

Engineering Enterprise Networks with SDN

Nitin Agarwal

A Thesis Submitted to
Indian Institute of Technology Hyderabad
In Partial Fulfillment of the Requirements for
The Degree of Master of Technology



Department of Computer Science and Engineering

June 2014

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

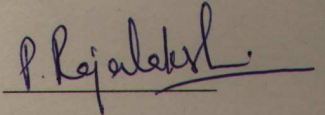
(Signature)

(Nitin Agarwal)

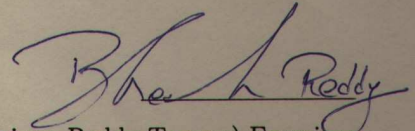
(Roll No.)

Approval Sheet

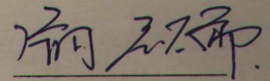
This Thesis entitled Engineering Enterprise Networks with SDN by Nitin Agarwal is approved for the degree of Master of Technology from IIT Hyderabad



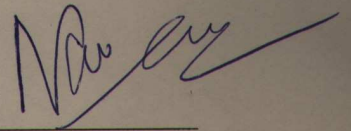
(Dr. P. Rajalakshmi) Examiner
Dept. of Electrical Engineering
IITH



(Dr. Bheemarjuna Reddy Tamma) Examiner
Dept. of Computer Science and Engineering
IITH



(Dr. Kotaro Kataoka) Adviser
Dept. of Computer Science and Engineering
IITH



(Dr. Naveen Sivadasan) Chairman
Dept. of Computer Science and Engineering
IITH

Acknowledgements

Foremost, I would like to express my sincere gratitude to my adviser Dr. Kotaro Kataoka for his valuable guidance, constant encouragement, motivation, enthusiasm, and immense knowledge. Without his patience and support, this dissertation would not have been possible. Many individuals contributed in many different ways to the completion of this thesis. I am deeply grateful for their support, feedback and constant encouragement. Finally, I thank my family for supporting me throughout my studies at the institute.

I would like to make a special mention of the excellent facilities provided by my institute, IIT Hyderabad.

Abstract

Today's networks are growing in terms of bandwidth, number of devices, variety of applications, and various front-end and back-end technologies. Current network architecture is not sufficient for scaling, managing and monitoring them. In this thesis, we explore SDN to address scalability and monitoring issue in growing networks such as IITH campus network. SDN architecture separates the control plane and data plane of a networking device. SDN provides a single control plane (or centralized way) to configure, manage and monitor them more effectively.

Scalability of Ethernet is a known issue where communication is disturbed by a large number of nodes in a single broadcast domain. This thesis proposes Extensible Transparent Filter (ETF) for Ethernet using SDN. ETF suppresses broadcast traffic in a broadcast domain by forwarding the broadcast packet to only selected port of a switch through which the target host of that packet is reachable. ETF maintains both consistent functionality and backward compatibility with existing protocols that work with broadcast of a packet.

Nowadays, flow-level details of network traffic are the major requirements of many network monitoring applications such as anomaly detection, traffic accounting etc. Packet sampling based solutions (such as NetFlow) provide flow-level details of network traffic. However, they are inadequate for several monitoring applications. This thesis proposes Network Monitor (NetMon) for OpenFlow networks, which includes the implementation of a few flow-based metrics to determine the state of the network and a Device Logger. NetMon uses a push-based approach to achieve its goals with complete flow-level details. NetMon determines the fraction of useful flows for each host in the network. It calculates out-degree and in-degree based on the IP address, for each hosts in the network. NetMon classifies the host as a client, server or peer-to-peer node, based on the number of source ports and active flows. Device Logger records the device (MAC address and IP address) and its location (Switch DPID and Port No). Device Logger helps to identify owners (devices) of an IP address within a particular time period.

This thesis also discusses the practical deployment and operation of SDN. A small SDN network has been deployed in IIT Hyderabad campus. Both, ETF and NetMon are functional in the SDN network. ETF and NetMon were developed using Floodlight which is an open source SDN controller. ETF and NetMon improve scalability and monitoring of enterprise networks as an enhancement to existing networks using SDN.

Contents

Declaration	ii
Approval Sheet	iii
Acknowledgements	iv
Abstract	v
Nomenclature	vii
1 Introduction	1
1.1 Scaling a Broadcast Domain of Ethernet	1
1.2 Network Flow Monitoring	2
1.3 Overview of the Work	3
1.4 Thesis Outline	3
2 SDN and OpenFlow	5
2.1 Software Defined Networking	5
2.2 OpenFlow Protocol	6
3 Scaling a Broadcast Domain of Ethernet	8
3.1 Handling a Large-scale Network	9
3.2 Related Work	10
3.3 System Design	11
3.4 System Implementation	12
3.4.1 Data Structures	13
3.4.2 Discovering DHCP Servers	13
3.4.3 Operating DHCP in Suppress Mode	14
3.4.4 Operating ARP Request in Suppress Mode	14
3.4.5 Flooding for ARP	15
3.4.6 Proactive Flow Rules	15
3.4.7 Mobility	16
3.4.8 Garbage Collection	16
3.4.9 Rate Limiter	16
3.4.10 IP Conflict Detection	16
3.5 Evaluation	17
3.5.1 Proof of Concept in Virtualized Environment	17
3.5.2 Estimated Scalability of ETF and Broadcast Domain	17

3.5.3	ETF Performance under Actual Deployment	18
3.6	Discussion	20
3.6.1	Improving the Performance	20
3.6.2	Scaling IPv6 Mechanisms	20
3.6.3	Avoiding ARP Poisoning	21
4	Network Flow Monitoring	22
4.1	Network Flow Metrics	23
4.2	Related Work	24
4.3	System Design	24
4.4	System Implementation	25
4.4.1	Data Store	25
4.4.2	SDN Controller	26
4.4.3	Data Collector	26
4.4.4	Device Logger	27
4.4.5	Fraction of Useful Flows	28
4.4.6	Out-degree, In-degree, Port-degree and Flow-count	30
4.4.7	Web Interface	31
4.5	Evaluation	31
4.6	Discussion	33
5	SDN in IITH Campus	36
6	Conclusions and Future Work	38
6.1	Conclusions	38
6.2	Future Work	38
	References	40

Chapter 1

Introduction

Network engineering consists of planning, designing and implementing computer networks. It also includes addressing and routing, performance tuning, resource management, traffic engineering and network monitoring. Today's networks are growing in terms of bandwidth, number of devices supported, various network access technologies and services used. Enterprise networks also have similar characteristics and they have thousands of users across a company's diverse geographical locations. They are complex as different sites are connected to each other and tight control policies are applied. They are diverse in network access technologies, such as LAN, Wi-Fi and VPN etc. Enterprise networks run various services such as video streaming, voice communication and sensor data etc. Current network architecture is not sufficient for scaling, managing and monitoring them. So it is natural to look for a new technology for managing, operating and engineering them. Software defined networking (SDN) is a networking architecture which has been gaining momentum in past few years. SDN provides a way for more flexible, programmable, vendor-neutral, cost-effective and innovative network architecture. It promises to simplify network management, monitoring and control.

1.1 Scaling a Broadcast Domain of Ethernet

Ethernet, a layer 2 networking technology, has been broadly used for interconnecting networks and end-systems. Emergence of Wi-Fi enabled devices, like laptops, tablets and smart phones has been pushing the growth of a layer 2 segment using Ethernet. In an IPv6 network, the size of each segment will be naturally much larger because of a /64 prefix length.

Current IP network architecture is based on protocols that actively use broadcast packets. According to Cisco, the maximum number of hosts in a broadcast domain should be between 200 and 500 [1]. This number depends on the broadcast based services which hosts run. For scalability of Ethernet, the impact of broadcast packets sent by a large number of end-systems is one of the major issues. In layer 2 segment of Ethernet, a broadcast domain is formed where a broadcast packet can reach. In a broadcast domain, broadcast packets will be flooded through all the connected ports of layer 2 switches and wireless interfaces of Wi-Fi access points connected to the domain. In CSMA/CA, the mechanism of RTS/CTS is not applied for transmitting a broadcast packet. Therefore, in the case when a large number of end-systems are connected to a single broadcast domain, there is a high possibility of collisions, resulting in performance reduction and unnecessary

consumption of bandwidth in the wireless network. This thesis considers a broadcast domain where a large number of hosts are connected directly using wires or through Wi-Fi access points as shown in Fig.1.1.

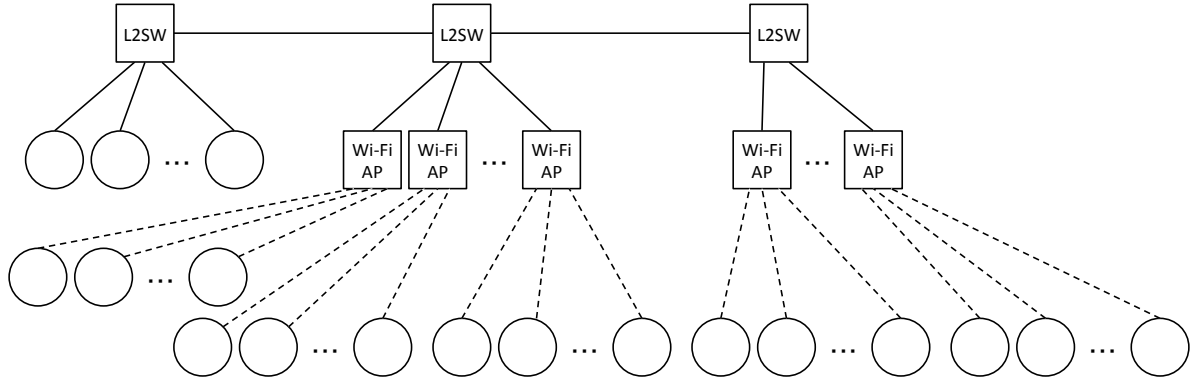


Figure 1.1: A large-scale broadcast domain where wired and wireless end-systems connect. A broadcast packet will reach all end-systems.

This thesis proposes Extensible Transparent Filer (ETF) for Ethernet using SDN to address the scalability issue of a single broadcast domain of Ethernet. ETF learns the existence of hosts in the network. Based on the information gathered and the contents of the broadcast packet, it determines the output port through which the target host of the broadcast packet can be reached. So instead of flooding the broadcast packet to all the active ports except incoming port, ETF forwards such a packet only through the selected port.

1.2 Network Flow Monitoring

Network monitoring is fundamental to examine the state of enterprise networks. Network monitoring is used for daily network management operations like traffic engineering, troubleshooting, anomaly detection, QoS support and accounting etc. Today's networks are large and complex. Distributed environment and resource constraints make network management rather difficult. In addition, implementing monitoring and diagnostic techniques causes significant overhead. Network flow monitoring provides a fine grained view of network traffic. There are many applications like anomaly detection, accounting and end user profiling etc., which require flow-level granularity of network traffic.

Ping, traceroute and Simple Network Management Protocol (SNMP) [2] are used for collecting the information to monitor the state of the network. Ping and traceroute are used to get the state of links, delay in the network or finding loops in routing etc. SNMP requires running an SNMP agent on each and every networking device and an SNMP-manager at the monitoring server. SNMP is based on pull based (request and response) architecture. It was developed to simplify network management. But it does not give flow-level granularity of network traffic.

In current network architecture, passive approaches (such as port mirroring and packet-sampling) are used to achieve this. Port mirroring requires devices to support SPAN ports. One copy of network traffic is sent to these ports which are connected to the monitoring server. Monitoring server receives atleast one copy of network traffic. It runs a tool similar to ntop [3] for analyzing the traffic. But

due to high bandwidth usage in the core network and limited number of SPAN ports supported per device, it is difficult to use in today's networks. NetFlow [4] and sFlow [5] are commonly used packet-sampling tools for network flow monitoring. NetFlow or sFlow capable devices capture each packet with some probability and selected packets are aggregated into flows. Flow reports are then sent to the monitoring server for analysis. These sampling based approaches are sufficient for applications which require coarse view of network traffic. But for fine granularity of network traffic, they are not good as they can miss several small flows. NetFlow was also designed to process each and every packet, but it causes high CPU overhead on networking devices, especially in the core network.

OpenFlow allows to control the traffic on per-flow basis. It maintains statistics corresponding to each flow. This thesis proposes a Network Monitor (NetMon), which calculates a few flow metrics to determine the state of the network, considering OpenFlow's flow-based architecture. NetMon determines the fraction of useful flows for each host in the network. It calculates the out-degree and in-degree for each host based on the IP address. NetMon classifies the host as a client, server, or peer-to-peer node, based on the number of source ports and active flows.

In traditional network monitoring, the focus was on the core network. But in an SDN network, each and every switch (including access layer switches) can be controlled through a single controller. NetMon implements a Device Logger to record the device (MAC address and IP address) and its location (Switch DPID and Port No). Device Logger helps to identify owners (devices) of an IP address within a particular time period.

NetMon is a distributed network monitoring system. Each switch supports OpenFlow protocol and sends OpenFlow messages to the controller. It uses a push-based approach to achieve its goals. NetMon provides a Web-GUI to visualize the state of the network.

1.3 Overview of the Work

This work includes the study of SDN and OpenFlow for engineering the enterprise network. ETF and NetMon are proposed for addressing the scalability and monitoring of the enterprise network. OpenFlow is used for implementing ETF and NetMon. ETF is proposed to scale a broadcast domain of Ethernet. ETF suppresses broadcast traffic in a broadcast domain by selecting an appropriate outgoing port of the switch through which the target host of a broadcast packet is reachable. NetMon is implemented for monitoring the enterprise network. As OpenFlow provides flow-based switching on Ethernet, NetMon focuses on calculating a few flow-based metrics to determine the state of the network. NetMon includes a Device Logger to track the location of hosts in the network. NetMon also provides a Web-GUI to query and visualize the state of the network. This work also discusses the deployment of SDN network in IIT Hyderabad campus. Both systems, ETF and NetMon are deployed and evaluated in the SDN network of IIT Hyderabad. The number of users connected to our SDN network is within a range of 20 to 50.

1.4 Thesis Outline

The thesis is structured as follows. Chapter 2 briefly explains about the SDN and OpenFlow. Extensible Transparent Filter (ETF) is proposed for scaling a broadcast domain in Chapter 3. Chapter 4 focuses on the implementation of Network Monitor (NetMon) for flow-based monitoring in

the enterprise network. Deployment of SDN in IITH campus is discussed in Chapter 5. Conclusions and future work are presented in Chapter 6.

Chapter 2

SDN and OpenFlow

Computer networks are large, complex and difficult to manage. Traditional networks are ossified considering their non-programmable, vertically integrated, closed and vendor specific architecture. It is difficult to control and manage the network as there is no centralized way to do so. Networking devices run complex and distributed control software that is typically closed and proprietary, and each device needs to be configured individually.

Software Defined Networking (SDN) paradigm promises to simplify the control and management of the network. SDN aims to make networks more simple, dynamic, open and programmable. OpenFlow [6] was the first open standard interface for implementing the SDN.

2.1 Software Defined Networking

A traditional networking device consists of data plane and control plane as shown in Figure 2.1. Data plane is used to forward a packet and control plane is used to determine where to forward the packet. For instance, in a learning switch, data plane is responsible for packet forwarding and control plane keeps a MAC table to determine the output port for the incoming packet. SDN architecture separates out the control plane and data plane of a networking device. In SDN architecture, control plane is programmable and logically centralized (known as the controller) which allows network administrators to control all the data-plane elements by writing a single control program. Network intelligence is centralized in the SDN controller which maintains a global view of the network. Switches communicate with a centralized controller through an open standard (such as OpenFlow). SDN facilitates the deployment of new services and protocols in the network, due to its vendor independence architecture and network virtualization. It also reduces the capital and operational costs for deploying and managing the network. Common SDN applications are network virtualization [7], network monitoring [8,9], load balancing [10], user authentication [11] and cloud or data center network [12] etc.

Figure 2.2 shows a logical view of SDN architecture. With a global view of the network at the controller, applications and policy-engines which are built on top of the controller, view networking devices as a single logical switch. Controller communicates with all the devices through an open-standard. Networking devices are simple and implement only basic packet forwarding mechanism.

SDN is not a new idea but has gained traction in recent times [14,15]. Many vendors (such

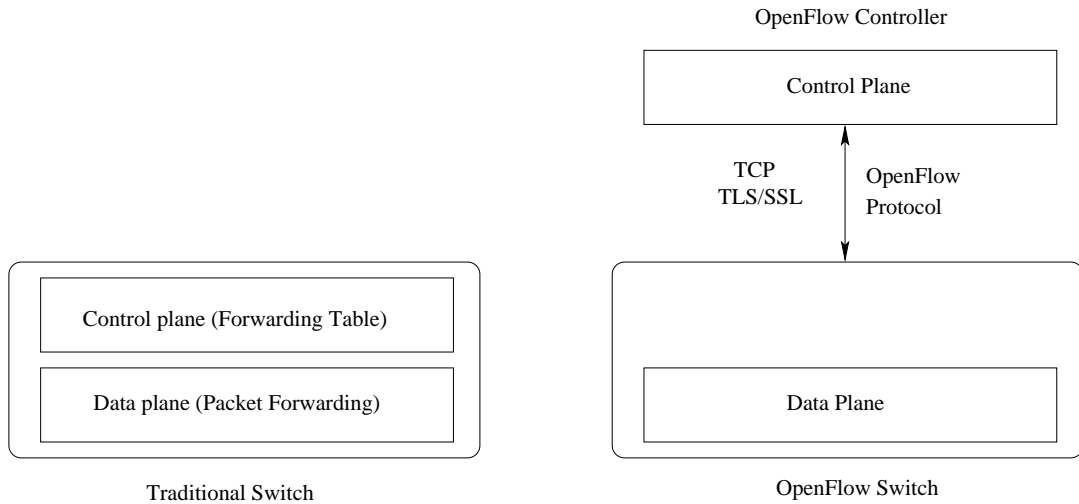


Figure 2.1: Traditional Switch vs OpenFlow Switch

as Cisco) have their proprietary implementations of the concept of SDN. OpenFlow is a widely accepted implementation of SDN across the industry and academic research communities. The OpenFlow protocol is open source and aims at making network programmable, innovative and vendor agnostic. One of the advantages of OpenFlow and its vendor independence is the rise of the concept of virtual switches. These are software level switches which are implemented usually as user-space or kernel-space software. One such example is Open vSwitch [16, 17] which implements the OpenFlow protocol. This enables any regular computer to be used as networking hardware and reduces the need to purchase expensive hardware from proprietary vendors.

2.2 OpenFlow Protocol

OpenFlow [6] is a protocol designed by the Open Networking Foundation(ONF) which promotes and adopts SDN through open standards development. OpenFlow was the first SDN standard to realize the concept of Software Defined Networking. The OpenFlow protocol is spoken between OpenFlow enabled switch (SDN switch) and OpenFlow Controller as shown in Figure 2.1. OpenFlow allows to control the network on per-flow basis in a fine-grained manner.

Table 2.1: A flow entry

Match Fields	Counters	Actions
--------------	----------	---------

Table 2.2: Match fields used to match packets against flow entries

Ingress Port	Ether src	Ether dst	Ether Type	VLAN Id	VLAN Priority	IP src	IP dst	IP ToS bits	TCP/UDP src port	TCP/UDP dst port
--------------	-----------	-----------	------------	---------	---------------	--------	--------	-------------	------------------	------------------

In OpenFlow protocol [6, 18], switches only consist of a forwarding plane that is equipped with

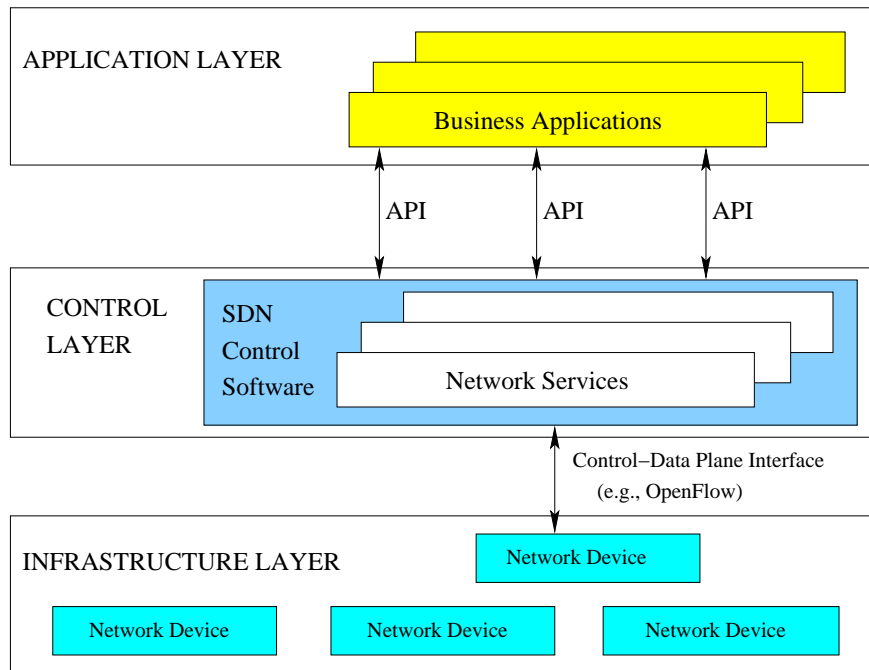


Figure 2.2: Software-Defined Network Architecture (Source: [13])

flow tables. A switch can have multiple flow tables. Each flow table contains several flow rules. Flow rules are similar to forwarding or routing rules in traditional switches and routers. Each packet is matched with flows rules in the flow tables. A flow rule includes a match, actions and counters as shown in Table 2.1. The OpenFlow protocol defines the fields which are included in the flow rules for matching. It currently supports matching up to the transport layer as shown in Table 2.2.

When the OpenFlow switch receives a packet and it has no matching flow rule for the packet, it forwards the packet to the controller through the *packet_in* message. The logic implemented in the controller then determines the actions for such packets. Depending on the logic, an OpenFlow switch can work as a router, switch, firewall, or network address translator etc. Controller either installs a flow rule on the switch by sending a *flow_mod* message or sends a *packet_out* message. If a flow rule is installed on a switch, then the *packet_in* message will not be sent for packets which match to that flow rule unless it is mentioned in the action explicitly. Once a flow rule is matched to a packet then counters corresponding to that flow are updated and corresponding actions are executed on that packet of the flow. The flow rules also have two timeout values: *Idle_timeout* and *Hard_timeout*, which control when the flow should be removed from the flow table of the switch automatically. Flows can also be removed by the controller explicitly. The OpenFlow protocol works on top of TCP and has support for TLS/SSL encryption.

Currently a few hardware vendors like Big Switch Networks, HP, and Pronto support OpenFlow in their hardware switches. Some of the available OpenFlow controllers are Floodlight [19], Ryu [20], Trema [21], NOX/POX [22] etc.

Chapter 3

Scaling a Broadcast Domain of Ethernet

Ethernet is a widely used technology for interconnecting networks and end-systems. Due to its simplicity in routing, ease of configuration and plug-and-play architecture, Ethernet is used as an elementary building block for a large network. An enterprise network using Ethernet will be simple and cost-effective. But a single Ethernet network can not scale to form a large enterprise network. The main reason for this is the broadcast traffic which consumes significant portion of network bandwidth.

This thesis proposes Extensible Transparent Filter (ETF) for Ethernet using SDN, that suppresses the broadcast traffic in a broadcast domain by selecting an appropriate outgoing port of the switch through which the target host of a broadcast packet is reachable. ETF maintains both consistent functionality and backward compatibility with existing networking protocols such as Address Resolution Protocol (ARP) [23], Dynamic Host Configuration Protocol (DHCP) [24] and other possible protocols that run on top of Ethernet and work with the broadcast of a packet.

ETF is a collection of SDN switches and a controller with a set of mechanisms to handle broadcast packets. Each of the SDN switch captures a broadcast packet and forwards it only to an appropriate outgoing port through which the target end-system is reachable. SDN controller learns about existence of hosts and determines the output port to reach the target host which needs to receive the broadcast packet without making other hosts receive it unnecessarily. ETF significantly suppresses the broadcast traffic in a broadcast domain and improves the scalability of Ethernet aiming the following achievements.

- 1) Scale the size of broadcast domain up to the maximum available capacity of MAC tables that routers, layer 2 switches and layer 3 switches can maintain for a particular domain. For example, achieving 4000 hosts (/20 subnet) and 8000 hosts (/19 subnet) in a single broadcast domain will be a very practical milestone for an enterprise network like a university campus.
- 2) Facilitate Wi-Fi hosts as a partial or even majority of the large-scale broadcast domain. ETF can localize the impact of broadcast packets within a collision domain formed by each Wi-Fi AP, even though many other APs are serving in the same broadcast domain.

3.1 Handling a Large-scale Network

Operational Disadvantage of Network Segmentation

If we aggressively fragment a large-scale network into small layer 2 segments, the impact of broadcast packets in each broadcast domain will be less than maintaining it as large to serve the same number of users. In the case that a layer 2 segment provides Wi-Fi connectivity, its size tends to be even smaller to isolate the collision domain from the many other hosts in the same broadcast domain. However, the routers or layer 3 switches need to handle more layer 3 subnets, the number of required VLANs will be more, and the utilization of address space will be less.

Flooding in a Broadcast Domain

Some important protocols in the Internet architecture use broadcast, that introduces flooding in a broadcast domain. DHCP is important to coordinate the network configuration dynamically. According to the observation in the campus network in Indian Institute of Technology Hyderabad (IITH), the major types of broadcast messages are ARP and NetBIOS Name Service (NBNS) [25] as shown in Figure 3.1. ARP and NBNS contribute by 31% and 57% respectively to the total broadcast traffic.

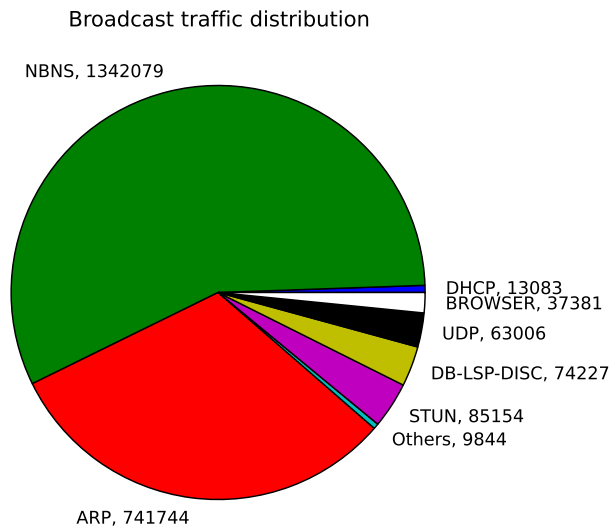


Figure 3.1: Distribution of broadcast traffic observed through 24-hours packet capture in IITH. The number of DHCP packet is significantly small because most of the end-systems have static IP configuration.

For example, Figure 3.2 shows that ARP generates significant number of broadcast packets in a single broadcast domain. If we deploy multiple Wi-Fi access points for serving a large number of clients in a single layer 2 segment, ARP broadcast packets would cause frequent interrupts which decrease the communication performance. NBNS is also expected to have a similar impact as ARP.

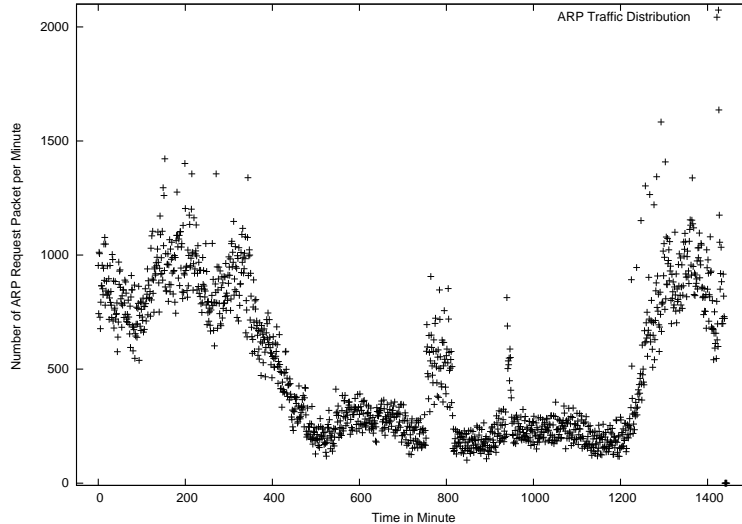


Figure 3.2: 24-hours (from 12:00 pm) observation result of ARP Requests, which are broadcasted in a subnet in IITH. The number of ARP Requests was 515 packets per minute on average and 2074 packets per minute were sent at its peak from 595 unique IP addresses in total. The top 3 popular targets in this subnet were WEB Proxy, default gateway of the subnet and DNS server.

SDN for a Large-scale Network

Due to vendor independence architecture of SDN, it facilitates the deployment of a large-scale network in an incremental manner. The behaviour of networking devices can be described in a uniform manner, irrespective of their vendors.

On scaling a broadcast domain, the benefit of SDN is its deeper inspection of packets than the traditional intelligent switches and flexibility of on-switch packet processing that can be enabled on production level switches. Snooping technologies are suboptimal in terms of scalability. For example, as we discuss in Section 3.6.2, filtering the control messages for IPv6 Address Auto Configuration with Multicast Listener Discovery (MLD) Snooping still has a potential for repeated near-floodings in the same domain [26]. SDN enables the fine-tuning of Ethernet with the minimised impact of flooding that the existing snooping technologies do not achieve.

3.2 Related Work

Scalability of Ethernet is a known issue and various approaches have been suggested in recent years to improve it.

Myers et al. [27] propose to eliminate the flooding in Ethernet by replacing broadcast-based protocols such as ARP and DHCP, with directory service integrated in switches called “local bridge”. Because their system requires end hosts to support the directory service, backward compatibility to the existing bootstrapping protocol is not maintained. Therefore, the seamless deployment is an issue.

EtherProxy [28] studies the impact of broadcast traffic and proposes the installation of dedicated devices to suppress the broadcast traffic. It caches protocol information carried by protocol

messages passing through it. For an ARP request, it responds with an ARP reply message. For a DHCP broadcast messages, it changes the destination MAC to a unicast address. If Etherproxy has any information in its cache then it passes the broadcast packet as is. Etherproxy considers the backward compatibility and it does not require any change to both clients and LAN equipments forming a broadcast domain. But Etherproxy breaks the end-to-end communication architecture and it requires modification of broadcast packets. Etherproxy is an additional device for connecting Ethernet bridges, and thus we need to install more devices as the number of Ethernet bridges increases.

SEATTLE [29] proposes floodless Ethernet architecture. In SEATTLE, ARP and DHCP are converted into unicast-based directory services. ARP is modified to return MAC address of a host and its “location”, which indicates the switch to which host is connected. DHCP relay is implemented on SEATTLE switch to avoid flooding of DHCP broadcast messages. Link-state algorithm was introduced to accelerate a packet delivery across SEATTLE switches. The major drawback of SEATTLE is its lack of interoperability with traditional Ethernet switches that do not understand the SEATTLE architecture.

MOOSE [30] introduces a hierarchical MAC address that contains 24 bits of switch identifier and 24 bits of host identifier. Location of a particular host is identified as the switch identifier in the hierarchical MAC address, and MOOSE runs inter-switch routing protocols in layer 2 so that a frame goes through the shortest path rather than the path determined by a spanning tree algorithm. The directory service of MOOSE called Enhances Lookup (ELK) suppresses DHCP and ARP broadcast messages similar to SEATTLE architecture. While MOOSE tries to provide high-performance switching architecture with very rich control mechanism in its back-end system, both the complexity of system and the operational cost of such dynamic features will be considerably high.

Zhao et al [31] introduce FSDM which can be adapted to complement MOOSE. FSDM caches the information of hosts in the SDN controller by maintaining two hash tables IP-MAC (IP address, Entry (MAC Address, hard-timeout, Flag)) and MAC- Map (MAC address, Location). The functionality of DHCP is integrated with SDN controller. In FSDM, the controller responds to ARP and DHCP messages. The FSDM expects that controller holds a global view of network and it discards ARP Request to a host that does not appear in its IP-MAC table. In this scenario, handling of static hosts still needs to be considered.

3.3 System Design

ETF is a collection of SDN switches and a controller that suppresses broadcast packets being flooded in a broadcast domain. Each switch has flow rules to match broadcast packets and to forward them to an appropriate outgoing port without flooding. Such flow rules are defined by the ETF running on the controller, either based on the expected destination written in packets or learned by looking into the specific fields of packets.

ETF maintains the end-to-end feature of communication by delivering broadcast packets from one end to another without any modification in it. ETF is transparent in layer 2 and maintains backward compatibility so that end-systems in the broadcast domain can operate the existing networking protocols such as ARP and DHCP.

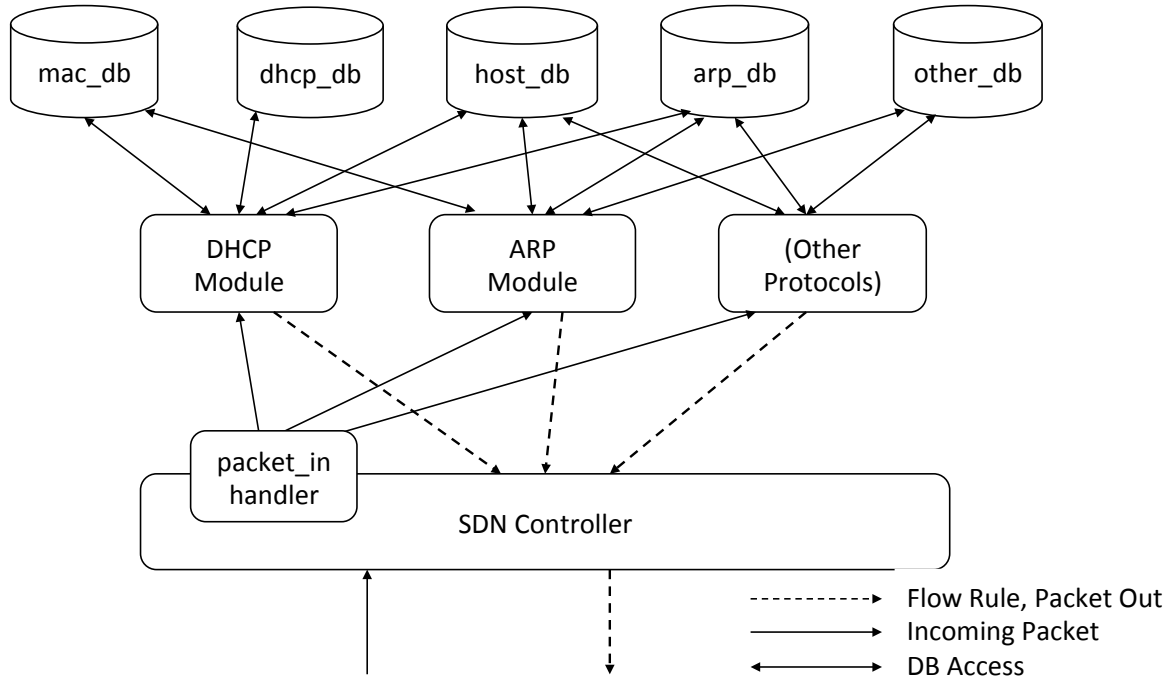


Figure 3.3: ETF System Design

Fig.3.3 shows the system design of ETF. The input to ETF is an incoming packet which is captured and forwarded by an SDN switch. The packet is passed to a corresponding protocol module to determine the action that the SDN switch should take. Each protocol module outputs the flow rule and injects it into the SDN switch through the secure channel. If ETF does not have any information about the target host, ETF will instruct the switch to flood the packet using *packet.out* message. All supported protocols access the same information base in order to reduce the frequency of broadcast, and to avoid the overhead of maintaining independent information bases for each protocol. For example, in the case of ARP and DHCP, most of the hosts, that connect to a broadcast domain for the first time, first perform DHCP and then ARP. By utilizing the MAC/IP address mapping learned through processing DHCP messages, the probability of broadcasting an ARP Request will be much less.

3.4 System Implementation

ETF is implemented as an OpenFlow controller module/application using OpenFlow 1.0 [18] protocol. According to the OpenFlow protocol, an OpenFlow switch forwards a packet to the controller if it has no matching flow rule for that packet, or if it has a flow rule explicitly stating to forward it to the controller. Such a packet is processed in the ETF module/application by using the *packet.in* handler. ETF then installs flow rules for broadcast packets, so that they are delivered in a unicast manner to the destination host whenever possible.

3.4.1 Data Structures

ETF maintains following data structures to efficiently store and retrieve the information about hosts and the network topology. It examines the packet header, and determines match and action primitive for any flow rule and installs it on the appropriate SDN switch(es).

- **dhcp_db** is a list of DHCP servers and information related to them such as IP address, Switch DPID and Port number.
- **host_db** is a hash table where *MAC Address* is used as a hash key to look up IP Address, Port No, Switch DPID and Last Access of a host.
- **mac_db** is a hash table maintained for every switch in the network, similar to how traditional switches maintain. The table takes *MAC Address* as the key to lookup the output port.
- **path_db** is a hash table where a pair (*source Switch DPID* and *destination Switch DPID*) is used as a hash key to get Port No on the source switch to reach the destination switch.
- **arp_db** is a hash table where *IP Address* is used as a hash key to lookup MAC address.

It is important to note that either *path_db* or *mac_db* is used to determine the output port to forward a packet. *path_db* requires the topology information of the network but it can reduce the flooding that takes place to build the *mac_db*. In current implementation of ETF, *mac_db* is used for forwarding packets.

3.4.2 Discovering DHCP Servers

Assume that the broadcast domain is in startup phase where

- 1) H_{C1} , connected to switch S_X via port a , broadcasts DHCP Discover to configure its network interface as shown in Figure 3.4, and
- 2) ETF does not have information about $DHCP_S$.

Once S_X receives DHCP Discover message from H_{C1} , it forwards the packet to ETF running on the SDN controller. ETF creates an entry for H_{C1} in the MAC table of S_X . ETF then looks up *dhcp_db* to find the available DHCP servers. S_Y and S_Z also go through the same process. If there are no active servers known by ETF, then ETF will let S_X , S_Y and S_Z flood the packet from all active ports except the port on which DHCP Discover was received. This flooding procedure happens only if no DHCP server is known to ETF.

When $DHCP_S$ receives DHCP Discover from H_{C1} , the $DHCP_S$ sends DHCP Offer. S_Y captures the packet and forwards it to ETF. ETF updates *dhcp_db*, *host_db* and *arp_db* to store the information that $DHCP_S$ is connected to S_Y through port a . If there are multiple DHCP servers in the same network, ETF puts an entry for each DHCP server in the databases.

The DHCP server in the network has been discovered. As the ETF has already learned the MAC address of H_{C1} for switch S_X , S_Y and S_Z in their respective *mac_db*, it will give S_Y a flow rule to forward the DHCP Offer to port b . Similarly, it gives a flow rule to S_X to forward the packet to port a .

ETF also inserts a proactive flow rule on each switch to forward the DHCP broadcast packet to an appropriate port through which $DHCP_S$ can be reached as discussed in Section 3.4.6.

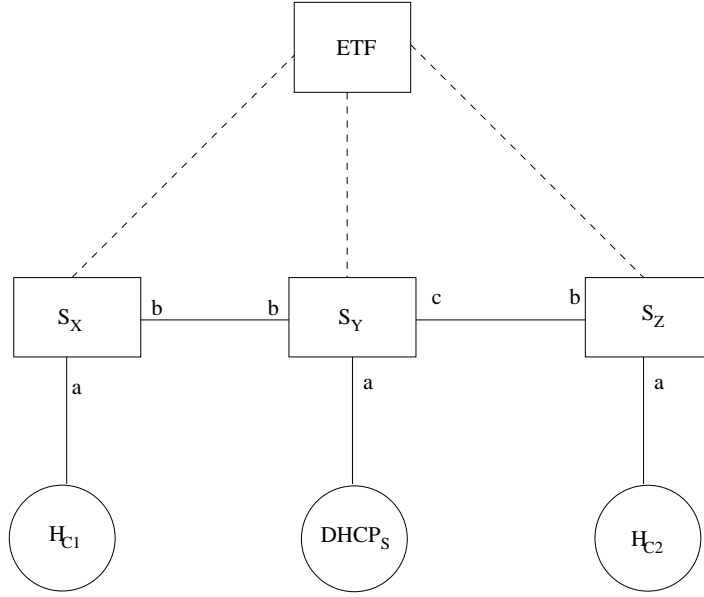


Figure 3.4: Example network diagram for processing DHCP packets between a DHCP client H_{C1} and a DHCP Server $DHCP_S$. S_X , S_Y and S_Z are the SDN switches interconnected with each other. ETF maintains the information of DHCP servers dynamically.

3.4.3 Operating DHCP in Suppress Mode

Once H_{C1} receives DHCP Offer, it broadcasts DHCP Request to inform all DHCP servers that the client takes IP address offered by a specific DHCP server. Switches S_X and S_Y will forward the packet to the port through which DHCP server is reachable, since the controller has already discovered the DHCP server in the network.

After $DHCP_S$ receives DHCP Request, it sends DHCP Ack to confirm the assignment of IP address to H_{C1} . S_Y and S_X forward the DHCP Ack using the same flow rules that were added during DHCP Offer, ETF will insert/update *host_db* and *arp_db* with the new IP address for H_{C1} . These entries are deleted if the host sends a DHCP DECLINE message.

Now a new host H_{C2} connected to S_Z asks for network configuration using DHCP. Although H_{C2} broadcasts DHCP Discover, it is sent via the port b on S_Z without flooding to any other ports. ETF learns about H_{C2} using the same process as H_{C1} . If a DHCP server and a client is connected to the same switch, the broadcast packet will be locally forwarded without being sent to the adjacent switches.

3.4.4 Operating ARP Request in Suppress Mode

Assume that H_{C1} conducts ARP to start communication with H_{C2} after the DHCP operation in both end-systems. H_{C1} broadcasts an ARP Request to look up the MAC address of H_{C2} . S_X receives the ARP Request and forwards it to ETF. ETF looks up H_{C1} and H_{C2} , that are the source and the target of ARP Request respectively, in *arp_db*. If there is already an entry of H_{C1} and H_{C2} , it updates Last Access in the database. ETF gets the MAC Address of target host and looks up *mac_db* of S_X to obtain the port to reach H_{C2} . ETF gives S_X a flow rule to forward such a packet

Table 3.1: Proactive Flow Rules given to forward DHCP broadcast packets to DHCP_S

Switch DPID	in_port	dl_type	dl_src	dl_dst	nw_proto	tp_src	tp_dst	action
S _X	any	IP	any	ff:ff	UDP	68	67	output:b,ETF:1500
S _Y	any	IP	any	ff:ff	UDP	68	67	output:a,ETF:1500
S _Z	any	IP	any	ff:ff	UDP	68	67	output:b,ETF:1500

Table 3.2: Proactive Flow Rules given to forward ARP Reply back to H_{C1}

Switch DPID	in_port	dl_type	dl_src	dl_dst	nw_proto	action
S _X	any	ARP	any	H _{C1}	Reply	output:a,ETF:1500
S _Y	any	ARP	any	H _{C1}	Reply	output:b,ETF:1500
S _Z	any	ARP	any	H _{C1}	Reply	output:b,ETF:1500

via port b and to ETF itself. S_Y and S_Z also go through the same process as S_X and ARP Request reaches H_{C2} .

ARP Reply is a unicast packet from H_{C2} to H_{C1} . Once S_Z receives ARP Reply from H_{C2} , the switch forwards the packet to ETF. ETF looks up H_{C2} in *host_db* and *arp_db* and update the Last Access. ETF will also give S_Z a flow rule to forward the packet via port b . S_Y and S_X will also go through the same process as S_Z .

3.4.5 Flooding for ARP

While ARP works in Suppress Mode for the end-systems which get IP configuration using DHCP, ETF does not have information about end-systems that use a static IP address. Therefore, when such an end-system is the target of ARP Request, the packet will be flooded to all ports of the switches except the port on which ARP Request was received. Having such a host issuing ARP Request does not result in a big overhead, because the switch connected to it forwards the ARP Request to ETF. ETF learns the source host and creates its entry in *arp_db* and *host_db*.

3.4.6 Proactive Flow Rules

Proactive flow rules allow each switch to know how to process a particular packet before the packet actually reaches it. Proactive flow rules avoid instances where each of the intermediate switches on the end-to-end path accesses the controller in order to process the same packet repeatedly, for obtaining the flow rule for the corresponding packet. ETF uses proactive flow rules to avoid the delay that an end-system will experience while ETF processes a packet.

Once ETF knows the outgoing port to reach DHCP server, ETF inserts a proactive flow rule to each of the switches as shown in Table 3.1. Each switch forwards DHCP broadcast messages to ETF and the outgoing port through which $DHCP_S$ is reachable. This approach can also be used for sending a DHCP message back from the DHCP server to a client. Because ETF learns that, for example, the client H_{C1} is connected to S_X via port a , using DHCP Discover message, ETF can give all switches a flow rule to reach back to H_{C1} immediately after S_X forwards a DHCP broadcast message from H_{C1} to ETF.

Once ETF captures an ARP Request, it can insert a proactive flow rule for the ARP Reply on each switch setting the outgoing port to reach back to the source of the request as shown in Table 3.2. As ETF receives each and every ARP message, it learns the source host of the ARP Reply and updates the databases accordingly.

3.4.7 Mobility

If a host moves within the same broadcast domain like, eg. a laptop computer moves from one Wi-Fi AP to another, the outgoing port to the host must be updated. The existing flow rules will not match to a packet sent by such a host. It will be forwarded to ETF which learns that the host has moved and updates the database to reflect the changes.

3.4.8 Garbage Collection

ETF specifies the *Idle_timeout* as 300 seconds for all flows other than ARP and DHCP. The *Idle_timeout* for ARP and DHCP flows depends on the DHCP lease time specified in the DHCP servers. If the *Idle_timeout* of the flow is smaller than the lease time, then the flow will be removed before the host renews its lease. This will lead to unnecessary processing at the controller.

ETF has a configurable parameter which determines the *Max Age* of an entry of a host in *host_db*. If an entry in *host_db* is not referred within the *Max Age*, i.e. $Current\ Time - Last\ Access > Max\ Age$, then the entry is given a second chance before the entry is deleted. When a second chance is given, ETF sends a unicast ARP probe, to the last known location of the host in the network. If ETF doesn't receive a ARP Reply before it performs the next garbage collection iteration, the entry will be deleted, and the corresponding entry for the host in *arp_db* is also deleted.

3.4.9 Rate Limiter

In case, when ETF doesn't have an entry for the target host of ARP Request in its *arp_db*, it applies a rate limiting policy on such ARP Requests. The policy is to allow only one such request within a *Rate Limit Interval* which is configurable. This is done to reduce the flooding of such packets in the network.

3.4.10 IP Conflict Detection

IP conflict module is implemented to improve the performance of ETF. First-come first-serve policy is applied for the use of an IP address. Users are never allowed to configure IP addresses which are reserved for a few important servers like DHCP server, DNS server and default gateway. IP conflict detection module maintains an ARP table for hosts in the network. Whenever an IP conflict is detected, it sends three ARP probe messages to the host, which has held the IP address currently, in a unicast manner. It puts this ARP probe along with time-stamp in an ARP probe message queue. If the host replies to ARP probe then newer hosts will not be allowed to use the same IP address. Controller installs flow rules to deny access to the conflicting hosts in the network.

3.5 Evaluation

3.5.1 Proof of Concept in Virtualized Environment

As a proof of concept, ETF was implemented in the lab setup using Trema 0.4.6 as SDN controller and Open vSwitch 1.4.3 as SDN Switch on two workstations. Basically this implementation enables all of the features that we have discussed in the previous section.

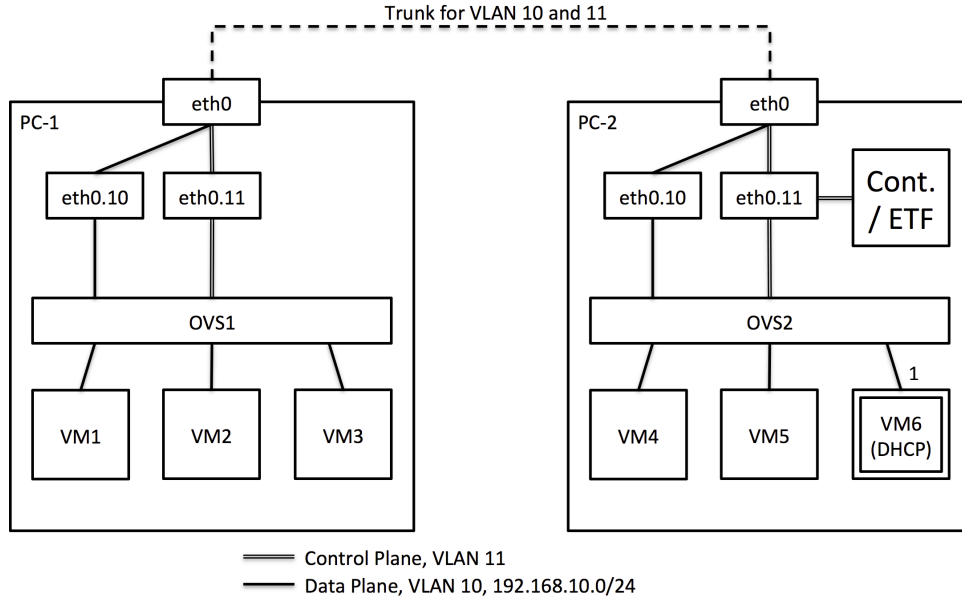


Figure 3.5: Test setup for proof of concept

As shown in Figure 3.5, two separate VLANs were used for the data plane and the control plane respectively. All VMs connected to VLAN 10, and VM6 served as a DHCP server. After all other VMs got the network configuration using DHCP, each of them conducts ping to every other VM. In this process, flooding was observed to occur only once for the first DHCP Discover packet to find the DHCP server. The rest of the experiment operated in Suppress Mode successfully, even though all VMs transmitted broadcast packets for performing DHCP and ARP. These protocols were successfully completed without any flooding.

3.5.2 Estimated Scalability of ETF and Broadcast Domain

In Suppress Mode of ARP and DHCP, broadcast packets will be delivered only through the port on the end-to-end path to its destination. Therefore, the number of outgoing port for such packets on each switch is one less than or equal to the number of DHCP servers respectively.

Assume that ETF operates in the broadcast domain which has N hosts and M switches, and *path_db* is used for forwarding the packet. *arp_db* and *host_db* will take $O(N)$ space, while *path_db* will take $O(M^2)$ space. ETF will suppress the number of broadcast messages from $O(N)$ without ETF to $O(1)$ for DHCP, and from $O(N^2)$ without ETF to $O(1)$ for ARP given that N hosts operates DHCP and communicate with each other on the same link.

3.5.3 ETF Performance under Actual Deployment

To evaluate the suppression performance of ETF, we observed the number of broadcast packets in the ETF-enabled broadcast domain.

Suppressing DHCP Packets

The following Figure 3.6 shows the 16 hours packet capture result of laptop computers connected to Wi-Fi AP1 and AP2, wired desktop computers and the DHCP server. In the observation period, we had as total of 97 unique hosts in the same broadcast domain where 60 were connecting to Wi-Fi AP1, 20 to Wi-Fi AP2, and the others directly to the SDN switches.

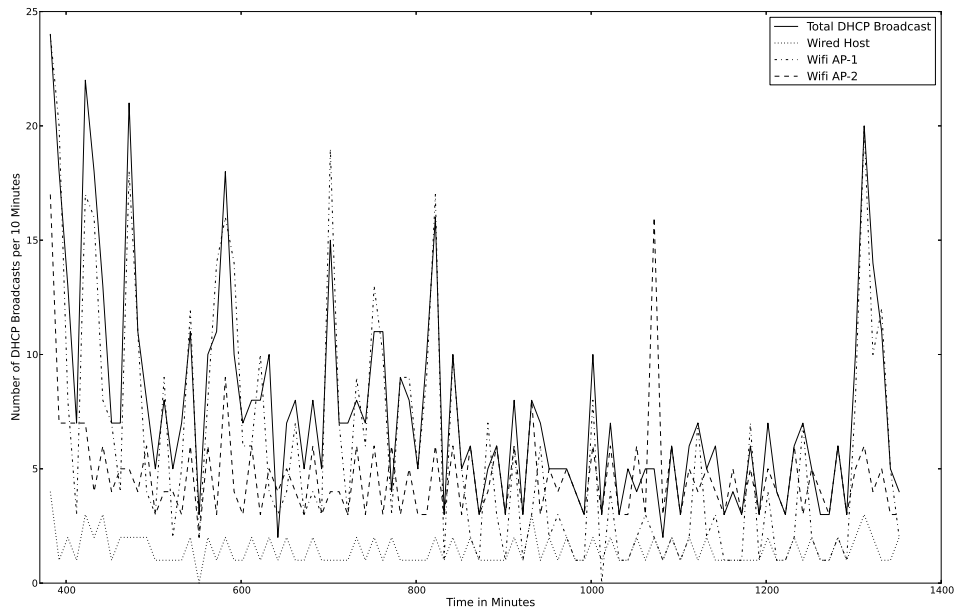


Figure 3.6: Number of DHCP broadcast packets per 10 minutes for wired, wireless clients and DHCP server.

The total number of DHCP broadcast packets which were observed at the DHCP server was 740 including 351 DISCOVERs, 141 OFFERs, 179 INFORMs, and 69 REQUESTs. The following Table 3.3 summarizes and gives the overall performance for suppressing DHCP.

Table 3.3: Overall Performance for Suppressing DHCP

Capture Point	Total Number	Avg. PPM	Max. PPM	Suppress Rate
DHCP Server	740	0.76	8	N/A
Wired Host	143	0.147	2	80.6%
Wi-Fi Host-1	577	0.59	10	22.0%
Wi-Fi Host-2	457	0.47	13	38.0%

While the wired host exhibited good efficiency to suppress the broadcast packets, we observed reasonably less suppress rate at each of the Wi-Fi hosts. In our deployment, the majority of end hosts were Wi-Fi users and the total number of users were not very large. Therefore in these particular conditions, the suppress rates at Wi-Fi hosts were not very high and subject to the size of collision domain. If we have more end hosts connecting to more Wi-Fi APs. the suppress rate will naturally be improved. This is because a broadcast packet that is filtered by ETF and destined only for a host under Wi-Fi AP1 will not reach AP2.

The packet capture at Wi-Fi hosts exhibited more number of broadcast packets than that we intended and some times the number was even more than that were observed at the DHCP server. One observation is that this happened most probably because several hosts faced packet loss at the SDN switch and such hosts repeatedly broadcast the packets. In current hardware switch, the location of flow rules depends on the match fields, so all the flows were stored in software only. We had a limit of 1000 pps (packet per second) for processing in software chain of switch. If switch receives more packets than it can process, it will drop the packets. By enabling the SDN switches to store flow rules in the hardware, then the switch will exhibit much better performance and such packet loss will not be caused.

Suppressing ARP Packets

The following Figure 3.7 and Table 3.4 show the 16 hours of packet capture result in the same time period with DHCP observation. The count for total number of ARP broadcast packets was maintained at the SDN controller.

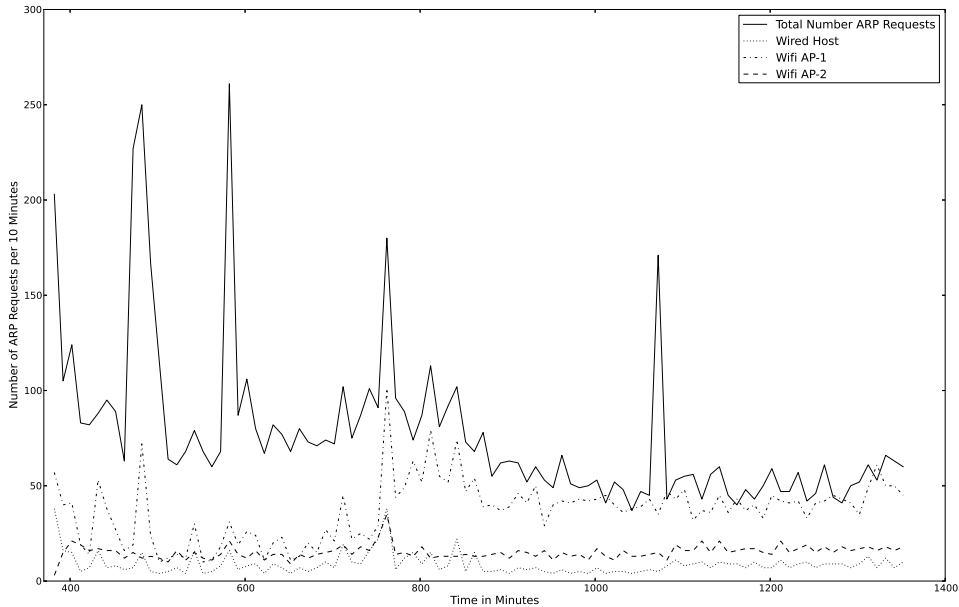


Figure 3.7: Number of ARP broadcast packets per 10 minutes for wired, wireless clients and SDN controller.

Table 3.4: Overall Performance for Suppressing ARP

Capture Point	Total Number	Avg. PPM	Max. PPM	Suppress Rate
SDN Controller	7502	7.72	147	N/A
Wired Host	873	0.90	21	88.3 %
Wi-Fi Host-1	3605	3.71	55	51.9 %
Wi-Fi Host-2	1467	1.51	19	80.4 %

ARP suppression performed well for the wired host while Wi-Fi hosts observed more broadcast packet again due to the size of collision domain. We observed that 2.1% of ARP Requests were Gratuitous ARP, which is not filtered under the current deployment. By checking the complete matching of IP/MAC including non-DHCP hosts, Gratuitous ARP can also be filtered more effectively. Another observation is that several hosts queried a host that didn't exist in the same domain. The fraction of such packets was 8.5%. In this case, such ARP requests will be flooded in the network, because of the possible non-DHCP host that has not sent any packet. The rate limit of 1 packet per two seconds is applied to ARP Requests to a non-existent host to avoid possible DoS attacks.

3.6 Discussion

3.6.1 Improving the Performance

As far as we observed, Windows and Mac OSX hosts tend to conduct Gratuitous ARP using broadcast more frequently than Linux hosts. Integrating "IP Conflict Detection" with ETF is one approach to filter Gratuitous ARP because ETF can maintain MAC/IP matching of each host. One possibility is that a DHCP host and a non-DHCP host may conflict for their IP address. We have discussed our approach for this in Section 3.4.10.

We are also examining interoperability of Proactive Flow Rule injection into the currently available SDN switches. This feature will contribute to the reduction of the frequency of *packet_in*, and reduce the response time for ARP.

3.6.2 Scaling IPv6 Mechanisms

Another scalability issue that a large scale broadcast domain will encounter is IPv6 operation.

An IPv6 host joins a solicited-node multicast address to perform Duplicate Address Detection (DAD) and Neighbor Discovery (ND). A solicited-node multicast address is determined using the last 24 bits of a node's unicast and anycast addresses [32]¹. Given that the first 24 bits of MAC address is used in the link-local IPv6 address, potentially 2^{24} hosts may join the same solicited-node multicast address using different IPv6 address on the same broadcast domain. In this case, even if MLD Snooping is enabled on each layer 2 switch, Neighbor Solicitation (NS) packets multicasted from one IPv6 node for DAD or ND can be transmitted from all possible ports that the switches have learned and, as a result, introduce near-flooding situations.

¹Note that DAD must not be performed for anycast addresses [33].

ETF will exhibit better performance on handling this IPv6 scalability than MLD snooping. NS packet for DAD and ND will be sent only through the ports toward the host of direct interest. In addition to MAC/IP address mapping that ETF maintains, ETF can deep inspect the upper layer information of the incoming NS packet. Therefore, ETF can detect a duplicate address and a target of ND without near-flooding packets.

3.6.3 Avoiding ARP Poisoning

As ETF maintains the information about all the hosts and switches that are present in network, we can configure static entry in the databases for certain important hosts like Default Gateway, WEB Proxy and DNS Server. These static entries will never be updated. This approach will make the ARP Poisoning attack difficult.

Chapter 4

Network Flow Monitoring

Network monitoring observes and analyzes the status and behavior of the end-systems. It is required to ensure that network is up and running as planned. The network flow information is very important to understand the network behavior, identify security threat and ensure QoS. For instance, flow level details give the information about the end-system such as who is sending, who is receiving, who is using what applications etc. Network flow monitoring supports several network management tasks such as anomaly detection, accounting, application identification, traffic engineering and detecting scans, worms, and bot-net activities.

Traditional approaches such as packet-sampling and mirroring the traffic, do not work best in today's network. The reasons are the followings:

- They are not scalable with the size of the network. Whenever the network is expanded, network administrators have to redesign the network monitoring plan. With the growth of the network, additional monitoring appliances will be required.
- They lack in term of coordination among networking devices. As there is no uniform architecture, it is also cumbersome and error-prone to configure each and every device manually.
- In traditional network monitoring, the focus was on the core layer of the network which makes difficult to monitor the behavior of end-hosts.

The challenges in designing a network monitoring system are as follows:

- How to define monitoring information.
- How to get the information from resource to the manager.
- How to process or get the meaningful information from the monitored data.
- How to present the information.

This thesis proposes NetMon, a network monitoring tool for OpenFlow networks. NetMon takes advantage of features provided by the OpenFlow protocol. NetMon implements a few flow based metrics such as fraction of useful flows, out-degree, in-degree, port-degree and flow-count. It also records the devices and their locations in the network.

NetMon is a distributed monitoring system. Each access layer switch to which hosts are connected either directly through wired connection or Wi-Fi APs, has a subset of flow rules in the network. Controller can also partition the flow rules equally on all the switches across the network as discussed in Section 4.6. These switches inform the controller about every new flow in a push-based manner. *packet_in* and *flow_removed* messages are used to indicate arrival and removal of a flow respectively, to the controller. NetMon calculates the flow statistics and records the devices and their locations in the database based on certain triggers. It provides a Web-GUI to visualize and query these statistics.

4.1 Network Flow Metrics

A network flow is a sequence of packets from a sending application to a receiving application. It is represented through an IP 5-tuple $\langle srcIP, dstIP, srcPort, dstPort, protocol \rangle$. For instance,

- **Flow A:** $\langle 192.168.252.220, 192.168.36.22, 31234, 3128, TCP \rangle$
- **Flow B:** $\langle 192.168.36.22, 192.168.252.220, 3128, 31234, TCP \rangle$

Flow B is called reverse-flow of flow A. NetMon considers the following flow metrics for the monitoring:

- A. *Fraction of Useful Flows:* Guha et al. [34] examines the health of the network via a new flow-based metric based on the fraction of useful flows generated by end-hosts. They consider a flow which explicitly fails or does not get a response from the intended destination, as non-useful. A TCP flow, which completes a 3-way handshake successfully, is marked as a useful flow. A UDP flow, which sees packets in both direction, is considered as a useful flow. The cause of non-useful flows can be misconfiguration, lack of awareness about services in the network, unnecessary broadcast traffic etc.
- B. *IP Out-degree and In-degree:* The IP out-degree of a host reveals the number of hosts to which the host is sending packets. The IP in-degree of a host reveals the number of hosts which are sending packets to the host. The IP out-degree and in-degree have many network applications such as detecting stealthy spreaders.
- C. *Port-Degree and Flow-Count:* The port-degree of a host shows the number of network processes that are running on the host. A graph between the port-degree and number of active flows can be used to classify a host as a client, server or peer-to-peer node.
- D. *Response Time:* Response time is defined as a time interval between a flow and its reverse-flow. It includes network latency and application level latency at the server. Response time shows the congestion in network and the load on server. In OpenFlow networks, controller is responsible for installing flow rules on SDN switches. Controller can maintain a time-stamp corresponding to each flow. It can determine the response time by subtracting the time-stamp of a flow from the time-stamp of its reverse-flow. Current implementation of NetMon does not calculate the response time of a flow.

4.2 Related Work

In the past, a lot of work has been done in the field of flow-based network monitoring. Anemone [35] uses the network topology information and the end-systems as real time ‘traffic sensors’ to support the flow-based network management. Karagiannis et al. [36] present an approach to profile the activity and behavior of end-hosts using the transport layer information of flows. Joumblatt et al. [37] discuss about the importance and challenges of packet capture at the end-host. Most of these approaches were not implemented as real-time monitoring system.

Sekar et al. [38] present a minimalist approach for network flow monitoring. They use flow sampling and sample-and-hold as sampling primitives and configure these primitives on routers using cSamp [39] in a coordinated fashion across the network. NetFlow [4] and sFlow [5] are commonly used technologies for implementing network flow monitoring. As they rely on sampling of packets, they can miss the several small flows and are not well suited for some applications such as [34] which require some specific packets involving connection setup phase of a TCP flow.

Network monitoring using OpenFlow has also been explored in recent years. OpenSAFE [9] routes the traffic for network analysis and requires separate monitoring appliances. OpenNetMon [40] uses adaptive polling for determining throughput, latency and packet loss. OpenSketch [41] is an SDN based measurement architecture similar to OpenFlow. A three stage pipeline (hashing, filtering, and counting) is implemented in the commodity switches. It provides a measurement library to use these sketches. Upgrade or replacement of SDN switches is required to support this. Our approach is similar to FlowSense [42]. FlowSense uses push based approach to determine the link utilization in the network. It uses only *packet_in* and *flow_mod* messages to gather the required information.

4.3 System Design

Fig.4.1 shows the system design of NetMon. In SDN network, the controller maintains the state of the network. Controller maintains the log for devices and calculates flow metrics. It stores the record for devices in a persistent database and prints flow statistics its output stream. Data Collector communicates with the controller using inter process communication mechanism and reads the statistics from the output stream of the controller. Data Collector parses them and stores them in appropriate data structures. It provides APIs to the NetMon for accessing these statistics. NetMon provides a Web-GUI to network administrator for visualizing the state of the network. Based on the request of user, NetMon system either accesses Device Logger database or Round Robin Database (RRD) [43] files. A few flow statistics are maintained in memory only, NetMon calls an appropriate method of Data Collector to access them.

Fig.4.2 shows the flow of messages in SDN controller. Controller receives the information about a new flow through *packet_in* message. Device Logger and Learning Switch receive *packet_in* messages. Device Logger reads the content of packet header and if required, generates a trigger to record the device in the *device_logger* database. A trigger is caused when a device is not registered, or changes its IP address or location, or in some other cases. Learning Switch receives such messages to determine the action that the SDN switch should take. It forwards the message to the Flow Parser module. Flow Parser stores all the active flows in its *flow_db* and responds to Learning Switch with a boolean flag, which indicates whether a flow rule should be installed on the switch or not. If

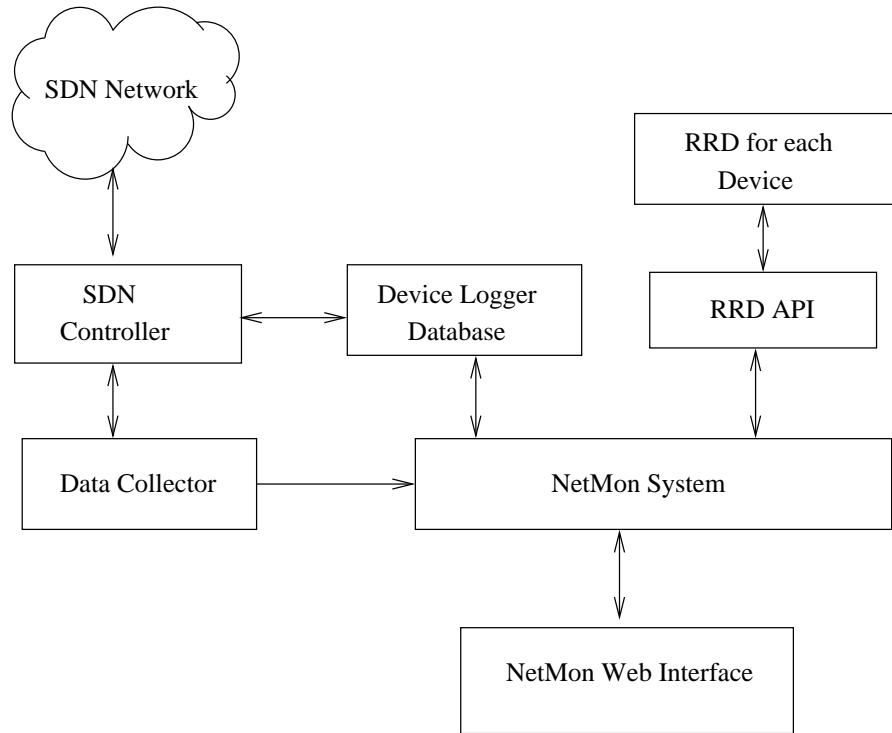


Figure 4.1: NetMon System Design

Flow Parser returns *true* then Learning Switch determines the output port for the flow and installs the flow rule on the switch. Learning Switch either sends a *flow_mod* or a *packet_out* message to the SDN switch. Whenever a *flow_removed* message is received, Flow Parser removes the corresponding flow from its *flow_db*.

4.4 System Implementation

NetMon is implemented as an application of OpenFlow. NetMon receives the information about every new TCP/UDP flow using *packet_in* messages. It determines whether a flow is useful or non-useful. It calculates out-degree and in-degree based on the IP address, for each hosts in the network. NetMon classifies the host as a client, server or peer-to-peer node, based on the number of source ports and active flows. NetMon uses RRD [43] to store the statistics for each host in the network. NetMon provides a basic Web-GUI to present the state of the network.

4.4.1 Data Store

NetMon uses Postgres [44] to keep record for devices in the network. SDN controller writes a record for a device whenever a trigger is generated. NetMon keeps flow statistics for each device in its own separate RRD [43] file. While maintaining a constant file size, RRD file is well-suited to store the time-series data and it also presents history of data. In the current implementation of NetMon, only the fraction of useful flows is stored in the RRD file. Other flow statistics are not stored in the persistent database and kept in memory only.

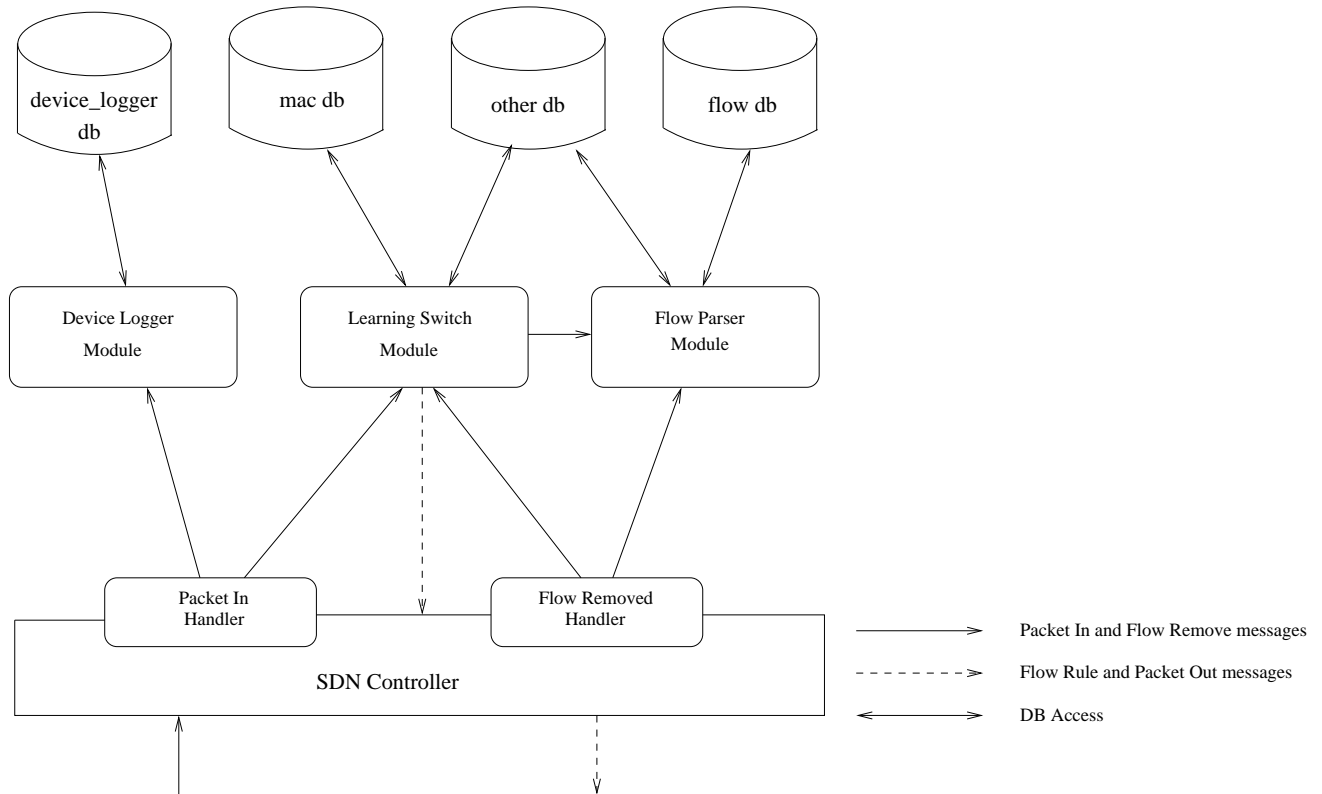


Figure 4.2: Flow of messages in SDN controller

4.4.2 SDN Controller

SDN controller receives control messages like *packet_in* and *flow_removed*. *Packet_in* messages are forwarded to Device Logger, Learning Switch and Flow Parser modules. Device Logger records the devices and their location in the network. Learning Switch is responsible for installing flows on switches based on the global network information. Learning Switch forwards the *packet_in* message to the Flow Parser. Flow parser keeps all the active flows in its *flow_db* table. It sends a boolean flag to Learning Switch, for deciding whether to install a flow rule on a switch or not. Whenever it receives *flow_removed* message, it removes the corresponding flow from its *flow_db*. Flow Parser calculates each flow-metric in a separate thread and periodically prints the flow-statistics on output stream of the controller.

4.4.3 Data Collector

Data Collector communicates with SDN controller through interprocess communication mechanism and reads the statistics messages from its output stream. It parses these statistics messages and stores them in appropriate data structures. If it receives a reset message from SDN controller, it clears data structures. Data collector also provides API for accessing these data structures.

4.4.4 Device Logger

Most of the servers like Web server, Proxy server etc. use IP address to keep the log about the user's request. IP addresses are dynamic and same IP address can be used by multiple users over time. So it is difficult to track the user based on the IP address. NetMon suggests an approach, Device Logger, to trace back the user.

Algorithm 1 Device Logger

Require: Device Logger Table $dl_table(TimeStamp, MACAddress, IPAddress, SwitchDPID, PortNo)$

Require: Packet In Message pi

Require: Switch DPID sw

Require: Input Port in_port

```
1:  $mac\_addr \leftarrow$  Source MAC address from  $pi$ 
2:  $ip\_addr \leftarrow$  Source IP address from  $pi$ 
3:  $r1 \leftarrow$  Record( $current\_time, mac\_addr, ip\_addr, sw, in\_port$ )
4:  $r2 \leftarrow$  Get the latest record and its time stamp from  $dl\_table$  which has IP Address  $ip\_addr$ 
5:  $r3 \leftarrow$  Get the latest record and its time stamp from  $dl\_table$  which has IP Address  $ip\_addr$  and
   MAC Address  $mac\_addr$ 
6: if  $r2 = null$  or  $r3 == null$  then {A new device joins the network or it uses an IP address first
   time}
7:   Insert record  $r1$  in  $dl\_table$ 
8:   return
9: end if
10: if  $r2.TimeStamp = r3.TimeStamp$  then
11:   if  $r3.PortNo = in\_port$  AND  $r3.SwitchDPID = sw$  then {Check the location of device}
12:     return
13:   else
14:     Insert record  $r1$  in  $dl\_table$ 
15:   end if
16: else
17:   Insert record  $r1$  in  $dl\_table$ 
18: end if
```

With SDN, controller has a global view of the network. Controller can track the hosts when they are connected to, or disconnected from access switches. Switch sends *port_up* and *port_down* messages to the controller when a port goes up and down respectively. These messages are not sufficient to track users who are connected through Wi-Fi APs. Device Logger receives *packet_in* message when a packet does not match to any flow rule. Controller keeps 5-tuple information $\langle TimeStamp, IPAddress, MACAddress, SwitchDPID, PortNo \rangle$ to record the device in the network. Device Logger stores the record in its persistent database whenever a trigger is generated. NetMon provides a Web-GUI to query about the devices and their locations for a given IP Address within a time range.

Device Logger implements Algorithm 1 to process *packet_in* messages. In Algorithm 1 line number 4 and 5 are used to determine whether a device has newly joined the network or the owner

of an IP address has changed since its last used. If a new device joins the network then record $r3$ will be *null* and Device Logger will put record for the device in the *device_logger* database. In line number 11, Device Logger checks whether the device has changed its location from the previous one. If the device changes its location, it will record for the device in its *device_logger* db.

4.4.5 Fraction of Useful Flows

NetMon adopts distributed and push-based approach to determine whether a flow is useful or not. Access layer switches keep 5-tuple $\langle srcIP, dstIP, srcPort, dstPort, protocol \rangle$ flow rules while other switches have flow rules with lesser granularity. Whenever a new flow arrives at the access layer switch, it sends first packet of that flow to the controller via *packet_in* message. Flow parser keeps that flow in its *flow_db* and marks it as useful or non-useful, based on available information. Algorithms 2 and 3 are used to determine whether a TCP or UDP flow is useful or not.

Flow parser keeps all active flows in its *flow_db* hash table. *flow_db* is a hash table where 5-tuple IP *flow* is used as key and *flowStats* is used as value. FlowStats keeps statistics corresponding to each flow.

Flow Parser sends a boolean flag to Learning Switch for decision regarding the installation of flow. For a TCP flow, when Flow Parser receives a SYN packet, it sends *false* value to the Learning Switch asking it not to install flow rule for this packet. This is done to ensure that controller receives all three packets of TCP handshake mechanism.

Algorithm 2 Determining whether a TCP flow is useful or not

Require: Flow Map $flow_db < Flow, FlowStats >$

Require: Packet pkt

```

1:  $flow \leftarrow$  Extract flow from  $pkt$ 
2:  $rflow \leftarrow$  Reverse flow of  $flow$ 
3:  $flags \leftarrow$  Extract TCP flags from  $pkt$ 
4: if  $flags == SYN$  and  $flags \neq ACK$  then
5:    $flowStats.useful = false$ 
6:    $flowStats.state = SYN$ 
7:    $flow\_db.put(flow, flowStats)$ 
8:   return
9: end if
10: if  $flags == SYN$  and  $flags == ACK$  then
11:    $flowStats.useful = false$ 
12:    $flowStats.state = SYNACK$ 
13:    $flow\_db.put(flow, flowStats)$ 
14:   return
15: end if
16: if  $flags == RST$  and  $flags == ACK$  then
17:    $flowStats.useful = false$ 
18:    $flowStats.state = RST$ 
19:    $flow\_db.put(flow, flowStats)$ 

```

```

20:  return
21: end if
22: if flags == ACK then
23:   if flow_db.containsKey(rflow) == true and flow_db.containsKey(flow) == true then
24:    flowStats1 = flow_db.get(flow)
25:    flowStats2 = flow_db.get(rflow)
26:    if flowStats1.state == SYN and flowStats2.state = SYNACK then
27:     flowStats1.useful = true
28:     flowStats1.state = ACK
29:     flowStats2.useful = true
30:     flowStats2.state = ACK
31:    return
32:   end if
33: end if
34: end if
35: if flow_db.containsKey(rflow) == false and flow_db.containsKey(flow) == false then
36:  flowStats.useful = true
37:  flowStats.state = TSR
38:  flow_db.put(flow,flowStats)
39:  return
40: end if
41: if flow_db.containsKey(rflow) == true and flow_db.containsKey(flow) == false then
42:  flowStats.useful = true
43:  flowStats.state = TRS
44:  flow_db.put(flow,flowStats)
45:  return
46: end if

```

Algorithm 3 Determining whether a UDP flow is useful or not

Require: Flow Map *flow_db* < *Flow*, *FlowStats* >

Require: Packet *pkt*

```

1: flow ← Extract flow from pkt
2: rflow ← Reverse flow of flow
3: if flow_db.containsKey(rflow) == false and flow_db.containsKey(flow) == false then
4:  flowStats.useful = false
5:  flowStats.state = USR
6:  flow_db.put(flow,flowStats)
7:  return
8: end if
9: if flow_db.containsKey(rflow) == true and flow_db.containsKey(flow) == false then
10:  flowStats.useful = true
11:  flowStats.state = URS
12:  flow_db.put(flow,flowStats)
13:  flowStats ← flow_db.get(rflow)
14:  flowStats.useful = true
15:  return
16: end if

```

In our implementation, flow and flowStats are represented as follows:

- **flow** is $\langle SourceIP, DestinationIP, SourcePort, DestinationPort, NetworkProtocol \rangle$.
- **flowStats** is $\langle Useful, FlowState, LastAccessTime \rangle$ where
 - **Useful** is a boolean flag which is *true* if flow is useful, otherwise *false*.
 - **Flow State** represents the state of flow.
 - **Last Access Time** keeps the time in milliseconds, when the flow was received at the controller.

Flow parser maintains one of the following states for a TCP flow:

1. **SYN**: When a TCP SYN packet arrives of a new flow, the flow is stored in the *flow_db* and its flow state is marked as SYN.
2. **RST**: When a TCP RST packet arrives of a new flow, the flow is added to *flow_db* and its flow state will be RST (Note that the reverse flow is already stored in the *flow_db* with state as SYN).
3. **SYNACK**: If a TCP SYNACK packet is received of a new flow and its reverse flow is already there in *flow_db* with flow state as SYN then this flow is stored in *flow_db* with flow state as SYNACK.
4. **ACK**: When a TCP ACK packet arrives, if the flow and its reverse flow are present in *flow_db* with flow states SYN and SYNACK respectively then both flows are put in ACK state.
5. **TSR**: A flow is kept in TSR state if the TCP packet matches neither of the above mentioned scenario. It is possible in the case when the flow and its reverse flow were removed by the switch due to *idle_timeout*.
6. **TRS**: It is same as TSR, just that if the reverse flow exists, then the state of flow will be TRS.

A useful TCP flow will have either ACK, TSR or TRS state. If a TCP flow is marked as failed after seeing RST packet, NetMon keeps such flow in *flow_db* but does not install any flow rule for such flow in OpenFlow switch. These flows are then later removed from *flow_db* explicitly. TSR and TRS states represent that the flow was removed due to *idle_timeout* in OpenFlow switch.

It maintains one of the following states for a UDP flow:

1. **USR**: A new UDP flow is registered in *flow_db* and its reverse flow is not stored in *flow_db*.
2. **URS**: Reverse flow is already stored in *flow_db*.

4.4.6 Out-degree, In-degree, Port-degree and Flow-count

Algorithm 4 is used to calculate the out-degree, port-degree and flow-count for each IP address. *While* loop in line number 1 is used to get the flow list corresponding to each IP address. In line number 9, *While* loop is used to iterate over all flows corresponding to an IP address. NetMon runs this algorithm to print these statistics once in a minute. Similar approach is used to get in-degree of each host in the network.

Algorithm 4 Calculating out-degree, in-degree, port-degree and flow-count

Require: Source IP stats hash table $srcHostStatsMap < IP, IPStats >$

Require: Set to keep destination IP $otherHostSet < IP >$

Require: Set to keep source port $srcPortSet < PortNo >$

```
1: while  $srcHostStatsMap$  has items do
2:    $i \leftarrow srcHostStatsMap.item()$ 
3:    $ip \leftarrow i.key()$ 
4:    $ipStats \leftarrow i.value()$ 
5:    $flowList \leftarrow ipStats.getFlowList()$ 
6:    $otherHostSet.clear()$  {Clears sets}
7:    $srcPortSet.clear()$ 
8:    $flowCount \leftarrow 0$  {Resets flowCount}
9:   while  $flowList$  has items do
10:     $flow \leftarrow flowList.item()$ 
11:     $srcPort \leftarrow flow.getSourcePort()$ 
12:     $destIP \leftarrow flow.getDestinationIP()$ 
13:     $flowCount \leftarrow flowCount + 1$ 
14:     $srcPortSet.add(srcPort)$ 
15:     $otherHostSet.add(destIP)$ 
16:   end while
17:   Output  $ip, srcPortSet.size(), otherHostSet.size()$  and  $flowCount$ 
18: end while
```

4.4.7 Web Interface

NetMon provides a web-GUI for visualizing the state of the network. Figure 4.3 shows form for tracing a device in the network. It expects start time, end time and an IP address, and displays the list of devices which have used that IP address in a given time range.

List of all the active hosts is shown in Figure 4.4. It displays IP address, MAC address, switch DPID and port number for each active host in the network. Figure 4.5 displays out-degree, in-degree, port-degree and flow count for each device in the network in a tabular form. NetMon allows to sort records based on any column (such as out-degree) of the table. These statistics are updated in every minute. NetMon also presents the graphs, between in-degree and out-degree, as well as between port-degree and active flows of all the hosts in the network.

NetMon presents the fraction of useful flows for each device in the network as shown in Figure 4.6. NetMon provides a historical view of these statistics. NetMon displays graphs for each device showing the fraction of useful flow in a time period of last day, week, month and year.

4.5 Evaluation

Device Logger

Device Logger is scalable and well-suited to track devices which have used an IP address in a given time range. NetMon puts record for a device only when a trigger is generated. This is done to minimize the number of entries in the database. In most of the cases, a device gets the same IP address whenever it joins the network. In such cases, NetMon does not put record for the device.

For 20000 users who have 5 devices each in the network, NetMon will take approximately 3.4 TB space for storing the records about devices in one year. We consider that each record takes 20

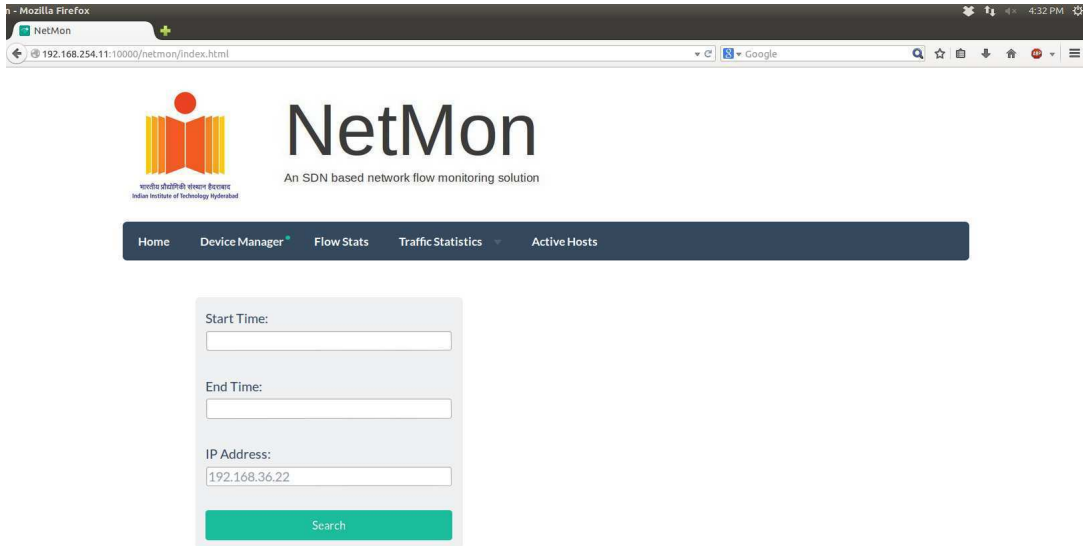


Figure 4.3: Form for tracing the device

bytes for storage and a device changes its location or IP address 5 times in a day. This number will further decrease, if the network has wired hosts with static IP addresses.

Fraction of Useful Flow

Table 4.1 shows the fraction of useful-flows generated by the end-systems in our SDN network. The total number of hosts which were connected to the SDN network, was 83. We found that in our network, use of proxy servers is the main cause for generating non-useful flows. Devices failed to update or install security patches due to hosts' inability to configure proxy settings. Other causes of non-useful flows are attempts to discover services, misconfiguration (such as incorrect DNS or proxy settings) and excessive connection retries. We also observe that default features of an application (such as LAN sync in Dropbox), which are not being used by a user, unnecessarily consumes bandwidth in the network.

Fraction of useful flows (in %)	Number of hosts (in %)
$\geq 95\%$	44.58 %
$\geq 80\%$ and $< 95\%$	27.41 %
$\geq 65\%$ and $< 80\%$	16.87 %
$\geq 65\%$ and $< 50\%$	3.61 %
$\leq 50\%$	7.23 %

Table 4.1: Health of hosts in IITH SDN network

The screenshot shows the NetMon web interface. At the top, there is a logo for NetMon and the text "An SDN based network flow monitoring solution". Below this is a navigation menu with options: Home, Device Manager, Flow Stats, Traffic Statistics, and Active Hosts. The main content area displays a table of active hosts. The table has four columns: IP Address, MAC Address, Switch DPID, and Port No. The table is sorted by IP Address in descending order. The data is as follows:

IP Address	MAC Address	Switch DPID	Port No
192.168.96.89	38:59:f9:dd:08:db	0x1f4a45d362c289c0	12
192.168.252.97	18:03:73:cb:12:7e	0x1f4a45d362c22500	14
192.168.252.93	f0:4d:a2:da:f7:44	0x1f4a45d362c22500	5
192.168.252.84	78:2b:cb:8d:37:1b	0x1f4a45d362c22500	24
192.168.252.82	f0:4d:a2:db:92:9f	0x1f4a45d362c22500	21
192.168.252.42	58:94:6b:8b:67:84	0x1f4a45d362c289c0	12
192.168.252.34	00:8c:fa:68:be:24	0x1f4a45d362c22500	2
192.168.252.27	f0:4d:a2:da:fe:17	0x1f4a45d362c22500	6
192.168.252.243	00:19:99:a9:c5:d7	0x1f4a45d362c289c0	20
192.168.252.239	f0:4d:a2:da:f7:04	0x1f4a45d362c22500	13
192.168.252.235	f0:4d:a2:da:fe:c0	0x1f4a45d362c22500	17
192.168.252.232	78:2b:cb:8d:36:dc	0x1f4a45d362c22500	8
192.168.252.221	78:2b:cb:8d:33:13	0x1f4a45d362c22500	7

Figure 4.4: Table for active hosts in the network

Out-degree and In-degree

High difference between out-degree and in-degree is an indication for error or malicious things happening in the network. In our network, proxy servers which are sitting outside the SDN network, have the highest out-degree and in-degree.

Port-degree and Flow count

Port-degree and flow count can be used to classify hosts as server, client or peer node. Servers have very low port-degree and high flow count while clients have high port-degree and low flow count. Peer nodes have high port-degree and moderate flow count.

4.6 Discussion

NetMon is designed for OpenFlow networks. In OpenFlow networks, switches trigger control messages like *packet_in* and *flow_removed* whenever a new flow arrives or a flow entry expires. NetMon requires flow rules to match source IP, destination IP, source port, destination port, and protocol fields with the incoming packet. For calculating a few flow metrics, it is required that controller should receive the first packet of every new TCP/UDP flow. So these fields can not be chosen for wildcard and OpenFlow switches will have a large number of flow rules in their flow tables. Flow tables of switches will be exhausted as switches have a limited high speed memory (such as TCAMs). Large number of distinct flows will also trigger many control packets. It will make the controller busy and it might not be able to process them in a timely fashion. In practice, the need for scalability pushes operators to increasingly adopt alternative approaches like distributed SDN controller or partition the flow rules among all switches across the network.

Host IP Address	TCP Sent	UDP Sent	Total Sent	TCP Received	UDP Received	Total Received	Out Degree	In Degree	Port Degree	Sent FlowCount	Received FlowCount
192.168.252.220	132669	9	132678	132116	4	132120	10	8	23	23	11
192.168.254.11	131949	0	131949	131929	0	131929	1	1	1	1	5
192.168.36.22	28588	0	28588	19534	0	19534	10	15	1	174	235
192.168.35.5	26932	0	26932	12086	0	12086	3	5	1	6	9
192.168.252.243	11206	0	11206	26023	0	26023	2	2	3	3	3
192.168.252.97	9778	0	9778	19455	0	19455	1	1	36	36	36
192.168.252.93	0	6187	6187	0	2	2	6	2	4	6	2
192.168.252.61	2842	0	2842	3727	0	3727	2	2	101	101	73
192.168.252.10	2343	472	2815	2561	11	2572	4	3	61	62	48
192.168.252.209	13	2311	2324	2	0	2	2	1	4	4	2
192.168.252.34	1814	0	1814	2369	0	2369	2	2	12	12	10
192.168.252.82	1386	0	1386	2202	0	2202	1	1	10	10	10
192.168.252.84	1311	0	1311	0	0	0	1	0	1	1	0

Figure 4.5: Table for out-degree, in-degree, port-degree and flow count

- A. *Distributed SDN Controller*: We can partition a large network into many smaller networks each having its own controller. We can also use a physically distributed but logically centralized controller. Some of the work has already done in this area such as [45].
- B. *Partitioning the Flow Rules*: NetMon requires that every new flow should reach the controller. If there are N switches along the path between source and destination IP pair then it will cause switches to send N control messages for a flow. In the implementation of NetMon, all other switches except the ingress switch have wildcard flow rule. So the controller gets the information about every new flow through only the ingress switch. Other switches have the flow rules in which only source IP and destination IP are matched. So they will not forward the same flow to the controller. In our approach, flow tables of ingress switches can exhaust. Instead of this, for every pair of hosts who communicate with each other, any one of the switches that fall in their path, can be marked as an authority switch. We can develop an algorithm, which takes the number of flows on each switch and source and destination IP as input and determines the authority switch for that IP pair. A similar approach is suggested in [46] for scalable flow-based networking. In current implementation of NetMon, an ingress switch is always assigned as an authority switch. Controller receives the information about the arrival or expiration of a flow through only the assigned authority switch.

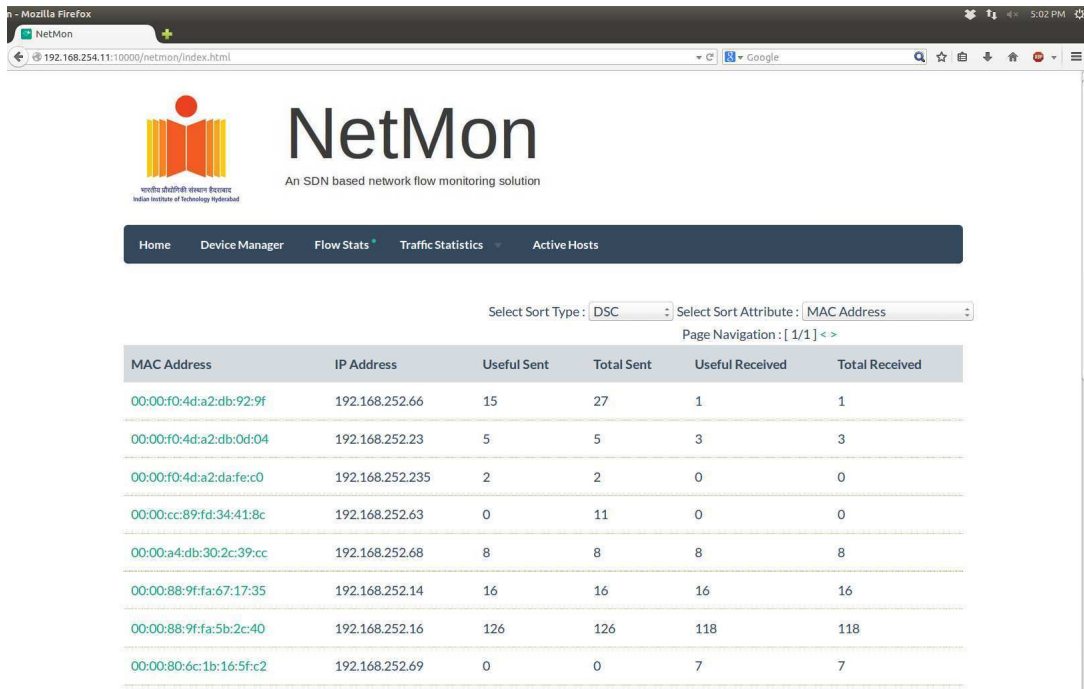


Figure 4.6: Table for fraction of useful flows

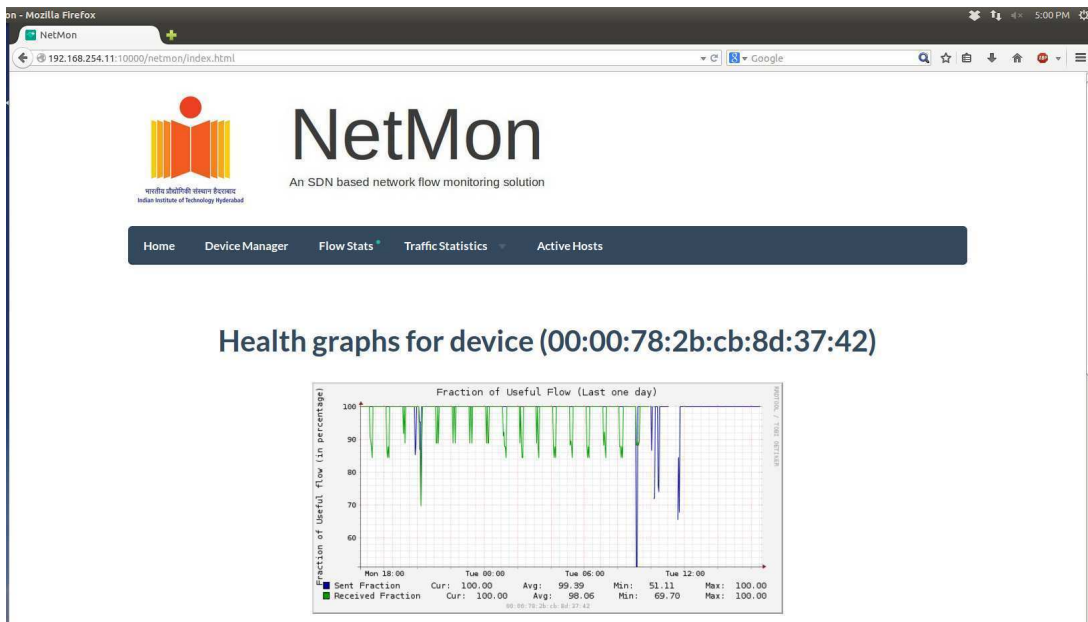


Figure 4.7: Graph for health of a host

Chapter 5

SDN in IITH Campus

In SDN, the network architecture is vendor neutral because the behavior of each switch is defined in a uniform manner (such as OpenFlow protocol). Vendor-agnostic architecture gives a big advantage in the deployment of an enterprise network, which is setup in an incremental fashion based on the expected number of users. As IIT Hyderabad (IITH) is a newly established university and the size of its network has been increasing every year, we also adopt an incremental approach for deploying the campus network.

The transition of existing networks to SDN will not be instantaneous considering the cost and risk factor etc. Panopticon [47] takes the topology of the network and cost for deploying SDN as input, and gives location of switches that need to be replaced with SDN switches. It tries to maximize the benefit of SDN while minimizing the cost, in a partial SDN deployment. Currently there are three categories of switches, viz., traditional switches, pure OpenFlow switch and hybrid switches. We use hybrid switches for deployment of SDN in IITH campus. Hybrid switches can work as, both, traditional switches and OpenFlow switches. In hybrid switches, a single VLAN can be run in OpenFlow while other VLANs in traditional mode using traditional protocols. Hybrid switches support fail-stand-alone mode, if OpenFlow controller is not working then they can switch back to traditional mode.

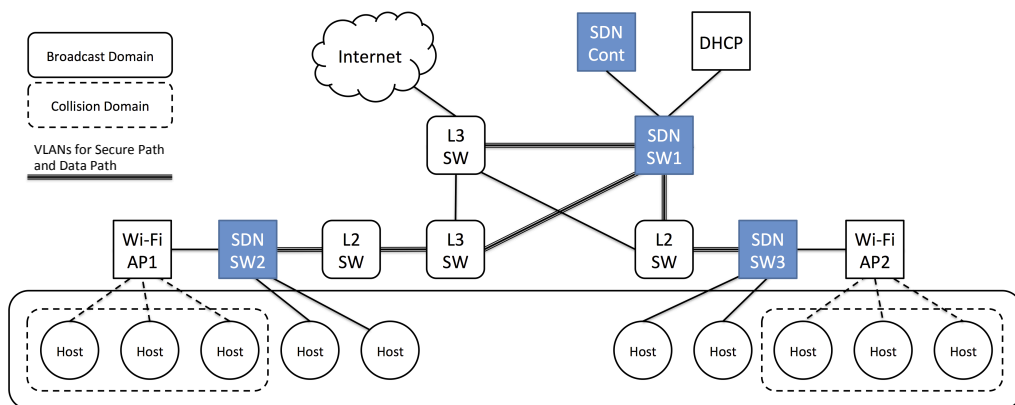


Figure 5.1: SDN Deployment Overlaying IIT Hyderabad Campus Network

In IITH, the SDN deployment is done in overlay mode where traditional LAN remains as is and

SDN traffic goes across the network through a particular VLAN. Currently, three SDN switches are installed as described in Figure 5.1 and two Wi-Fi APs in bridge mode are connected to them. The SDN network forms a single broadcast domain. The following TABLEs 5.1 and 5.2 describe the hardware and software specifications of SDN switches and the controller that are introduced in the campus network.

Table 5.1: SDN Controller Specification for Deployment

Model	Lenovo ThinkPad X201s
Operating System	Ubuntu 12.04 LTS
CPU	Intel Core i7 L620 2.0 GHz
RAM	8GB
NIC	Intel 82577LM Gigabit Ethernet Network Connection
SDN Controller	Floodlight 0.90
OpenFlow Version	1.0

Table 5.2: SDN Controller Specification for Deployment

Switch Model	HP J9573A 3800-24G-Poe+-2SFP+
Firmware version	KA.15.13.0005

Both ETF and NetMon are deployed in the SDN network of IITH. The number of users connected to SDN network is within a range of 20 to 50. In future, we will deploy a few more SDN switches and more Wi-Fi APs connected to them.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Large scale broadcast domains using Ethernet indeed simplify network design. However, while broadcast is actively used to operate various networking protocols, the increase of broadcast traffic will result in the reduction of communication quality. To overcome such a trade-off, this thesis proposed Extensible Transparent Filter (ETF) that suppresses broadcast traffic in a broadcast domain using SDN. SDN enabled on-switch filtering of broadcast packets that will reach only the intended destination without disturbing other end-systems. ETF supports the suppression of DHCP and ARP packets and its deployment has been done using production SDN switches. In both DHCP and ARP, even though the SDN switches receive large number of broadcast packets, ETF exhibited good performance to wired hosts up to 88%, and also localized the impact of broadcast packet within the collision domain of each of Wi-Fi APs.

Network monitoring is required to ensure that network is up and running as planned. Many network management applications require flow-based network monitoring. Traditional approaches are not scalable with the network growth. This thesis implements a tool NetMon for flow-based network monitoring. In NetMon, flow rules are distributed among all the switches which send the statistics to the controller in a push-based manner. It utilizes the features of the OpenFlow protocol and does not require any additional hardware for monitoring purpose. NetMon records the devices and their locations in the network. This helps the network administrator to trace the device and its location, based on an IP address and a given time range. NetMon calculates the fraction of useful flows, out-degree, in-degree, port-degree, and flow-count for each host in the network. NetMon has also been deployed in ITTH SDN network. We observed that only 44.58 % of the total end-systems generate 95 % or more useful-flow.

6.2 Future Work

As future work, performance tuning of ETF and NetMon will be done by carefully examining the configuration parameters on both applications, SDN controller and SDN switches. Many of the available hybrid hardware switches do not have clean implementation of OpenFlow, so it is also important to examine the impact of this on performance and functions of systems. We will test

ETF in a much larger network comprising of 10 Wi-Fi APs for a longer period. With OpenFlow 1.3, we would like to filter the broadcast traffic of DAD and ND in IPv6 and some other broadcast based protocols.

We will develop a partitioning algorithm for distributing flow rules almost equally on all SDN switches as discussed in Section 4.6. Currently NetMon presents raw data about the out-degree, in-degree, port-degree and flow-count. We would like to develop a network monitoring application which uses these raw data. In future, we will calculate a few more flow-based metrics (such as response time) for determining the state of the network.

References

- [1] P. Oppenheimer. Top-down network design. Cisco Press, 2004.
- [2] J. Case, M. Fedor, M. Schoffstall, and J. Davin. RFC 1157: Simple network management protocol (SNMP). *IETF, April* .
- [3] L. Deri and S. Suin. Effective traffic measurement using ntop. *Communications Magazine, IEEE* 38, (2000) 138–143.
- [4] B. Claise. Cisco systems NetFlow services export version 9 .
- [5] P. Phaal and M. Lavine. sflow version 5 2004.
- [6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, (2008) 69–74.
- [7] S. Dabkiewicz, R. van der Pol, and G. van Malenstein. OpenFlow network virtualization with FlowVisor .
- [8] L. Jose, M. Yu, and J. Rexford. Online measurement of large traffic aggregates on commodity switches. In Proc. of the USENIX HotICE workshop. 2011 .
- [9] J. R. Ballard, I. Rae, and A. Akella. Extensible and scalable network monitoring using opensafe. *Proc. INM/WREN* .
- [10] R. Wang, D. Butnariu, J. Rexford et al. OpenFlow-based server load balancing gone wild 2011.
- [11] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: dynamic access control for enterprise networks. In Proceedings of the 1st ACM workshop on Research on enterprise networking. ACM, 2009 11–18.
- [12] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang. Meridian: an sdn platform for cloud network services. *Communications Magazine, IEEE* 51, (2013) 120–127.
- [13] O. M. E. Committee et al. Software-Defined Networking: The New Norm for Networks. *ONF White Paper. Palo Alto, US: Open Networking Foundation* .
- [14] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN. *Queue* 11, (2013) 20.

- [15] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turetli. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys and Tutorials (Under Review)* .
- [16] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending Networking into the Virtualization Layer. In *Hotnets*. 2009 .
- [17] Open vSwitch. <http://openvswitch.org/> 2014.
- [18] ONF Specifications OpenFlow Switch Specification Version 1.0.0. <http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf> 2009.
- [19] Floodlight Java based OpenFlow Controller. <http://www.projectfloodlight.org/floodlight/>.
- [20] Ryu. <http://osrg.github.io/ryu/>.
- [21] Trema. an open source modular framework for developing openflow controllers in ruby/c. <https://github.com/trema/trema> 2013.
- [22] POX. <http://www.noxrepo.org/pox/about-pox/>.
- [23] D. Plummer et al. An Ethernet address resolution protocol (RFC 826). *Network Working Group* .
- [24] R. Droms. RFC 2131 dynamic host configuration protocol, March 1997. *Obsoletes RFC1541. Status: DRAFT STANDARD 3*.
- [25] N. W. Group et al. RFC 1001 Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods. Technical Report 1987.
- [26] M. Christensen, K. Kimball, and F. Solensky. RFC 4541 Considerations for Internet Group Management Protocol (IGMP) and Multicast Listener Discovery (MLD) Snooping Switches. *Status: Informational, May* .
- [27] A. Myers, E. Ng, and H. Zhang. Rethinking the service model: Scaling Ethernet to a million nodes. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Networking*. Citeseer, 2004 .
- [28] K. Elmeleegy and A. L. Cox. Etherproxy: Scaling ethernet by suppressing broadcast traffic. In *INFOCOM 2009, IEEE*. IEEE, 2009 1584–1592.
- [29] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: a scalable ethernet architecture for large enterprises. In *ACM SIGCOMM Computer Communication Review*, volume 38. ACM, 2008 3–14.
- [30] M. Scott, A. Moore, and J. Crowcroft. Addressing the Scalability of Ethernet with MOOSE. In *Proc. DC CAVES Workshop*. 2009 .
- [31] J. Wang, W. Zhao, S. Yang, J. Liu, T. Huang, and Y. Liu. FSDM: Floodless service discovery model based on Software-Defined Network. In *Communications Workshops (ICC), 2013 IEEE International Conference on*. IEEE, 2013 230–234.

- [32] R. M. Hinden and S. E. Deering. IP version 6 addressing architecture .
- [33] S. Thomson and T. Narten. RFC 2462 IPv6 Stateless Address Autoconfiguration, 1998 .
- [34] S. Guha, J. Chandrashekar, N. Taft, and K. Papagiannaki. How healthy are today’s enterprise networks? In Proceedings of the 8th ACM SIGCOMM conference on Internet measurement. ACM, 2008 145–150.
- [35] R. Mortier, R. Isaacs, and P. Barham. Anemone: using end-systems as a rich network management platform. In Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data. ACM, 2005 203–204.
- [36] T. Karagiannis, K. Papagiannaki, N. Taft, and M. Faloutsos. Profiling the end host. In *Passive and Active Network Measurement*, 186–196. Springer, 2007.
- [37] D. Joumblatt, R. Teixeira, J. Chandrashekar, and N. Taft. Perspectives on tracing end-hosts: a survey summary. *ACM SIGCOMM Computer Communication Review* 40, (2010) 51–55.
- [38] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement. ACM, 2010 328–341.
- [39] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. cSamp: A System for Network-Wide Flow Monitoring. In NSDI, volume 8. 2008 233–246.
- [40] N. L. van Adrichem, C. Doerr, and F. A. Kuipers. OpenNetMon: Network Monitoring in OpenFlow Software-Defined Networks .
- [41] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In Proceedings 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI, volume 13. 2013 .
- [42] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha. Flowsense: Monitoring network utilization with zero measurement cost. In *Passive and Active Measurement*. Springer, 2013 31–41.
- [43] T. Oetiker. RRDtool 2005.
- [44] M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM* 34, (1991) 78–92.
- [45] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. ACM, 2013 7–12.
- [46] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. *ACM SIGCOMM Computer Communication Review* 40, (2010) 351–362.
- [47] D. Levin, M. Canini, S. Schmid, and A. Feldmann. Toward Transitional SDN Deployment in Enterprise Networks .