - cp ~/ofwrt13/trunk/bin/ar71xx/openwrt-ar71xx-mikrotik-vmlinux-initramfs.elf /srv/tftp/

Setup the Host

An HTTP web server is required on the host, e.g. mini-httpd in Ubuntu. Install the mini-httpd web server

- sudo apt-get install mini-httpd
- Edit /etc/default/mini-httpd change the following

```
# Start daemon?
# 0 = no
# 1 = yes
START=1
```

- Edit /etc/mini-httpd.conf and change

```
# On which host mini_httpd should bind?
host=192.168.1.3
```

```
# Run in chroot mode?
chroot # yes
#nochroot # no
# We are the web files stored?
# Please change this to your needs.
data_dir=/home/arman/Img-ready-to-flash/
# I suggest to copy the files in a separate folder, so you can keep a track of them
# so copy following two images to your desired folder
#openwrt-ar71xx-mikrotik-DefaultNoWifi-rootfs.tar.gz // remove DefaultNoWifi if it has it in
the #name
#openwrt-ar71xx-mikrotik-vmlinux-initramfs.elf
#openwrt-ar71xx-mikrotik-vmlinux-lzma.elf
```

- Restart the mini-httpd web server and other services
  - sudo /etc/init.d/mini-httpd restart
  - sudo /etc/init.d/networking restart
  - sudo /etc/init.d/atftpd restart
  - sudo /etc/init.d/isc-dhcp-server restart

Now it is time to install the image on router board

Run Wireshark, sniffing on eth1, to watch network traffic.
- sudo wireshark # without sudo you won't see the ports

Boot the RouterBOARD 750GL via TFTP:
- Connect an ethernet cable between port 1 on the RouterBOARD 750GL and the TFTP server.
  - Other ports won't work
- Press the small "RES" button and plug in the power cable.
- The "PWR" and "ACT" lights will illuminate. Then "ACT" will flash, and finally stop.
- Now release the "RES" button.
- The 750GL should request a DHCP address, receive 192.168.1.99, download openwrt-ar71xx-nand-vmlinux-initramfs.elf via TFTP, and boot OpenWRT.

Use Wireshark to ensure everything's working; look for the TFTP "DATA Packet" packets to verify the ramdisk download.

The device will boot OpenWRT and use IP address 192.168.1.1.

Swap the cable to a different port to access OpenWRT. I used port 2 (port 1 won't work). Then telnet to the device:

- telnet 192.168.1.1

Now that the 750GL has booted to ramdisk, the last step is flashing OpenWRT to make it permanent.

- wget2nand http://192.168.1.3

```
root@OpenWrt:/# wget2nand http://192.168.1.3
Connecting to 192.168.1.3 (192.168.1.3:80)
kernel              100% |*******************************|  2717k  0:00:00 ETA
Connecting to 192.168.1.3 (192.168.1.3:80)
rootfs.tgz          100% |*******************************|  1222k  0:00:00 ETA
Erasing filesystem...
Mounting /dev/mtdblock2 as new root and /dev/mtdblock1 as kernel partition
Copying kernel...
Preparing filesystem...
...
Cleaning up...
Image written, you can now reboot.  Remember to change the boot source to Boot from
Nand
```

Now that we have openwrt running lets connect the router to the internet and install necessary packages.

Let's first create a password for the router so we can connect to it using ssh.

- passwd

After you changed the password the connection will reset and it won't accept telnet anymore so use ssh to connect to the router

- ssh -l root 192.168.1.1

It is easier to prepare the configuration in the host and then download it to the router. Now from the host copy the network config files to the router

- scp network root@192.168.1.1:/etc/config/

Sample of one of the device configuration:

```
config interface 'loopback'
        option ifname 'lo'
        option proto 'static'
        option ipaddr '127.0.0.1'
        option netmask '255.0.0.0'

config globals 'globals'
        option ula_prefix 'fd1c:640f:6cb2::/48'

config interface 'lan'
        option ifname 'eth0.1'
        option type 'bridge'
        option proto 'static'
        option ipaddr '192.168.1.1'
        option netmask '255.255.255.0'
        option ip6assign '60'

config interface 'lan2'
        option proto 'static'
        option ifname 'eth0.2'
        option ipaddr '192.168.2.13'
        option netmask '255.255.255.0'
        option gateway '192.168.2.20'ç




        option broadcast '192.168.2.255'

config interface 'lan3'
option proto 'static'
option ifname 'eth0.3'

config interface 'lan4'
option proto 'static'
option ifname 'eth0.4'

config interface 'wan'
     option ifname 'eth0.5'
     option proto 'static'
     option ipaddr '147.83.118.147'
     option netmask '255.255.0.0'
     option gateway '147.83.117.1'
     option dns '147.83.2.3'

config switch
option name 'switch0'
option reset '1'
option enable_vlan '1 2 5'

config switch_vlan
option device 'switch0'
option vlan '1'
option vid '1'
option ports '0t 1'

config switch_vlan
option device 'switch0'
option vlan '2'
option vid '2'
option ports '0t 2'

config switch_vlan
option device 'switch0'
option vlan '3'
option vid '3'
option ports '0t 3'

config switch_vlan
option device 'switch0'
option vlan '4'
option vid '4'
option ports '0t 4'

config switch_vlan
option device 'switch0'
option vlan '5'
option vid '5'
option ports '0t 5'
```

Now we have internet and we should be able to ping outside world.

Connect to the router and start openflow:

- /etc/init.d/openflow start

You should see something like this

```
root@OpenWrt:/etc# /etc/init.d/openflow start
eth0.1,eth0.2,eth0.3,eth0.4
Configuring OpenFlow switch for out-of-band control
RTNETLINK answers: No such file or directory
Dec 19 18:04:16|00001|datapath|ERR|eth0.2 device has assigned IP address
192.168.2.13
```

Please note that after r34793 /etc/functions.sh → /lib/functions.sh so if you are using an old version change it!

I simply copy the function.sh from lib to etc. so you can do the same otherwise you will get an error

- /sbin/ofdown: .: line 4: can't open '/etc/functions.sh'
- /sbin/ofup: .: line 5: can't open '/etc/functions.sh

The openflow config file is located at /etc/config/openflow, and looks something like this

```
config 'ofswitch'
        option 'dp' 'dp0'
        option 'ofports' 'eth0.1 eth0.5 eth0.3 eth0.4'
        option 'ofctl' 'tcp:192.168.2.20:6633'
        option 'mode'  'outofband'
```

Change the controller ip to the ip of your computer which host the controller.

Useful reminder about some of files and folders location:

- /etc/config/openflow # openflow config file
- /etc/config/network # network config file
- /lib/openflow # Openflow Bash files
- /lib/functions.sh # copy this tp /etc/