Network Monitoring with Software Defined Networking

Towards OpenFlow network monitoring

Vassil Nikolaev Gourov

Master of Science Thesis

Delft University of Technology

Network Architectures and Services

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science
Network Architectures and Services Group

Network Monitoring with Software Defined Networking

Towards OpenFlow network monitoring

Vassil Nikolaev Gourov
4181387

August 30, 2013
M.Sc. Thesis No:

Abstract

Network monitoring is becoming more and more important as more Internet Service Providers and Enterprise networks deploy real-time services, like voice and video. Network operators need to have an up-to-date view of the network and to measure network performance with metrics like link usage, packet loss and delay, in order to assure the quality of service for such applications. Obtaining accurate and meaningful network statistics helps the service providers to estimate the "health" of their network, to find service degradation due to congestion and could even use them for routing optimization. Finally, a more accurate picture of the nature of the Internet traffic is important for continued research and innovation.

In this thesis, Software Defined Networking is used as a unified solution to measure link utilization, packet loss and delay. Currently, there is no single solution capable to measure all the mentioned metrics, but a collection of multiple different methods. They all need separate infrastructure which needs additional installations and expenses. Furthermore, those methods are not capable to meet all the requirements for a monitoring solution, some are not accurate or granular enough, others are adding additional network load or lack scalability.

Software Defined Networking is still in an early development stage. Unfortunately, obtaining network statistics is a problem that has not been discussed very much. Nevertheless, OpenFlow protocol is already gaining a lot of popularity and it could be used as a unified monitoring solution. Its built-in features are already sufficient to provide accurate measurements. This thesis explores what are the current monitoring methods with and without Software Defined Networking and how monitoring could be archived in an OpenFlow enabled topology. It proposes methods to measure link usage, packet loss and delay. In terms of link usage statistics several approaches are proposed to reduce the generated overhead. Furthermore, the thesis also suggests how to measure how packet loss in networks that use OpenFlow.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1-1 Research Motivation

Since the establishment of the first point-to-point communication in the 1950s, data communications have experienced fast development, and have on their own facilitated research and technology advancement worldwide. The ARPANET, which is considered the predecessor of today's Internet, was founded during the end 70s and early 80s. When the networks that are the foundation of the Internet were first designed, they were very simple (i.e. Ethernet, invented 1973 and was commercially introduced in 1980 [27]). Multiple inter-networking protocols, which we still use, were introduced during those initial days, such as the Internet Protocol (IP) and the Transmission Control Protocol (TCP). Throughout the years, people have discovered the application of the Internet and its penetration worldwide has grown exponentially. Its success is undeniable, it has now become a part of our daily life and billions of people worldwide use it [20]. IP networks are now also widely used for business needs, such as e-commerce, online banking, etc.

The network and protocol designers considered only the services and applications available at that time, such as mail, telephony and file transfer. They tried to build an architecture where resources are socially shared with other users. Unfortunately, there are some design drawbacks that need to be considered. What was simple and straightforward in the beginning started to change slowly and today the Internet has evolved into a huge structure, an interconnection of thousands of networks. A lot of new services have been introduced, such as video streaming, social networking, online gaming, bank transfers, etc. Every time a new service was introduced, it came with new control requirements, which in turn led to increasing network complexity. Currently, networks based on the old design have become too complicated and hard to manage, they are also too expensive [5]. Furthermore, there is no guarantee that any new architecture would not result in a similar problem a decade from now. Every time when a new infrastructure capable of solving the past problems was introduced, new problems came up. A solution that is able to meet the future requirements as they arise is needed. This is where the philosophy of Software Defined Networking (SDN) may play an important role.

New services need an environment that is capable to dynamically adjust to their demands, that is capable to provide them more than best effort point-to-point interconnection. While this has been of a minor importance during the past years, at a certain point in time people and organizations will get frustrated with the current best effort services. The performance of applications depends on the efficient utilization of network resources. Currently, Internet Service Providers (ISPs) have simply oversubscribed additional capacity. In the past this has been a possible way to avoid network congestion, packet delays and losses [35]. Unfortunately, the dramatic increase of Internet traffic in the past years suggests that this approach is not feasible for the future as demands are growing exponentially. For example, one service that has experienced an incredible growth during the last couple of years is video streaming over the Internet. Its popularity has increased with an impressive rate up to the point that it forms about 50 percent of the Internet traffic [47]. Additionally, it continues growing and it is expected to reach 90 percent over the next years [4]. People constantly use the Internet to watch TV shows, video tutorials, amateur videos, live events, etc. Such applications require certain Quality of Service (QoS) guarantees to work properly.

A key aspect for network management, in order to reach QoS and for tasks like network security and traffic engineering, is accurate traffic measurement. During the last decade this has been an active research field, as it has become extremely difficult to obtain direct and precise measurements in IP networks due to the large number of flow pairs, high volume of traffic and the lack of measurement infrastructure [60]. Current flow-based measurements use too many additional resources (i.e. bandwidth and central processor unit (CPU) time), and other existing monitoring solutions require additional changes to the configuration of the infrastructure, which is expensive, again bringing additional overhead. It is now obvious that there is a need of management tools capable to provide an accurate, detailed and real-time picture of what is actually happening on the network and, in the same time, are cheap and easy to implement.

1-2 Research Question

Whenever network management is considered there are two main stages: monitoring and control. Thus, the first step to achieve a scheme capable of battling those problems is to design a suitable monitoring solution. The problem that this thesis addresses is how to monitor network utilization efficiently in real time in the context of SDN and more specifically OpenFlow. Hence, the main goals of this thesis are:

- Use Software Defined Networking as a lever to meet the future networking demands,
- by designing a monitoring solution capable to measure network utilization, delay and packet loss and
- evaluate it via an implementation using OpenFlow.

Subsequently, the research goal of this thesis is to provide an answer to the following questions:

- How can monitoring be achieved with Software Defined Networking?
- What kind of improvements could SDN bring compared to present solutions?

1-3 Thesis Structure

This thesis follows a standard research and development method. Initially, the problem is formulated after investigation of the existing work and a review of the SDN technology. Chapter 2 goes trough the development of the SDN during the years, followed by a preliminary research of the SDN philosophy and current advances. Chapter 3 deals with a literature review of the available network monitoring solutions. Once the investigation is finished, it is time to build a conceptual architecture and a prototype. Chapter 4 focuses on the implementation of a network monitoring solution in a real test environment using the OpenFlow protocol. First, an overall architecture of the building components is given and each component is discussed separately. Chapter 5 answers the question on how traffic monitoring should be achieved in SDN. The chapter starts with describing the design objectives and assumptions. It explains in details the architectural trade-offs. Last, in Chapter 6 the monitoring prototype is evaluated. The chapter describes multiple evaluation scenarios, carefully designed to measure different aspects of the monitoring solution. The final chapter, Chapter 7 presents conclusions and future work.

Chapter 2

Software Defined Networking

The goal of this chapter is to provide insight on the philosophy of a new network paradigm, Software Defined Networking. The chapter starts with a brief summary of the history and findings that led to recent advances in the field, explaining why SDN is needed and what inherited problems it should overcome. Secondly, an explanation of the SDN basics is provided. The chapter finishes with an overview of OpenFlow, the protocol that is currently considered as the SDN standard.

2-1 Network Evolution and Problems

Throughout the past decades the Internet has changed the way people live. Users enjoy its simplicity and the ability to communicate world-wide. While using the Internet still remains the same for the end users with almost no major changes, the underlying infrastructure has undergone a significant change. New services that require certain network quality have emerged and the ISPs had to adapt. It can be safely said that networking has been a transformative event since the beginning. The overall architecture turned out to be a big success due to its simplicity and effectiveness, but the supporting network infrastructure has been slowly becoming a problem. Networks are growing large and need to support a lot of new applications and protocols. Currently, they have become complex to manage which results in expenses due to maintenance and operations, as well as, in network-downtime due to human errors [28] or due to network intrusions [15]. Furthermore, certain new services are not capable to operate in a best effort (BE) environment (i.e. voice and video delivery), where resources are socially shared, resulting in delays and losses.

To meet the demands of the emerging new applications, a lot of efforts have been made. Several new techniques and changes in the infrastructure architecture were suggested, each time the network operators had to invest substantial amount of money (e.g. MPLS or IPv6 enabled devices). Furthermore, a lot of work has been done in the direction of integrating new services in the old architectures, that were never designed with those applications in mind.

There are some inherited limitations (i.e. slow service development and upgrade cycle), which make it impossible to meet current market requirements. With every new product on the market the network complexity increases and instead of facing the real problem, it is avoided. The current network architecture relies on devices where the control plane and data plane is physically one entity, the architecture is coupled to the infrastructure. Furthermore, every node within the current network architectures needs to be separately programmed to follow the operator's policies. It can be safely stated that network control is too complex. Additionally, the companies that provide network devices had full control over the firmware and the implementation of the control logic, as a result it is very difficult to assure vendor inter-operability and flexibility, sometimes close to impossible. The constraints that limit the networks evolution include:

- **Complexity**: Over the past decades a lot of new protocols were defined each one solving a specific network problem. As a result, for any changes in the network topology or implementation of a new policy, the network operators need to configure thousands of devices and mechanisms (update ACLs, VLANs, QoS, etc.) [41].

- **Scalability**: The demand for data grows exponentially, traffic patterns are changing and are no longer predictable. Due to these changes, the cloud computing and the spawn of new bandwidth hungry applications (i.e. content delivery), during peak hours the demand for bandwidth reaches a level, dangerously close to the maximal network capacity. However, having complexity problems also leads to scalability problems, because their networks are no longer capable to continue growing at the same speed. Furthermore, network providers are not be able to continue investing into new equipment endlessly as they have already been heavily investing during the past decades into infrastructure. Meanwhile, the innovation cycle speeds up, which means additional expenditures for more hardware, but the service prices are shrinking [41].

- **Dependability**: The network operators need to tailor the network to their individual environment, which requires inter-vendor operability, but very often such operability is not provided by the equipment vendors.

In [11] is stated that an ideal network design should be exactly the opposite of what current networks have become, or simply should involve hardware that is: Simple, Vendor-neutral and Future-proof. The software remains agile, which means, capable of supporting all the current network requirements (i.e. access control, traffic engineering). At the same time, it should be able to meet any future requirements. New network architectures are proposed, however the initial cost to implement them over the already existing infrastructure is too big. The lesson is that network operators need a solution that can be implemented without changing too much infrastructure and spending a lot of money. This is where SDN kicks in. The SDN approach tries to solve the issues for the network providers by introducing a more centralized control system for the whole network.

2-2 Overview

2-2-1 History

The idea of a centralized network architecture is not something new. A lot of research initiatives have been working on the idea of split architectures during the past three decades. It goes back to the 1980s when Network Control Point (NCP) [46], a circuit switched network was introduced. Up until then all call control was managed by the circuit switches the calls went through. However, there were a number of problems, like limited processing power of the single switches, limited visibility of the network the switch had and the limited amount of upgrades the switch could have. These problems were addressed by the NCP, which simply is a processing platform external to the switch. A similar approach is later used as basis for many voice features and is still in use today, for example, within the SS7 protocol suite.

At the end of the 90s another project emerged known as the Tempest [33]. It suggests a way to delegate control to multiple parties over the same physical ATM network. The project revolves around the idea that every service has specific needs and would require separate control architecture. The main goal is to split the network into multiple virtual networks, which could be used in a way suitable for a specific application. Shifting the question of how to fit the new service within the current architecture to what is the best architecture for some particular new service.

Around a decade ago AT&T's researchers started working on a platform called "IP-NCP", a platform separated from the infrastructure, where routing decisions could take place. Subsequently, 8 years ago, what is now known as the Routing Control Platform (RCP) [8] became reality. RCP selects the Border Gateway Protocol (BGP) routes for an autonomous system (AS) by collecting information about the topology and the external destinations. Afterwards, it is capable to perform route selection and assign them to the routers using the iBGP protocol. This approach solves the scalability problem that current IP networks have with BGP. Approximately at the same time frame, a research paper suggesting a new conceptual architecture called 4D [19] was published. Together with the RCP the 4D architecture could be considered to be the forefathers of the SDN notion and the hype that followed afterwards.

More recently, the notion has gained even more widespread fame with the adoption of OpenFlow (OF) [31] (see section 2.2.3) and the SANE [10] and Ethane [9] projects. Those two projects defined an enterprise architecture and they are the predecessors of OpenFlow as it is known today. The RCP concept is also re-examined using SDN abstraction and the specifics of the OF protocol. In [45] a prototype of the RouteFlow Control Platform (RFCP) is presented. The authors suggest a hybrid model to assure smooth migration towards SDN and to verify its efficiency, aBGP (Aggregated BGP) is implemented. The paper suggests use cases which can be achieved by using the RFCP platform, such as Traffic Engineering and Optimal Best Path reflection.

2-2-2 Architecture

In principle, network equipment is logically composed of a data and control plane, each of them accomplishing a special task. The data plane is also known as forwarding plane and takes care of forwarding packets through the network device, while the control plane is concerned with the decision logic behind. It uses the forwarding policies defined by the ISPs for each packet, to take decision which is the next hop or/and final destination. Hence, the underlying infrastructure could be split into two components: hardware and software that controls the overall behaviour of the network and its forwarding layer.

The Software Defined Networking approach decouples the control plane from the network equipment and places it in a logically "centralized" network operating system (NOS), also very often referred to as controller. A protocol is used to interconnect the two separated planes, providing an interface for remote access and management. Figure 2-1 illustrates the basic SDN architecture. The architecture varies with the implementation and depends of the type of network (i.e. data-centre, enterprise and wide area network) and its actual needs. The physical infrastructure consists of switching equipment capable to perform standard tasks, such as buffering, queueing, forwarding, etc. Although, referred to as physical environment, many network deployments use also software switches (virtualization, mainly in data-centres with virtual servers). While some discussion are going on how exactly the control plane should be achieved (distributed or centralized), a common agreement is reached that it "decides" what has to happen within the network, either proactively or reactively. Finally, the controller could be closed or could provide Application Programmable Interface (API). The main idea behind SDN is to abstract the architecture and provide an environment where it is easy to build new applications. This would reduce the development time for new network applications and allow network customization based on specific needs.

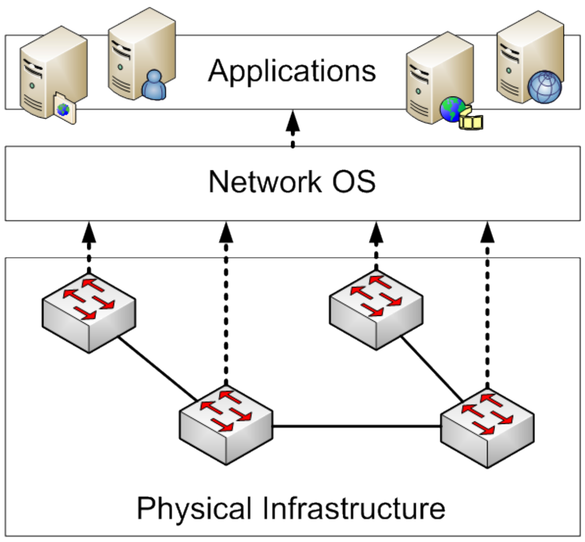


Figure 2-1: Basic Architecture of a Software Defined Network

Such architecture tries to achieve the following goals:

- **Interoperability**: Centralized control over the SDN enabled devices from any vendor throughout the whole network.
- **Simplicity**: the complexity issues are eliminated. The network control is easier and more fine grained, which results in an increased reliability and security, in a more granular network control. Flow based control allows control over each session, user, device and application.

- **Innovativeness**: The abstraction of the network services from the network infrastructure allows network operators to tailor the behaviour of the network and program new services faster than before. The entire structure becomes much more evolvable.

Unfortunately, there are also some issues that need to be faced. The very centralized nature of the SDN architecture raises scalability doubts. The main problem in such a centralized architecture is the bottleneck that could be formed between the infrastructure and the controller. Furthermore, there are also concerns for a bottleneck with the switching equipment, in terms of forwarding capacity (table memory) and the overhead that could be created by constant reactive invocation of the control plane. This problem brings discussions about how distributed/centralized the SDN architecture should be.

2-2-3 OpenFlow

OpenFlow [31] is an implementation of a communication protocol that interconnects the forwarding with the data plane. It enables part of the control operations to run on an external controller. The architecture is based on the SDN paradigm, it uses flow tables installed on switches, a controller and a protocol for the controller to talk securely with switches (encrypted using TLS). Figure 2-2 presents the basic scope of an OF enabled switch. The idea is to manage the switches (more precisely the switch flow tables) remotely using the OpenFlow protocol. This protocol allows the controller to install, update and delete rules in one or more flow tables, either proactively or reactively. The following subsection describes each component separately, the switches and the controllers and their interaction.

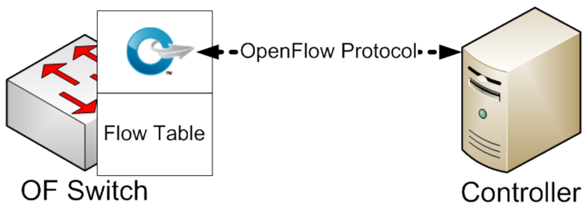


Figure 2-2: OpenFlow Switch Architecture [17]

Data-Plane

The flow table of each OF switch contains a set of match fields, counters and instructions. The header of every packet that enters the switch, together with the ingress port, is compared against the flow table. If there is a match, the switch executes the specified actions (i.e. forwards the packet). In case there is no match the controller needs to take decision and install a new flow table entry. The switch sends a flow requests message to the controller containing the new packet header. The controller takes decision on what to do with the packet and sends instructions to the switch. This new rule specifies the actions that need to be taken for this packet. The whole process could be summarized in four steps and is illustrated in Figure 2-3.

A **match** is whenever the values in the header fields of the new packet are the same as those defined in the flow table. If a field is assigned the value of ANY then it matches any header

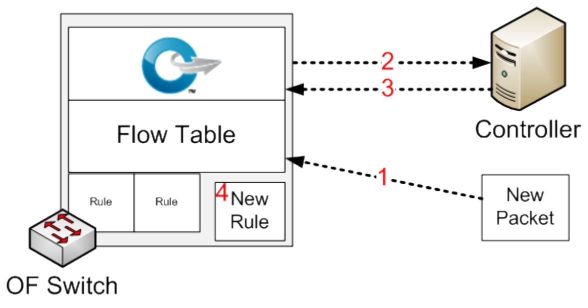


Figure 2-3: OpenFlow flow setup [57]

values. Table 2-1 gives the fields of the new packet that are compared to the switch flow tables. There might be multiple matches, then the entry with the exact match has higher priority over the wildcard entries. If there are multiple entries with the same priority, the switch is allowed to choose between them [17].

Table 2-1: Fields from packets used to match against flow entries [17]

Type
Ingress Port
Metadata
Ethernet source address
Ethernet destination address
Ethernet type
VLAN id
VLAN priority
MPLS label
MPLS traffic class
IPv4 source address
IPv4 destination address
IPv4 protocol/ARP opcode
ToS bits
Transport source port/ICMP type
Transport destination port/ICMP code

During the last years the research community has focused its efforts towards SDN and the OF protocol. A new non-profit consortium was launched in 2011 known as The Open Networking Foundation (ONF). Its main goal is to establish a SDN ecosystem, by introducing standards and solutions. By the end of 2012 it had already published OF switch specification version 1.3.1 [17], describing the requirements for an OF switch.

Multiple equipment vendors (like Brocade, IBM, NEC) have introduced OpenFlow enabled switches to the market. There are also some options for building up a OF switch for research purposes. First, in [37] there is a description on how to build an OF enabled hardware switch using the NetFPGA platform. Linux Ethernet Switches can also work with OF protocol, but the performance is limited. Additionally, OF is not limited to hardware only, there are multiple projects for software switches. The one that remains most cited and is compatible with most of the other projects is the Open vSwitch [44]. An open source software switch targeted at virtual server environments. A more detailed list of commodity switches and software switches that are complaint with OF can be seen in [32].

Control-Plane

The main task for the controller is to add and remove entries from the switch flow-tables. Figure 2-4 provides a scheme of a basic controller. The controller interacts with a set of switches via OpenFlow using the so called Southbound interface. For security reasons it supports encrypted TLS and exchanges certificates with every switch. This varies with the implementation as it is not specified. The controller is responsible for all the essential functions such as service and topology management. It could be enhanced with additional features or it could provide information to external applications. Currently, the so called northbound communication is not standardized. Some efforts are made to enhance the abstraction level by designing network programming languages on top of the controllers. Examples of such languages are Frenetic [16] and Procera [55]. Finally, via the West and Eastbound interfaces the controller is capable to communicate with other controllers. There are already a lot of proposals for this interaction (e.g. Hyperflow [52]).

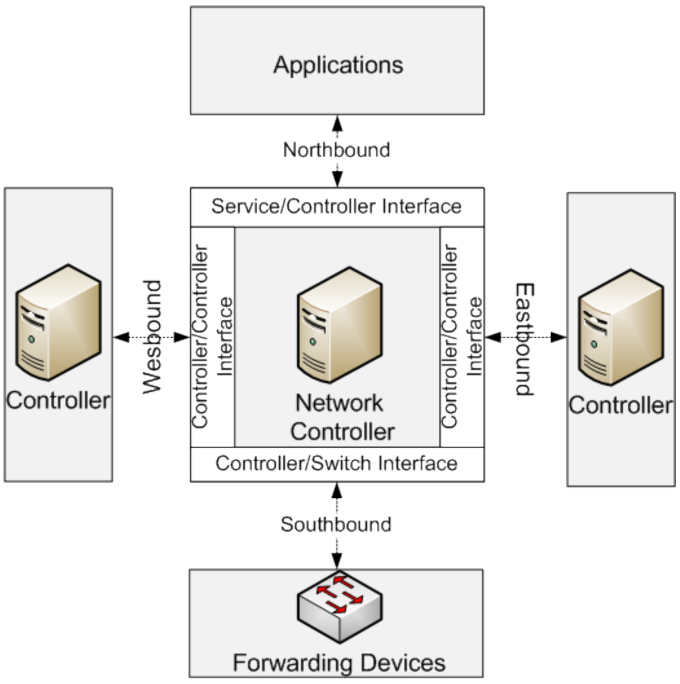

Applications
Northbound
Service/Controller Interface
Controller
Westbound
Controller/Controller Interface
Network Controller
Controller/Controller Interface
Eastbound
Controller
Controller/Switch Interface
Southbound
Forwarding Devices

Figure 2-4: Scheme of OpenFlow controller

A lot of designing and implementation efforts are focused on the controller, since it is the most important element of the whole architecture. Throughout the past years a number of new controllers have been released. Table 2-2 depicts the most important existing open-source controller projects, released for research and testing purpose (a more detailed list can be seen in [32]). NOX [21] is the first OpenFlow controller. It has been used as a base for new controllers and numerous experimental applications use it. It is designed for a campus network. The NOX creators made both C++ and Python interfaces, which made it hard to use and a lot of users were only interested into using the Python interface. POX [30] was designed to address that, to create a framework that makes it easier to write new components. Floodlight is also picking up a lot of steam, an open source Java-based controller released by BigSwitch Networks with the help of Beacon's creator. It could be said that it is the first open source commercial controller. It is capable to work with both physical and virtual switches. Finally, Onix [29] is a framework for a distributed controller which runs on a cluster of one or more physical servers, each of which may run multiple instances. Based on the foundations of the Onix controller new proprietary OpenFlow controllers are under development by several companies (NEC, BigSwitch Networks, Nicira Networks, Google and Ericsson). There are a lot of projects dealing with different problems the SDN architecture faces. The main ongoing discussions are about centralized vs distributed state (affecting the granularity) and reactive vs proactive decisions. The only conclusion that can be drawn at this moment is that there is not a one size fits all solution, as different approaches are more suitable for specific application environments (i.e. data centres or enterprise networks).

Table 2-2: Open-Source OF Controllers

Name	**Language**	**Developer**
Beacon [14]	Java	Stanford University
Floodlight [40]	Java	Big Switch Networks
Maestro [61]	Java	Rice University
NOX [21]	C++, Python	Nicira
POX [30]	Python	Nicira

Applications

While the exact SDN architecture is not yet decided and it remains under investigation, another hot topic is the applications that could be designed around it. A lot of research efforts are targeting to integrate old applications that proved to be useful within the legacy networks, but were also rather expensive or complicated to implement in the production networks. This section briefly presents some of the applications currently being built over the existing open controllers or considered by the industry. The idea is to illustrate the wide use of OF in different network environments.

For **enterprise and campus networks** a network management application which is capable of enforcing **security** policies such as access control, packet inspection. For example, SANE [10] inspects the first packet of every new flow that enters the network. For **flow control** and traffic engineering by flow prioritization, route selection, congestion control, load balancing. There are some examples of OpenFlow applications deployed in campus networks like Ethane [9] and wireless networks like Odin [50]. Multiple projects are already offering network researchers an open source software router. Projects like RouteBricks, Click, Quagga and XORP bring benefits to the network community by offering extensible forwarding plane. QuagFlow [39] is built with the idea to partner Quagga with OpenFlow. Since then it has evolved and is now known as RouteFlow (RF) [38]. It uses a virtual view of the topology where the routing engine calculates the data path and executes the resulting flow commands into the real network topology. There are also some suggestions for **network monitoring** applications that collect network utilization statistics for the entire network [53].

Data centres also found use for SDN and OpenFlow. Server virtualization is already years ahead and the next logical step is the network virtualization. SDN would enable hypervisor scalability, automated virtual machine (VM) migration, faster communication between the multiple virtual servers, bandwidth optimization, load balancing [56], energy consumption [23]. For example, MicroTE [6] uses real-time host-based traffic monitoring to take appropriate routing decisions every second for both what is considered predictable and unpredictable traffic. Routing decisions are taken using a weighted form of Equal Cost Multipath (ECMP) approach along the K equal hop length paths between the source and the destination or based on bin-packing heuristics. Furthermore, for the **infrastructure networks** it would make bandwidth calendaring and demand placement possible. Those uses are necessary for data centre replication or for any service that requires guarantees and network resource coordination (content delivery or cloud services).

Chapter 3

Network Monitoring

Using Software Defined Networking could solve some of the current monitoring problems in today's IP networks. Furthermore, there is insufficient work on how measurements should be achieved in an OpenFlow enabled network. This chapter discusses basic traffic measurement approaches used today by ISPs. Initially, it provides definition of network metrics, as well as, their errors and uncertainties. The chapter ends with an overview of the current monitoring approaches. Furthermore, it reviews not only the current monitoring methods, but the proposed SDN approaches, as well.

3-1 Measurements

Every day millions of computers connect to the Internet, generating a huge amount of traffic. This process is dynamic in the sense that the amount of traffic is still increasing, changing in size and new applications are also constantly joining the flow. While this process continues, it is fundamental for ISPs to keep up to date knowledge about what is going on within their part of the network. Measuring the traffic parameters provides a real view of the network properties, an in-depth understanding of the network performance and the undergoing processes. Network monitoring is crucial for QoS criteria and assures that the network systems function properly, therefore granting Service Level Agreements (SLA) in a dynamic, constantly changing environment. The ability to obtain real traffic data allows to analyse network problems, generate traffic matrices, optimize the network using traffic engineering techniques or even upgrade it based on future predictions. Finally, a proper network view allows the routing algorithms to take more appropriate decisions, increasing the resource utilization and decreasing the congested nodes/links.

3-1-1 Metric Definition

An explanation of what to measure is needed, before diving into measuring classifications and monitoring tools that are out there. This subsection tries to give a proper definition for all the metrics that will be used later in this thesis. The definitions for delay and packet loss are based on the IPPM Framework [43].

Link Usage and Utilization: Link usage is expressed as the number of correctly received IP-layer bits sent from any source and passing trough link L during the interval of time T. There is a distinction between capacity and usage. While the link capacity is a theoretical value, a representation of the maximum link usage, the link usage is a variable number that changes. It is the average rate of bits that go trough the link and are correctly received by the other node attached to it. Therefore, the link utilization is expressed as the current link usage over the maximum link capacity and is a number from zero to one. When the utilization is zero it means the link is not used and when it reaches one it means the link is fully saturated [12]. Throughout this thesis the term utilization is used as a synonym for link usage and is presented with the number of bytes that pass trough a link in a specific moment.

Packet delay: Some applications are directly influenced by the network delay, their performance degenerates under large end-to-end delay, which makes it an important metric for proper QoS and user experience assurance. The total network delay is composed of propagation delay, processing delay, transmission delay and queueing delay. Propagation delay depends on the signal speed and the link's length. Processing delay is the time it takes each forwarding node to find a match in the routing table and move the packet from the ingress to the egress interface. Transmission delay is the time it takes the node to push the entire amount of packet bits (counts from the first bit that reaches the transmission line) on the link. Finally, queueing delay is the time packets spend waiting in buffers and queues due to prioritizing certain traffic classes or congested links.

Delay is the total time it takes for a packet to travel from a source node to its destination node. The time measurement starts as soon as the first bit is sent and finishes counting when the last bit is received by the destination. Therefore, the one-way delay is the difference ΔT between the time T_1 the packet started transmitting and the time T_2 when the packet was entirely received [1].

Delay Uncertainties and Errors; Delay is extremely difficult to measure since it can be as 10μs up to millisecond range. Therefore, measuring nodes need accurate clock synchronization and often they should rely on Global Positioning System (GPS) clocks instead of Network Time protocol (NTP) servers. There are four sources of error and uncertainty in today's network [36]. Accuracy defines how close a given node's clock is to the coordinated universal time (UTC). Resolution is how precisely a clock can measure time, i.e. some clocks are capable to measure only with millisecond precision. When multiple clocks are involved they can disagree on what time it is, therefore they have synchronization issues. Finally, over time clocks are drifting, thus influencing their accuracy and synchronization.

Packet loss is the event when certain amount of packets sent from a source node fail to reach their destination node. This may be caused by link failures and changes in the topology or by network congestion. One-way packet loss is expressed as a number from 0 to 1 (sometimes also presented in percent). It is exactly one when all the packets do not reach their destination and zero when all were correctly received [2].

There are some additional terms that are related to the direct measurement of those metrics and are frequently used in the following chapters.

A **flow** is a unidirectional data communication between two applications that is uniquely defined by the following five identificators: source IP, destination IP, the number of the transportation protocol that is used and, if available, the source and destination ports. A **link** is the physical interconnection between two neighbouring nodes.

Traffic Matrix(TM); A must have for Traffic Engineering (TE) is the Traffic Matrix. Simply said, it contains information about the amount of data transmitted between each pair of network nodes. Obtaining the traffic matrix has been a big problem for the last decade, mainly because the network measurement techniques that are used in today's IP networks have some limitations. For example, they lack scalability or granularity. Furthermore, the best method that could be used for TM estimation is based on direct measurements, but it is prevented by the demand estimation paradox. Different source-destination pairs, often have a number of common links and it is hard to separate which traffic goes where. Once the TM is known, there are various possibilities for network improvements, such as, increase the link bandwidth, implement load balancing, fine tune and optimize the Interior Gateway Protocol (IGP) metrics and change the BGP exit points.

3-1-2 Types of Measurements

Traditionally, many different measurement techniques are used. The main types of measurements and the trade-offs they bring are discussed in this subsection. It is important to note that OpenFlow already provides means to implement any of the methods or combine them if needed, while traditionally every type of measurement requires separate hardware or software installations.

Active vs Passive methods

In the world of network measuring there are two distinct groups of measurement methods: passive and active methods. Passive measurement methods consist of counting the network traffic without injecting additional traffic in the form of probe packets. Those methods are good in the sense that they do not generate additional network overhead, unfortunately, the reliability of the gathered data depends on installing additional hardware (traffic monitors). On the other hand, active measurements are achieved by generating additional packets. They are sent trough the network, where their behaviour is monitored. Some of the most well known active measurement methods are traceroute and ping.

Both active and passive measurement schemes are useful for network monitoring purposes and for collecting large amounts of statistical data. However there are several aspects that need to be taken into account when designing a new monitoring application: overhead and accuracy. For example, active measurements contribute with additional network load. Additional traffic could cause inaccurate results and it is unclear how unbiased those methods are.

Application and Network methods

There is also difference on which Open System Interconnection (OSI) layer the measurements are being taken. Network layer measurements use infrastructure components (i.e. routers and switches) to account for statistics. Since the Network layer is responsible to assure end-to-end delivery it can provide more accurate data about delay, jitter, throughput and losses. Network layer methods are mainly used by ISPs to include new services and improve network design. Unfortunately, this approach is not granular enough. It lacks the ability to differentiate between applications (i.e. accounts only for the network-layer protocols). Furthermore, it requires additional infrastructure or network changes. This is why Application layer measurements are attracting attention. They are easier to deploy and are application specific, therefore granular. Since they are operating on the upper layer it is also possible to use them for better service delivery. Unfortunately, this method requires access to end devices, which ISPs normally do not have.

3-2 Literature Review

Traffic measurements have been a field of interest for years in the computer networks community . Normally, it involves measuring the amount and type of traffic in a particular network. Today, different techniques are used to measure link usage, end-to-end delay and packet loss. This section is divided into two subsections, describing both the monitoring systems that are currently actively deployed and the new measurement approaches utilizing SDN capabilities. The first subsection focuses on methods that are currently used by ISPs to monitor their own infrastructure. Unfortunately, their options are very limited and give only partial network information. Recently, some new suggestions take advantage of the functionality provided by SDN, which are discussed in the second subsection.

3-2-1 Current Internet

The first group of measurements is based on **port counters**. Today, ISPs mainly use Simple Network Management Protocol (SNMP) [49] for link load measurements. SNMP counters are used to gather information about packet and byte counts across every individual switch interface. The SNMP protocol has been in development since 1988, and it is now incorporated in most network devices. A poller periodically sends requests towards every device in an IP network. The obtained information is then available on a central structure known as Management Information Base (MIB). Unfortunately, some concerns exist on how often a switch can be queried (limited to once every 5 minute) and the overall resource utilization (SNMP queries may result in higher CPU utilization than necessary). Furthermore, information gathered from SNMP counters lacks insight into the flow-level statistics and hosts behaviour, thus, obtained monitoring information simply lacks granularity. Furthermore, SNMP is unable to measure other metrics, therefore, it is not good enough for a single monitoring solution. This leads to additional measurement infrastructure for packet loss and delay.

For a scalable real-time monitoring solution flow-based measurements such as NetFlow and sFlow [48] rely on **packet sampling**. The method collects periodic samples for every flow, which does not give the most accurate results, but is sufficiently accurate. Samples are taken for every specific number of packets that go through the network (i.e. every 200th packet). Every 5th minute the router sends the flow statistics to a centralized collector for further analysis. To avoid synchronization problems (cache overflow, etc.) the traffic is separated into bins of interval size of multiple minutes.

Deep Packet Inspection (DPI) is another method that is heavily used within network monitoring, for security reasons and also for high speed **packet statistics**. The method consists of directing traffic or mirroring (via span port) it towards devices that inspect the header of every packet and collect statistical information. Unfortunately, few network devices support it, so very often additional hardware installations are required.

All monitoring techniques mentioned above use direct measurement approaches and have problems that are summarized in Table 3-1.

Table 3-1: Measurement Issues [13]

Issue	**Example**
High Overhead	NetFlow CPU usage
Missing data	LDP ingress counters not implemented
Unreliable data	RSVP counter resets, NetFlow cache overflow
Unavailable data	LSPs not cover traffic to BGP peers
Inconsistent data	timescales differences with link statistics

Today, delay and packet loss data is mainly obtained by application measurements. Unfortunately, ISPs normally do not have access to the end devices and the applications they use, therefore they are unable to use methods that involve end hosts. The common practice to measure those two parameters is to use *ping*. It uses the round trip time (RTT) by sending Internet Control Message Protocol (ICMP) requests, in other words, it sends a number of packets from a source node to a destination node and measures the time it takes for it to return back. For example, Skitter [26] uses beacons that are situated throughout the network to actively send probes to a set of destinations. The link delay is calculated by finding the difference between the RTT measures obtained from the endpoints of the link. However, using such a strategy to monitor the network delay and packet losses requires installing additional infrastructure, because every beacon is limited to monitor a set of links. Furthermore, using this method accounts additional inaccuracy and uncertainties.

Passive measurements are another widely used methods for packet and delay monitoring. An example of passive monitoring is given in [18] and consists of capturing the header of each IP packet and timestamp it before letting it back on the wire. Packet tracers are gathered by multiple measurement points at the same time. The technique is very accurate (microseconds), but requires further processing in a centralized system and recurrent collecting of the traces, which generates additional network overhead. Furthermore, every device needs accurate clock synchronization between every node (GPS clocks). Another similar approach is used to measure packet losses [51]. It tags uniquely each packet when it passes trough the source node and accounts if it was received in the end node.

3-2-2 SDN

The OpenFlow protocol is capable of not only controlling the forwarding plane, but also to monitor the traffic within the network.

OpenTM [53] estimates a TM, by keeping track of the statistics for each flow and polling directly from the switches situated within the network. The application decides which switch to query on runtime and converges to 3% error after 10 queries. In the paper presenting it, several polling algorithms are compared for a querying interval of 5 seconds [53].

Another active measurement technique is suggested in [3]. The authors use the fact that every new flow request has to pass through the controller. This allows to route the traffic towards one of multiple traffic monitoring systems, record the traffic or analyse it with an Intrusion Detection System (IDS) or simply said a firewall.

OpenFlow's "PacketIn" and "FlowRemoved" messages sent by the switch to the controller carry information about when a flow starts and end. Additionally, the "FlowRemoved" packet account for the duration and the number of packets/bytes of each flow. This features is used for passive measurements in FlowSense [58]. The measurements are evaluated from three prospectives: accuracy (compared to polling), granularity (estimate refresh) and staleness(how quickly can the utilization be estimated)

Furthermore, there are suggestions to implement a new SDN protocol that is oriented towards statistic gathering. Some efforts in this direction are shown in [59], where a new software defined traffic measurement architecture is proposed. To illustrate the capabilities of this approach the authors implement five measurement tasks on top of an OpenSketch enabled network. The measurement tasks are detection of: heavy hitters (small number of flows account for most of the traffic), superspreader (a source that contacts multiple destinations), traffic changes, flow size distribution, traffic count.

Chapter 4

Working with POX

While OpenFlow is not designed for monitoring purposes, it has some features capable of providing a real-time network monitoring. This chapter gives an in depth overview of the prototype, that is proposed and implemented in this thesis, using OpenFlow and POX. The chapter starts with some architectural assumptions that are needed for the implementation. Then it continues by giving concrete details, explaining how the POX controller works and the additional modules that were required.

4-1 Assumptions

Since, SDN is a new paradigm, some architectural aspects are still under investigation. In order to pay more attention on the research problems that were already outlined, the following two architecture **assumptions** are made.

First, one centralized controller manages all switches and handles all the control operations. There are indications that one controller is perfectly capable to handle current networks. For example, a research on how many controllers are needed and where should they be placed is presented in [24]. Multiple real network topologies taken from [7] are evaluated there and the presented results show that for most of the cases one controller covers suffices. Multiple projects propose a distributed controller system that still acts as a centralized entity. For example, Hyperflow [52] is an application built over NOX. To solve the controller scalability problem, it uses multiple controllers spread over the network. They act as a decision element and manage their own network partition. Meanwhile, the Hyperflow application instance on the top of each controller takes care of the global network view across the whole network. The Onix framework [29], which suggests three strategies to solve the scalability problems. First, the control applications are capable to partition the workload in a way that adding instances reduces work without merely replicating it. Second, the network that is managed by one Onix instance is presented as a single node to the other cluster's network information base. This reduces the amount of information required within a single Onix instance, thus allowing a hierarchical structure of Onix clusters. Finally, Onix suggests that applications that use the network should also be capable to obtain a complete view of the consistency and durability of the network state, thus, are also able to take decisions [29].

Finally, there are no scalability issues for the controller and the switches. The centralized controller is capable to deal with all the traffic and there are no scalability issues at all. Maestro [61] is a controller build with the idea to improve and solve the scalability problems that NOX has. It exploits parallelism in order to achieve near linear performance on multi-core processors. Another improvement in this direction is NOX-MT [54], multithreaded successor of NOX. The authors of [54] use cbench, a tool designed to test performance in terms of flows per second the controller can handle via latency and throughput measurements, to address the performance issues of current network controllers. While NOX-MT handles 1.6 million requests per second with an average response time of 2ms, the authors clearly state this should be used as a lower bound benchmark for a single controller.

4-2 OpenFlow Monitoring Architecture

In this section, the architecture of the OpenFlow monitoring application is described in depth. The section starts with presenting an overview of the different components that are needed to build a monitoring application. Afterwards, every building component is explained and discussed separately. To illustrate and confirm the monitoring abilities, a prototype of the component was implemented as a Python application for POX [30]. Which, in few words, is a Python based OpenFlow controller that can be used for fast implementation of network control applications.

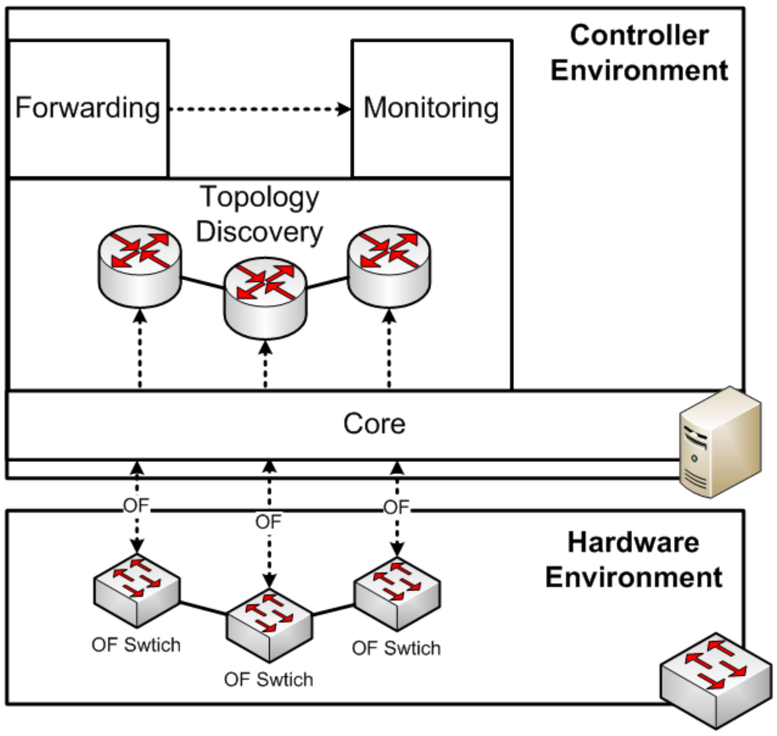


Figure 4-1: OpenFlow prototype

The monitoring application works as a core component of the controller, therefore, it has access to all the available information, including routing decisions. It is also capable to directly interact with each switch that supports OpenFlow. Figure 4-1 illustrates the architecture of the prototype. First, the discovery component builds a topology view. It is responsible to build a graph representation of the network topology. Furthermore, there is a virtual switch instance created for every new switch that connects to the controller. Each instance stores switch specific information. The application waits until both the forwarding application and the topology view are started and work properly. Then it is fired up. When this happens it is first registered as a core component of the controller, gaining access to the view-from-the-top information provided by the topology view. Everything known to the controller is also known to the monitoring component: active switches and their parameters, active flows and global routing information, link status, etc.

4-2-1 POX APIs

The Core object serves as the kernel of the POX API. It provides a lot of functions some of which are unique for POX and others are just wrapped around some other functionalities, that are already part of Python. The main goal of the Core is to provide interaction between the different components. Basically, instead of importing one component into another and vice-versa, the Core supplies data from all components that are registered with it. This provides a flexible way to interchange components and add new functionalities. Apart from **registering components** in the Core, POX supports some more functionalities which are used for the prototype implementation:

- **Raising events**: In POX, a particular piece of code listens for an event and is called once certain objects(i.e. new flow request from a switch) raise this event. In other words, whenever something important happens (i.e. new node joins the network or new packet enters it) the controller takes note of this event by raising a warning, which could be used to trigger an application or .
- **Working with packets**: Since POX is a networking controller, most of the applications that use it also work with packets. Thus, POX provides with means to easily construct and analyse (de-construct) packets.
- **Threads and Tasks**: POX uses its own library for cooperation between the components, which eliminates the need to worry about synchronization between modules.

Figure 4-2 illustrates the initialization phase for every component described in this chapter. As soon as the controller is started all the components register within the Core. Once this is done it becomes idle waiting for events.

4-2-2 Topology

One of the main responsibilities of POX is to interact with OpenFlow switches. For this purpose, a separate component (openflow.of_01) is registered to the Core as soon as the controller is fired up. Another component (openflow.discovery) uses LLDP messages to discover

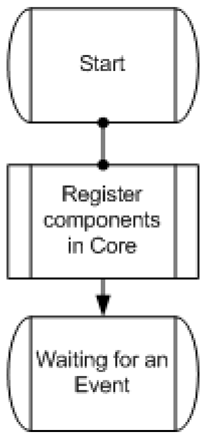


Figure 4-2: Initialization

the network topology (switch discovery). Every time a new switch is discovered an event "*ConnectionUp*" is fired. Furthermore, the controller associates it with a *Connection* object. This could later be used by other components to communicate with the corresponding switch. Apart from connection, each switch is also assigned a unique Datapath ID (DPID) which is used for data plane identification. This DPID could be defined by the MAC or be a totally different identification. As soon as there is a new switch event another *Switch* object is created. It is used to store the connection and DPID information for this specific switch or other relevant information. This object is also "listening" for switch specific events. The topology module is responsible to generate a graph representation of the physical network and keep it up to date. This graph is updated every time when a new node joins or leaves, based on "*ConnectionUp*" and "*ConnectionDown*" events. Figure 4-3 illustrates the whole process.

4-2-3 Forwarding

Each *Switch* object is "listening" for events from its corresponding switch. The forwarding module requiers one specific event called "**Packet In**". Every time an edge switch registers a new packet and does not have a matching table entry for it, it sends a request to the controller, which contains the packet header and a buffer ID. This event indicates to the controller that there is a new flow in the network. The path calculation can be done using any route calculation algorithm. It can be done in a reactive or a proactive way. In this case, Dijkstra Shortest Path [35] algorithm is used. The metric used for routing decisions is the inverse capacity. The controller assigns the route to the flow, installing table rules to match on every switch on the path by sending a *ofp_flow_mod* command. Additionally, the forwarding component is responsible to track every new flow together with its route. It keeps information locally about every flow until a "*FlowRemoved*" event fires up. This happens when a switch removes a flow entry from its table, because it was deleted or expired (idle or hard time out). The diagram on Figure 4-4 presents the whole forwarding process.

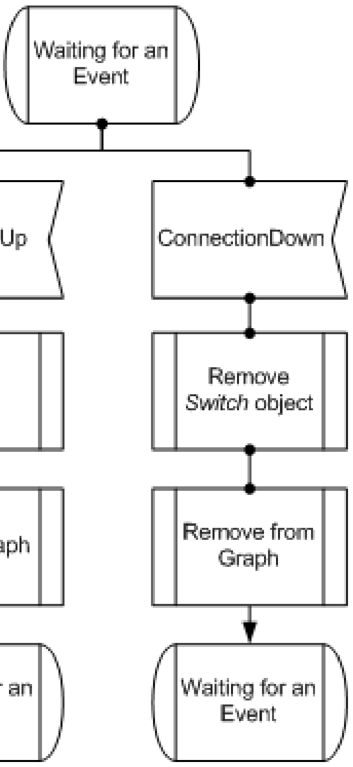


Figure 4-3: Topology view

4-3 Overview

At this point, only the monitoring component is left. It will be discussed in more details in the next chapter. Figure 4-5 presents the basic scheme. To summarise it shortly, the first thing every OpenFlow switch does, once it is started, is to establish a connection with the designated controller. The switch gives its state and link information. This allows the controller to keep a global up-to-date network view. Once a packet enters the network and no matching rule for it exists, it is forwarded towards the controller. The controller inspects the packet and determines how to handle it. Normally, the controller would install new flow entry in every switch table that needs it and then return the packet to its source node. This means that the controller has topology view of the network and information about the active flows (IP source/destination, port source/destination, ToS, etc.) and the routes they take trough the network. Each switching device within the network contains activity counters, i.e. for OpenFlow there are separate table, port, flow and queue counters. The flow and route information should be used as input parameters of the monitoring component. It is responsible to poll network devices, which in terms should return the requested information. Another option is to implement a passive measurement and wait for the switches to send statistics once the flow has expired. The monitoring component should make use of the two statistical gathering approaches. The final output should be data for link utilisation, delay and packet loss.

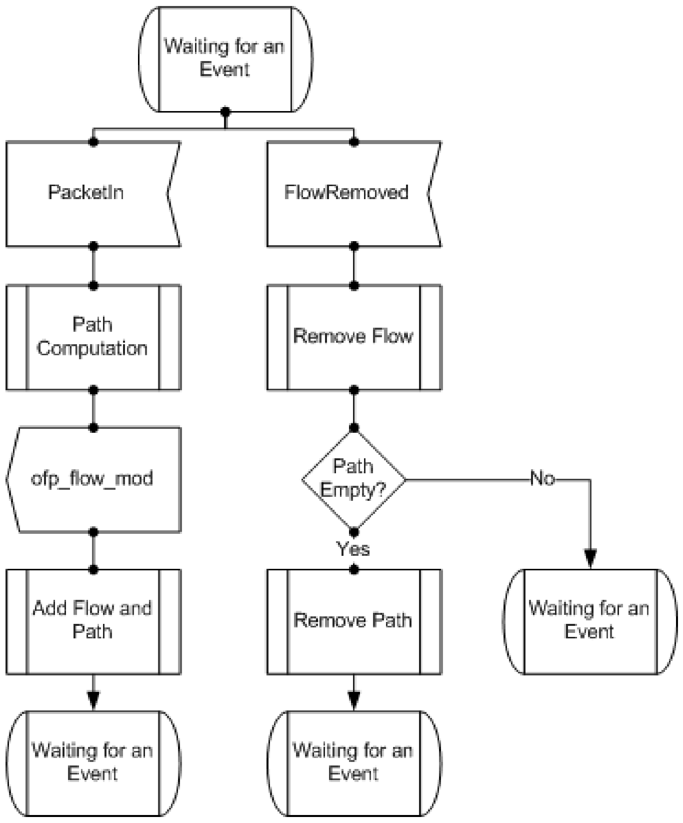


Figure 4-4: Forwarding

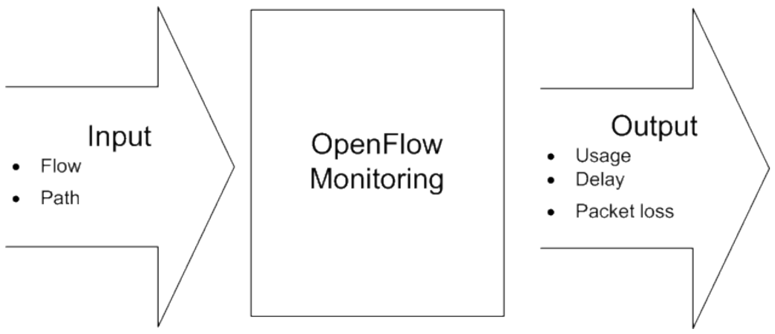


Figure 4-5: Basic diagram of the monitoring component

Chapter 5

OpenFlow network monitoring

While OpenFlow is not designed for monitoring purposes, it has some features capable of providing real-time network monitoring. The previous chapter presented the application architecture and an explanation on how it works. The main goal of this chapter is to design a solution that is able to monitor the network usage, delay and packet losses, track traffic flows and applications in use, and show which users and devices are present and active. Such an application can be used to generate Traffic Matrix (traffic demands for each different traffic class is also a possibility) or for security purposes. Furthermore, the SDN controller requires a network monitoring tool in order to efficiently forward network traffic and to achieve certain QoS constraints.

5-1 Objectives

A network monitoring system should be able to observe and display up-to-date network state. Several monitoring solutions are already capable to do that in one or another way. To meet the specific challenges ISPs face, the following design requirements are considered:

- **Fault detection**: Whenever a link or node failure happens, the network monitoring system should be warned as soon as possible. This option is already incorporated in OpenFlow, where the controller is alerted every time there is a topology change. Therefore, the monitoring system should only get data from the controller.
- **Per-link statistics**: ISPs require statistics for every link. This would allow them to meet the SLAs and assure QoS within the boundaries of their network, without bandwidth over-provisioning. As seen in the previous chapter there are ways to measure link utilisation, packet-loss and delay, but those ways require additional infrastructure and have more drawbacks, which leads to the overhead requirements:
- **Overhead**: The proposed solutions should not add too much network overhead. The overhead should scale no matter how big the network is (as long as the controller can

handle them) or the number of active flows at any moment. The component should be able to obtain statistics based on the routing information, thus, sending a query requests only to those devices that are currently active.

- **Accuracy**: A big difference between the reported network statistics and the real amount of used capacity should be avoided. Utilisation monitoring solutions that directly poll statistical data from the network are considered to present the best results in terms of accuracy, but are not granular enough. The methods that are used should be capable to achieve accuracy comparable or better than the current implementations.
- **Granularity**: The system should be able to account for different type of services. It should be able to make distinction between flows that have specific needs, i.e. require special care (bandwidth, delay, etc.). Furthermore, it should make distinction between applications, as well as, clients. This is another feature that is already integrated in OpenFlow.

Subsequently, the main goal of the project is to reduce the generated overhead as much as possible, without too much degradation of the measurement accuracy.

5-2 Measuring Methods

By using SDN to implement a network monitoring system some of the objectives are already met. Since every device communicates with the controller, there is real-time view on the network status, including links, nodes, interfaces, etc. Furthermore, it provides sufficient granularity and it is capable to monitor the utilization of every link within a given network without sampling any packet or adding more overhead to any of the switches.

5-2-1 Real-time link utilization

OpenFlow allows granular view of the network, but this is done by generating additional network/switch load. Obtaining flow statistics is a task that requires polling for information for every flow separately. In this section the following ways for its improvement are proposed:

- **Aggregate flows**; Generate only one query per set of flows that share the same source-destination path instead of polling statistics for every flow separately.
- **Data collection schemes**; Poll different switches, thus reducing the overhead on a single switch/link and spreading it evenly.
- **Adaptive polling**; Using a recurrent timer does not accommodate traffic changes and spikes. Hence, an adaptive algorithm that adjusts its query rate could enhance the accuracy and reduce the overhead.

Finally, a suitable passive measurement technique, capable to reduce the monitoring overhead even more, is described in FlowSense [58]. Unfortunately, it indicates the link utilization once the flow expires. While this is perfect for traffic matrix generation, it is not always usable to take real-time forwarding decisions.

Counters

In the OpenFlow switch specifications [17] it is written that switches have to keep counters for port, flow table/entry, queue, group, group bucket, meter and meter band. Table 5-1 presents the Per Flow Entry counters used in this thesis. Furthermore, in order to follow the statistics for more than one flow, there is an option to bundle multiple flows in a group and observe their aggregated statistics.

Table 5-1: Counters [17]

Counter	**Description**
Received Packets	Counts the number of packets
Received Bytes	Counts the number of bytes
Duration (seconds)	Indicates the time the flow has been installed on the switch in seconds
Duration (nanoseconds)	Counts time the flow has been alive beyond the seconds in the above counter

The OpenFlow specifications also state that "counters may be implemented in software and maintained by polling hardware counters with more limited ranges" [17]. The switch manufacturers and designers decide when software counters are updated and how they are implemented in hardware, meaning that the accuracy varies per switch vendor.

Flow statistics

Polling for each flow separately could result into scalability problems since the total number of queries depends on the number of active flows. Which varies depending on the network and increases with its growth. For example, the data from a 8000 host network [42] suggests no more than 1.200 active flows per seconds and another source [37] show a number below 10.000 flows for a network with 5.500 active users.

The first step, in order to reduce the overhead is to send a single query for all flows sharing the same path. For a solution that scales better and is not subject to the total number of active flows, the queries should be sent based on the active source-destination pairs, bundling all the flows together. Thus, the maximum number of queries for network with N edge nodes never exceeds the maximum number of source-destination pairs $N * (N - 1)$. Furthermore, only if every flow has separate source-destination pair (goes trough a different route) the amount of queries equals the number of flows.

Obtaining link utilisation statistics is done by sending a number of queries based on active switch source-destination flow paths. This is done in a recurring manner. The frequency at which queries are send to the switches can be specified. The controller polls for all flows and/or ports of a chosen switch. The flow level statistics that have the same ingress and egress switch and follow the same network route are added together in order to obtain the aggregated throughput.

Data Collection

One of the most important aspects for a monitoring system that relies on active measurements, is the polling algorithm. While flows traverse the network they are subject to packet losses and drops, this means that switches that the flow passes may register different rates. Furthermore, switches are subject to different traffic loads, etc. An important question is how utilization information should be gathered. The following introduces some possible querying schemes.

Last Switch query: The normal practice is to query the last switch on the path of the flow. Polling the last switch assures that the flow has passed thought the network and has been subjected to any packet losses due to link degradation. It is the most accurate polling scheme for traffic matrix generation. Unfortunately, when only the edge of the network is polled, additional load on the edge switches is generated.

As a side note, in a partial SDN deployment, the most viable monitoring solution is to place SDN enabled switches on the network edge. Once all the edge switches support a SDN protocol such as OpenFlow, the traffic that passes across the network is known (source, destination, ports, etc.).

Round-Robin query: Another algorithm that could be used is uses Round Robin scheduling. In the context of information polling, it consists of loading each switch on the path equally, by simply polling them in a circular order without giving priority to any of the switches. While the best polling scheme is to query the edge switches all the time, it also generates load only on those switches, a Round-Robin scheduling algorithm reduces the load on the edge switches, trading it off with some additional inaccuracy. Furthermore, this algorithm can be combined with another scheme that also takes into consideration the past values (measurements), link statistics, etc.

Adaptive polling

Setting up a never changing recurrent timer has some drawbacks. Since the recurrent timer is a fixed value it is unable to adjust fluctuating flow arrival rate. In case that flows arrive too fast and the recurrent is large, the monitoring system would miss link utilization changes, thus reducing the overall accuracy. If the flows arrive slower then the recurrent timer would bring unneeded network overhead.

In order to solve the above problems the monitoring system should focus on the bandwidth trend instead of the changes. This method is already suggested and evaluated in [34]. The results presented there show decreased overhead. Unfortunately, the method is not entirely suitable for SDN. The controller does not know the link utilization before sending a query to the switch. Therefore, taking decisions based solely on link utilization is not feasible. On the other hand, the controller learns when new flows are sent trough the network. Hence, this is why the adaptive timer should be based on the flow arrival rate, more precisely, increase the polling frequency whenever there is a new flow and then take further decisions based on the link utilization. Thus, decision taking could depend on reaching a threshold value, as well. This value could be obtained by comparing the mean throughput since the flow arrival with its last measured utilization. For the monitoring component it is enough to increase the polling frequency whenever there is "PacketIn" and "FlowRemoved" or whenever exceptional increase of link utilization is measured without any specific event.

5-2-2 Link packet loss

As the amount of real time traffic is growing (mainly voice and video), packet loss is an increasingly worrying problem. Dropped packets decrease the quality of applications (causing pixelation or terminated sessions). A lot of enterprises lack proper infrastructure and tools to measure packet loss and they do not even know it is there. Furthermore, this is one of the metrics that could be used for constraint QoS routing.

Some active measurement proposals are used in real networks. In terms of SDN packet loss measurement could be improved by implementing more switch capabilities to account for them (i.e. automatically sending probes between each other and accounting link statistics). Unfortunately, this would add more CPU and link overhead. Therefore, in this section a novel approach to measure link packet loss is proposed, capable to eliminate the overhead. While in the previous section the main concept focused on real time and active flow measurements, here passive measurements are used. This is based on the idea that packet loss metrics can be generalized on per class basis without loss of accuracy. Furthermore, measuring the packet loss for every single flow would not be viable. To estimate a stable and accurate link metric, that does not fluctuate too much, a set of measurements are required. Thus, a metric that represents most of the packets, without accounting for the anomalous changes or the statistical outliers. Finally, in an active network flows terminate every second, so the obtained measurements would still be real-time.

On a new flow arrival and when the switch does not have any rules installed, the first packet is sent towards the controller. The controller is then responsible to decide what to do with the packet and eventually install table rules on each switch on the path of the flow. Once the flow is finished each switch indicates this with another message to the controller. The whole process is illustrated in Figure 5-1. The flow is installed at time t_0 with a *"FlowMod"* message sent from the controller towards every switch on the route of the flow. At time t_1, t_2, up to t_N(where N is the amount of switches), the controller receives *"FlowRemoved"* messages. Those messages indicate the flow has expired and some specific statistics for the flow, such as, the number of bytes, packets and the flow duration. Measuring the packet-loss relies on the fact that each switch sends this information based on its own counters.

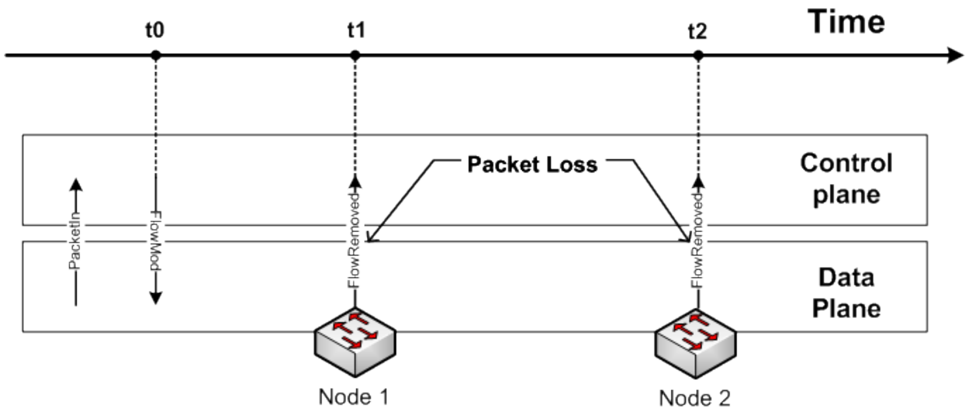


Figure 5-1: Calculating the packet loss percentage

Each switch has separate flow counters, but it counts different amount of bytes. This is because there are link losses, congested links, etc. Receiving flow information from every switch allows to compare their counter statistics and calculate the number of bytes that were lost. Whenever messages that the flow has expired from the same flow are received their recorded packet bytes are compared. This comparison allows to determine the packet losses for this particular flow $((1 - packets_{Node2}/packets_{Node1})$. The technique is sufficient to determine what the current link state for this traffic class is. In case there is a need for flow packet loss, every node could send periodically its counters to the controller.

5-2-3 Link delay

Delay is another powerful metric that indicates the network health. It is often used in SLAs between customers and the network provider. Like packet loss, it should report the latency experienced by most of the packets that enter the system.

At this point, OpenFlow does not offer any new way for network delay measurements. Thus, the used measurement techniques would be the same that were described in the literature review (Chapter 3), sending active probes from one node to another. Nevertheless, SDN could still bring improvements, because it would not require the use of special beacons or synchronization between the nodes. The transmission, processing and skew delay remain problem

The proposed method is illustrated on Figure 5-2. The controller generates an User Datagram Protocol (UDP) packet and sends it towards Node 1 at t_0, installing a new rule in Node 1 flow table. This rules indicates to Node 1 to send this UDP packet to Node 2. At t_1 the packet flows from Node 1 to Node 2. Since no rule installed in Node's 2 flow table, the switch returns the packet to the controller at t_2. The controller keeps constant connection with every node, therefore, knows the delay between it and the nodes. This allows to determine the delay between every pair of nodes. The method uses each node as a beacon, thus, does not require additional infrastructure. Finally, since only the controller does the measurements there is no need for synchronization between the nodes.

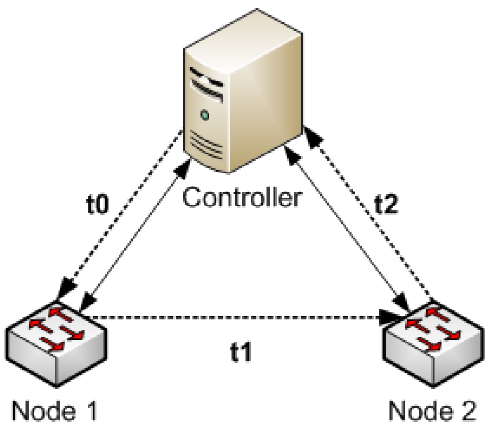


Figure 5-2: Measuring the link delay

5-3 Monitoring

After describing the design of the monitoring component and what it should achieve, it is now time to give some insight on the implementation concept. Only the packet loss and link utilization methods were implemented, since there is nothing new suggested for the delay measurements. The monitoring component released in POX reacts every time there is a "PacketIn", "FlowRemoved", "StatusReplay" or when a polling timer expires.

Whenever, a new packet enters the system it is sent to the controller. The controller has a complete view of the network and based on the link weights decides where to route this packet. The monitoring component registers every "PacketIn" event and creates a unique identification based on the flow information. Additionally, a separate ID is used to distinguish between the network paths. Every flow is assigned to a certain path. In few words, the monitoring component keeps track of every flow that enters and the path it follows through the network. Furthermore, every *Switch* object also account the flows that pass through it. This information is later used to determine the link utilization.

In order to execute a piece of code in the future or assign a recurring event the monitoring component uses the Timer class incorporated in POX. In case, this is the first packet that uses this route, the monitoring component starts a polling timer for every second. Whenever the timer expires it fires an event. During this event a data collection algorithm is used (Round Robin or Last Switch). Those two algorithms present a trade-of between accuracy and overhead. . Afterwards, a message "*StatusRequest*" to the chosen switch is sent. This is the query requesting statistics for all the flows that follow the same path. Every path has a separate timer.

When a switch receives a "*StatusRequest*" message it generates a response. The "*StatusReply*" message contains the information obtained from the switch counters. On flow level it gives the duration of the flow (in nanoseconds), packet and byte count. Port statistics give more information about the state (both transmitted and received) such as number of dropped packets, bytes, errors and collisions. The controller obtains information for every flow that follows the same path. A part from that, the polling timer is also adjusted. The controller tracks the time that passed since the last flow routed trough this path was registered, as this time increases, the polling timer also increases. Currently, the controller polls every second for the first five seconds, then every five seconds until the 15th second, moving to 15 seconds until the end of the first minute and polling once per minute when there has not been any flow activity for over a minute.

When the switch removed a flow entry from its table, because it was deleted or expired (idle or hard time out), it also raises a "*FlowRemoved*" event. Such event means that this flow is no longer active and the monitoring component does not need to account for it any more. The controller receives a massage that indicates the whole duration of the flow together with the data statistics for this particular flow. This is used to obtain packet loss information. The whole process is described in Figure 5-3

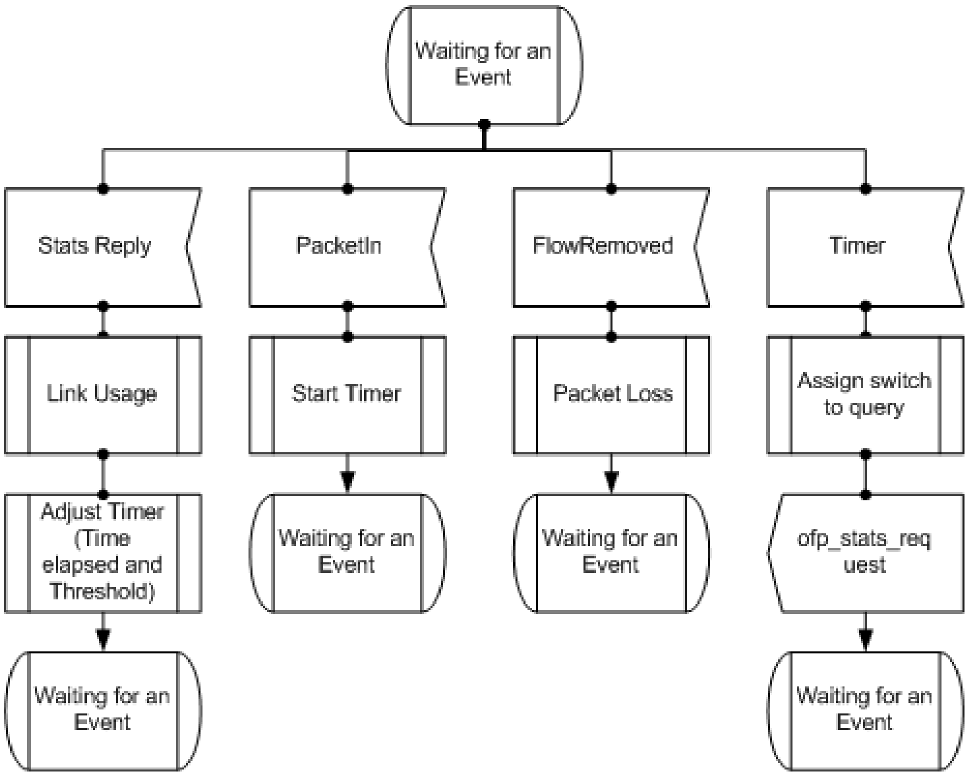


Figure 5-3: Monitoring

Chapter 6

Evaluation

The following chapter presents the evaluation results from the proposed traffic monitoring component. The chapter begins with a description of the experiment scenario and the tools that were used to build it and finishes with the obtained results. First, the proposed link utilizations improvements are evaluated. Each experiment is made using an emulator and then repeated within the physical topology. Throughout this chapter only the results from the testbed are shown. The chapter finishes with evaluation of the suggested packet loss method, trying to answer the question if the method is valid, how accurate is it and how much does it depend on the flow parameters (duration and rate).

6-1 Test Environment

The emulation uses an Intel Core i5 computer with four 2.53 GHz cores and 4 GB RAM and Mininet 2 [22]. Mininet is a container-based emulator able to create realistic virtual topology. The containers mechanism uses groups of processes that run on the same kernel and yet use separate system resources, like network interfaces and IDs. Thus, every emulated switch or host creates its own process. Furthermore, network links can be assigned specific link properties such as bandwidth and packet-loss. Unfortunately, like most emulators there are also some drawbacks. Processes in Mininet do not run in parallel, instead they use time multiplexing. This may cause delayed packet transmission, so it is not suitable for time accurate experiments.

The testbed is installed on servers that have Intel(R) Xeon(TM) processor with four 3.00 GHz cores. Every switch uses separate physical machine with Ubuntu 12.04.2 LTS operating system. The testbed uses Open vSwitch [44] as OpenFlow enabled switch. Traffic is generated by the Iperf application. A network testing tool capable to create TCP and UDP traffic between two hosts, one acting as client and the other as server. It measures the end-to-end (either uni- or bi-directional) throughput, packet loss and jitter. NetEm [25] is used, in order to emulate link packet losses and delay. It is an Linux kernel enhancement that uses the queue discipline integrated from version 2.6.8 (2.4.28) and later.

6-2 Link Utilization

The topology used for the Link Utilization tests is illustrated in Figure 6-1. This topology is enough to determine if all the suggestions from the previous chapter (Chapter 5) work as expected. Any additional infrastructure would not give different results. Each link of the topology is configured to have 100 ms delay and 1 % packet loss. The OpenFlow network controller used in the experiment is POX. In order to evaluate the suggested methods, one or multiple UDP flows are created between Node 1 towards Node 3. The experimental evaluation aims to prove the systems accuracy and reduced overhead.

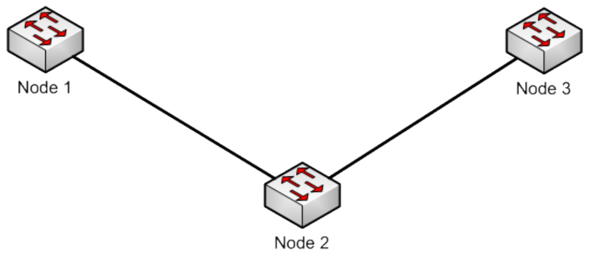


Figure 6-1: Evaluation Topology

6-2-1 Data collection schemes

The first experiment aims to compare the two polling algorithms discussed in the previous chapter to obtain throughput. Samples were taken once every 5 seconds for the duration of an hour. Initially, a single UDP flow with data rate of 2 Mbps is sent from Node 1, through the testbed.

Figure 6-2 illustrates the measured flow rate when only the last switch is polled. The error compared to the Iperf Server report (See B-2 for it) averages to 2.86% and never exceeds 3.4% for the whole duration of the experiment when polling the last switch only. Furthermore, there is only an average of 0.2% difference (never exceeding 1%) difference compared to the results reported by Wireshark (See B-1). The reason for this variation of the readings between Iperf and Wireshark is because they operate on different layers of the OSI model, thus, Wireshark accounts for the packet headers. Hence, the Wireshark readings are more accurate. Furthermore, little traffic fluctuations can be seen on the measured results. Those are because the emulated packet loss vary, therefore, data rate is not always the same. The main reason for the spikes in the beginning are due to the forwarding module. At first packets are buffered by the first switch while the controller is taking routing decisions. This problem is outside the scope of the thesis.

Figure 6-3 illustrates the obtained results when using Round-Robin scheduling. Average deviation from the Iperf Server report (See B-4) is 2.88% and 0.5% from Wireshark (See B-3). Because the controller queries all switches one by one there are differences between their traffic counters, this is causing the fluctuation of the readings. Those differences are subject to the packet losses and in a network where there are no losses the difference would be really small.

Figure 6-4 compares the two querying schemes. For this experiment there is almost no difference. This is because there are only 2% packet loss. For networks where link losses

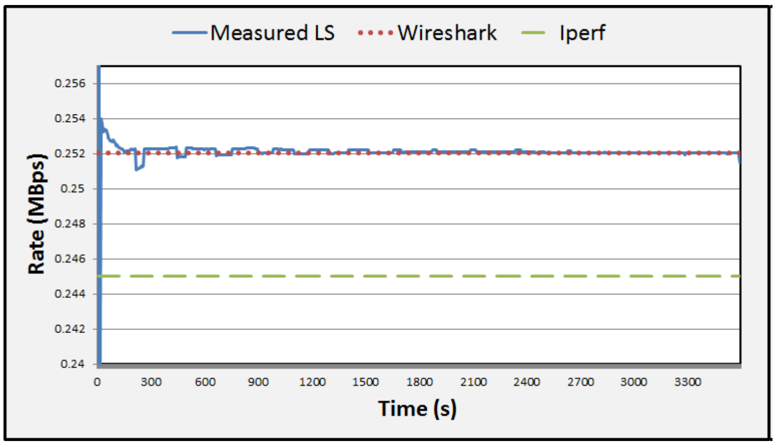


Figure 6-2: Last Switch query

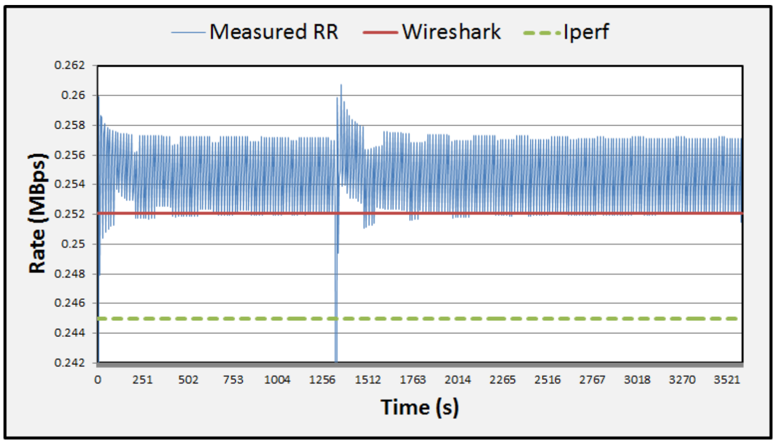


Figure 6-3: Round Robin scheduling test results

are bigger the difference between those two algorithm will also increase. There is a small fluctuation around the 1430th second of the experiment, the fluctuation is also registered by Wireshark (Figure B-5), which means that the problem is not cause of the monitoring prototype.

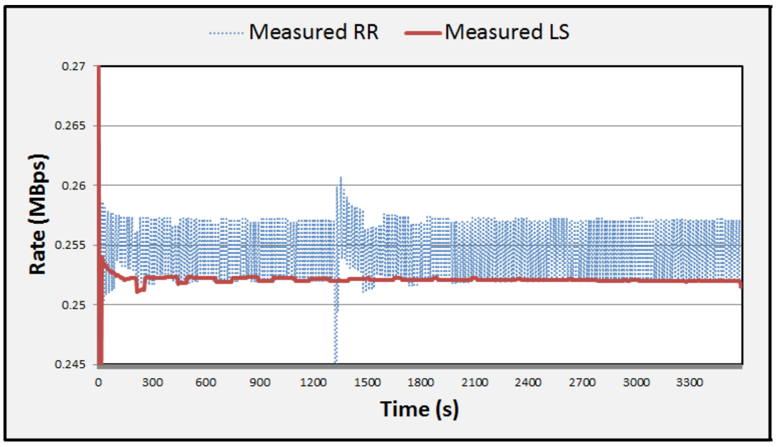
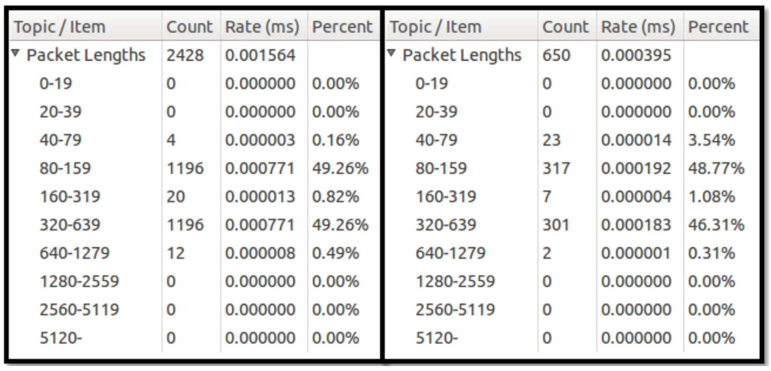


Figure 6-4: Comparison of the two querying schemes

6-2-2 Flow vs Aggregate statistics

In this section a comparison for added overhead is made, between polling for statistics for every single flow separately or querying only once for all the flows that share the same source and destination node. Four UDP flows are sent from Node 1 to Node 3.

Figure 6-5 compares the differences of overhead between the polling approach suggested in this project (right graph) and the OpenTM uses (left graph). In the first case, when all the flows follow the same path the controller queries only once per query interval. Meanwhile, for the other case, there are four queries. The results prove that using aggregate flow query method decreases the overhead that is generated.



Topic / Item	Count	Rate (ms)	Percent
▾ Packet Lengths	2428	0.001564	
0-19	0	0.000000	0.00%
20-39	0	0.000000	0.00%
40-79	4	0.000003	0.16%
80-159	1196	0.000771	49.26%
160-319	20	0.000013	0.82%
320-639	1196	0.000771	49.26%
640-1279	12	0.000008	0.49%
1280-2559	0	0.000000	0.00%
2560-5119	0	0.000000	0.00%
5120-	0	0.000000	0.00%

Topic / Item	Count	Rate (ms)	Percent
▾ Packet Lengths	650	0.000395	
0-19	0	0.000000	0.00%
20-39	0	0.000000	0.00%
40-79	23	0.000014	3.54%
80-159	317	0.000192	48.77%
160-319	7	0.000004	1.08%
320-639	301	0.000183	46.31%
640-1279	2	0.000001	0.31%
1280-2559	0	0.000000	0.00%
2560-5119	0	0.000000	0.00%
5120-	0	0.000000	0.00%

Figure 6-5: Comparison of the link overhead per polling scheme: OpenTM approach (left) versus Aggregated statistics approach (right)

Figure 6-6 (Wireshark results B-6 and Iperf results B-8) and 6-7 (Wireshark results B-7 and Iperf results B-9) presents the measured results. The average deviation is 1% (LastSwtich) and 1.01% (RoundRobin) for Wireshark. This results prove that the proposed improvement reduces the generated overhead, while the statistics remain accurate. It is important to note there is a small increase of the difference between the reported by Wireshark and the measurement results. This is because the flows are measured separately and summed afterwards, which increases the average error, but makes the results more stable reducing the random fluctuations.

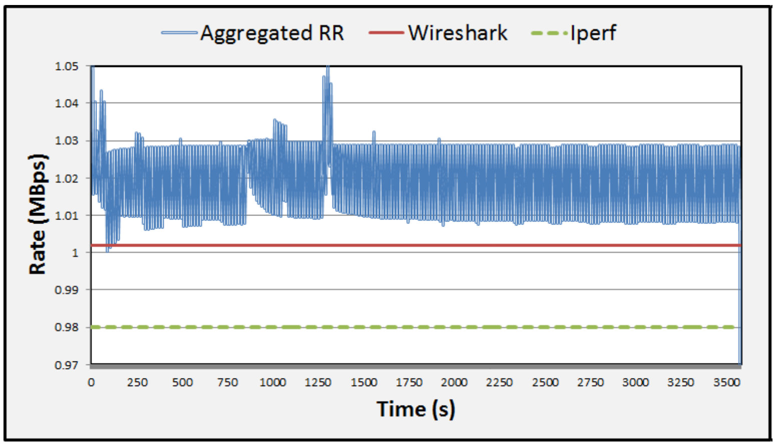


Figure 6-6: Aggregate flow statistics test results for Round Robin

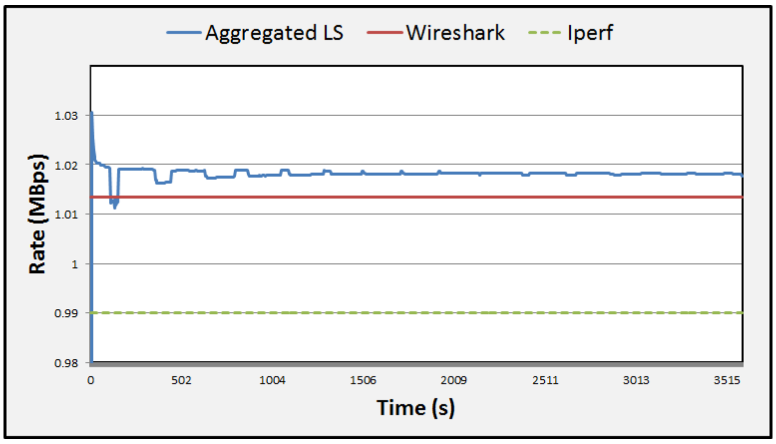


Figure 6-7: Aggregate flow statistics test results for Last Switch

6-2-3 Adaptive vs Recurring polling

The frequency at which switches could be polled is also a factor that affects the accuracy, hence, it is also evaluated. In order to test how much of influence it is, two flows are sent from Node 1 to Node 3 (two UDP flows worth of 2 Mbps each). Each flow has different duration, the first flow continues for 80 seconds and the second flow has a duration of 5 seconds. Polling frequencies are directly linked with the overhead, the more queries, the higher load. Figure 6-8 shows the obtained data rate for the duration of this experiment for both 2s and 10s recurring interval. Once again the main reason for the spikes in the beginning of each flow are due to the forwarding module. Packets are buffered by the first switch and then released altogether when the controller is installs the new flow rule. This is why the graph looks really inaccurate. Still, the figure shows how the polling intervals affect the accuracy of the measurements. This is really dependent on how long a traffic flow continues, which varies between the different kinds of traffic within the network. For this case recurrent timer with 2 seconds interval gives better results than the timer with 10 seconds interval.

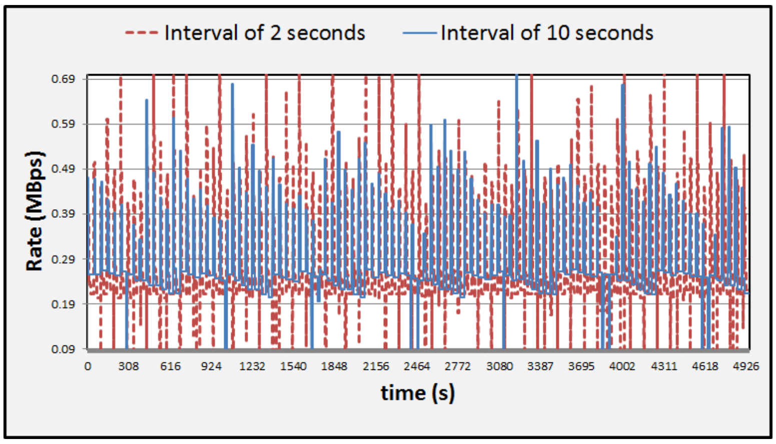


Figure 6-8: Different polling intervals - Data rate measurements with 2s and 10s recurring interval

A static recurrent timer does not deal with the problem well enough. Whenever there is a flow with long duration, the recurrent timer polls unnecessary, adding more overhead. As soon as there is a short lived flow, the recurrent timer misses it. In the previous chapter it was proposed to use adaptive technique instead of recurrent timer. Figure 6-9 presents the measured results.

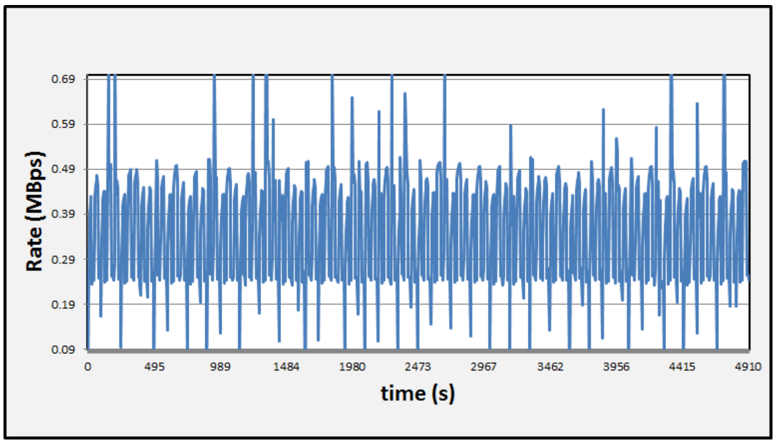


Figure 6-9: Different polling intervals - data rate measurements with adaptive polling

Figure 6-10 and 6-11 compare the results from such a scheme with the results from 2 and 10 second recurring statistical requests. They show more consistency obtaining better accuracy than the 2s interval timer and using 825 probes less. It still uses 350 probes more than the 10 second interval scheme, but this could be improved. For a single experiment the adaptive technique does 15 requests more than the recurrent with 10 seconds interval. The implementation with shorter interval does 28 queries more than the adaptive query scheme. The proposed adaptive method eliminates the random factor by reacting on flow arrival and not on a recurring timer. This method gives better results in terms of accuracy and overhead.

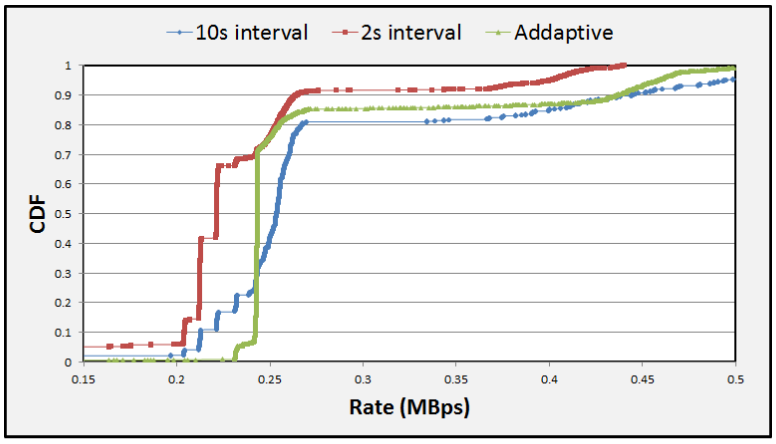


Figure 6-10: CDF for different polling intervals

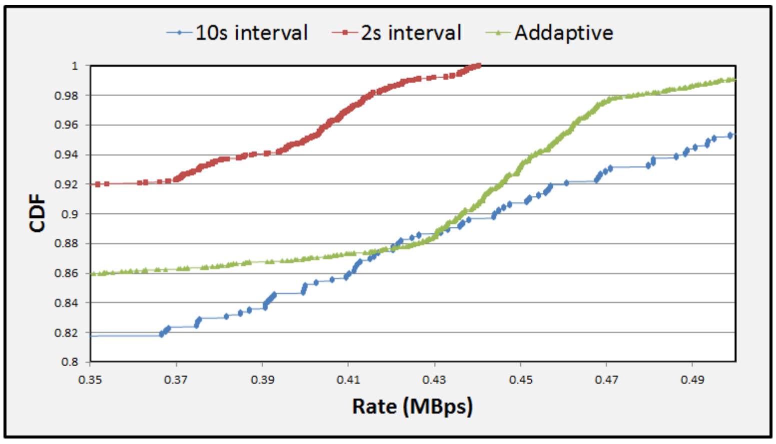


Figure 6-11: CDF for different polling intervals - a closer look

6-2-4 Validation

All the suggested methods compared to the initial method in a more complex scenario. The real traffic sent through the network is illustrated on Figure 6-12 to the right and the physical topology is to the left. The experiment uses multiple flows with different source-destination pairs. An UDP flow with data rate of 3 Mbps is sent from Node 1 to Node 2 for the duration of 100 seconds. At the same time and for the same duration another UDP flow is sent from Node 1 to Node 3. This flow has data rate of 4 Mbps. Finally, three UDP flows are sent from Node 1 to Node 4, each worth of 2 Mbps. Two of the flows continue for 100 seconds and the third has duration of 50 seconds.

The idea behind this experiment is to use together both the aggregate statistics method and

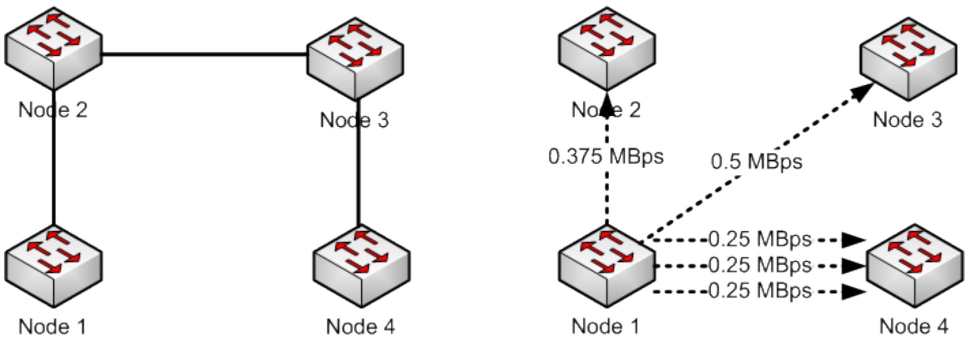


Figure 6-12: A more complex evaluation topology - Physical topology (left) and Flows sent through the topology (right)

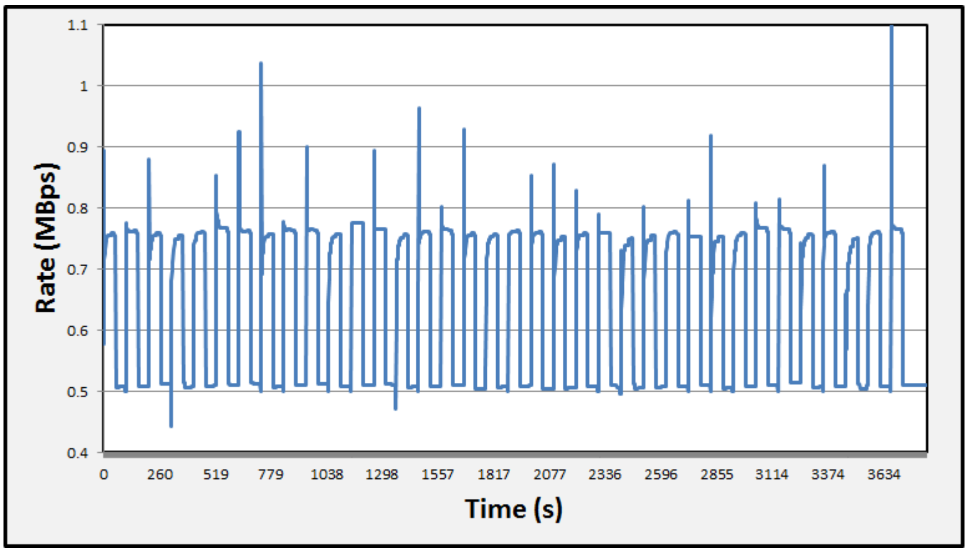


Figure 6-13: Data Rate for the link between Node 3 to Node 4

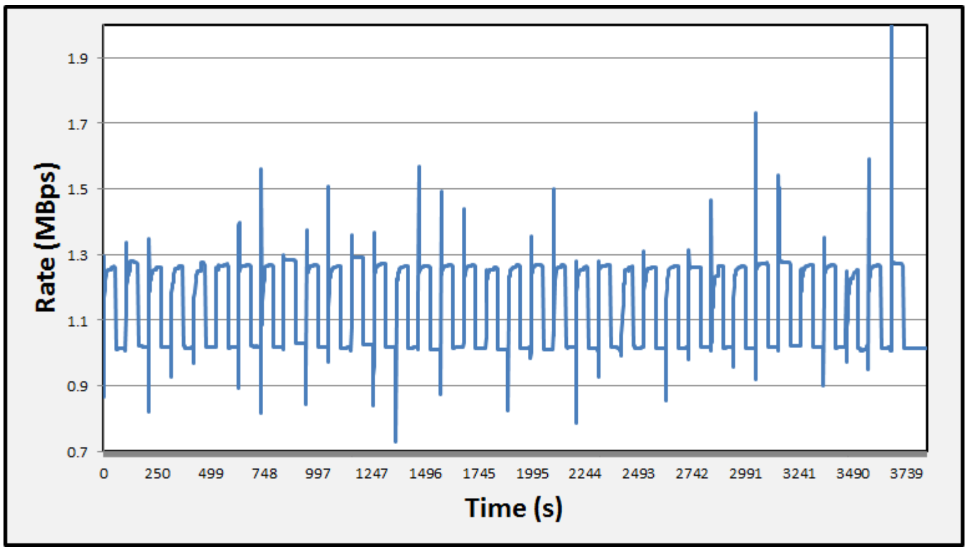


Figure 6-14: Data Rate for the link between Node 2 to Node 3

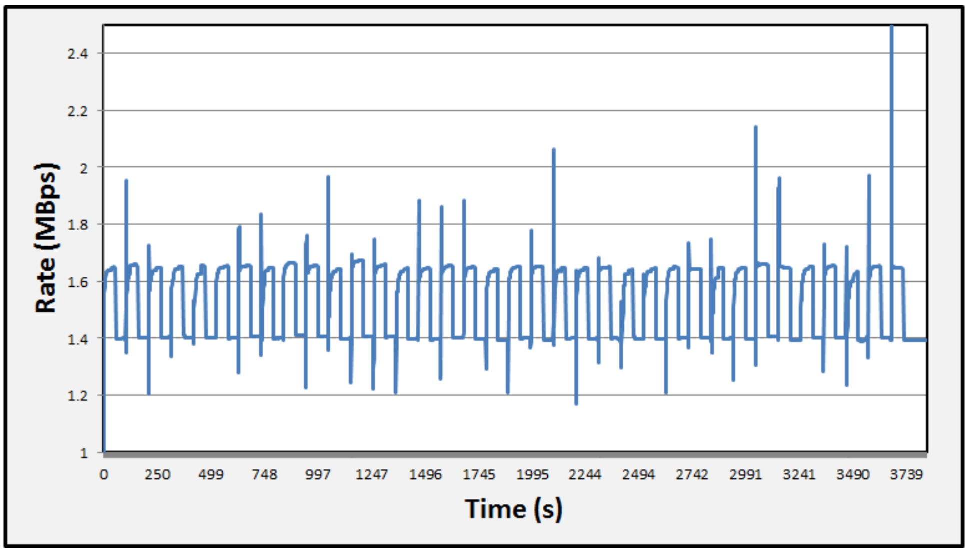


Figure 6-15: Data Rate for the link between Node 1 to Node 2

the adaptive methods proposed in this thesis and to validate that they can work with more than one flow. The obtained statistics are illustrated on Figures 6-15 for the link between Node 1 and Node 2 (example Wireshark report B-12), 6-14 for the link between Node 2 to Node 3 (example Wireshark report B-11) and 6-13 for the link between Node 3 and Node 4 (example Wireshark report B-10). The results are not different than the ones from the previous experiments and show that both methods are capable to work with more than one source-destination pairs and to provide accurate results.

6-3 Packet Loss

Apart from new link measurement methods the thesis also proposes a new method capable to measure the packet loss. The topology used for this experiment is illustrated in Figure 6-16. The topology is enough to validate that the method works properly and to determine its accuracy under different circumstances. Each test is first held in Mininet environment and then repeated in the testbed.

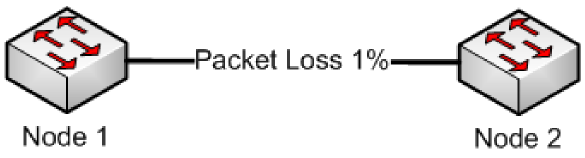


Figure 6-16: Evaluation Topology

6-3-1 Validation

In order to confirm that the proposed measuring approaches for packet loss work, 1000 consecutive flows worth of 6 Mbps are sent from Node 1 to Node 2. The link is set to emulate

1% packet loss.

The results are shown on Figures 6-17 (See also Figure B-13 for Iperf Server report). As it can be seen from the server output the packet-loss varies from flow to flow. However, the packet-loss distribution shows really promising results, an average of 0.99% losses per flow and standard deviation of ±0.34. The deviation is expected since NetEm uses normal distribution for packet loss emulation.

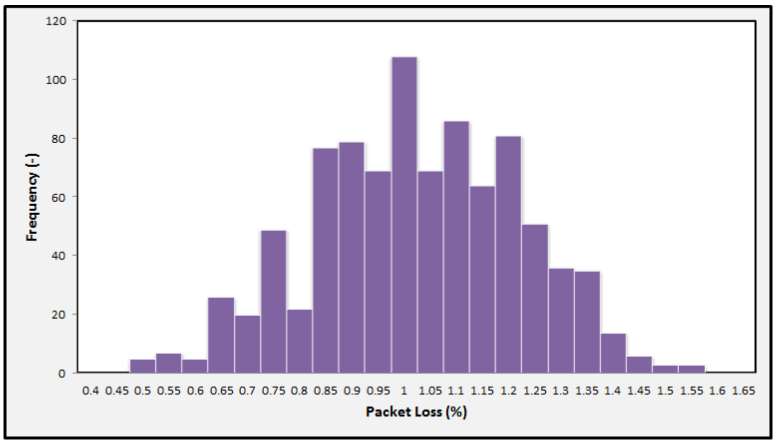


Figure 6-17: Packet loss Distribution

In order to determine exactly how accurate the method is 18 flows are recorded (Iperf Server report) and then compared with the measured packet loss. The first measurement consists of sending flows worth of 64 Kbps for the duration of 195 seconds (average call duration). Results can be seen on Figure 6-18. They match perfectly the Iperf Server report (See Figure B-14). Finally, the second set of measurements emulates short term Video connection using MPEG 2 with data rate of 10 Mbps (Figure 6-19). Ten flows that are set to continue each for 2 minutes are recorded. The results from both measurements prove that the proposed measurement method gives perfect accuracy.

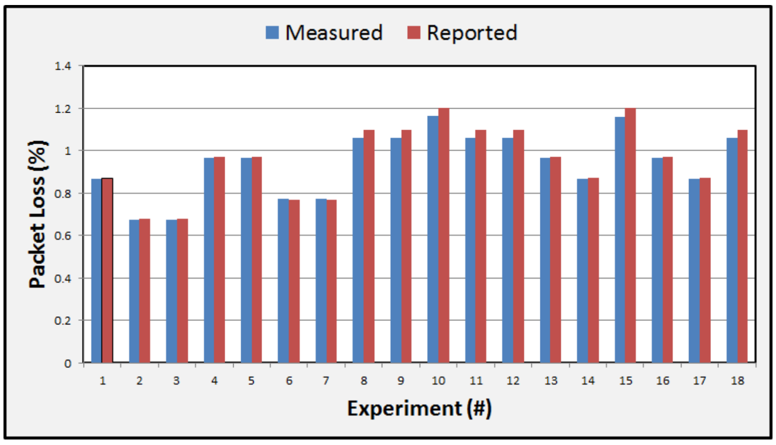


Figure 6-18: Example of packet loss measurements for flows of 64 kbps

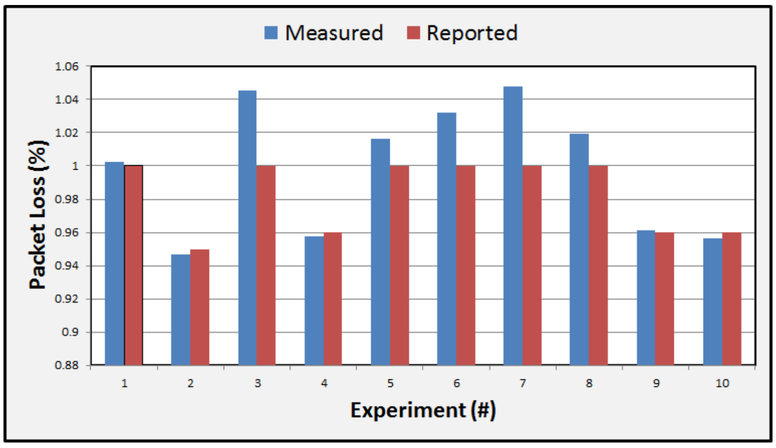


Figure 6-19: Example of packet loss measurements for flows of 10 Mbps

6-3-2 Dependency

The proposed methodology is capable to measure packet losses accurately, but what would happen if the flows are short lived or have lower throughput. Furthermore, if the method would be used to determine link metrics and for SLAs it should give overall performance and not the spikes that tend to occur from time to time. A thousand consecutive flows with either different duration or data rate are sent from Node 1 to Node 2. The link is set to emulate 1% packet loss. This experiment measures the packet losses under different network loads and flow durations. A thousand measurements are done for every parameter change (both in flow duration or in data rate).

The first experiment measures how the accuracy changes with the flow duration. Figure 6-20 presents the results. As it can be seen the more the flow duration increases the more accurate the results are. Thus, flows that continue longer are more trustworthy. In this case flows with duration of 20 seconds show packet loss close to the flows of 60 seconds. From this experiment it is clear that really short lived flows with a duration of a second are not sufficiently accurate for a metric estimation. The final experiment measures the accuracy in terms of data rate. Figure 6-20 illustrates the results which are analogue to the previous experiment. With the increase of the data rate the resulting graph is more granular and converges closer to the 1% packet loss. Thus, the longer the flow continues or the bigger it is, the closer to the emulated packet loss it gets. It is also obvious that one measurement would not be enough to estimate a packet loss metric.

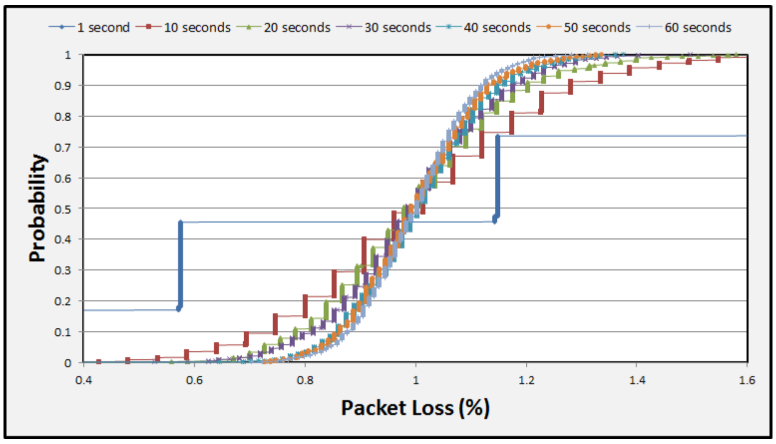


Figure 6-20: Example of packet loss measurements for flows with different duration

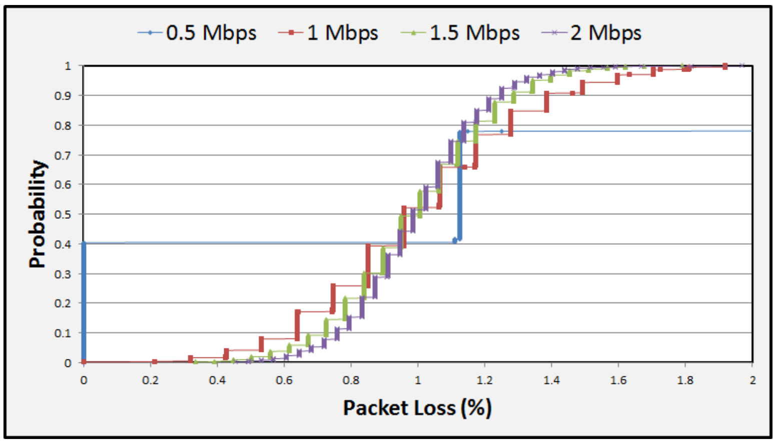


Figure 6-21: Example of packet loss measurements for flows with different data rate

Chapter 7

Conclusions and Future Work

7-1 Conclusions

This thesis explores the concept of network monitoring implemented in SDN architectures. This new network architecture provides a significant advantage over other because it is easier to tune up and introduce new functionalities. In terms of network monitoring, SDN allows to build a monitoring solution adjusted to the specific network needs. By using SDN the monitoring system is capable to obtain a complete view of the network that includes nodes, links and even ports. Furthermore, the solution is capable to obtain fine grained and accurate statistics, for every flow that passes trough the network.

This thesis builds on the previous work within the field, including some improvements over the link utilisation measurement method. First, a different data collecting scheme is proposed to reduce the overhead imposed over a single switch and to distribute the load over all the nodes through the network equally. Afterwards, in order to reduce the network overhead, a technique that aggregates all flows that go through the same network route is evaluated. The last improvement suggests to base the polling decisions not on some recurrent interval, but on the occurrence of certain events instead, eliminating the need of trade-off between overhead and accuracy. There is more additional overhead, for the cases when the polling requests are too often and decreased accuracy when the polling interval is too long.

Finally, this thesis proposes a new measuring method for packet loss, which, has proven to be accurate. This method is capable to determine the packet loss percentage for each link and also for any path. Furthermore, it is a passive method, thus imposing zero additional network overhead. It is not influenced by the network characteristics like the active probing methods that currently exist. The method is capable to provide statistics for every different type of service that passes trough the network.

Thus, this thesis answers the question of how monitoring can be achieved in SDN by implementing a monitoring component that uses the OpenFlow protocol to communicate with the hardware equipment. The component is responsible to poll switches for link usage and to gather packet loss statistics. The suggested approach brings the following improvements over the already existing monitoring solutions. Starting from a more up to date view of the network, including the state of the switches, ports and links, a fine grained and yet very accurate measurements with the price of low overhead.

7-2 Future Work

Possible extensions to the measurement schemes suggested in this thesis could be considered. The accuracy could be improved based on a combination of past statistics, link characteristics or weighted measurements results without imposing additional overhead. The adaptive timer requires more tuning, therefore, more research would be necessary on when more samples are needed and when less. Furthermore, more experiments in a real environment (i.e. with real traffic) are needed to fully evaluate the proposed measurement approaches. For the suggested packet loss method some questions need to be answered. For example, how much data is enough to take that the reported percentage of packet losses is not a random spike and how long before the data is too old to be considered valid.

Once there are suitable monitoring systems capable to provide the necessary performance and usage statistics, the next phase is the network optimization phase. The main goal of TE is to enhance the performance of an operational network, at both traffic and resource level. Network monitoring takes an important part in TE by measuring the traffic performance parameters. Additionally, today's traffic engineering in service provider networks works on coarse scale of several hours. This gives big enough time frame for offline TM estimation or it's deduction via regressed measurements. Unfortunately, this approach is not always viable, current IP traffic volume changes within seconds (or even miliseconds), which could lead to congestion and packet losses at the most crucial moment.

Since SDN is a new architecture still gaining popularity, there are also some questions that need to be answered in terms of routing. Obtaining an accurate and real time view of the network could bring more benefits and open more options. Monitoring the network is the first step towards a SDN forwarding protocol capable to provide sufficient QoS for all types of applications and traffic.

Bibliography

[1] G. Almes, S. Kalidindi, and M. Zekauskas. A One-way Delay Metric for IPPM. RFC 2679 (Proposed Standard), September 1999.

[2] G. Almes, S. Kalidindi, and M. Zekauskas. A One-way Packet Loss Metric for IPPM. RFC 2680 (Proposed Standard), September 1999.

[3] Jeffrey R. Ballard, Ian Rae, and Aditya Akella. Extensible and scalable network monitoring using OpenSAFE. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[4] Ali Begen, Tankut Akgul, and Mark Baugher. Watching Video over the Web: Part 1: Streaming Protocols. *IEEE Internet Computing*, 15(2):54–63, March 2011.

[5] Michael H. Behringer. Classifying network complexity. In *Proceedings of the 2009 workshop on Re-architecting the internet*, ReArch '09, pages 13–18, New York, NY, USA, 2009. ACM.

[6] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 8:1–8:12, New York, NY, USA, 2011. ACM.

[7] Rhys Bowden, Hung X. Nguyen, Nickolas Falkner, Simon Knight, and Matthew Roughan. Planarity of data networks. In *Proceedings of the 23rd International Teletraffic Congress*, ITC '11, pages 254–261. ITCP, 2011.

[8] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association.

[9] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, August 2007.

[10] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: a protection architecture for enterprise networks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[11] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: a retrospective on evolving SDN. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 85–90, New York, NY, USA, 2012. ACM.

[12] P. Chimento and J. Ishac. Defining Network Capacity. RFC 5136 (Informational), February 2008.

[13] Paolo Lucente Clarence Filsfils, Arman Maghbouleh. Best Practices in Network Planning and Traffic Engineering. Technical report, CISCO Systems, 2011.

[14] David Erickson. Beacon, URL: https://openflow.stanford.edu/display/Beacon/Home. Online, July 2013.

[15] eSecurityPlanet.com Staff. Sasser worms continue to threaten corporate productivity, URL: http://www.esecurityplanet.com/alerts/article.php/3349321/Sasser-Worms-Continue-to-Threaten-Corporate-Productivity.htm. Online, May 2004.

[16] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. *SIGPLAN Not.*, 46(9):279–291, September 2011.

[17] The Open Networking Foundation. OpenFlow Switch Specification v1.3.1 URL: https://www.opennetworking.org/images/stories/downloads/specification/openflow-spec-v1.3.1.pdf. Online, September 2012.

[18] Chuck Fraleigh, Sue Moon, Bryan Lyles, Chase Cotton, Mujahid Khan, Deb Moll, Rob Rockell, Ted Seely, and Christophe Diot. Packet-Level Traffic Measurements from the Sprint IP Backbone. *IEEE Network*, 17:6–16, 2003.

[19] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, October 2005.

[20] Miniwatts Marketing Group. World Internet Usage Statistics, URL: http://www.internetworldstats.com/stats.htm, December 2012.

[21] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[22] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[23] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: saving energy in data center networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[24] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. *SIGCOMM Comput. Commun. Rev.*, 42(4):473–478, September 2012.

[25] Stephen Hemminger. Abstract Network Emulation with NetEm, 2005.

[26] B. Huffaker, D. Plummer, D. Moore, and k. claffy. Topology discovery by active probing. In *Symposium on Applications and the Internet (SAINT)*, pages 90–96, Nara, Japan, Jan 2002. SAINT.

[27] IEEE. History of Ethernet, URL: http://standards.ieee.org/events/ethernet/history.html. Online, July 2013.

[28] Z Kerravala. Configuration management delivers business resiliency. Technical report, The Yankee Group, 2002.

[29] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[30] Murphy McCauley. About POX, URL: http://www.noxrepo.org/pox/about-pox/. Online, 2013.

[31] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[32] Marc Mendonça, Bruno Nunes Astuto, Xuan Nam Nguyen, Katia Obraczka, and Thierry Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks, June 2013. In Submission In Submission.

[33] J. E. Van Der Merwe, S. Rooney, I. M. Leslie, and S. A. Crosby. The Tempest - A Practical Framework for Network Programmability. *IEEE Network*, 12:20–28, 1997.

[34] B. Lekovic & P. V. Mieghem. Link state update policies for Quality of Service routing. In *Proc. 8th IEEE Symp. on Communications and Vehicular Technology in the Benelux (SCVT2001)*, pages 123–128, October 2001.

[35] Piet Van Mieghem. *Data communications networking.* Techne Press, 2006.

[36] D. Mills. Network Time Protocol (Version 3) Specification, Implementation, 1992.

[37] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 1–9, New York, NY, USA, 2008. ACM.

[38] Marcelo R. Nascimento, Christian E. Rothenberg, Marcos R. Salvador, Carlos N. A. Corrêa, Sidney C. de Lucena, and Maurício F. Magalhães. Virtual routers as a service: the RouteFlow approach leveraging software-defined networks. In *Proceedings of the 6th International Conference on Future Internet Technologies*, CFI '11, pages 34–37, New York, NY, USA, 2011. ACM.

[39] Marcelo Ribeiro Nascimento, Christian Esteve Rothenberg, Marcos Rogério Salvador, and Maurício Ferreira Magalhães. QuagFlow: partnering Quagga with OpenFlow. *SIGCOMM Comput. Commun. Rev.*, 40(4):441–442, August 2010.

[40] Big Switch Networks. Floodlight, an SDN controller, URL: http://www.projectfloodlight.org/floodlight/. Online, July 2013.

[41] ONF. Software-defined Networking: The New Norm for Networks. White Paper, April 2012.

[42] Ruoming Pang, Mark Allman, Mike Bennett, Jason Lee, Vern Paxson, and Brian Tierney. A first look at modern enterprise traffic. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, IMC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[43] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP Performance Metrics. RFC 2330 (Informational), May 1998.

[44] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. e.a.: Extending networking into the virtualization layer. In *In: 8th ACM Workshop on Hot Topics inNetworks (HotNets-VIII).New YorkCity,NY(October 2009*, 2009.

[45] Christian Esteve Rothenberg, Marcelo Ribeiro Nascimento, Marcos Rogerio Salvador, Carlos Nilton Araujo Corrêa, Sidney Cunha de Lucena, and Robert Raszuk. Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 13–18, New York, NY, USA, 2012. ACM.

[46] R. E. Staehler B.J. Yokelson S. Horing, J. Z. Menard. Stored program controlled network: Overview. *Bell System Technical Journal*, 61:1579–1588, September 1982.

[47] Sandvine. Global internet phenomena report. Technical report, Sandvine, 2011.

[48] sFlow. Traffic Monitoring using sFlow, URL: http://www.sflow.org/sFlowOverview.pdf. Online, July 2013.

[49] W. Stallings. SNMP and SNMPv2: the infrastructure for network management. *Comm. Mag.*, 36(3):37–43, March 1998.

[50] Lalith Suresh, Julius Schulz-Zander, Ruben Merz, Anja Feldmann, and Teresa Vazao. Towards programmable enterprise WLANS with Odin. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 115–120, New York, NY, USA, 2012. ACM.

[51] Silver Peak Systems. How to Accurately Detect and Correct Packet Loss, URL: http://www.silver-peak.com/info-center/how-accurately-detect-and-correct-packet-loss. Online, July 2013.

[52] Amin Tootoonchian and Yashar Ganjali. HyperFlow: a distributed control plane for OpenFlow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[53] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. OpenTM: traffic matrix estimator for OpenFlow networks. In *Proceedings of the 11th international conference on Passive and active measurement*, PAM'10, pages 201–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[54] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. In *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[55] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM.

[56] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[57] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2):136–141, 2013.

[58] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. FlowSense: monitoring network utilization with zero measurement cost. In *Proceedings of the 14th international conference on Passive and Active Measurement*, PAM'13, pages 31–41, Berlin, Heidelberg, 2013. Springer-Verlag.

[59] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 29–42, Berkeley, CA, USA, 2013. USENIX Association.

[60] Qi Zhao, Zihui Ge, Jia Wang, and Jun Xu. Robust traffic matrix estimation with imperfect information: making use of multiple data sources. *SIGMETRICS Perform. Eval. Rev.*, 34(1):133–144, June 2006.

[61] T. S. Eugene Ng Zheng Cai, Alan L. Cox. Maestro: A System for Scalable OpenFlow Control. Technical report, Rice University, 2011.

Appendix A

Glossary

ACL: Access Control List

API: Application Programmable Interface

AS: Autonomous System

BE: Best Effort

BGP: Border Gateway Protocol

CPU: Central Processor Unit

UTC: Coordinated Universal Time

DPID: Datapath identification

DPI: Deep Packet Inspection

ECMP: Equal Cost Multipath

GPS: Global Positioning System

ID: Identification

IGP: Interior Gateway Protocol

ICMP: Internet Control Message Protocol

IP: Internet Protocol

ISP: Internet Service Provider

IDS: Intrusion Detection System

LDP: Label Distribution Protocol

LSP: Label-Switched Path

MIB: Management Information Base

NCP: Network Control Point

NOS: Network Operating System

NTP: Network Time protocol

OSI: Open System Interconnection

OF: OpenFlow

QoS: Quality of Service

RSVP: Resource Reservation Protocol

RTT: Round Trip Time

RF: RouteFlow

RFCP: RouteFlow Control Platform

RCP: Routing Control Platform

SLA: Service Level Agreements

SNMP: Simple Network Management Protocol

SDN: Software Defined Networking

TE: Traffic Engineering

TM: Traffic Matrix

TCP: Transmission Control Protocol

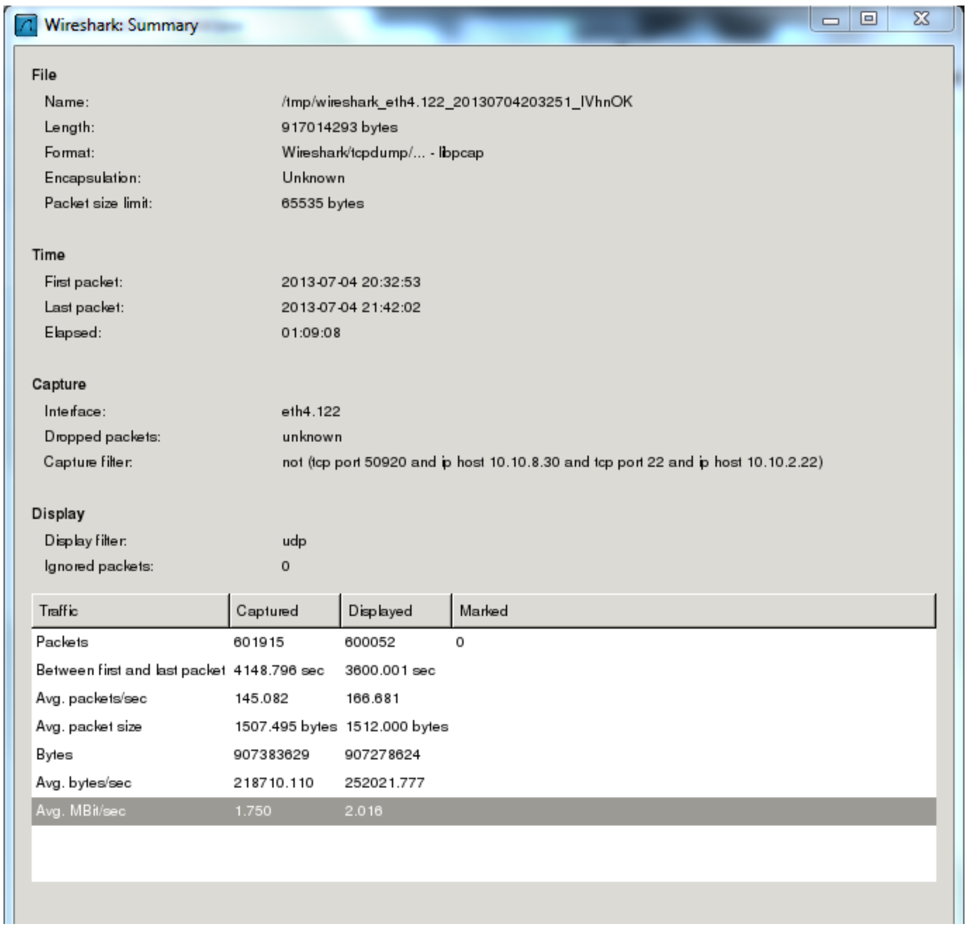
UDP: User Datagram Protocol

VLAN: Virtual Local Area Network

VM: Virtual Machine

Appendix B

Results



Wireshark: Summary

File

Name:	/tmp/wireshark_eth4.122_20130704203251_lVhnOK
Length:	917014293 bytes
Format:	Wireshark/tcpdump/... - libpcap
Encapsulation:	Unknown
Packet size limit:	65535 bytes

Time

First packet:	2013-07-04 20:32:53
Last packet:	2013-07-04 21:42:02
Elapsed:	01:09:08

Capture

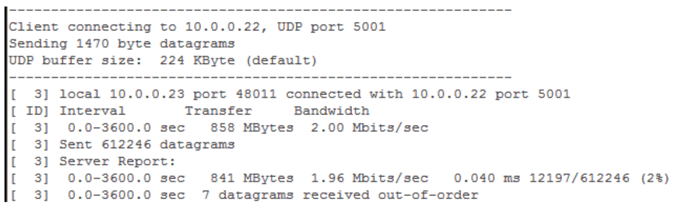
Interface:	eth4.122
Dropped packets:	unknown
Capture filter:	not (tcp port 50920 and ip host 10.10.8.30 and tcp port 22 and ip host 10.10.2.22)

Display

Display filter:	udp
Ignored packets:	0

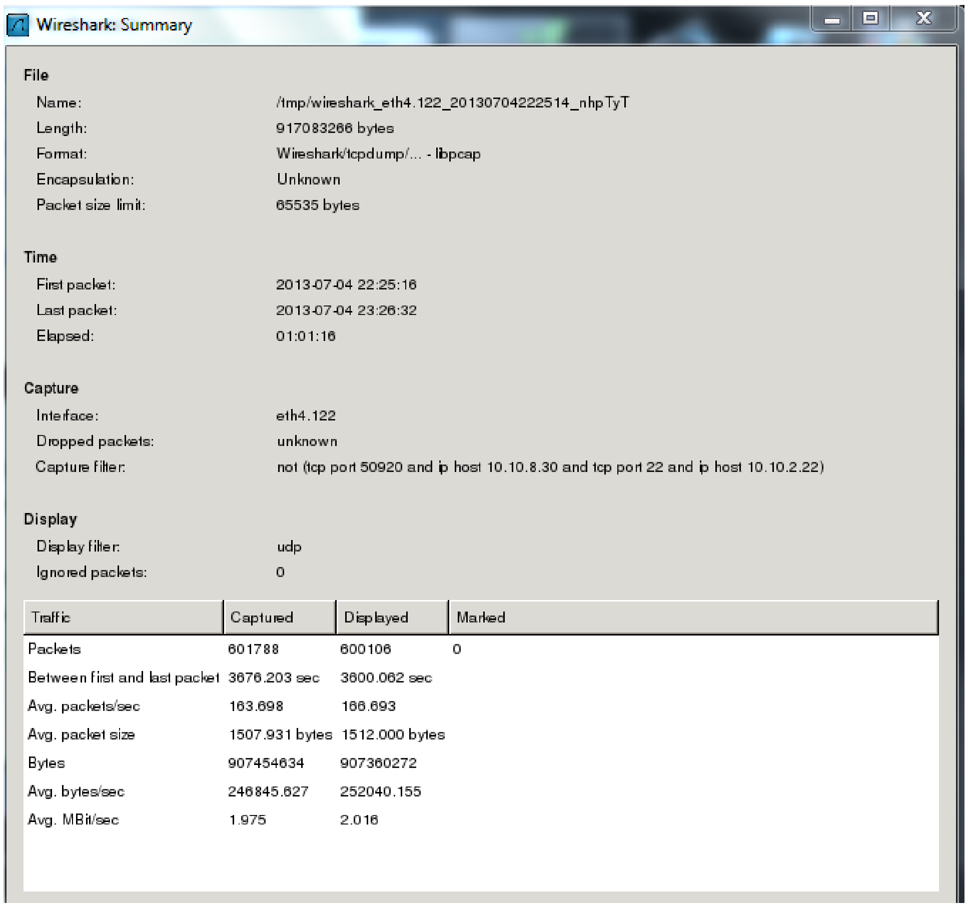
Traffic	Captured	Displayed	Marked
Packets	601915	600052	0
Between first and last packet	4148.796 sec	3600.001 sec	
Avg. packets/sec	145.082	166.681	
Avg. packet size	1507.495 bytes	1512.000 bytes	
Bytes	907383629	907278624	
Avg. bytes/sec	218710.110	252021.777	
Avg. MBit/sec	1.750	2.016	

Figure B-1: Wireshark report for Last Switch measurements



```
------------------------------------------------------------
Client connecting to 10.0.0.22, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  224 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.23 port 48011 connected with 10.0.0.22 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[  3] Server Report:
[  3]  0.0-3600.0 sec   841 MBytes  1.96 Mbits/sec   0.040 ms 12197/612246 (2%)
[  3]  0.0-3600.0 sec  7 datagrams received out-of-order
```

Figure B-2: Iperf Server report for Last Switch measurements



Wireshark: Summary

File

Name:	/tmp/wireshark_eth4.122_20130704222514_nhpTyT
Length:	917083266 bytes
Format:	Wireshark/tcpdump/... - libpcap
Encapsulation:	Unknown
Packet size limit:	65535 bytes

Time

First packet:	2013-07-04 22:25:16
Last packet:	2013-07-04 23:26:32
Elapsed:	01:01:16

Capture

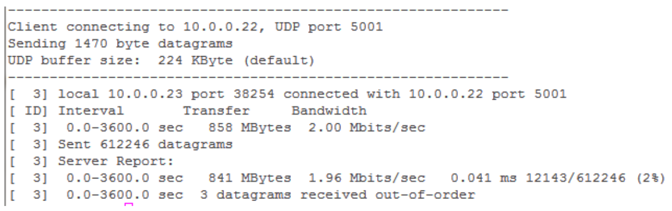
Interface:	eth4.122
Dropped packets:	unknown
Capture filter:	not (tcp port 50920 and ip host 10.10.8.30 and tcp port 22 and ip host 10.10.2.22)

Display

Display filter:	udp
Ignored packets:	0

Traffic	Captured	Displayed	Marked
Packets	601788	600106	0
Between first and last packet	3676.203 sec	3600.062 sec	
Avg. packets/sec	163.698	166.693	
Avg. packet size	1507.931 bytes	1512.000 bytes	
Bytes	907454634	907360272	
Avg. bytes/sec	246845.627	252040.155	
Avg. MBit/sec	1.975	2.016	

Figure B-3: Wireshark report for Round Robin measurements



```
------------------------------------------------------------
Client connecting to 10.0.0.22, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  224 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.23 port 38254 connected with 10.0.0.22 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[  3] Server Report:
[  3]  0.0-3600.0 sec   841 MBytes  1.96 Mbits/sec   0.041 ms 12143/612246 (2%)
[  3]  0.0-3600.0 sec  3 datagrams received out-of-order
```

Figure B-4: Iperf Server report for Round Robin measurements

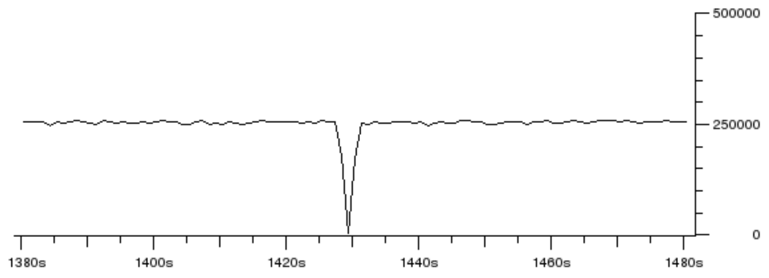
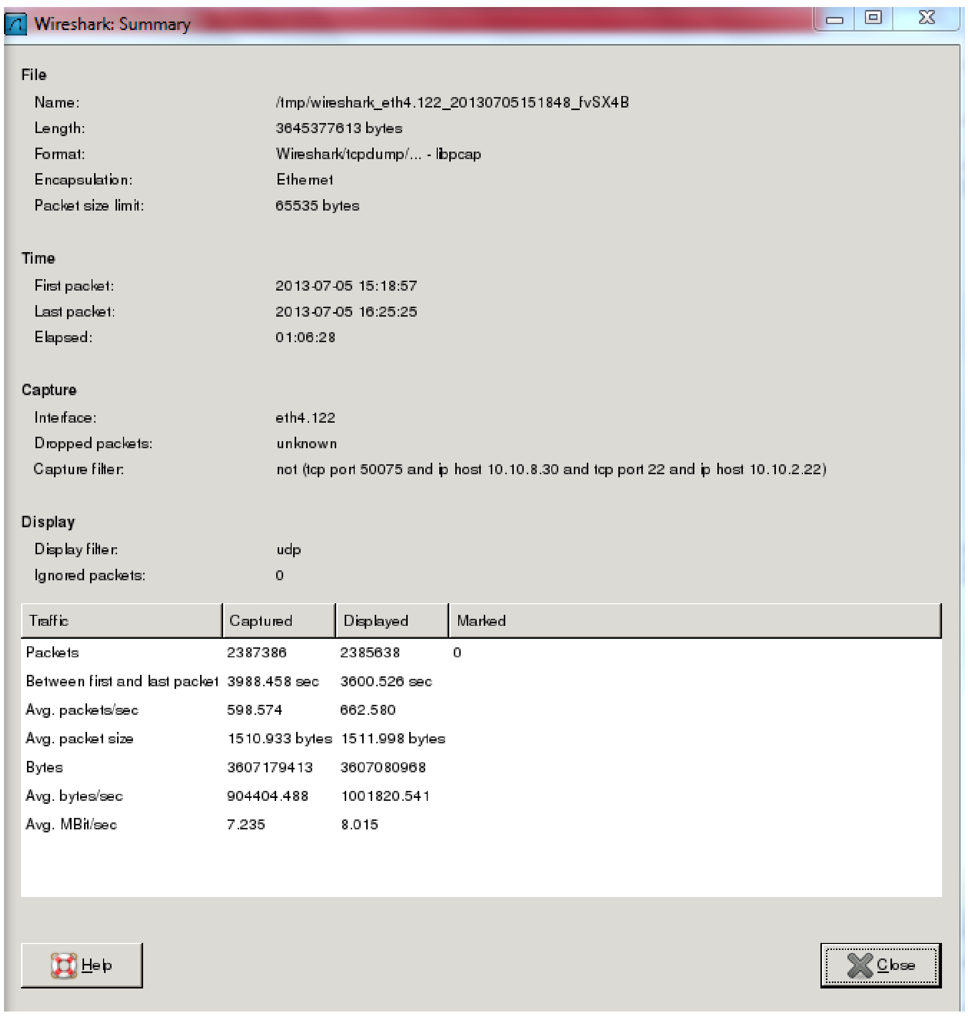


Figure B-5: Round Robin scheduling Wireshark results



File

Name:	/tmp/wireshark_eth4.122_20130705151848_fvSX4B
Length:	3645377613 bytes
Format:	Wireshark/tcpdump/... - libpcap
Encapsulation:	Ethernet
Packet size limit:	65535 bytes

Time

First packet:	2013-07-05 15:18:57
Last packet:	2013-07-05 16:25:25
Elapsed:	01:06:28

Capture

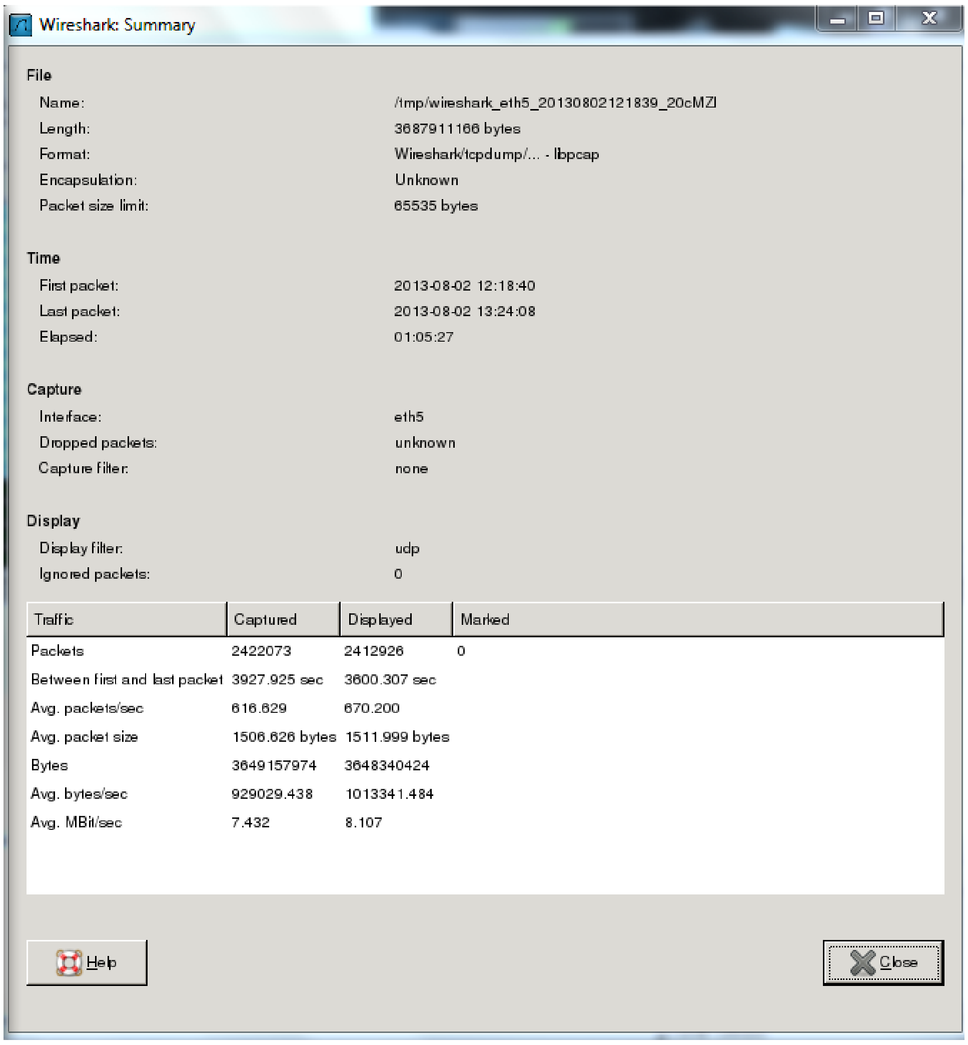
Interface:	eth4.122
Dropped packets:	unknown
Capture filter:	not (tcp port 50075 and ip host 10.10.8.30 and tcp port 22 and ip host 10.10.2.22)

Display

Display filter:	udp
Ignored packets:	0

Traffic	Captured	Displayed	Marked
Packets	2387386	2385638	0
Between first and last packet	3988.458 sec	3600.526 sec	
Avg. packets/sec	598.574	662.580	
Avg. packet size	1510.933 bytes	1511.998 bytes	
Bytes	3607179413	3607080968	
Avg. bytes/sec	904404.488	1001820.541	
Avg. MBit/sec	7.235	8.015	

Figure B-6: Wireshark report for Aggregated measurements when using Round Robin



File

Name:	/tmp/wireshark_eth5_20130802121839_20cMZl
Length:	3687911166 bytes
Format:	Wireshark/tcpdump/... - libpcap
Encapsulation:	Unknown
Packet size limit:	65535 bytes

Time

First packet:	2013-08-02 12:18:40
Last packet:	2013-08-02 13:24:08
Elapsed:	01:05:27

Capture

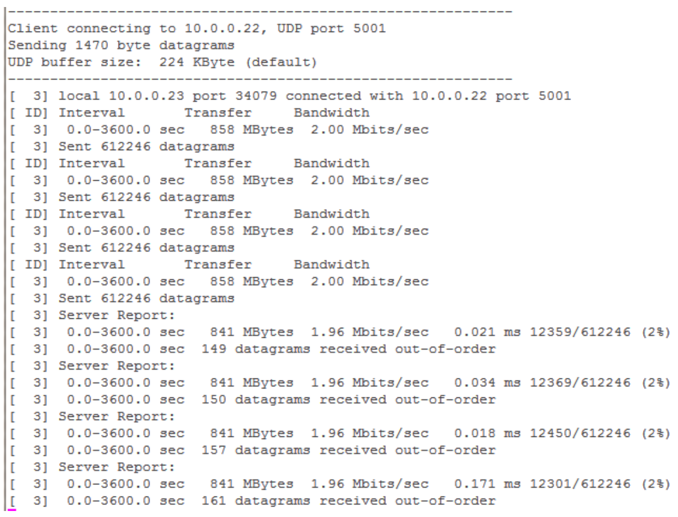
Interface:	eth5
Dropped packets:	unknown
Capture filter:	none

Display

Display filter:	udp
Ignored packets:	0

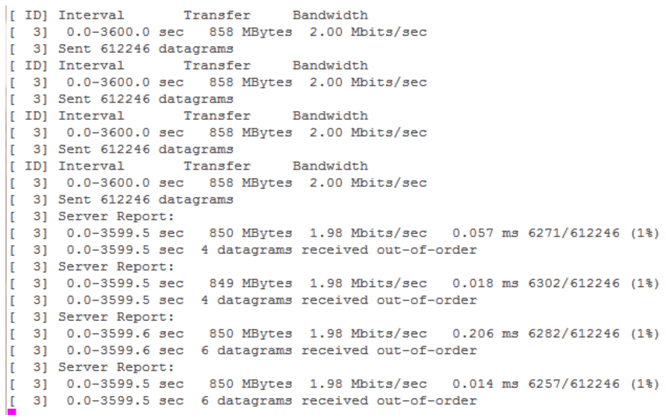
Traffic	Captured	Displayed	Marked
Packets	2422073	2412926	0
Between first and last packet	3927.925 sec	3600.307 sec	
Avg. packets/sec	616.629	670.200	
Avg. packet size	1506.626 bytes	1511.999 bytes	
Bytes	3649157974	3648340424	
Avg. bytes/sec	929029.438	1013341.484	
Avg. MBit/sec	7.432	8.107	

Figure B-7: Wireshark report for Aggregated measurements when using Last Switch



```
------------------------------------------------------------
Client connecting to 10.0.0.22, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  224 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.23 port 34079 connected with 10.0.0.22 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[  3] Server Report:
[  3]  0.0-3600.0 sec   841 MBytes  1.96 Mbits/sec   0.021 ms 12359/612246 (2%)
[  3]  0.0-3600.0 sec  149 datagrams received out-of-order
[  3] Server Report:
[  3]  0.0-3600.0 sec   841 MBytes  1.96 Mbits/sec   0.034 ms 12369/612246 (2%)
[  3]  0.0-3600.0 sec  150 datagrams received out-of-order
[  3] Server Report:
[  3]  0.0-3600.0 sec   841 MBytes  1.96 Mbits/sec   0.018 ms 12450/612246 (2%)
[  3]  0.0-3600.0 sec  157 datagrams received out-of-order
[  3] Server Report:
[  3]  0.0-3600.0 sec   841 MBytes  1.96 Mbits/sec   0.171 ms 12301/612246 (2%)
[  3]  0.0-3600.0 sec  161 datagrams received out-of-order
```

Figure B-8: Iperf Server report for Aggregated measurements when using Round Robin



```
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-3600.0 sec   858 MBytes  2.00 Mbits/sec
[  3] Sent 612246 datagrams
[  3] Server Report:
[  3]  0.0-3599.5 sec   850 MBytes  1.98 Mbits/sec   0.057 ms 6271/612246 (1%)
[  3]  0.0-3599.5 sec  4 datagrams received out-of-order
[  3] Server Report:
[  3]  0.0-3599.5 sec   849 MBytes  1.98 Mbits/sec   0.018 ms 6302/612246 (1%)
[  3]  0.0-3599.5 sec  4 datagrams received out-of-order
[  3] Server Report:
[  3]  0.0-3599.6 sec   850 MBytes  1.98 Mbits/sec   0.206 ms 6282/612246 (1%)
[  3]  0.0-3599.6 sec  6 datagrams received out-of-order
[  3] Server Report:
[  3]  0.0-3599.5 sec   850 MBytes  1.98 Mbits/sec   0.014 ms 6257/612246 (1%)
[  3]  0.0-3599.5 sec  6 datagrams received out-of-order
```

Figure B-9: Iperf Server report for Aggregated measurements when using Last Switch

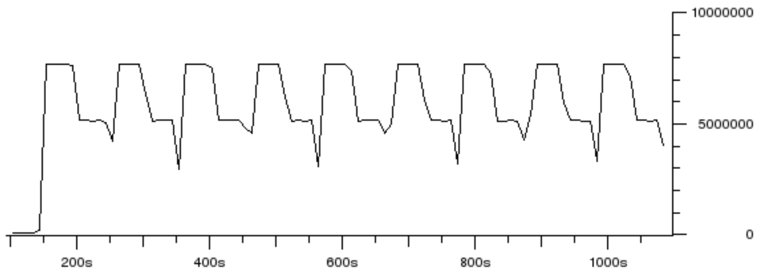


Figure B-10: Data Rate for the link between Node 3 to Node 4 as shown by Wireshark

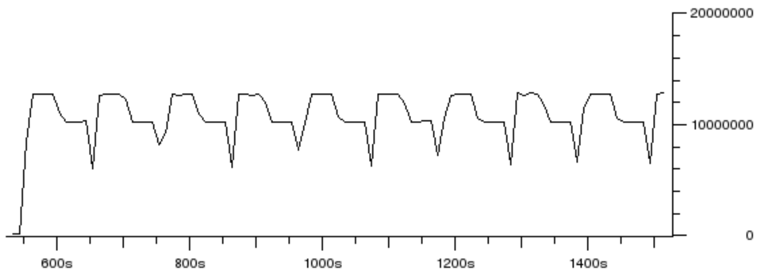


Figure B-11: Data Rate for the link between Node 2 to Node 3 as shown by Wireshark

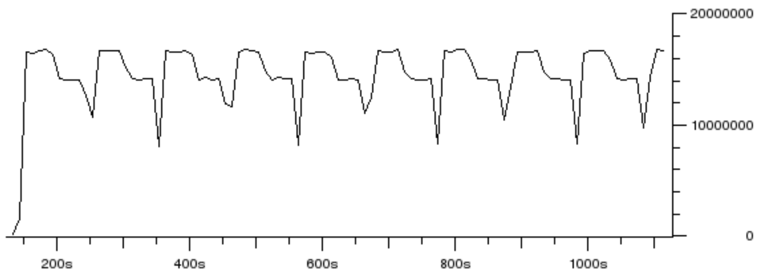
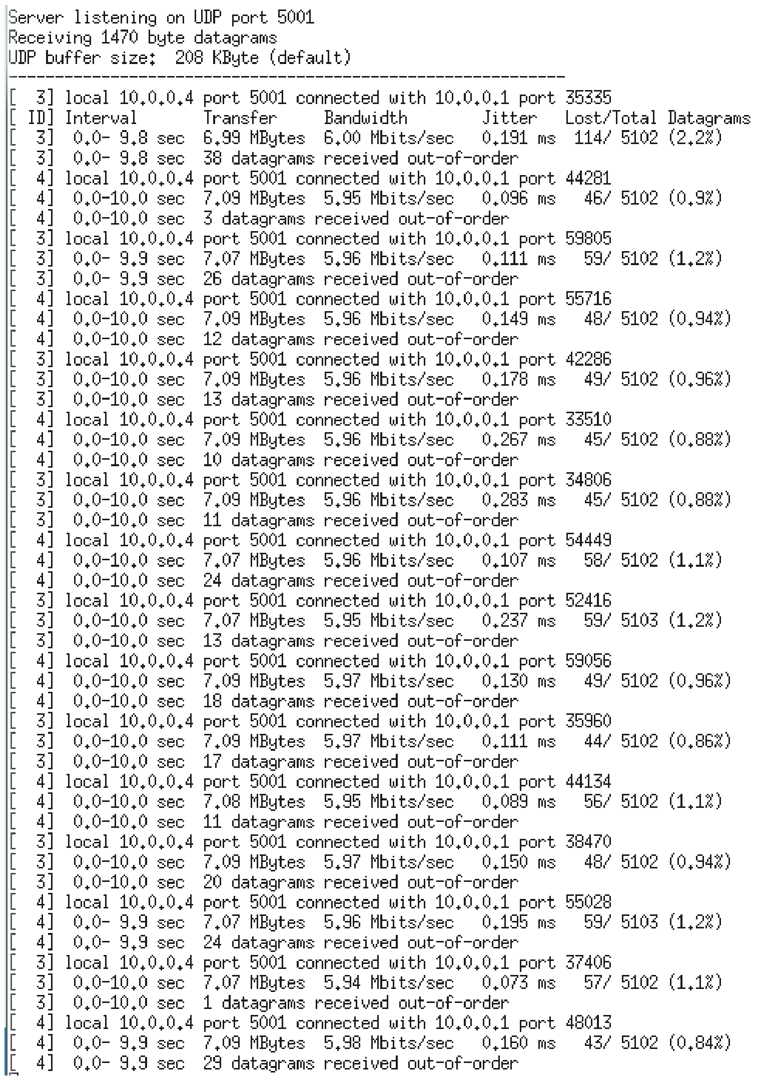
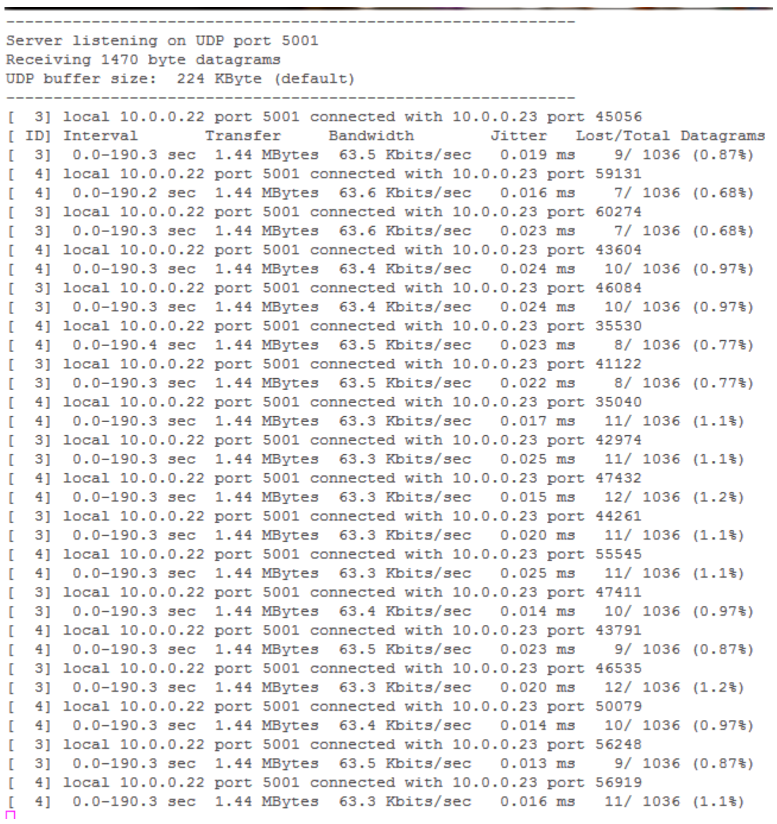


Figure B-12: Data Rate for the link between Node 1 to Node 2 as shown by Wireshark



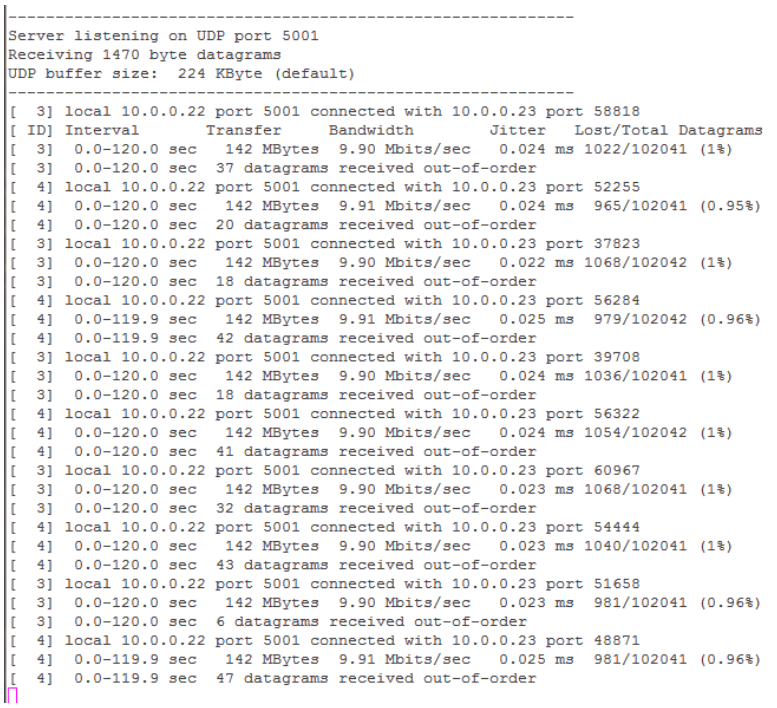
```
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 35335
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0- 9.8 sec  6.99 MBytes  6.00 Mbits/sec   0.191 ms  114/ 5102 (2.2%)
[  3]  0.0- 9.8 sec  38 datagrams received out-of-order
[  4] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 44281
[  4]  0.0-10.0 sec  7.09 MBytes  5.95 Mbits/sec   0.096 ms   46/ 5102 (0.9%)
[  4]  0.0-10.0 sec  3 datagrams received out-of-order
[  3] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 59805
[  3]  0.0- 9.9 sec  7.07 MBytes  5.96 Mbits/sec   0.111 ms   59/ 5102 (1.2%)
[  3]  0.0- 9.9 sec  26 datagrams received out-of-order
[  4] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 55716
[  4]  0.0-10.0 sec  7.09 MBytes  5.96 Mbits/sec   0.149 ms   48/ 5102 (0.94%)
[  4]  0.0-10.0 sec  12 datagrams received out-of-order
[  3] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 42286
[  3]  0.0-10.0 sec  7.09 MBytes  5.96 Mbits/sec   0.178 ms   49/ 5102 (0.96%)
[  3]  0.0-10.0 sec  13 datagrams received out-of-order
[  4] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 33510
[  4]  0.0-10.0 sec  7.09 MBytes  5.96 Mbits/sec   0.267 ms   45/ 5102 (0.88%)
[  4]  0.0-10.0 sec  10 datagrams received out-of-order
[  3] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 34806
[  3]  0.0-10.0 sec  7.09 MBytes  5.96 Mbits/sec   0.283 ms   45/ 5102 (0.88%)
[  3]  0.0-10.0 sec  11 datagrams received out-of-order
[  4] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 54449
[  4]  0.0-10.0 sec  7.07 MBytes  5.96 Mbits/sec   0.107 ms   58/ 5102 (1.1%)
[  4]  0.0-10.0 sec  24 datagrams received out-of-order
[  3] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 52416
[  3]  0.0-10.0 sec  7.07 MBytes  5.95 Mbits/sec   0.237 ms   59/ 5103 (1.2%)
[  3]  0.0-10.0 sec  13 datagrams received out-of-order
[  4] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 59056
[  4]  0.0-10.0 sec  7.09 MBytes  5.97 Mbits/sec   0.130 ms   49/ 5102 (0.96%)
[  4]  0.0-10.0 sec  18 datagrams received out-of-order
[  3] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 35960
[  3]  0.0-10.0 sec  7.09 MBytes  5.97 Mbits/sec   0.111 ms   44/ 5102 (0.86%)
[  3]  0.0-10.0 sec  17 datagrams received out-of-order
[  4] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 44134
[  4]  0.0-10.0 sec  7.08 MBytes  5.95 Mbits/sec   0.089 ms   56/ 5102 (1.1%)
[  4]  0.0-10.0 sec  11 datagrams received out-of-order
[  3] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 38470
[  3]  0.0-10.0 sec  7.09 MBytes  5.97 Mbits/sec   0.150 ms   48/ 5102 (0.94%)
[  3]  0.0-10.0 sec  20 datagrams received out-of-order
[  4] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 55028
[  4]  0.0- 9.9 sec  7.07 MBytes  5.96 Mbits/sec   0.195 ms   59/ 5103 (1.2%)
[  4]  0.0- 9.9 sec  24 datagrams received out-of-order
[  3] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 37406
[  3]  0.0-10.0 sec  7.07 MBytes  5.94 Mbits/sec   0.073 ms   57/ 5102 (1.1%)
[  3]  0.0-10.0 sec  1 datagrams received out-of-order
[  4] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 48013
[  4]  0.0- 9.9 sec  7.09 MBytes  5.98 Mbits/sec   0.160 ms   43/ 5102 (0.84%)
[  4]  0.0- 9.9 sec  29 datagrams received out-of-order
```

Figure B-13: Example server output



```
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  224 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 45056
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-190.3 sec  1.44 MBytes  63.5 Kbits/sec   0.019 ms    9/ 1036 (0.87%)
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 59131
[  4]  0.0-190.2 sec  1.44 MBytes  63.6 Kbits/sec   0.016 ms    7/ 1036 (0.68%)
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 60274
[  3]  0.0-190.3 sec  1.44 MBytes  63.6 Kbits/sec   0.023 ms    7/ 1036 (0.68%)
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 43604
[  4]  0.0-190.3 sec  1.44 MBytes  63.4 Kbits/sec   0.024 ms   10/ 1036 (0.97%)
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 46084
[  3]  0.0-190.3 sec  1.44 MBytes  63.4 Kbits/sec   0.024 ms   10/ 1036 (0.97%)
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 35530
[  4]  0.0-190.4 sec  1.44 MBytes  63.5 Kbits/sec   0.023 ms    8/ 1036 (0.77%)
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 41122
[  3]  0.0-190.3 sec  1.44 MBytes  63.5 Kbits/sec   0.022 ms    8/ 1036 (0.77%)
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 35040
[  4]  0.0-190.3 sec  1.44 MBytes  63.3 Kbits/sec   0.017 ms   11/ 1036 (1.1%)
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 42974
[  3]  0.0-190.3 sec  1.44 MBytes  63.3 Kbits/sec   0.025 ms   11/ 1036 (1.1%)
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 47432
[  4]  0.0-190.3 sec  1.44 MBytes  63.3 Kbits/sec   0.015 ms   12/ 1036 (1.2%)
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 44261
[  3]  0.0-190.3 sec  1.44 MBytes  63.3 Kbits/sec   0.020 ms   11/ 1036 (1.1%)
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 55545
[  4]  0.0-190.3 sec  1.44 MBytes  63.3 Kbits/sec   0.025 ms   11/ 1036 (1.1%)
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 47411
[  3]  0.0-190.3 sec  1.44 MBytes  63.4 Kbits/sec   0.014 ms   10/ 1036 (0.97%)
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 43791
[  4]  0.0-190.3 sec  1.44 MBytes  63.5 Kbits/sec   0.023 ms    9/ 1036 (0.87%)
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 46535
[  3]  0.0-190.3 sec  1.44 MBytes  63.3 Kbits/sec   0.020 ms   12/ 1036 (1.2%)
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 50079
[  4]  0.0-190.3 sec  1.44 MBytes  63.4 Kbits/sec   0.014 ms   10/ 1036 (0.97%)
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 56248
[  3]  0.0-190.3 sec  1.44 MBytes  63.5 Kbits/sec   0.013 ms    9/ 1036 (0.87%)
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 56919
[  4]  0.0-190.3 sec  1.44 MBytes  63.3 Kbits/sec   0.016 ms   11/ 1036 (1.1%)
```

Figure B-14: 64kbps tests server output



```
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  224 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 58818
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-120.0 sec   142 MBytes  9.90 Mbits/sec   0.024 ms 1022/102041 (1%)
[  3]  0.0-120.0 sec  37 datagrams received out-of-order
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 52255
[  4]  0.0-120.0 sec   142 MBytes  9.91 Mbits/sec   0.024 ms  965/102041 (0.95%)
[  4]  0.0-120.0 sec  20 datagrams received out-of-order
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 37823
[  3]  0.0-120.0 sec   142 MBytes  9.90 Mbits/sec   0.022 ms 1068/102042 (1%)
[  3]  0.0-120.0 sec  18 datagrams received out-of-order
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 56284
[  4]  0.0-119.9 sec   142 MBytes  9.91 Mbits/sec   0.025 ms  979/102042 (0.96%)
[  4]  0.0-119.9 sec  42 datagrams received out-of-order
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 39708
[  3]  0.0-120.0 sec   142 MBytes  9.90 Mbits/sec   0.024 ms 1036/102041 (1%)
[  3]  0.0-120.0 sec  18 datagrams received out-of-order
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 56322
[  4]  0.0-120.0 sec   142 MBytes  9.90 Mbits/sec   0.024 ms 1054/102042 (1%)
[  4]  0.0-120.0 sec  41 datagrams received out-of-order
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 60967
[  3]  0.0-120.0 sec   142 MBytes  9.90 Mbits/sec   0.023 ms 1068/102041 (1%)
[  3]  0.0-120.0 sec  32 datagrams received out-of-order
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 54444
[  4]  0.0-120.0 sec   142 MBytes  9.90 Mbits/sec   0.023 ms 1040/102041 (1%)
[  4]  0.0-120.0 sec  43 datagrams received out-of-order
[  3] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 51658
[  3]  0.0-120.0 sec   142 MBytes  9.90 Mbits/sec   0.023 ms  981/102041 (0.96%)
[  3]  0.0-120.0 sec  6 datagrams received out-of-order
[  4] local 10.0.0.22 port 5001 connected with 10.0.0.23 port 48871
[  4]  0.0-119.9 sec   142 MBytes  9.91 Mbits/sec   0.025 ms  981/102041 (0.96%)
[  4]  0.0-119.9 sec  47 datagrams received out-of-order
```

Figure B-15: 10Mbps tests server output