

# **SDN ASSISTED ROUTING IN IOT NETWORKS**

*A Project Report Submitted in Partial Fulfillment of  
the Requirements for the degree of*

**Bachelor of Technology  
in  
Computer Science & Engineering  
by**

**Arup Kar (Roll No. CSB17031)  
Rishabh Trivedi (Roll No. CSB17063)  
Masood Rehman (Roll No. CSB17070)**

**Guided by  
Dr. Nabajyoti Medhi**



**DEPARTMENT OF COMPUTER SCIENCE AND  
ENGINEERING  
SCHOOL OF ENGINEERING  
TEZPUR UNIVERSITY**

# ACKNOWLEDGEMENT

---

In completing this project, we had to take the help and guideline of some respected people, who deserve our greatest gratitude. The completion of this project gives us much pleasure.

We would like to show our gratitude to **Dr. Nabajyoti Medhi**, Assistant Professor, Department of Computer Science & Engineering, Tezpur University for providing us with this opportunity to take up the project under his guidance and providing us with good guidelines for the project throughout numerous consultations.

Many people, especially our classmates and team members itself, have made valuable comment suggestions on this proposal which gave us an inspiration to improve our assignment.

Lastly we would like to thank our parents for their support and motivation during this project.

We thank all the people for their help directly and indirectly to complete our project.

**ARUP KAR**

**RISHABH TRIVEDI**

**MASOOD REHMAN**

# **PREFACE**

---

This project report has been prepared in partial fulfillment of the requirement for Project I Programme for B.Tech in Computer Science and Engineering for the academic year 2020-21. The purpose of this program is to acquaint the students with practical applications of theoretical concepts taught to them during their course.

During our project we have learned about the various methodologies and technologies of Software Defined Networks. Every topic from crust to core was explained to us and this report is a result of what we have learned and implemented. Information in this report is gathered from different sources like project proposals, online websites and instruction manuals.

This document gives reader an insight of Software Defined Networks. Also it provides some ideas about the technological advancement with Computer Network. Interactive visuals provide an ease to reader for understanding process and doesn't let reader to get bored as traditional technical text does. I have tried my best to eliminate all mistakes and misrepresentation of facts but since it is natural for humans to make mistakes, so I ask your pardon in advance for any such mistake.

To create and run example topologies mininet was used, RYU controller was used as an SDN controller and Open vSwitch was used.

Thank you.

# TABLE OF CONTENT

---

Topic	Page No.
ACKNOWLEDGEMENT	i
PREFACE	ii
LIST OF FIGURES	iv
1. INTRODUCTION	1
2. LITERATURE REVIEW	2
3. PROPOSED SOLUTIONS	3
3.1. SADFIR Routing Protocol	3
3.2. Queue Implementation	3
3.3. Meter Table Implementation	5
4. THEORY	7
4.1. Quality of Service	7
4.1.1. Techniques to improve the quality of service	7
4.1.2. QoS criteria for IoT	11
4.2. Internet of Things	11
4.3. Software Defined Networks	13
4.4. Clusters	14
4.5. Ryu Controller	15
4.6. Mininet	16
4.7. Open vSwitch	17
5. METHODOLOGY	20
5.1. Architecture/Topology	20
5.2. Implementation	22
6. CONCLUSION	32
6.1. Future Work Possible	32
7. BIBLIOGRAPHY	33
APPENDIX: Code	34

# LIST OF FIGURES

---

Figure No.	Title	Page No.
3.1.	Priority Queue	4
3.2.	Meter table entry	5
3.3.	Meter Band	6
4.1. (a)	FIFO Queuing	8
4.1. (b)	Tail Drop	8
4.2.	Priority Queuing	8
4.3.	Weighted Fair Queuing	9
4.4.	Leaky Bucket	9
4.5.	Token Bucket	10
4.6.	RYU SDN controller logo	15
5.1.	Network Topology	20
5.2.	Nodes and Links in the Network	21
5.3.	Terminal running the topology	22
5.4.	Running the Ryu application in terminal	23
5.5.	Adding the meter table	24
5.6.	Adding switch_arp to flow entry	25
5.7.	Adding switch_flow1 to flowentry	26
5.8.	Adding switch_flow2 to flowentry	27
5.9.	Executing command in terminal to check flow table	28
5.10.	Executing command in terminal to check meter table	28
5.11.	Starting UDP server in h1 & h3 and performing traffic test from h1 to h3	29
5.12.	Performing traffic test from h2 to h1	30
5.13.	Finally checking meter table	31
5.14.	h1 ping h2	31

# 1. INTRODUCTION

---

The internet of things (IoT) is a system of interrelated computing devices, mechanical and digital machines or objects which are provided with unique identifiers (UIDs) and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. Devices and objects with built in sensors are connected to an Internet of Things platform, which integrates data from the different devices and applies analytics to share the most valuable information with applications built to address specific needs.

Software-Defined Networking (SDN) is an architecture that is dynamic, manageable, cost-effective and adaptable. It is ideal for high-bandwidth, dynamic nature of today's applications. SDN attempts to centralize network intelligence in one network component by disassociating the forwarding process of network packets (data plane) from the routing process (control plane) enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. SDN enables the programming of network behavior in a centrally controlled manner through software applications using open APIs. By opening up traditionally closed network platforms and implementing a common SDN control layer, operators can manage the entire network and its devices consistently, regardless of the complexity of the underlying network technology.

In IoT networks we can get flexibility and programmability by using SDN without troubling underlying architecture of existing implementations. In SDN based IOT, the routing protocol is proactively computed by the control plane. By overcoming heterogeneity of gadgets the interoperability of IoT gadgets can be resolved utilizing SDN.

The SDN controller manages and supervises the entire network. The central position of the SDN controller makes it appropriate to have a global vision of the network topology and conditions, performing system control, for example, directing and QoS control. The controller can decide the best directing choices and enter these into forwarding table.

## 2. LITERATURE REVIEW

---

The term "Internet of things" was coined by Kevin Ashton of Procter & Gamble in 1999, though he prefers the phrase "Internet for things". But the idea of IoT was first detailed in 2005, by the International Telecommunication Union (ITU) expressing the future society will be an "omnipresent system society" where bunches of smart articles can be associated with the web.

Demands for a better success rate of IoT have been rising in the industry. This has insisted the researchers to investigate opportunities promised by SDN proposed solutions to develop flexible management of IoT that separate the control logic from the network nodes or actuators. These advanced features of SDN-based central management of IoT architecture are attractive enough to gain researchers' attention as SDN-based management systems are being developed for every type of mainframe networks.

The SDN controller managing the network provides with the development of very flexible network management protocols. Accordingly, IoT nodes are capable of being reprogrammed for priority operations. So, various researchers have been investigating and proposing solutions for the management of IoT networks and many of these solutions have used the SDN approach.

In the past few years various routing protocols for IoT have been introduced. Some of them being Situation-Aware Routing Protocol Switching for SDN-based WSNs, Non-linear Weight Particle Swarm Optimization (NWPSO) algorithm-based routing protocol, QoS-aware routing mechanism for OpenFlow-enabled WSNs, SDN Enabled SPIN Routing Protocol and SDN-based Application-aware Distributed adaptive Flow Iterative Reconfiguring (SADFIR) routing protocol.

In the above mentioned routing protocols except for the SADFIR routing protocol all other routing protocols have two paramount downsides being lack of application awareness and distributed nature cluster-formulation. To bring about these two features SADFIR was introduced. The SADFIR routing protocol's main purpose was to manage heterogeneity-awareness and application-sensitivity of sensed information.

## 3. PROPOSED SOLUTIONS

---

### 3.1. SADFIR Routing Protocol

While researching for different ways to use SDN in IoT routing we stumbled across SDN-Based Application-aware Distributed adaptive Flow Iterative Reconfiguring (SADFIR) routing protocol. It extends the distributed property of clustering formulation by engaging the network nodes of Internet of Things (IoT).

SADFIR routing protocol computes the network topology with multiple SDN-controllers at the control layer and computes the residual resources to save computational cost of IoT devices. It improves the existing features and characteristics of SDN-based WSNs by providing us with network scalability, application awareness, heterogeneity awareness, topology discovery, low computational cost and periodic reconfiguration making it a very good solution for SDN based IoT routing.

The implementation of SADFIR is divided into three phases namely - Network Topology Management Phase (NTMP), Network Settling Phase (NSP) and Network Forwarding Phase (NFP).

This solution while being a good option was very complex and required a lot of time investment for us to successfully implement it and being complex it also gave us limited scope of future work. As it was one of the best SDN based IoT routing solutions, adding on top of it with limited time and our limited knowledge was not feasible. So, we decided to take a different approach and start from scratch.

### 3.2. Queue Implementation

The first solution we came up with was the use of queue. Queues are buffers in devices that hold data to be processed. Queues provide bandwidth reservation and prioritization of traffic as it enters or leaves a network device. If the queues are not emptied, they overflow and drop traffic.



A queue is also a collection of objects waiting to be processed, one at a time. In the networking environment, packets are queued up into the memory buffers of network devices like routers and switches. Packets in a queue are usually arranged in first-in, first-out order, but various techniques may be used to prioritize packets or ensure that all packets are handled fairly, rather than allowing one source to grab more than its share of resources.

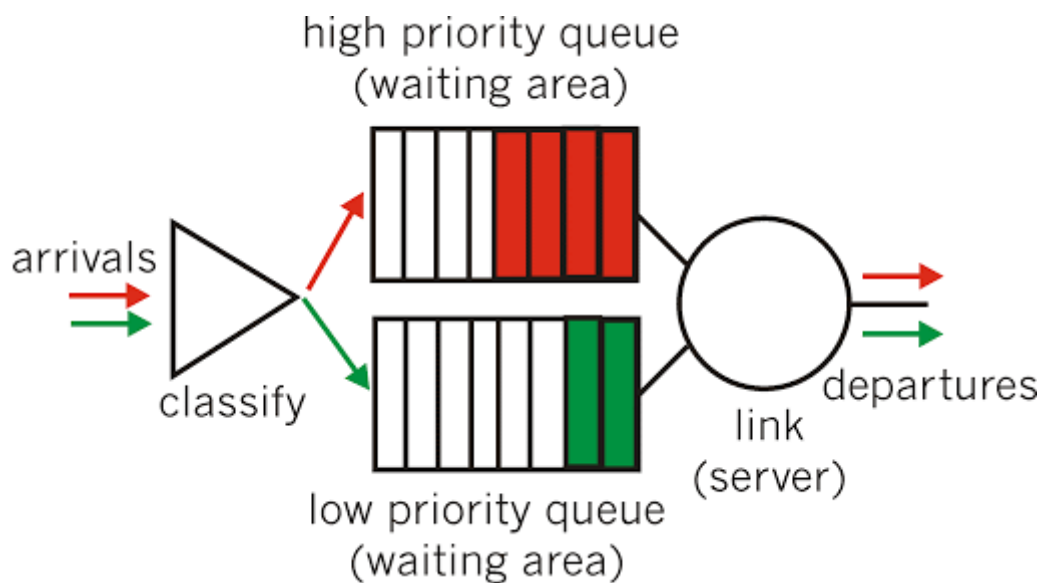


Fig 3.1. Priority Queue

OpenFlow queues have wide support across the board. They are supported on most software switch implementations (e.g. Open vSwitch (OVS), ofsoftswitch, etc), and many hardware switch vendors also support queues. Although queues are very useful, they are defined outside the OpenFlow protocol. OpenFlow merely provides a wrapper around existing switch queuing mechanisms in order to inform the controller of said queues. Queues must be defined/instantiated by the switch out of band. This means that for example if we are using OVS, then we must set up the queues with OVS commands prior to attempting to use them with OpenFlow. This is analogous to adding ports to an OpenFlow instance or virtual switch. OpenFlow 1.0 and 1.1 support queues with guaranteed minimum rates, while OpenFlow 1.2 and above support both minimum and maximum rates for a given queue.

Since queues are not managed by OpenFlow, OpenFlow is only able to query queue statistics from the switch. So, we only get limited Quality-of-Service (QoS) support through a simple

queuing mechanism. Thus, limiting our options if we opt for queues. So, we decided to look for a different approach.

### 3.3. Meter Table Implementation

The final solution we came up with was to use OpenFlow Meters. Meters are switch elements that can measure and control the rate of packets. The meter triggers a meter band if the packet rate or byte rate passing through the meter exceed a predefined threshold. If the meter band drops the packet, it is called a Rate Limiter.

A meter table consists of meter entries, defining per-flow meters. Per-flow meters enable OpenFlow to implement various simple QoS operations, such as rate-limiting, and can be combined with per-port queues to implement complex QoS frameworks.

OpenFlow 1.3 introduces meters to the OpenFlow protocol. Meters complement the queue framework already in place in OpenFlow by allowing for the rate-monitoring of traffic prior to output. More specifically, with meters, we can monitor the ingress rate of traffic as defined by a flow. Flows can direct packets to a meter using the goto-meter OpenFlow instruction, where the meter can then perform some operation based on the rate it receives packets.

A meter measures the rate of packets assigned to it and enables controlling the rate of those packets. Unlike queues that are attached to ports, Meters are attached directly to flow entries. Any flow entry can specify a meter in its instruction set, the meter measures and controls the rate of the aggregate of all flow entries to which it is attached. Multiple meters can be used in the same table, but in an exclusive way (disjoint set of flow entries). Multiple meters can be used on the same set of packets by using them in successive flow tables.

Meter Identifier	Meter Band	Counter
------------------	------------	---------

Fig 3.2. Meter table entry

A meter entry in the meter table consists of three components :-

- Meter identifier: an unsigned integer of 32 bits used to uniquely identify a meter
- Meter bands: an unordered list of meter bands. Meter bands specify the way to process the packets and the rate of the band.

- Counters: These are updated when the packets are processed by a meter

Band Type	Rate	Counter	Type Specific Arguments
-----------	------	---------	-------------------------

Fig 3.3. Meter Band

Each meter band contains :-

- Band type: This defines how packets are processed
- Rate: meter bands are identified by meters by their rate. It is the lowest rate to which the band can be applied
- Counters: These updates when a packet is processed by a meter band
- Type specific arguments: Some band types have some optional type specific arguments

Meters can assure that at least a minimum amount of traffic is allowed between a pair of hosts even in the presence of a lot of noisy traffic. So, after reviewing the functions and utilities of meters we decided to use them to set priority for flows depending on the source from which they came. This priority was then used to define the routing by the controller.

## 4. THEORY

---

### 4.1. Quality of Service (QoS)

Quality of Service refers to the capability of a network to provide better service to selected network traffic over various technologies, including Frame Relay, Asynchronous Transfer Mode (ATM), Ethernet and 802.1 networks, SONET and IP-routed networks that may use any or all of these underlying technologies. It is particularly important for the transport of traffic with special requirements like voice over IP (VOIP).

The goal of QoS is to provide priority including dedicated bandwidth, controlled jitter and latency and improved loss characteristics. It is also important to ensure that by providing priority for one or more flows, other flows do not fail.

QoS is also used to achieve outcomes such as:-

- Prioritize network and application traffic, guaranteeing high priority to important traffic or limiting non-essential traffic.
- Achieve equal bandwidth sharing among different subnets, classes or users in a network.
- Allocate bandwidth externally or internally or both, applying QoS to both upload and download traffic or to only one of them.
- Ensure low latency for customers and revenue-generating traffic in an enterprise environment.
- Perform traffic profiling of applications to ensure bandwidth usage.

#### 4.1.1. Techniques to improve the quality of service

- Scheduling
  - First in First Out (FIFO) Queuing: The first packet that arrives at a router is the first packet to be transmitted. This is illustrated in Figure 4.1. (a), which shows a FIFO with “slots” to hold up to eight packets. Given that the amount of buffer space at each router is finite, if a packet arrives and the queue (buffer space) is

full, then the router discards that packet, as shown in Figure 4.2. (b). This is done without regard to which flow the packet belongs to or how important the packet is. This is sometimes called tail drop, since packets that arrive at the tail end of the FIFO are dropped.

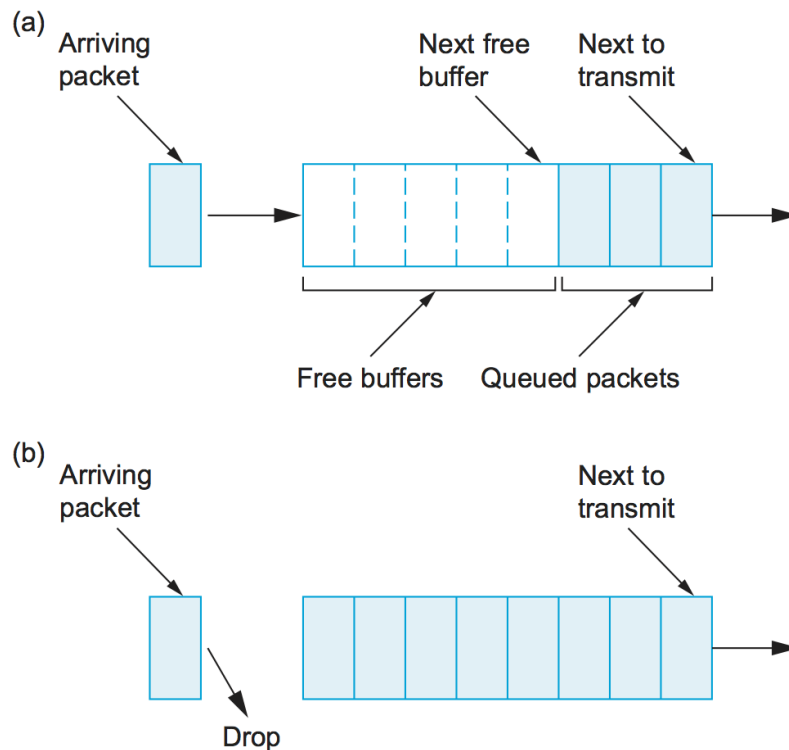


Fig 4.1. (a) FIFO Queuing, (b) Tail Drop

- Priority Queuing: A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

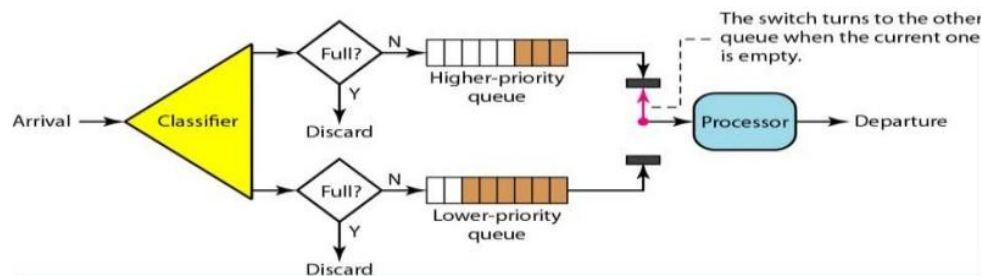


Fig 4.2. Priority Queuing

- Weighted Fair Queuing: Weighted fair queueing (WFQ) is a method of automatically smoothing out the flow of data in packet-switched communication networks by sorting packets to minimize the average latency and prevent exaggerated discrepancies between the transmission efficiency afforded to narrowband versus broadband signals. In WFQ, the priority given to network traffic is inversely proportional to the signal bandwidth. Thus, narrowband signals are passed along first, and broadband signals are buffered.

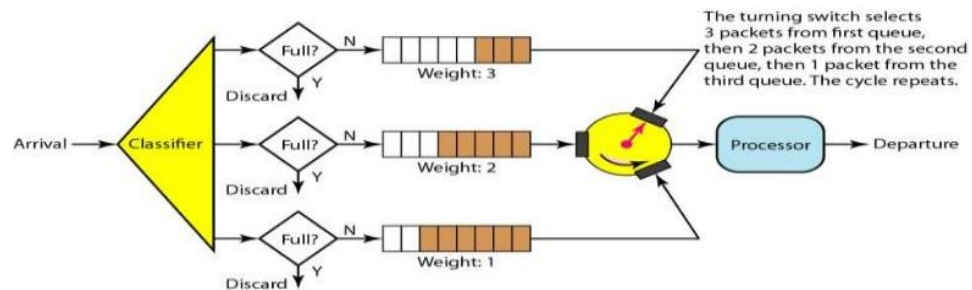


Fig 4.3. Weighted Fair Queuing

- Traffic Shaping:

- Leaky Bucket: Leaky bucket is a bucket with a hole at bottom. Flow of the water from bucket is at a constant rate which is independent of water entering the bucket. If bucket is full, any additional water entering in the bucket is thrown out. This same technique is applied to control congestion in network traffic.

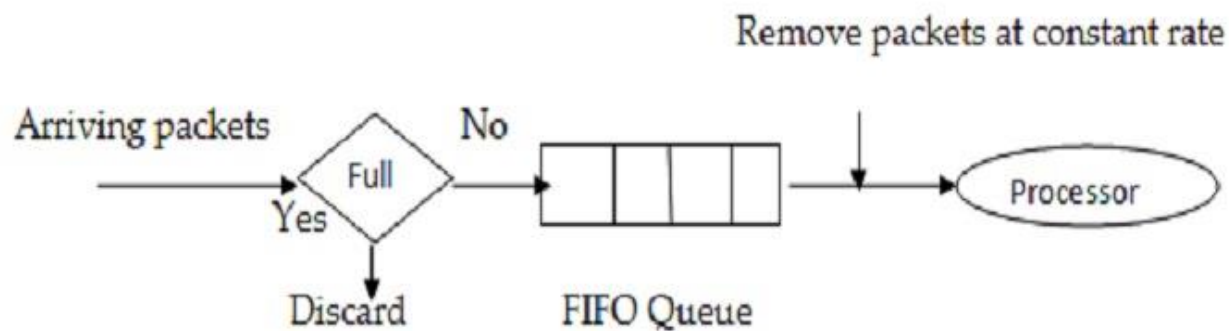


Fig 4.4. Leaky Bucket

- Every host in the network is having a buffer with finite queue length
- Packets which are put in the buffer is full are thrown away. The buffer may drain onto the subnet either by some number of packets per unit time, or by some total number of bytes per unit time.

- A FIFO queue is used for holding the packets.
  - If the arriving packets are of fixed size, then the process removes a fixed number of packets from the queue at each tick of the clock.
  - If the arriving packets are of different size, then the fixed output rate will not be based on the number of departing packets.
  - Instead it will be based on the number of departing bytes or bits.
- Token Bucket: The token bucket is an algorithm used in packet switched computer networks and telecommunications networks. It can be used to check that data transmissions in the form of packets conform to defined limits on bandwidth. It can also be used as a scheduling algorithm to determine the timing of transmissions that will comply with the limits set for the bandwidth.

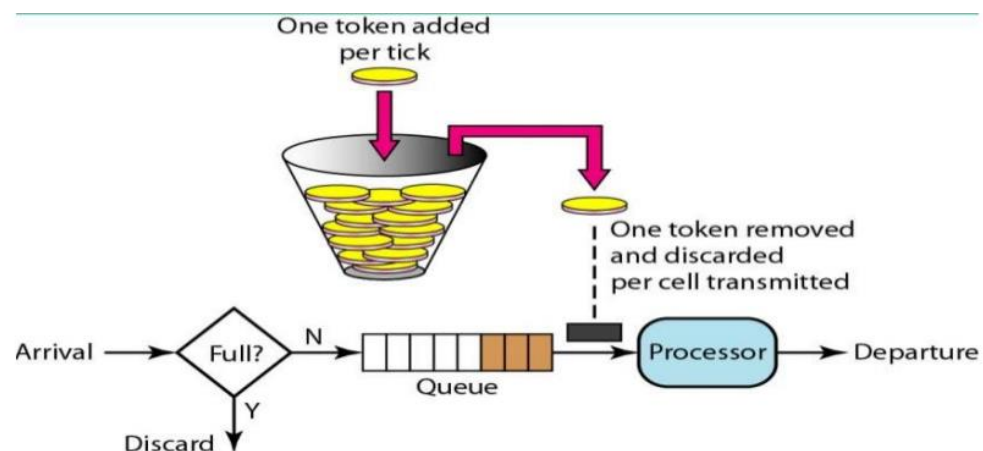


Fig 4.5. Token Bucket

The token bucket algorithm can be conceptually understood as follows:

- A token is added to the bucket every  $1/r$  seconds.
- The bucket can hold at the most  $b$  tokens. If a token arrives when the bucket is full, it is discarded.
- When a packet (network layer PDU) of  $n$  bytes arrives,
- If at least  $n$  tokens are in the bucket,  $n$  tokens are removed from the bucket, and the packet is sent to the network.

- If fewer than  $n$  tokens are available, no tokens are removed from the bucket, and the packet is considered to be non-conformant.
- **Resource Reservation:** Reserving resources such as buffer, bandwidth, CPU time etc beforehand. The Resource Reservation Protocol (RSVP) is a transport layer[1] protocol designed to reserve resources across a network using the integrated services model. RSVP operates over an IPv4 or IPv6 and provides receiver-initiated setup of resource reservations for multicast or unicast data flows.
- **Admission Control:** Admission control is a validation process in communication systems where a check is performed before a connection is established to see if current resources are sufficient for the proposed connection.

#### **4.1.2. QoS criteria for IoT**

The Internet of Things is expected to radically reshape many processes in a broad range of domains, from personal to industrial. In many of these heterogeneous scenarios, IoT systems will need to guarantee required levels of reliability and latency in order to provide high-quality services to end users. Quality of service support in IoT system will demand for explicit support at different levels. At the network level, on the one hand, specific technical communication standards will be necessary to ensure timed and reliable data delivery. At the application level, instead, dedicated support from application protocols and design of novel resource allocation algorithms will be mandatory to cope with concurrent access and implement proper management of resources.

## **4.2. Internet of Things (IoT)**

IOTs are basically internet technologies connecting devices and tools to the internet by means of wireless technologies. It involves unification of technologies such as low power embedded systems, cloud computing, big data, machine learning etc.

“In the 2000s, we are heading into a new area of ubiquity, where the users of the internet will be counted in billions and where humans may become the minority as generators and receivers of traffic. Instead, most of the traffic will flow between devices and all kinds of ‘things’, thereby



creating a much wider and more complex ‘Internet of things’.”

(The Internet of Things,ITU Internet Report 2005)

Some characteristics of IOT involve -

- Efficient,scalable architecture
- Unambiguous naming and addressing system
- Abundance of sleeping nodes and mobiles
- Intermittent connectivity

It is expected that IOT will involve applications by the millions,smart systems by billions and sensors by the trillions in the near future. Some common terminologies used in IOT involve IOT gateway, IOT LAN, WAN, PROXIES and Nodes. Some of the IOT network configurations are restricted to local areas where they have to communicate to the internet via a gateway.

Some functional components of IOT involve -

- Component for interaction and communication with other IOT devices.
- Component for processing and analysis of operations.
- Component for internet interaction.
- Component for handling web services of applications.
- Component to integrate application services.
- User Interface to access IOT.

Some of the challenges faced while developing an IOT device are security, scalability, bandwidth management and complexity management such as SDN.

Communication between the IOT device and the outside world dictates the network architecture. Choice of communication technology dictates the IOT devices hardware requirement and costs. Due to the presence of numerous applications of IOT enabled devices, a single networking paradigm not sufficient to address all the needs of the consumer or the IOT device.

Overall,IOT devices involve wide flexibility with the internet. Further they can be deployed in large scale and ensure connectivity with billions of devices in real time

### 4.3. Software Defined Networks (SDN)

SDN is a new approach to computer networking which promises to simplify network control, management and support much innovation through network programmability. It separates the control plane and the data plane. This separation helps in better management of the network with efficient handling of the network traffic on different planes of the SDNs architecture. The data plane in SDN forwards network traffic based on the control plane instructions. The SDN controller builds network intelligence by observing the data plane forwarding entities and other SDN agents. No doubt, the centralized control helps in better network management; however, it always becomes a bottleneck when it comes to exchanging large volumes of data. Moreover, due to the centralized architecture of the controller, it experiences overhead as the number of user increases. Consequently, the controller becomes an obstacle to the smooth provision of service, and if the controller itself fails, the switch that it had been managing can no longer be controlled. Moreover, the SDN controller act as a single point of failure because all the forwarding decisions are dependent directly on the controller. Once the SDN controller or the switches-to-controller links fail, the entire network may collapse.

The SDN architecture is:-

- directly programmable
- centrally managed
- agile and
- programmatically configured.

The scalability, reliability, inter-operability and fault tolerance remains a challenge in centralized network architectures. However, the positive aspect of SDN is that it is centralized but highly flexible and programmable at the same time. The network programmability aspects of SDNs makes unique. Moreover, the SDNs support multiple distributed SDN controllers to be connected to a network serving as backup controllers in the time of a failure. Moreover, multiple controllers allow load sharing when a single controller is overwhelmed with numerous flow requests. Furthermore, multiple controllers can reduce the latency, increase the scalability and fault tolerance, and provide availability in SDN deployment. However, the main problem with this approach is to maintain the consistency among various distributed controllers. The network

applications will be treated improperly by the distributed controllers because of inconsistency among the controllers concerning global view of the network states. In addition, multiple controllers create controller resource management problems, including controller state distribution, data sharing, consistency, and long propagation delay among multiple controllers which limits the network convergence time as well as affects the ability of the controller to respond to the various network events in minimal time such as PACKET\_IN messages.

## **4.4. Clusters**

Cluster networks are two or more computing devices working together for a common computing purpose. These networks take advantage of the parallel processing power of the computing devices. In addition to the increased processing power, shared computing resources in a cluster network have the capability to provide scalability, high availability, and failover capabilities should one computing device have a problem. There are three basic types of computing clusters: load-balancing clusters, high availability clusters, and high-performance clusters.

Nodes are computing systems in load balancing clusters. The workload of the network is spread over these nodes to increase the computing performance of the network. From a user perspective, the nodes function as one computer system.

High-availability cluster networks or failover clusters are two or more computing nodes that provide redundancy in case of hardware or software failure. When a computing system fails, its operations are transferred to the redundant node to provide continuous computing services.

High-performance clusters use the parallel processing power of the multiple cluster nodes to provide high performance computing. This allows the nodes to work together on a common problem. It is often a good solution for businesses whose networks have high processing requirements but that have limited budgets.

Clustering architectures can vary greatly depending on the networking technologies used and the purpose of the computing system. There are three primary clustering architectures: mirrored disk, shared disk, and shared nothing configurations. A mirrored-disk cluster architecture replicates stored application data to a backup storage site. Its basic purpose is to provide high availability

of the computing resources as well as disaster recovery in case of some type of computing failure.

A shared-disk cluster network uses central input/output (I/O) devices that are accessible to all nodes in the cluster. Usually, they are used for sharing disk storage for files and databases. Some shared-disk configurations distribute information across all the nodes in a cluster, whereas other configurations use a central metadata server.

A shared-nothing clustering architecture has independent and self-sufficient nodes. Each node has its own memory and I/O devices. It does not provide concurrent disk access from multiple nodes, because only one node needs access to the storage at any one time. Often each node in this type of architecture is responsible for a different set of network tasks. A shared-nothing cluster network can scale to hundreds of nodes.

#### **4.5. RYU Controller**

RYU is like a component based software defined networking framework. It provides software components with well defined API. This makes it easy for developers to create new network management and control applications. It supports various protocols like openflow, OVSDb, BGP.



Fig 4.6. RYU SDN controller logo

There are various inbuilt applications in RYU and RYU Manager listens on openflow ports (6653) and are in listening states. RYU applications are in python script and they can run in multiples on RYU in a single initiation.

Some RYU controller command lines are:-

- RYU-manager--help # To know all the available options
- RYU-manager--verbose # To enable the debug logs
- RYU-manager--ofp-tcp-listen-port 6634 # To use custom openflow port number
- RYU-manager --observe-links # To use Topology Discovery

IT administrators write specific applications which can communicate with the RYU Controller. The RYU Controller interacts with the forwarding plane in order to modify how the network will handle traffic flows using protocols such as OpenFlow.

## 4.6. Mininet

Mininet is responsible for creating a realistic virtual network which runs kernel,switch and applications on a single machine (including cloud and VMs) instantly with a single command.

```
> sudo mininet
```

With Mininet CLI and API,we can easily interact with our network,customise it or share it with others or even deploy it on real hardware. As such, Mininet is useful for development, teaching and research.

Some functions of Mininet include:

- Providing a simple,cheap network testbed to develop OpenFlow applications
- Enabling multiple concurrent developers who can work on the same topology independently.
- Enabling complex topology testing.We don't even need to wire up a physical network.
- A topology-aware and OpenFlow-aware CLI for running network wide tests or debugging.
- Supporting arbitrary custom topologies even including parametrized topologies.

- usability out of the box without programming.
- Providing Python API for network creation and experimentation which is straightforward and easily extensible.

Mininet networks run real code and as such the code we develop on Mininet including for OpenFlow Controller or modified switches or host can be used for real world testing with minimal changes.

Some advantages of Mininet include faster loading time, large scalability and providing more bandwidth. Even then, this large giant hub of a tool is not without its limitations. Mininet based networks can't exceed the CPU or bandwidth available yet. Also Mininet requires the Linux OS, without which it will be unable to run OpenFlow switches or applications.

## 4.7. Open vSwitch

Open vSwitch is a multi layered production quality virtual switch licensed under the Apache 2.0 license. Its design enables massive network automation through programmatic extension. Even then, it still supports standard management interfaces and protocols such as NETFLOW, sFLOW etc. Its design also enables distribution over multiple physical servers.

However there exists a Linux Switch L2 which is fast and reliable in order for the hypervisor to communicate with the outside world. Thus it is only natural to speculate over the need for Open vSwitch. The answer lies in the fact that Open vSwitches can be targeted at multi server virtualization deployments, something for which the previous stack is not well suited for. These environments are often characterized by highly dynamic end-points, the maintenance of logical abstractions, and sometimes integration with or offloading to special purpose switching hardware.

The following characteristics and design considerations help Open vSwitch to live up to its name:-

- **The Mobility of State:**  
All network states (including traditional soft states, L3 forwarding state, policy routing state, ACL, QoS policy etc) associated with a network entity should be easily identifiable and migratable between different hosts.

Open vSwitch has support for both configuring and migrating both slow (configuration) and fast network state between instances. For example, if a VM migrates between end-hosts, it is possible to not only migrate associated configuration (SPAN rules, ACLs, QoS) but any live network state (including, for example, existing state which may be difficult to reconstruct). Furthermore, Open vSwitch state is typed and backed by a real data-model allowing for the development of structured automation systems.

- **Responding to network dynamics**

Virtual environments are often characterized by high-rates of change which include VMs coming and going, VMs moving backwards and forwards in time, changes to the logical network environments, and so forth.

The features in Open vSwitch allow a network control system to respond and adapt as the environment changes including simple accounting and visibility support such as NetFlow, IPFIX, and sFlow. Moreover, Open vSwitch supports a network state database (OVSDB) that supports remote triggers. Therefore, a piece of orchestration software can "watch" various aspects of the network and respond if/when they change. This can be used for example, to respond to and track VM migrations.

Open vSwitch also supports OpenFlow as a method of exporting remote access to control traffic. There are a number of uses for this including global network discovery through inspection of discovery or link-state traffic (e.g. LLDP, CDP, OSPF, etc.).

- **Hardware Integration**

Open vSwitch's forwarding path is designed to be compliant to "offloading" packet processing to hardware chipsets. This allows for the Open vSwitch control path to be able to both control a pure software implementation or a hardware switch.

The advantage of hardware integration is not only performance within virtualized environments but also helps in managing virtualized hosting environments.

- **Maintenance of Logical Tags**

Multiple methods for specifying and maintaining tagging rules are included, all of which are accessible to a remote process for orchestration. Furthermore, in many cases these tagging rules are stored in an optimized form so they don't have to be paired with a heavyweight network device. This allows, for example, thousands of tagging or address remapping rules to be configured, changed, and migrated.

In many ways, Open vSwitch targets a different point in the design space than previous hypervisor networking stacks, focusing on the need for automated and dynamic network control in large-scale Linux-based virtualization environments.



## 5. METHODOLOGY

---

### 5.1. Architecture/Topology

Network topology is the schematic description of the arrangement of the physical and logical elements of a communication network.

Network topology refers to the manner in which the links and nodes of a network are arranged to relate to each other. Topologies are categorized as either physical network topology, which is the physical signal transmission medium, or logical network topology, which refers to the manner in which data travels through the network between devices, independent of physical connection of the devices.

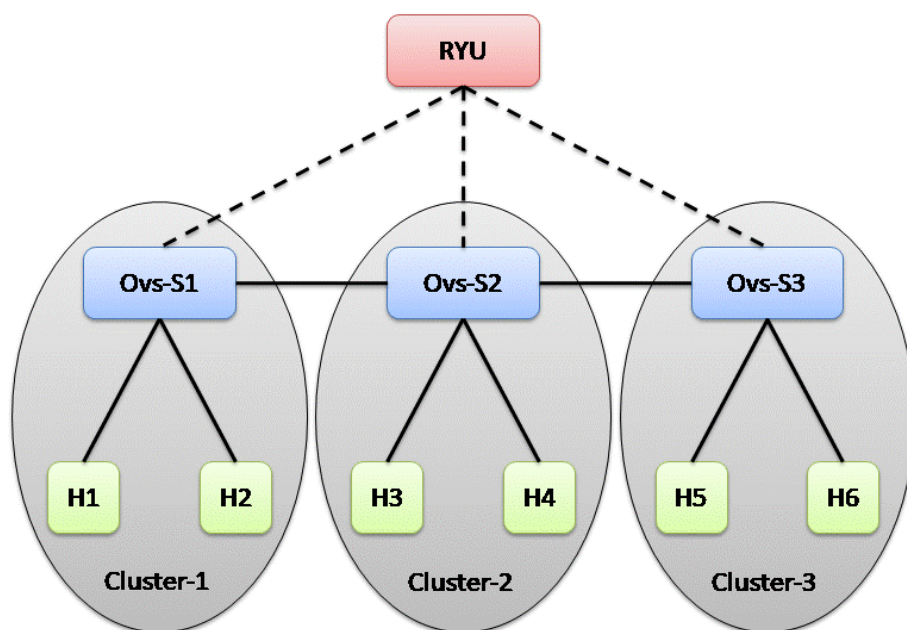


Fig 5.1. Network Topology

```
arup@arup: ~/Desktop/sdn
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s2) (h5, s3) (h6, s3) (s1, s2) (s2, s3)
*** Configuring hosts
h1 h2 h3 h4 h5 h6
*** Starting controller
c1
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> nodes
available nodes are:
c1 h1 h2 h3 h4 h5 h6 s1 s2 s3
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s1-eth2 (OK OK)
h3-eth0<->s2-eth1 (OK OK)
h4-eth0<->s2-eth2 (OK OK)
h5-eth0<->s3-eth1 (OK OK)
h6-eth0<->s3-eth2 (OK OK)
s1-eth3<->s2-eth3 (OK OK)
s2-eth4<->s3-eth3 (OK OK)
mininet> 
```

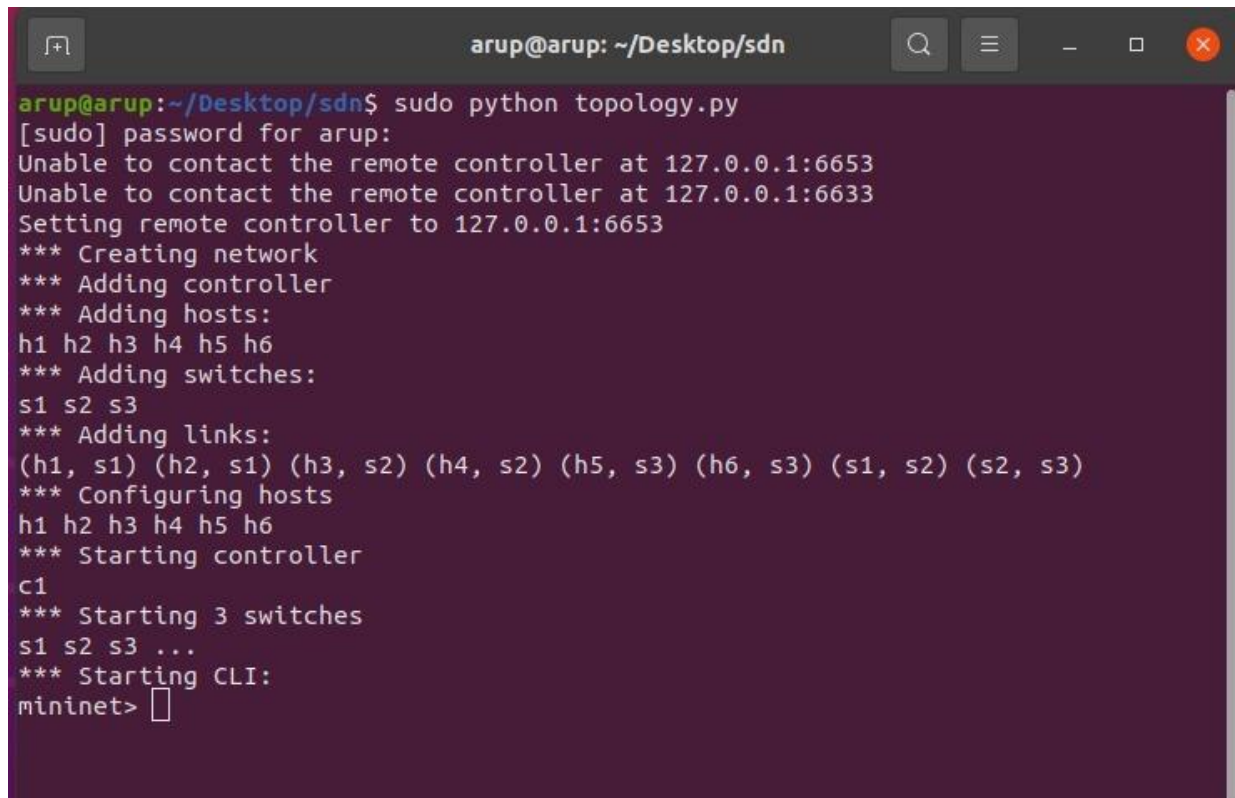
Fig 5.2. Nodes and Links in the Network

Our topology consists three Open vSwitches (OVS) – s1, s2 and s3 each having a pair of stations {h1, h2}, {h3, h4} and {h5, h6} respectively. All the switches are managed and maintained by a single SDN RYU controller.

## 5.2. Implementation

The steps for the implementation are:-

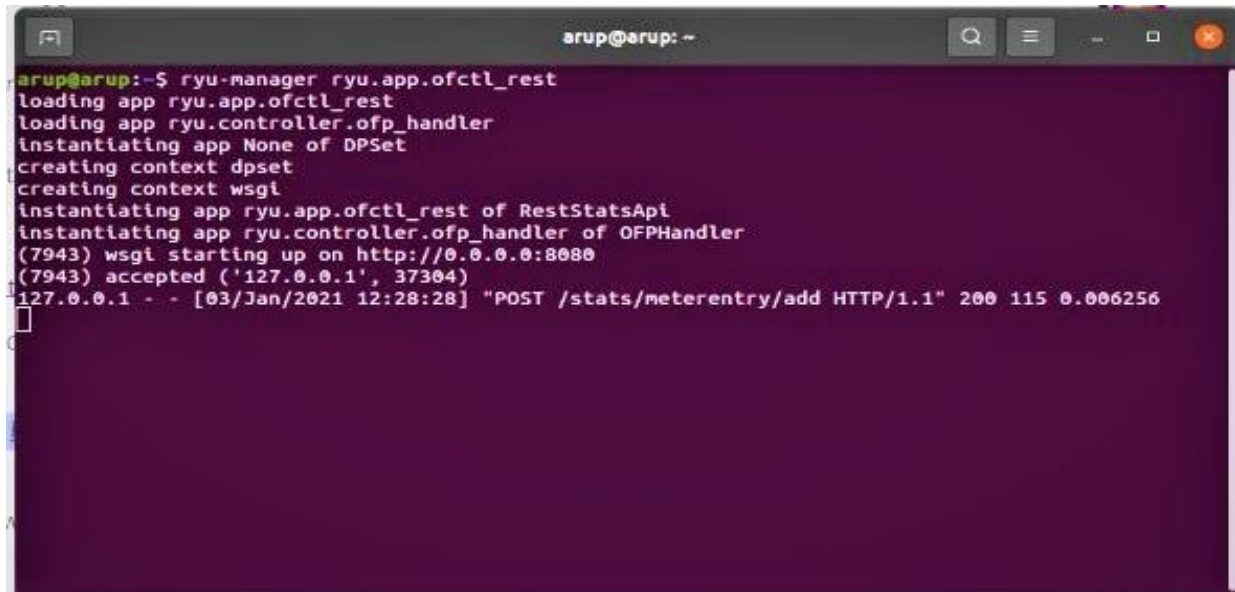
- i. Create the topology in mininet and execute it.



```
arup@arup: ~/Desktop/sdn
arup@arup:~/Desktop/sdn$ sudo python topology.py
[sudo] password for arup:
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s2) (h5, s3) (h6, s3) (s1, s2) (s2, s3)
*** Configuring hosts
h1 h2 h3 h4 h5 h6
*** Starting controller
c1
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> 
```

Fig 5.3. Terminal running the topology


- ii. Run the RYU applications.
  - a. RYU-manager RYU.app.ofctl\_rest

A terminal window titled 'arup@arup: ~' with a dark background and light green text. The window shows the command 'ryu-manager ryu.app.ofctl\_rest' being executed. The output shows the application loading various modules: 'loading app ryu.app.ofctl\_rest', 'loading app ryu.controller.ofp\_handler', 'instantiating app None of DPSet', 'creating context dpset', 'creating context wsgi', 'instantiating app ryu.app.ofctl\_rest of RestStatsApi', and 'instantiating app ryu.controller.ofp\_handler of OFPHandler'. It then shows the WSGI server starting on 'http://0.0.0.0:8080' and accepting a connection from '127.0.0.1' at port 37304. The final line shows an HTTP POST request to '/stats/meterentry/add' with a status of 200 and a response time of 0.006256 seconds.

```
arup@arup:~$ ryu-manager ryu.app.ofctl_rest
loading app ryu.app.ofctl_rest
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app ryu.controller.ofp_handler of OFPHandler
(7943) wsgi starting up on http://0.0.0.0:8080
(7943) accepted ('127.0.0.1', 37304)
127.0.0.1 - - [03/Jan/2021 12:28:28] "POST /stats/meterentry/add HTTP/1.1" 200 115 0.006256
```

Fig 5.4. Running the RYU application in terminal

- iii. Configure and add the meter table.
  - a. Curl -X POST http://localhost:8080/stats/meterentry/add -d '@addmeter.json'

A terminal window titled 'arup@arup: ~/Desktop/sdn/s1' with standard window controls. The command 'cat addmeter.json' has been executed, displaying the following JSON content:

```
arup@arup:~/Desktop/sdn/s1$ cat addmeter.json
{
  "dpId": 1,
  "flags": "KBPS",
  "meter_id": 1,
  "bands": [
    {
      "type": "DROP",
      "rate": 1000
    }
  ]
}
```

Fig 5.5. Adding the merter table

- iv. Confirm a rate limit of 1000 kbps.

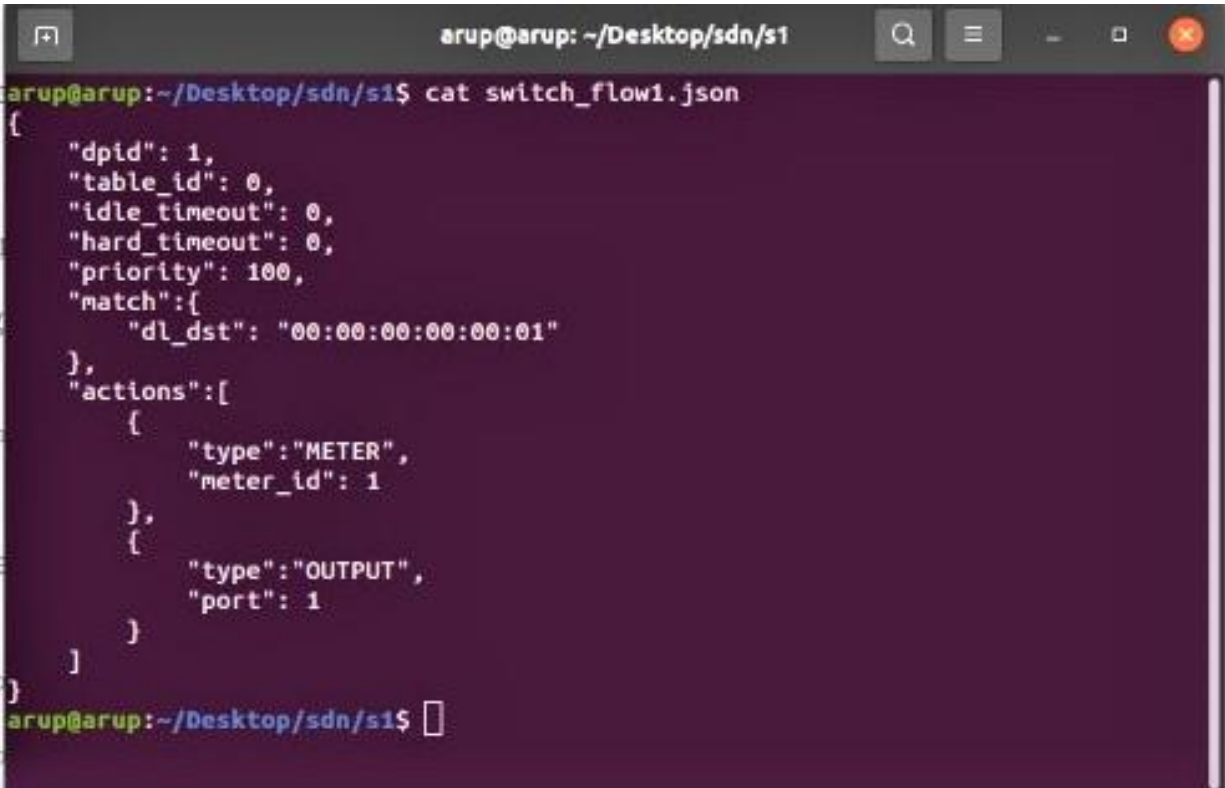
- v. Add the flows for cluster 1 using following commands:-
  - a. `curl -X POST http://localhost:8080/stats/flowentry/add -d '@switch_arp.json'`



```
arup@arup: ~/Desktop/sdn/s1
arup@arup:~/Desktop/sdn/s1$ curl -X POST http://localhost:8080/stats/meterentry/add -d '@addmeter.json'
arup@arup:~/Desktop/sdn/s1$
arup@arup:~/Desktop/sdn/s1$ ls
addmeter.json  switch_arp.json  switch_flow1.json  switch_flow2.json
arup@arup:~/Desktop/sdn/s1$ cat switch_arp.json
{
  "dpid": 1,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 100,
  "match":{
    "dl_dst": "ff:ff:ff:ff:ff:ff"
  },
  "actions":[
    {
      "type":"OUTPUT",
      "port": 4294967291
    }
  ]
}
```

Fig 5.6. Adding switch\_arp to flow entry

- b. `curl -X POST http://localhost:8080/stats/flowentry/add -d '@switch_flow1.json'`

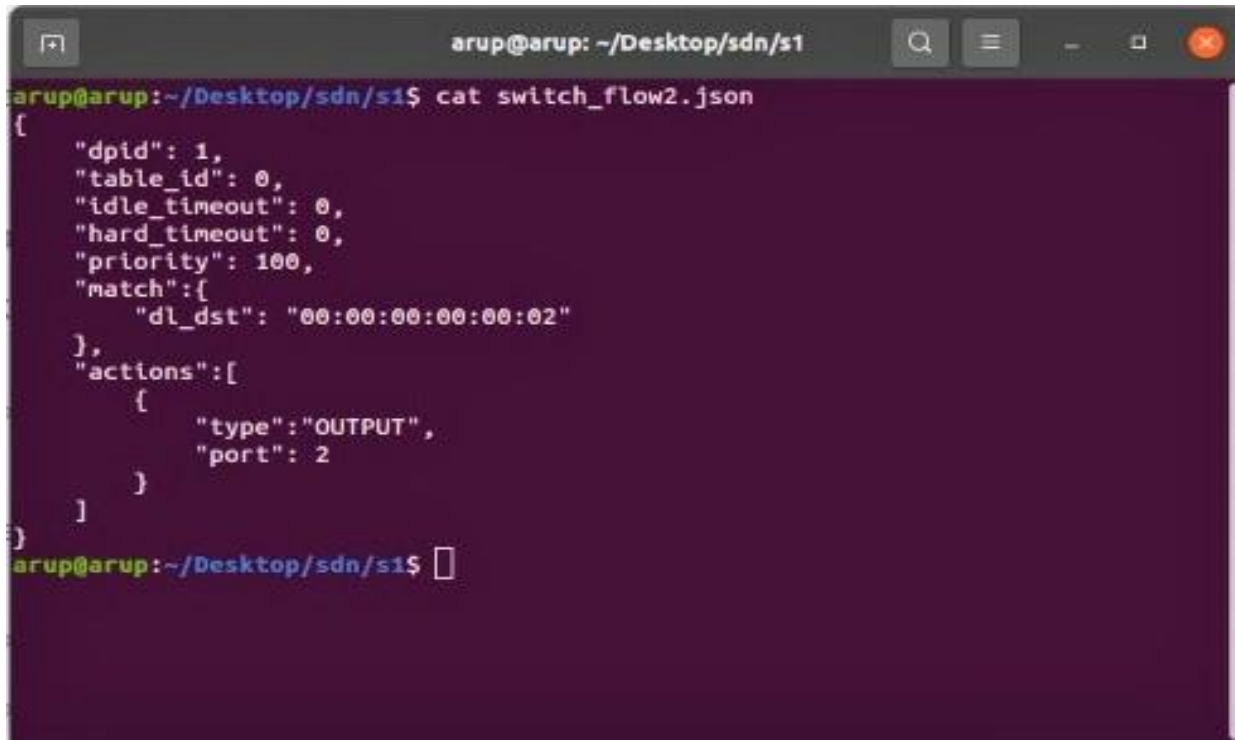
A terminal window titled 'arup@arup: ~/Desktop/sdn/s1' with search, menu, and window control icons. The command 'cat switch\_flow1.json' has been executed, displaying the following JSON content:

```
arup@arup:~/Desktop/sdn/s1$ cat switch_flow1.json
{
  "dpid": 1,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 100,
  "match":{
    "dl_dst": "00:00:00:00:00:01"
  },
  "actions":[
    {
      "type":"METER",
      "meter_id": 1
    },
    {
      "type":"OUTPUT",
      "port": 1
    }
  ]
}
```

```
arup@arup:~/Desktop/sdn/s1$
```

Fig 5.7. Adding switch\_flow1 to flowentry

- c. `curl -X POST http://localhost:8080/stats/flowentry/add -d '@switch_flow2.json'`

A terminal window titled 'arup@arup: ~/Desktop/sdn/s1' with search, menu, and window control icons. The command 'cat switch\_flow2.json' has been executed, displaying the following JSON content:

```
arup@arup:~/Desktop/sdn/s1$ cat switch_flow2.json
{
  "dpid": 1,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 100,
  "match":{
    "dl_dst": "00:00:00:00:00:02"
  },
  "actions":[
    {
      "type":"OUTPUT",
      "port": 2
    }
  ]
}
```

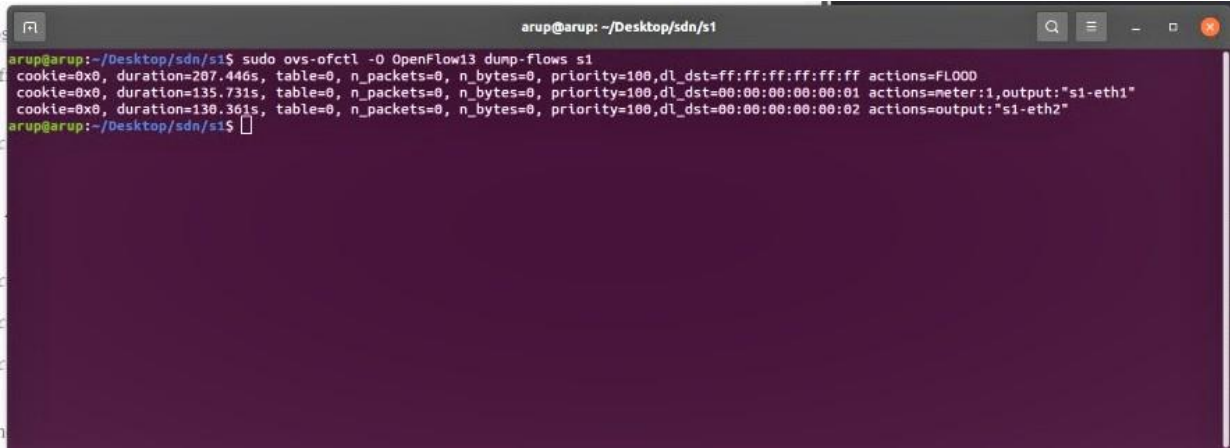
```
arup@arup:~/Desktop/sdn/s1$
```

Fig 5.8. adding switch\_flow2 to flowentry

In the switch\_flow1, we are applying the meter rate limit, it means h1 host is applied with a rate limit. H1 host receives only 1000 kbps traffic.



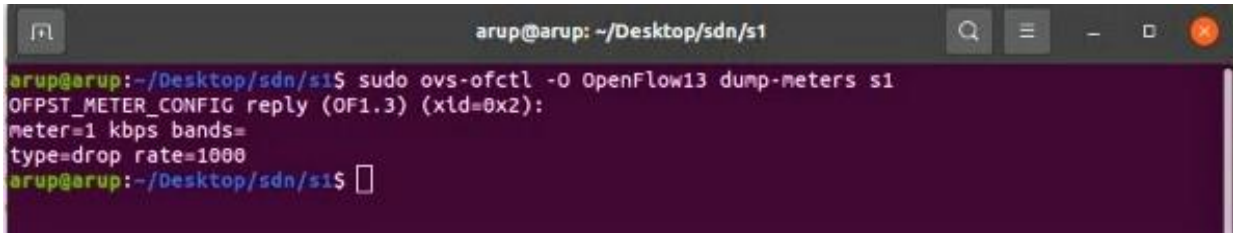
- vi. Next, check the flow table.
  - a. `sudo ovs-ofctl -O OpenFlow13 dump-flows s1`

A terminal window titled 'arup@arup: ~/Desktop/sdn/s1' showing the command 'sudo ovs-ofctl -O OpenFlow13 dump-flows s1' being executed. The output displays two flow entries. The first entry has a cookie of 0x0, a duration of 207.446s, and actions of FLOOD. The second entry has a cookie of 0x0, a duration of 135.731s, and actions of meter:1, output: 's1-eth1'.

```
arup@arup:~/Desktop/sdn/s1$ sudo ovs-ofctl -O OpenFlow13 dump-flows s1
cookie=0x0, duration=207.446s, table=0, n_packets=0, n_bytes=0, priority=100,dl_dst=ff:ff:ff:ff:ff:ff actions=FLOOD
cookie=0x0, duration=135.731s, table=0, n_packets=0, n_bytes=0, priority=100,dl_dst=00:00:00:00:00:01 actions=meter:1,output:"s1-eth1"
cookie=0x0, duration=130.361s, table=0, n_packets=0, n_bytes=0, priority=100,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth2"
arup@arup:~/Desktop/sdn/s1$
```

Fig 5.9. Executing command in terminal to check flow table

- vii. Then, check the meter table.
  - a. `sudo ovs-ofctl -O OpenFlow13 dump-meters s1`

A terminal window titled 'arup@arup: ~/Desktop/sdn/s1' showing the command 'sudo ovs-ofctl -O OpenFlow13 dump-meters s1' being executed. The output shows the meter configuration: meter=1 kbps bands=, type=drop rate=1000.

```
arup@arup:~/Desktop/sdn/s1$ sudo ovs-ofctl -O OpenFlow13 dump-meters s1
OFPST_METER_CONFIG reply (OF1.3) (xid=0x2):
meter=1 kbps bands=
type=drop rate=1000
arup@arup:~/Desktop/sdn/s1$
```

Fig 5.10. Executing command in terminal to check meter table

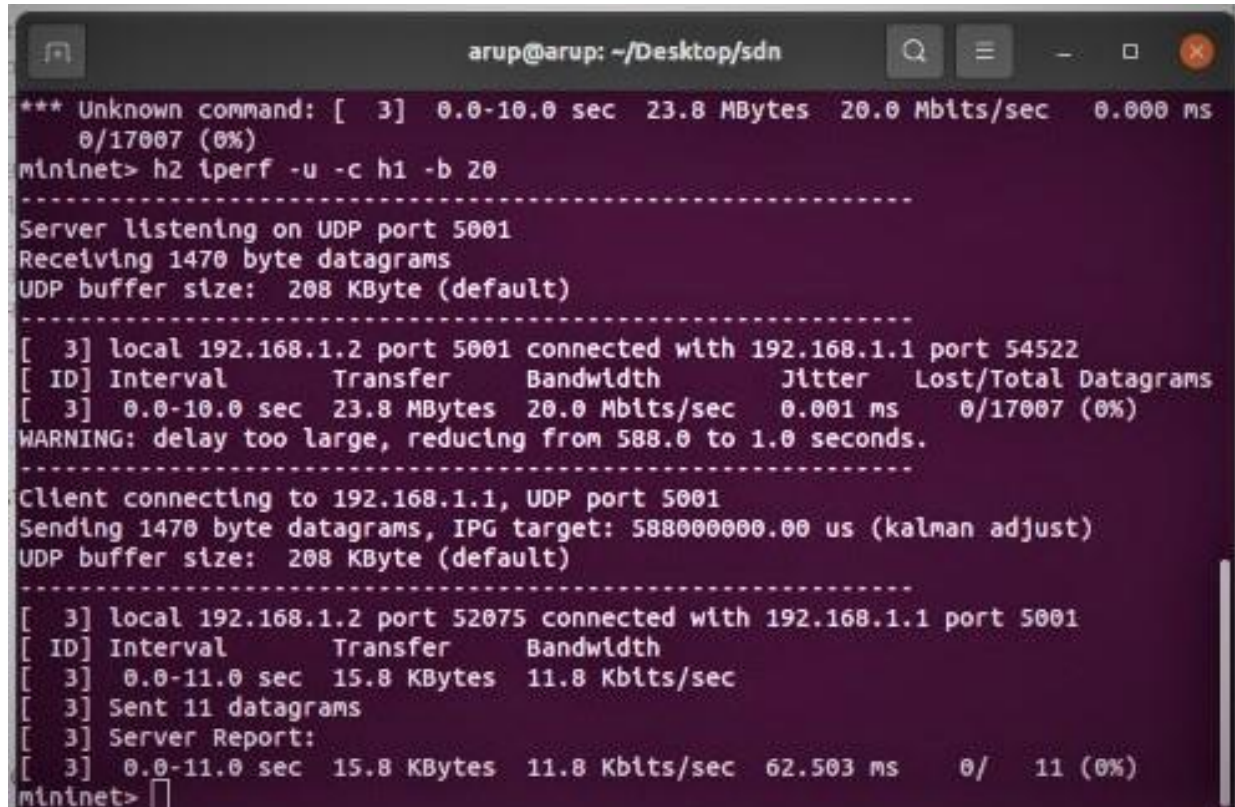
- viii. Start the UDP server in h1 and h3.
  - a. h2 iperf -u -s &
  - b. h1 iperf -u -s &
- ix. Performd the 100 Mbps UDP traffic test for h1 and h2. (meaning h1 pushes 10Mbps traffic and h2 recieves this traffic)
  - a. h1 iperf -u -c h2 -b 20m



```
arup@arup: ~/Desktop/sdn
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> h2 iperf -u -s &
mininet> h1 iperf -u -s &
mininet> h1 iperf -u -c h2 -b 20m
-----
Server listening on UDP port 5001
Recelving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
Client connecting to 192.168.1.2, UDP port 5001
Sending 1470 byte datagrams, IPG target: 588.00 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 3] local 192.168.1.1 port 54522 connected with 192.168.1.2 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3]  0.0-10.0 sec  23.8 MBytes 20.0 Mbits/sec
[ 3] Sent 17007 datagrams
[ 3] Server Report:
[ 3]  0.0-10.0 sec  23.8 MBytes 20.0 Mbits/sec  0.000 ms  0/17007 (0%)
mininet> [ 3]  0.0-10.0 sec  23.8 MBytes 20.0 Mbits/sec  0.000 ms  0/17007
(0%)
```

Fig 5.11. Starting UDP server in h1 & h2 and performing traffic test from h1 to h2

- x. Perform the 10 Mbps UDP traffic test for h2 and h1.
  - a. h2 iperf -u -c h1 -b 20



```
arup@arup: ~/Desktop/sdn
*** Unknown command: [ 3] 0.0-10.0 sec 23.8 MBytes 20.0 Mbits/sec 0.000 ms
0/17007 (0%)
mininet> h2 iperf -u -c h1 -b 20
-----
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 3] local 192.168.1.2 port 5001 connected with 192.168.1.1 port 54522
[ ID] Interval      Transfer    Bandwidth      Jitter    Lost/Total Datagrams
[ 3] 0.0-10.0 sec 23.8 MBytes 20.0 Mbits/sec 0.001 ms    0/17007 (0%)
WARNING: delay too large, reducing from 588.0 to 1.0 seconds.
-----
Client connecting to 192.168.1.1, UDP port 5001
Sending 1470 byte datagrams, IPG target: 588000000.00 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 3] local 192.168.1.2 port 52075 connected with 192.168.1.1 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3] 0.0-11.0 sec 15.8 KBytes 11.8 Kbits/sec
[ 3] Sent 11 datagrams
[ 3] Server Report:
[ 3] 0.0-11.0 sec 15.8 KBytes 11.8 Kbits/sec 62.503 ms    0/ 11 (0%)
mininet>
```

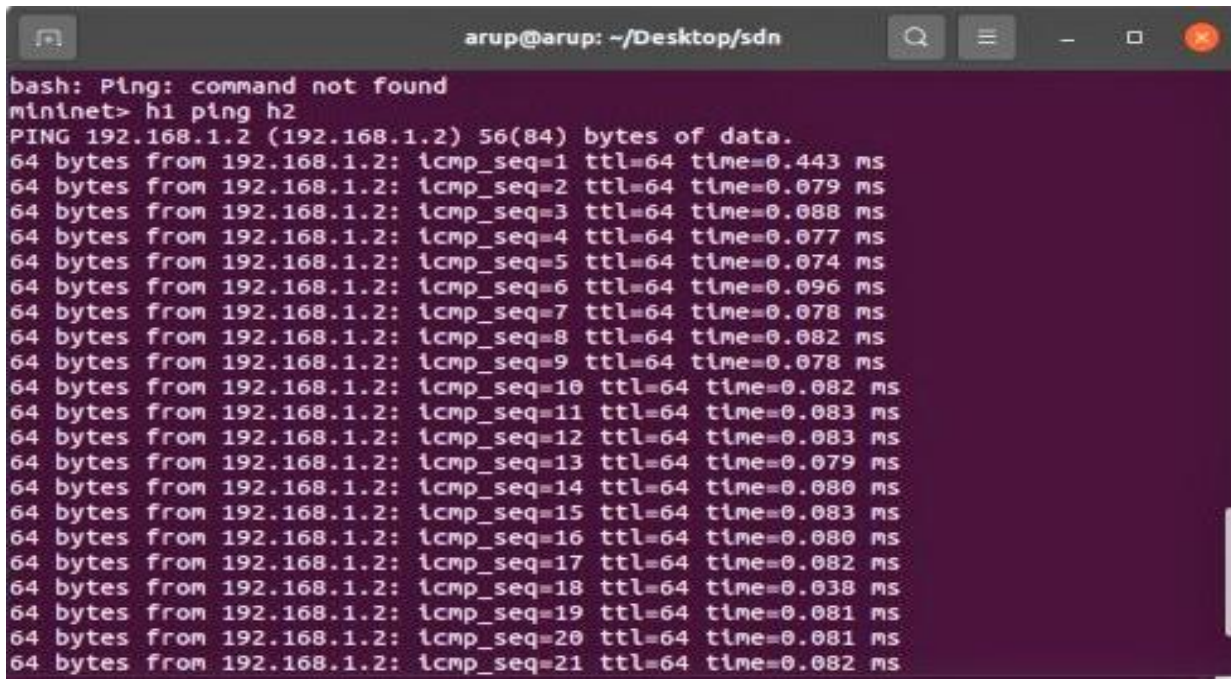
Fig 5.12. Performing traffic test from h2 to h1

- xi. Check the meter table
  - a. `ovs-ofctl -O OpenFlow13 meter-stats s1`
- xii. Ping h2 from h1
- xiii. Perform these same steps for the other switches while applying different priorities



```
arup@arup: ~  
arup@arup:~$ ovs-ofctl -O OpenFlow13 meter-stats s1  
ovs-ofctl: /var/run/openvswitch/s1.mgmt: failed to open socket (Permission denied)  
arup@arup:~$  
arup@arup:~$ sudo ovs-ofctl -O OpenFlow13 meter-stats s1  
[sudo] password for arup:  
OFPST_METER reply (OF1.3) (xid=0x2):  
meter:1 flow_count:1 packet_in_count:17 byte_in_count:19824 duration:710.566s bands:  
0: packet_count:0 byte_count:0  
arup@arup:~$
```

Fig 5.13. Checking meter table



```
arup@arup: ~/Desktop/sdn  
bash: Ping: command not found  
mininet> h1 ping h2  
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data:  
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.443 ms  
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.079 ms  
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.088 ms  
64 bytes from 192.168.1.2: icmp_seq=4 ttl=64 time=0.077 ms  
64 bytes from 192.168.1.2: icmp_seq=5 ttl=64 time=0.074 ms  
64 bytes from 192.168.1.2: icmp_seq=6 ttl=64 time=0.096 ms  
64 bytes from 192.168.1.2: icmp_seq=7 ttl=64 time=0.078 ms  
64 bytes from 192.168.1.2: icmp_seq=8 ttl=64 time=0.082 ms  
64 bytes from 192.168.1.2: icmp_seq=9 ttl=64 time=0.078 ms  
64 bytes from 192.168.1.2: icmp_seq=10 ttl=64 time=0.082 ms  
64 bytes from 192.168.1.2: icmp_seq=11 ttl=64 time=0.083 ms  
64 bytes from 192.168.1.2: icmp_seq=12 ttl=64 time=0.083 ms  
64 bytes from 192.168.1.2: icmp_seq=13 ttl=64 time=0.079 ms  
64 bytes from 192.168.1.2: icmp_seq=14 ttl=64 time=0.080 ms  
64 bytes from 192.168.1.2: icmp_seq=15 ttl=64 time=0.083 ms  
64 bytes from 192.168.1.2: icmp_seq=16 ttl=64 time=0.080 ms  
64 bytes from 192.168.1.2: icmp_seq=17 ttl=64 time=0.082 ms  
64 bytes from 192.168.1.2: icmp_seq=18 ttl=64 time=0.038 ms  
64 bytes from 192.168.1.2: icmp_seq=19 ttl=64 time=0.081 ms  
64 bytes from 192.168.1.2: icmp_seq=20 ttl=64 time=0.081 ms  
64 bytes from 192.168.1.2: icmp_seq=21 ttl=64 time=0.082 ms
```

Fig 5.14. h1 ping h2

## 6. CONCLUSION

---

Through this project we have gained a lot of insight in the workings of SDN and how it can be used to improve routing and quality of service for networks. It explains how an SDN controller (RYU) can be used in conjunction with an Open vSwitch to perform quality of service prioritize policy. As we can see meter table can be used to set multiple priority to traffic and then priority queuing can be performed on those traffic. We gained insight on meter bands and on their working. Due to limited amount of time and limited knowledge at the beginning, not all planned functionalities were able to be incorporated in this project. So, this project can be further improved by adding these functionalities as a scope for future work.

### 6.1. Future Work Possible

- Implementing this design in IoT networks while keeping in mind the QoS criteria required for IoT networks.
- The feature of smart traffic control can be added for better routing as the topology of the IoT network evolves and grows.
- IoT networks are developing at a very fast rate and standard routing protocols may have a hard time keeping up with this growth. So, trying to make the SDN based routing application aware would help it cope up with the growth of IoT networks.
- Heterogeneity aware clustering algorithms can be used that take advantage of the presence of node heterogeneity in terms of energy in order to prolong the network lifetime. This aims to balance load in the network and reduce energy consumption of sensor nodes



## 7. BIBLIOGRAPHY

---

- [1] Ayesha Shafique , Guo Cao, Muhammad Aslam, Muhammad Asad and Dengpan Ye, “Application-Aware SDN-Based Iterative Reconfigurable Routing Protocol for Internet of Things (IoT),” 22 June 2020
- [2] Carynthia Kharkongor, T. Chithralekha and Reena Varghese, “A SDN Controller with Energy Efficient Routing in the Internet of Things (IoT),” 2016
- [3] KNET Solutions (Online Training Centre), “RYU SDN Crash Course - Book,” <https://learning.knetsolutions.in/docs/RYU/>
- [4] Open Networking Foundation, “OpenFlow Switch Specifications,” Version 1.3.0, 25 June 2020

## APPENDIX: Code

---

### **topology.py**

```
from mininet.topo import Topo

from mininet.net import Mininet

from mininet.log import setLogLevel

from mininet.cli import CLI

from mininet.node import OVSSwitch, Controller, RemoteController, OVSKernelSwitch

from time import sleep
```

```
class SingleSwitchTopo(Topo):

    "Single switch connected to n hosts."

    def build(self):

        s1 = self.addSwitch('s1', protocols='OpenFlow13', cls=OVSKernelSwitch)
        s2 = self.addSwitch('s2', protocols='OpenFlow13', cls=OVSKernelSwitch)
        s3 = self.addSwitch('s3', protocols='OpenFlow13', cls=OVSKernelSwitch)
        h1 = self.addHost('h1', mac="00:00:00:00:00:01", ip="192.168.1.1/24")
        h2 = self.addHost('h2', mac="00:00:00:00:00:02", ip="192.168.1.2/24")
        h3 = self.addHost('h3', mac="00:00:00:00:00:03", ip="192.168.1.3/24")
        h4 = self.addHost('h4', mac="00:00:00:00:00:04", ip="192.168.1.4/24")
        h5 = self.addHost('h5', mac="00:00:00:00:00:05", ip="192.168.1.5/24")
        h6 = self.addHost('h6', mac="00:00:00:00:00:06", ip="192.168.1.6/24")

        #addlink(hostname,switchname,hostport,switchport)
```

```
self.addLink(h1, s1)
self.addLink(h2, s1)
self.addLink(h3, s2)
self.addLink(h4, s2)
self.addLink(h5, s3)
self.addLink(h6, s3)
self.addLink(s1, s2)
self.addLink(s2, s3)
```

```
if __name__ == '__main__':
    setLogLevel('info')
    topo = SingleSwitchTopo()
    c1 = RemoteController('c1', ip='127.0.0.1')
    net = Mininet(topo=topo, controller=c1)
    net.start()
    net.get('s1').start([c1])
    net.get('s2').start([c1])
    net.get('s3').start([c1])
    CLI(net)
    net.stop()
```



### **addmeter.json (for switch 1)**

```
{  
  "dpid": 1,  
  "flags": "KBPS",  
  "meter_id": 1,  
  "bands": [  
    {  
      "type": "DROP",  
      "rate": 1000  
    }  
  ]  
}
```

### **switch\_arp.json (for switch 1)**

```
{
  "dpid": 1,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 100,
  "match": {
    "dl_dst": "ff:ff:ff:ff:ff:ff"
  },
  "actions": [
    {
      "type": "OUTPUT",
      "port": 4294967291
    }
  ]
}
```

## switch\_flow1.json

```
{
  "dpid": 1,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 100,
  "match": {
    "dl_dst": "00:00:00:00:00:01"
  },
  "actions": [
    {
      "type": "METER",
      "meter_id": 1
    },
    {
      "type": "OUTPUT",
      "port": 1
    }
  ]
}
```

## switch\_flow2.json

```
{
  "dpid": 1,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 100,
  "match": {
    "dl_dst": "00:00:00:00:00:02"
  },
  "actions": [
    {
      "type": "OUTPUT",
      "port": 2
    }
  ]
}
```

### **addmeter.json (for switch 2)**

```
{  
  "dpid": 2,  
  "flags": "KBPS",  
  "meter_id": 2,  
  "bands": [  
    {  
      "type": "DROP",  
      "rate": 1000  
    }  
  ]  
}
```

### **switch\_arp.json (for switch 2)**

```
{
  "dpid": 2,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 50,
  "match": {
    "dl_dst": "ff:ff:ff:ff:ff:ff"
  },
  "actions": [
    {
      "type": "OUTPUT",
      "port": 4294967291
    }
  ]
}
```

### switch\_flow3.json

```
{
  "dpid": 2,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 50,
  "match": {
    "dl_dst": "00:00:00:00:00:03"
  },
  "actions": [
    {
      "type": "METER",
      "meter_id": 2
    },
    {
      "type": "OUTPUT",
      "port": 3
    }
  ]
}
```

## switch\_flow4.json

```
{
  "dpid": 2,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 50,
  "match": {
    "dl_dst": "00:00:00:00:00:04"
  },
  "actions": [
    {
      "type": "OUTPUT",
      "port": 4
    }
  ]
}
```



### **addmeter.json (for switch 3)**

```
{  
  "dpid": 3,  
  "flags": "KBPS",  
  "meter_id": 3,  
  "bands": [  
    {  
      "type": "DROP",  
      "rate": 1000  
    }  
  ]  
}
```

### **switch\_arp.json (for switch 3)**

```
{  
  "dpid": 3,  
  "table_id": 0,  
  "idle_timeout": 0,  
  "hard_timeout": 0,  
  "priority": 25,  
  "match": {  
    "dl_dst": "ff:ff:ff:ff:ff:ff"  
  },  
  "actions": [  
    {  
      "type": "OUTPUT",  
      "port": 4294967291  
    }  
  ]  
}
```

## switch\_flow5.json

```
{
  "dpid": 3,
  "table_id": 0,
  "idle_timeout": 0,
  "hard_timeout": 0,
  "priority": 25,
  "match": {
    "dl_dst": "00:00:00:00:00:05"
  },
  "actions": [
    {
      "type": "METER",
      "meter_id": 3
    },
    {
      "type": "OUTPUT",
      "port": 5
    }
  ]
}
```

## switch\_flow6.json

```
{  
  "dpid": 3,  
  "table_id": 0,  
  "idle_timeout": 0,  
  "hard_timeout": 0,  
  "priority": 25,  
  "match": {  
    "dl_dst": "00:00:00:00:00:06"  
  },  
  "actions": [  
    {  
      "type": "OUTPUT",  
      "port": 6  
    }  
  ]  
}
```