Dynamic Bandwidth Allocation in an SDN Environment Using QoS Queues and OpenFlow

Dr. Radhika N*, Abhilash V†, Niranjan Kumar K S‡, Phanindra Meduri§ and Nikhil Chandra Sai Ram¶
Dept of Computer Science & Engineering,
Amrita School Of Engineering, Coimbatore
Amrita Vishwa Vidyapeetham
India
Email: *n_radhika@cb.amrita.edu, †abhilash.venky@gmail.com, ‡niranjankumar26081998@gmail.com,
§phanindrameduri37@gmail.com, ¶nikhilchandra432@gmail.com

***Abstract*—A Software Defined Network can be used to separate the data and control planes of networking components and to create a centralized control plane, called the SDN Controller which acts as the brain of the network. The controller governs the actions of the individual components of the network. It uses two types of API to communicate with the networking components, namely, northbound and southbound APIs. A southbound APIl is used by the controller to send configurations and flow-rules to the connected networking components. By taking advantage of these flow-rules, a network can be made more dynamic and software controlled. In the current networking scenario, users gain access to the internet via an ISP who allocates a fixed bandwidth to the user regardless of the bandwidth usage by the user. This is disadvantageous to both end user and the ISP as bandwidth is not conserved during idle times. The intent of this research paper is to discuss the steps in building a network application working in an SDN environment which allows users to request the controller for a fixed amount of bandwidth for a specified amount of time to any other host that is connected to the topology. The network application is built on top of an SDN controller, and works by configuring QoS queues and flow-rules onto connected networking components with help of the southbound API OpenFlow. The configured QoS queues limit bandwidth to minimum & maximum values, and the flow-rules en-queue traffic from different hosts onto the created QoS queues. The client side of the network application may run on any node in the topology and is designed to send bandwidth requests to the controller. The controller responds to the client-application with response codes representing the success/failure of bandwidth allocation. We start by presenting the fundamentals of SDNs, followed by implementation of the network application in a scalable network topology deployed as an SDN.**

I. Introduction

Software-defined networking (SDN) a type of networking architecture that makes networking agile and flexible. The main aim of SDN is to make a computer network open and programmable.[1] SDN gives administrators the ability to manage and provision network services from a centralized location. An organization may control the network behavior for specific types of applications they run by using SDN. Say, traffic engineering and security find good applications in SDN. The primary advantage while using SDN is, computer networks can evolve at the speed of growth of the software. Using traditional networking, configuration of network is difficult to automate, hence application based network configuration is a challenge. In a world of automation and evolving technology, traditional networking which involves manual configuration of networking hardware, struggles to keep up with dynamic networking requirements of modern software applications. For the same reason, technological companies are starting the use of SDNs rather than traditional networks. The reason why SDNs are dynamic and hence powerful is because to the fact that in SDNs, the control plane and the data plane are disjoint in nature and are programmable whereas in traditional networks, they are combined together and are non-programmable. Consequently, in SDNs, programs can be written which can take advantage of the programmable control plane and data plane in networking devices, so as to configure them dynamically as per networking requirements of users/applications[2].

The field of SDN is very powerful and can be used to control network parameters such as bandwidth of hosts based on data using queues in networking devices. Bandwidth optimization is a key parameter to be taken care of in order to increase the performance of the network and hence is the reason of choice of our research.A bandwidth control method implemented in SDN includes obtaining data for one or more services in the network, wherein each of the one or more services is controlled by an associated user-agent[3]. With this as a central idea in mind, we set course to develop a network application which is aimed at optimizing bandwidth usage of a host in a software defined network aimed at allocating exactly the amount of bandwidth needed for a host and not more. One of the methods of achieving this is via user requests for bandwidth allocation which we demonstrate in this paper. We wish to extend this research into creating a network application for automatic bandwidth allocation for multiple applications on the same host, which ultimately would benefit the user and ISP at the same time.

II. FUNDAMENTALS OF SDN

A. SDN — Working & Architecture

A software-defined network attempts to build a computer network by separating it into two segments: control plane and data plane.[4] The control plane is the part of a network that carries signaling traffic and is responsible for routing.

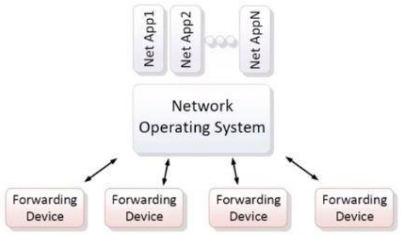

Fig. 1. SDN Architecture.

Control packets originate from or are destined for a router. Functions of the control plane include system configuration and management. This plane is generally used to manage configurations of devices connected to the SDN on a remote basis. The second segment is the data plane which is responsible for forwarding traffic to its final destination. The control plane dictates which path flows will take before they reach the data plane. This is done through the use of a flow protocol. This segment is where an administrator interacts with the SDN and actually manages the network. There is a third plane called the Management plane. The management plane of a networking device is the element of a system that configures, monitors, and provides management, monitoring and configuration services to all layers of the network stack and other parts of the system. In traditional networking, all three planes are implemented as a single component within the operating system/firmware of routers and switches. SDN decouples the data and control planes, and provides a software implementation of the control plane rather than hardware, which enables programmatic access and, as a result, makes network configuration and administration much more flexible.

The SDN Architecture centralizes the control plane which acts as the brain for the network. The centralized control plane, referred to as the SDN Controller, is represented as the Network Operating System in Fig. I.

B. Components of SDN Architecture

1) Forwarding devices: Forwarding devices are the networking components/devices that are connected to the SDN controller. These devices can be hardware switches with programmable interfaces or software switches. Since their interface is programmable and are connected to the SDN controller, these devices operate based on the instructions received from the SDN controller.[1] Examples of operation of forwarding device include : the forwarding device using layer 2 addresses to forward packets (switch), the forwarding device using layer 3 addresses to route packets (router), the forwarding device having the functionality to use both layer 2 and layer 3 addresses to process and forward packets (a multi-layer switch). We demonstrate the working of an OpenFlow switch with the help of Fig. 2

The Open Networking Foundation defines that an OpenFlow switch is one that contains one or more flow tables each of which hold flow entries.[1] It is notable that there must

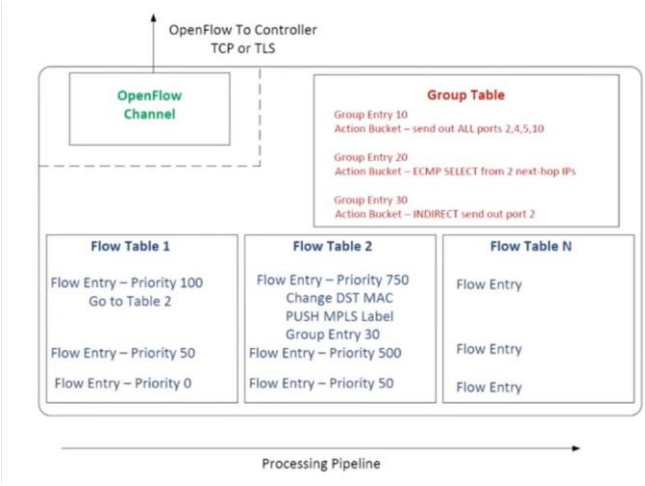

Fig. 2. Inside an OpenFlow Switch.

be a minimum of one flow table to hold flow entries in an OpenFlow switch. Flow entries in a flow table have certain match criterion to match with incoming packets and they also have instructions as to what to do with the matching packets, such as change source and destination MAC addresses, or go to a later flow table to match another flow entry. Flow entries have priorities assigned to them which will be useful if a packet matches multiple flow entries, in which case the flow entry with the highest priority will match the packet. There is also a Group Table with group entries. Group tables have action buckets in them with more actions to perform on matching. Multiple flow entries can map to a single group table entry, and changing the behaviour of this group entry essentially changes the action of multiple flow entries. There is a connection from the OpenFlow switch to the controller through the OpenFlow Channel. Communications such as flow addition/removal, status information, health checks and other OpenFlow protocol messages are done through the OpenFlow channel. The underlying protocol maybe TCP/TLS.

2) Southbound Interface : A southbound interface allows the SDN Controller to communicate with the networking devices connected to it. In SDN, the southbound interface is the OpenFlow (or alternative) protocol specification. Its main function is to enable communication between the SDN controller and the nodes in the network (both physical and virtual switches and routers) so that the controller can discover network topology, define network flows and implement requests relayed to it via northbound APIs. A few southbound interfaces have been listed in Fig. 4. We demonstrate further, the working of OpenFlow protocol used in the development of our network application.

The OpenFlow protocol is a standardized protocol for interacting with the forwarding behaviours of switches from various vendors. This provides us a way to control switch behaviour dynamically using code. OpenFlow can be used as the Southbound interface for any SDN controller.

Assume the topology shown in Fig. 3, where host h4 is

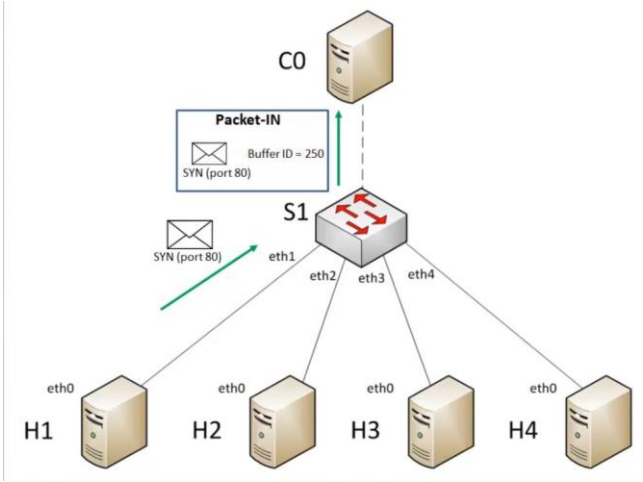

Fig. 3. A flow-table miss for a packet

running a web server, and h1 sends a HTTP GET request to the host h4. Ignoring ARP queries, when the http syn reaches OpenFlow enabled switch S1, since S1 initially has an empty flow table, a table-miss occurs. A table-miss causes S1 to forward the incoming packet to the SDN controller C0. Although, it does not necessarily send the entire packet to controller S0, rather, it buffers the packet data at an address and sends the address of the same along with the headers required for path calculation to the SDN controller as a separate packet called PACKET-IN. Fig. 3 depicts a table-miss. The controller might respond with either a PACKET-OUT packet, which instructs the switch what to do with the packet, or it may send a Flow-MOD packet, which is basically a packet that contains information for a new flow-entry to be installed in the flow-table of the switch. The packet-out message contains details such as match/mask, buffer ID, idle timeout, hard timeout, action and priority.

The reason for choosing OpenFlow as the southbound API is, Open Flow-based SDN architecture is the best at abstracting the underlying infrastructure from the applications that use it by decoupling the network control and data planes permitting the network to become programmable and manageable. OpenFlow switching is already being incorporated into a number of infrastructure designs, both physical and virtual, as well as SDN controller software. OpenFlow design supports policy-based flow management within a network making it particularly well suited to use cases satisfied by pushing predefined policies to implement network segmentation[7].

3) Application Interfaces/ Northbound Interface: SDN Controller provides abstraction of the underlying network by providing an API interface for the users who need information about the network or want to control the network. Network applications can take advantage of this northbound API provided by the SDN controller to query information about the network and control network parameters dynamically as and when the user needs to do so. For example, in an enterprise data center, functions of northbound APIs may include management

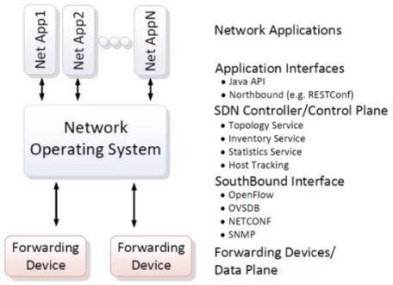

Fig. 4. Network Applications in a Software Defined Network.

solutions for automation and orchestration, and the sharing of actionable data between systems.[1]

4) Applications: Network applications in an SDN use the northbound APIs provided by the SDN Controller to deliver dynamic networking capabilities to the end user. There may be many network applications running in parallel that request the SDN controller for network resources using the northbound API.[1] An example of an application may be a network application that gives a complete map of the network by querying the SDN controller using the northbound API, which would in-turn use a southbound API such as OpenFlow to query the underlying network. Fig. 3. gives a reference as to how a network application manipulates the infrastructure of the network.

C. SDN vs. Traditional Networking

In a traditional network, every device has a control plane and a data plane within the device. The data plane is populated by the control plane, which is usually called the brain of the network. Control planes of various network devices talk to each other using distributed protocols, and push the data to their respective data planes. Since both control plane and data plane are coupled within the device, there is no way to make changes to the data plane without manipulating the control plane. This manipulation is done usually through a command line interface to the network device. This brings us to the conclusion that, in order to configure a traditional network, we need to configure the control plane of each networking device in the network, and this configuration could be limited to the feature set offered by the manufacturer of the networking hardware (custom commands/services provided by the manufacturer). In contrast, in an SDN network, there is a logically centralized control plane, with a global view of the whole network. Hence, the data planes of all the network nodes that come under this control plane can be modified centrally.[1] However, it's worthwhile to note that, the SDN Controller itself may be architecturally complex, as it deals with centralizing control planes of many nodes.

III. A NETWORK APPLICATION FOR BANDWIDTH ALLOCATION

The objective of writing this paper is to build a comprehensive bandwidth allocation program which can guarantee

a specified amount of bandwidth between two hosts for a specified amount of time. The program is built in an SDN environment wherein the controller acts as the central brain running whose tasks include configuration of networking components under the chosen custom topology, offering a northbound API for hosts that require bandwidth allocation, running the bandwidth allocation algorithm, configuration of QoS flow entries on to the networking components and returning response codes to clients. The end hosts of the topology act as clients requesting bandwidth from the network to a destination client within the topology. The tasks of the client include creating a bandwidth request to destination for a specific amount of time, sending the bandwidth request to controller using the northbound API, receiving and processing response codes from the controller.

Building this system for bandwidth allocation is done in three parts : the first is to build a scalable topology for the implementation of the network application, the second being, developing the code for the SDN controller, and the third being developing the client application designed to access the northbound API written over the SDN Controller. For testing purposes, entire network is simulated using software network simulators. We have used MININET network simulator to create a network topology over which our applications can run.

A. *Building a Custom Topology in a Network Emulator*

1) MININET Network Emulator: MININET is a network simulator which can be used with Windows and Linux. It allows one to launch a virtual network with switches, hosts and an SDN Controller. It can be used for SDN Controller testing and applications. In order to run. MININET supports research, development, learning, prototyping, testing, debugging, and any other tasks that could benefit from having a complete experimental network on a laptop or other PC. MININET in Windows operating system, a virtual box such as VMware is used. It is notable that MININET provides a simple and inexpensive network test bed for developing OpenFlow applications, supports arbitrary custom topologies and also provides a straightforward and extensible Python API for network creation and experimentation.

2) Custom Topology: The network topology over which we have developed the network application is highly scalable. The topology consists of three programmable switches, and any number of host systems (user choice), and an SDN controller controller connected to the network. The python API provided by MININET for network creation is used to form a virtual network whose topology is shown in Fig. 5. Although the image depicts a router S1 between switches S2 and S3, when it comes to SDN, each components function/role in the network is determined by the SDN controller. In the case of our application, the controller has been programmed to configure S1 with flows that make it a router, and S2 and S3 with flows that make them switches. There can be any number of hosts connected to switch S2 (limited by ports on S2) and the same

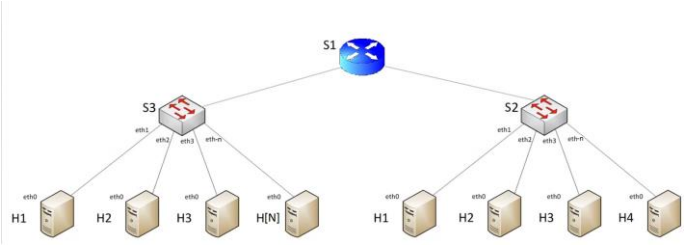

Fig. 5. Custom topology created inside MININET network emulator.

applies for switch S3. Switches S2 and S3 themselves are connected to S1 acting as the router between the two networks.

B. *Programming the SDN Controller*

Programming the SDN Controller itself has various parts to it. Each part of the SDN controller contributes to running the network application in part. The various parts of programming the SDN controller include identification and classification of routers and switches within topology, defining switch operations, defining router operations, programming the bandwidth allocation algorithm and lastly, requesting and responding to bandwidth allocation requests. On completion of the above mentioned steps, our network application would be complete Programming the SDN Controller is the most important part of developing this network application. There are various tools to program SDN OpenFlow Controllers, namely, Floodlight, Ryu, POX, NOX, etc. Each of these SDN Controllers support different versions of OpenFlow, are written in different programming languages and have differences in performance. The SDN Controller of choice for the network application we have developed is POX based on its packet handling capacity, by varying the packet size, number of packets and arrival pattern in the IP traffic flows[6].

POX provides a framework for communicating with SDN switches using either the OpenFlow or OVSDB protocol. Developers can use POX to create an SDN controller using the Python programming language. It is a popular tool for teaching about and researching software defined networks and network applications programming[8].

We have chosen POX as the ideal SDN Controller for our application as it offers simple control using the Python programming language to send flow modifications to the network components in the underlying network. This metric was chosen because, reconfiguration of bandwidth involves repeated addition/removal of QoS flow rules onto networking equipment which involves programming flow modification rules in several places in the implementation of the code. POX SDN Controller offers a relatively simple functions to achieve the task mentioned above.[8]

1) Identification & Classification of Switches & Routers: Every networking component that connects to the controller has a *connection dpid* and a port number in which the connection was received. These two parameters are used by the SDN Controller to identify the networking components

as either switches or routers. As soon as the identification is successfully done, each of the identified programmable switch is configured with the functions of either a switch or a router.

2) Switch Operations: The network components which need to be assigned the functionality of a switch essentially must have the abilities to work in Layer 2 of the OSI model. In the sample topology used to implement our network application consists of two networking components that function as switches. All of the Layer 2 functions required by a switch are implemented inside a python class called Switch. Within this class, a function called act like switch contains the code for the fundamental operation performed by switches, which involve the manipulation of a python dictionary called mac to port, which acts as the forwarding table of the switch. This function works by forwarding packets out to a single port when destination MAC is identified and associated with certain port of the switch, forwarding packets out multiple ports when destination MAC address is not associated with any port of the switch, have L2 learning ability (learn to associate MAC addresses to switch ports). Another important operation built into the code of the Switch class is push flow label function which is key to making this system efficient. Whenever a decision regarding the forwarding of a packet out the port of a certain switch in the network is made, the controller not only instructs the switch in question to forward the packet out the corresponding port, but also pushes a flow label with an arbitrary timeout which matches incoming packets having destination port of the currently matched packet with the action of the flow instructing the switch to output the packet to the current output port. By this, as soon as the SDN Controller associates a MAC address to an output port, the switch will have to ask the SDN controller for packet specific actions related to this destination MAC address only once, after which the flow label pushed by the SDN controller to the switch will be used to identify the output port for forwarding within the switch itself. This makes the forwarding process highly efficient for the switch as the switch directly obtains the output port from the flow entry as soon as it reads the destination MAC address from the packet.

3) Router Operations: The network components which are involved in the reading and manipulation of layer 3 header are assigned the functions of a router. In the sample topology used to implement our network application, there is a single networking component that acts a router having two interfaces connecting two networks. The functions that must be performed by the router for the successful functioning of the topology are implemented within a python class called Router.The router maintains several data structures which are involved in the routing process, namely, routing table, ARP table, packet buffer table and mac_ to_ port table. Similar to the implementation of the switch, the router too has a act_ like _router function which contains the code for the fundamental layer 3 operations performed by a router such as routing between networks using the internal data structures of the class. The router too is programmed to send flow labels which have packet matching conditions involving source host and destination host with actions to take, to the associated networking components.This makes the router function efficiently as the router queries the SDN controller for actions to take on an incoming packet from a new host at the maximum of one time, after which it successfully receives a flow label which takes care of the routing actions until its timeout expires. It is also worthwhile to note that, the interfaces of the router act as the default gateway of the corresponding networks. The router uses these interfaces to send ARP Requests and handles ARP queries from connected hosts by using functions implemented within the Router class.

4) Bandwidth Allocation Algorithm: The network application we build runs within the Router class implemented in the SDN controller code. This is because, bandwidth allocation eventually involves pushing QoS flow rules to the router, and the router will be the central key networking component which is responsible for effecting a custom bandwidth on its links. The SDN controller sends default flows to the connected router to redirect all packets sent with the destination IP address as the router's interface IP address and with destination port number sent to 9999 over the transport layer protocol UDP. By doing so, the router redirects all such matching packets to the SDN controller without any further processing of the packet by the router. The SDN controller hence opens UDP port 80 at both of the interfaces, to which client applications may send bandwidth allocation requests. On receiving the bandwidth allocation requests, the controller sends this packet to a function called *handleBandwidthRequest*, which takes the entire packet as its argument. The tasks performed by this function include parsing the request packet, checking validity of request including source/destination IP validation, allocation of bandwidth which is done by pushing of QoS flow rules to the router and sending a UDP response to the requesting host regarding the status of request. takes actions on the underlying network and eventually sends a response code to the requesting client. The tasks performed by the bandwidth allocation algorithm are elaborated further below :

Step 1 - Parsing the bandwidth request : The controller expects a python dictionary as the payload of the packet given as input to the bandwidth allocation function. The function parses the packet by initially extracting it's UDP payload which is a python dictionary. From this dictionary it extracts the requestID, source IP, destination IP, requested bandwidth and timeout.

Step 2 - Validating request parameters : The controller checks whether the source and destination IP addresses are valid, in which case it proceeds to the next step, else, it sends the corresponding error code as a response to the source IP address. The source checks if the requested bandwidth is available by comparing requested bandwidth to the remaining available bandwidth on the corresponding links. The algorithm proceeds further in case the requested bandwidth is available, else it sends the corresponding error code to the source IP address.

Step 3 - Allocation of Bandwidth : This is the most important step in the bandwidth allocation algorithm. Our

network application uses the concept of QoS Queues in a networking device to implement allocation of custom bandwidths. A queue is used to store traffic until it can be processed or serialized. Both switch and router interfaces have ingress (inbound) queues and egress (outbound) queues. There are two types of queues, hardware queues and software queues. When an egress queue is created for an interface of router, the upper and lower bandwidth can be configured through the software. In the implementation of this network application, the SDN Controller uses the concept of configuring software queues onto Open vSwitches along with OpenFlow to match specific types of packets and en-queue them onto newly created queues which have been configured with user specified bandwidth. Note that, all traffic leaving the link of a router are en-queued by default onto a queue which egresses traffic at full link bandwidth. Initially, as soon as the algorithm determines that there is enough bandwidth on a link, it checks if the traffic from the source host is already being en-queued onto some non-default queue, that is, it checks whether certain bandwidth has already been allocated by the algorithm to the source host. It does so by checking the *allocated_bandwidths* dictionary which is a part of the Router class.

In the case of not finding an allocated bandwidth for a source host, the algorithm first determines the output port to reach the destination host through the router. Following this, it creates a software egress-queue configured with the user requested bandwidth to be the minimum and maximum bandwidth of the queue on the determined output port. Note that this queue has a unique queue id within this port. Then, it pushes a flow label that matches the source and destination IP address as in the request, with the corresponding action to en-queue the packet onto this newly created queue by referencing its unique queue id. Note that this flow rule is pushed to the router with a timeout value from the timeout extracted from the request dictionary. This ensures that, the requested bandwidth will be allocated to the host only for the requested length of time as the flow gets removed after the length of the timeout. After successfully pushing the queue, it updates its local data structures, namely the *allocated_bandwidths* dictionary and *available_bandwidths* dictionary. At this point, the algorithm also starts a timer thread to keep track of how long the requested bandwidth is allocated. On expiration of this timer, the algorithm once again updates its data structures with new values as the flow gets automatically removed because of flow timeout. The algorithm then proceeds to constructing a response packet for the source host.

In case the algorithm finds a previously allocated bandwidth for the source host, then it first sends an OpenFlow message to the router to delete the queue configuration previously created and updates local data structures, thereby removing any bandwidth already allocated to the source host.It then proceeds with allocation of the newly requested bandwidth configuration. If it finds that there is not enough bandwidth to allocate the new request, it restores the previously allocated bandwidth and sends an appropriate response to the source host.

```
Data: BW_REQUEST
Result: RESPONSE_CODE
while True do
    if newRequestReceived() then
        srcIP, dstIP, requestedBW, timeout =
          ParseRequest(BW_REQUEST);
        if isAvailable(requestedBW) then
            if isBandwidthAllocated(srcIP) then
                deAllocateBandwidth(srcIP);
            end
            queue =
              createQueue(QUEUE_ID,requestedBW);
            assignQueue(DEVICE_ID,
              EGRESS_PORT);
            flow_rule = newFlowRule();
            flow_rule.match(source:srcIP,destination:dstIP);

            flow_rule.action(enqueue:QUEUE_ID);
            flow_rule.timeout(timeout);
            sendFlowRule(DEVICE_ID);
            responsePacket =
              response(SUCC_RES_CODE,
              REMAINING_BW);
            sendResponse(srcIP, responsePacket);
        else
            responsePacket =
              response(FAIL_RES_CODE,
              EXCEPTION_INFO);
            sendResponse(srcIP, responsePacket);
        end
    end
end
```

Algorithm 1: Bandwidth Allocation Algorithm

Step 4 - Sending UDP response to source host : The network application running on the SDN Controller responds to clients based on a set of response codes developed for this application. On successful allocation of bandwidth, the algorithm constructs a UDP packet with a response code along with the remaining available bandwidth and sends it to the source IP and port number 10000 over the transport layer protocol UDP. In case the allocation of bandwidth fails for a certain reason, it sends the appropriate response code with relevant additional information to the source host. The client application running in the source host is expected to receive and parse the UDP response.

The sequential operations performed by the network application are written in the form of an algorithm (Algorithm 1) in a high level. Using this as a basis, the core of the bandwidth allocating network application is developed.

C. Programming the Client Application

The client application is agent using which the user communicates with the SDN Controller. The client application runs on the hosts in the topology, whose task is obtain input from the user, build bandwidth requests, send requests

to the controller, and wait for response from the controller. The client application is written in python. It obtains the bandwidth request value, time duration and destination IP from the user through command line arguments. After obtaining theses parameters, it constructs a bandwidth request packet consisting of an incrementally generated requestID, source IP, destination IP, bandwidth request and timeout. The constructed packet is sent to the router's interface IP address (the default gateway of the host) into port 9999 over UDP protocol. It then waits for the controller to process the the request and waits for response. The client application waits for response packets from the SDN Controller over port 10000. On receiving a response from the controller, it parses the response packet, which is again a python dictionary consisting of a response code and additional information. It maps the response code to a description and displays the response code, its description and additional information sent by the controller.

IV. RESULTS

The network application was successfully developed and was able to support allocation of bandwidth from one host to another host within the network topology with any custom bandwidth requested by the source host.

A. *Ping reachability in the network*

The topology was successfully created and all nodes of the network were testing using ping. The mininet command *pingall* is used for the same.

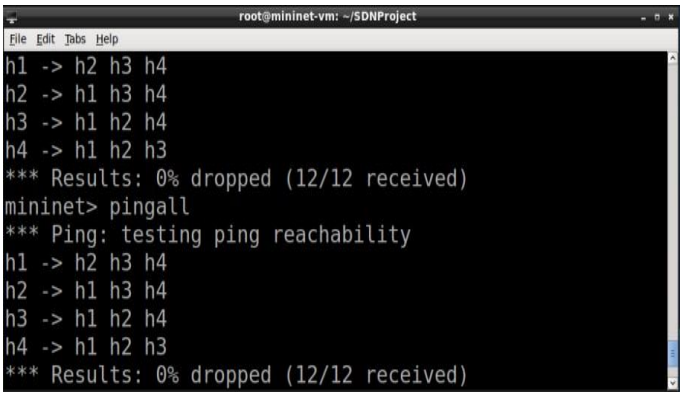

Fig. 6. Ping testing the entire network

B. *Efficiency in SDN using Flow Rules*

When a switch needs to forward a packet, it sends the first packet to the SDN Controller. The controller then pushes a flow rule to the device, after which the switch need not query the controller as to what to do with such types of packets henceforth. The flow label pushed by the SDN Controller will match such packets locally in the switch and forwarding immediately happens making the entire process highly efficient. This instantaneous drop in delay while forwarding packets is seen in the sudden drop in ping response time as seen in Fig. 7

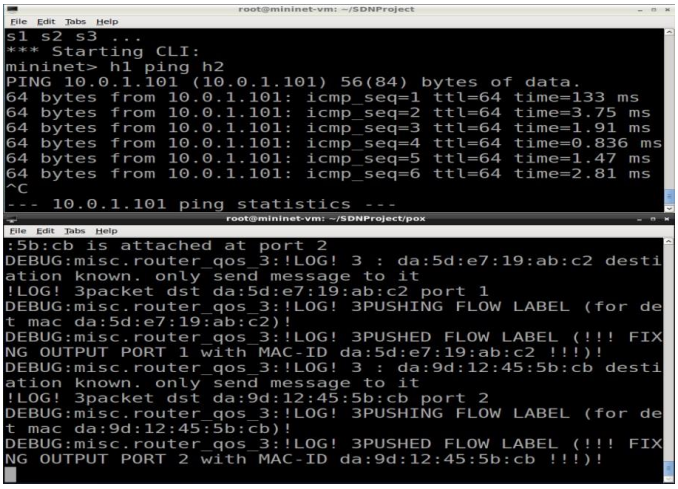

Fig. 7. Efficiency after pushing Flow Rule

The visualisation of drop in delay of ping packets as soon as flow rules are pushed using SDN Controller to the OVS is presented in Fig. 8 by recording ICMP Response times with respect to ICMP_Seq_Number

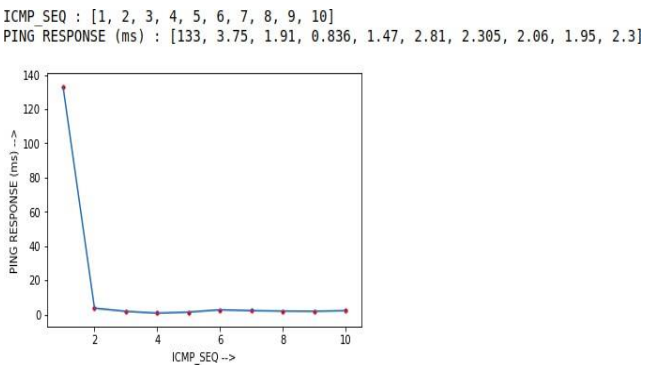

Fig. 8. Graph showing drop in latency of ping packets

C. *Bandwidth Allocation*

The bandwidth allocation algorithm and the client application used to request certain amount of bandwidth from the controller which was tested. The requested bandwidth was successfully allocated as seen in the following demonstration.

1) Bandwidth Allocation Demonstration: Figures 9 and 10 are individual screen shots of the client application and SDN Controller respectively. The client application (Fig 9) builds the bandwidth request as specified in the command line arguments and sends the request to the SDN Controller and waits for a response, which it receives in a short while containing the response code for the requested resource. The SDN Controller (Fig 10) receives the request and checks for availability of bandwidth after which it prints logs of the data structures that it modifies. It eventually sends a response to the client application requesting it.

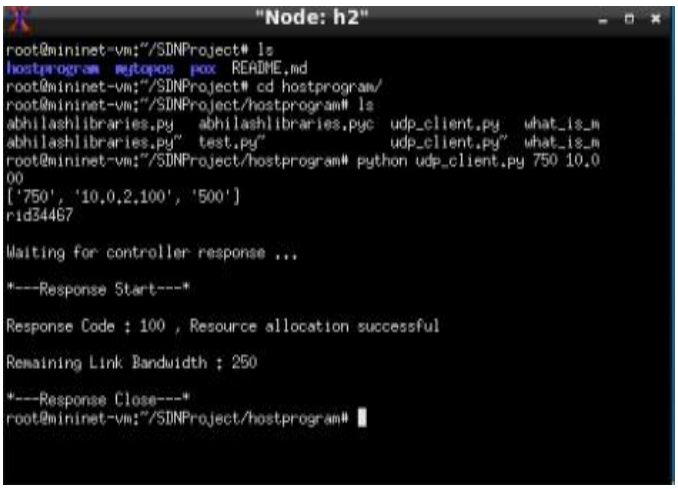

Fig. 9. Client application request-response

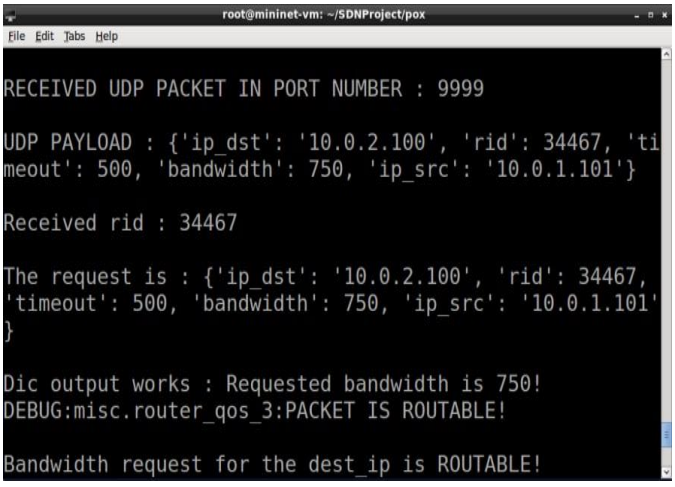

Fig. 10. SDN Controller receiving and processing bandwidth request

Fig 11 is a screenshot of the logs of the SDN Controller confirming that the requested bandwidth is available and allocates the bandwidth following which it builds a response to be sent to the source host.

D. Scenario Based Demonstration

1) Scenario A - Successful Bandwidth Allocation : Scenario A(a) : Single Host Requesting Bandwidth: This scenario represents the working of the network application when there is only a single requesting host in the network. Through this scenario, the exact bandwidth values allocated can be measured as an ideal case. Refer to Fig. 14 which contains the successful allocation of a bandwidth of 200 Mbps from source host h2 to destination host h4 in the other network. The figure also has performance testing logs done using *iperf* tool

A graph has been plotted for better visualization of the exact bandwidth obtained on requesting 200 Mbps. Refer to Fig 15.

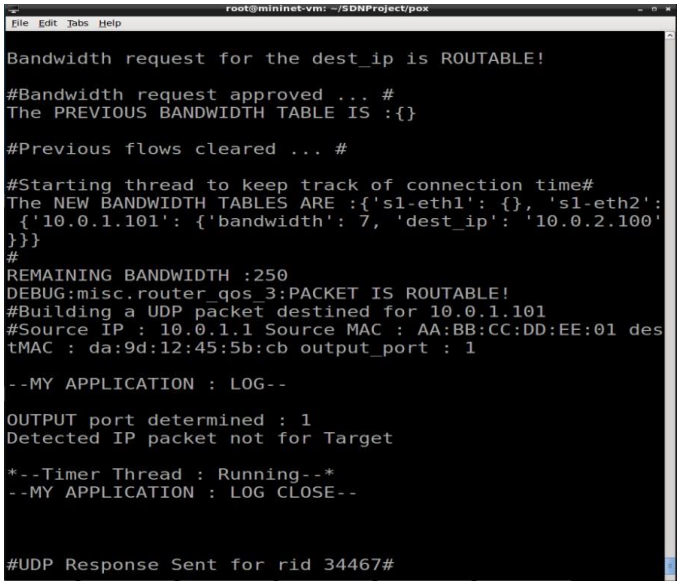

Fig. 11. SDN Controller receiving and processing bandwidth request

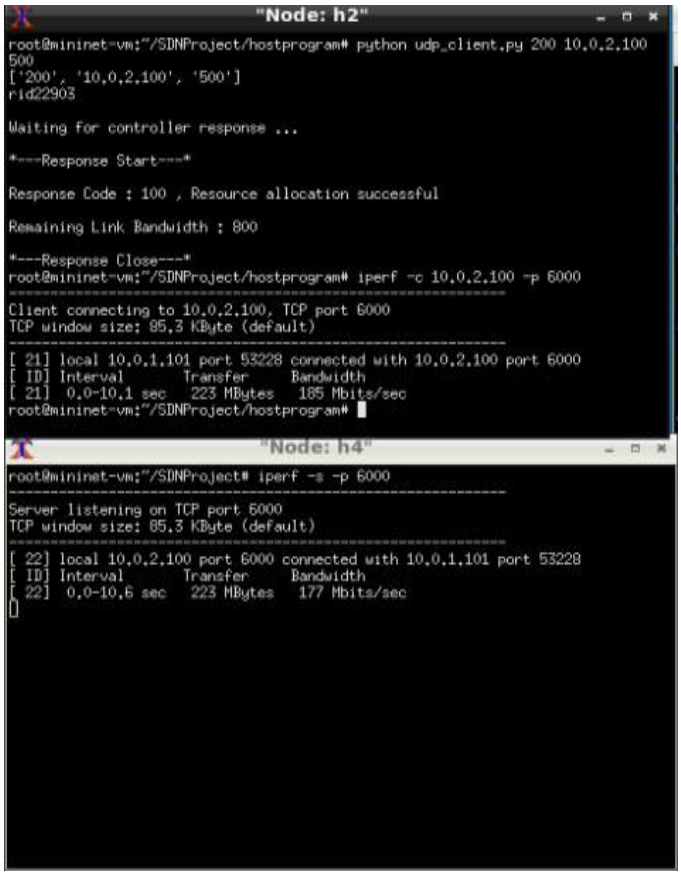

Fig. 12. Performance testing of the allocated bandwidth

Fig 16 contains finer visualisation of actual bandwidth obtained on successful allocation of 200 Mbps to a host. The plot values are extracted from the output of the *iperf*

IPERF_TEST_ID : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
RECORDED BANDWIDTH (Mbps) : [182, 183, 183, 179, 191]

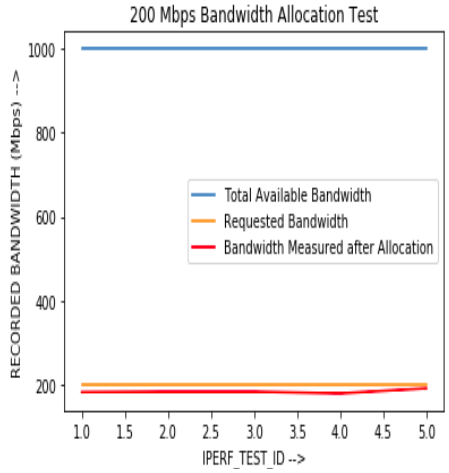

Fig. 13. Graphical visualisation of the obtained bandwidth

performance testing tool.

IPERF_TEST_ID : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
RECORDED BANDWIDTH (Mbps) : [182, 183, 183, 179, 191]

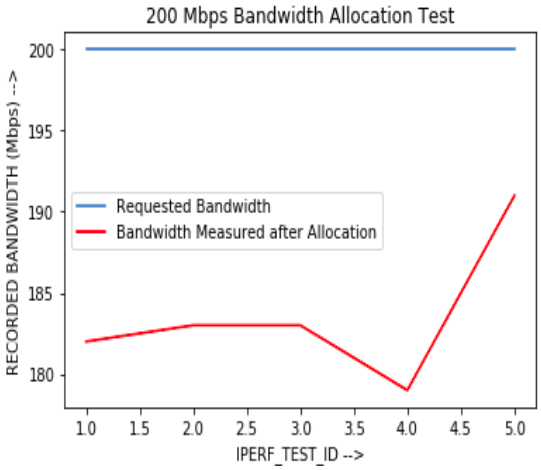

Fig. 14. Finer visualisation of actual bandwidth obtained on allocation of 200 Mbps bandwidth

Refer to Fig 17 which contains an extensive account of testing of allocation of a bandwidth of 500 Mbps from source host h2 to destination host h4.

Scenario A(b): Multiple Hosts Requesting Bandwidth: The application works just as good when two hosts simultaneously request for certain amount of bandwidth from the SDN Controller. The network applications creates separate queues for each host and hence the requested bandwidth value is exactly allocated for each requesting host.

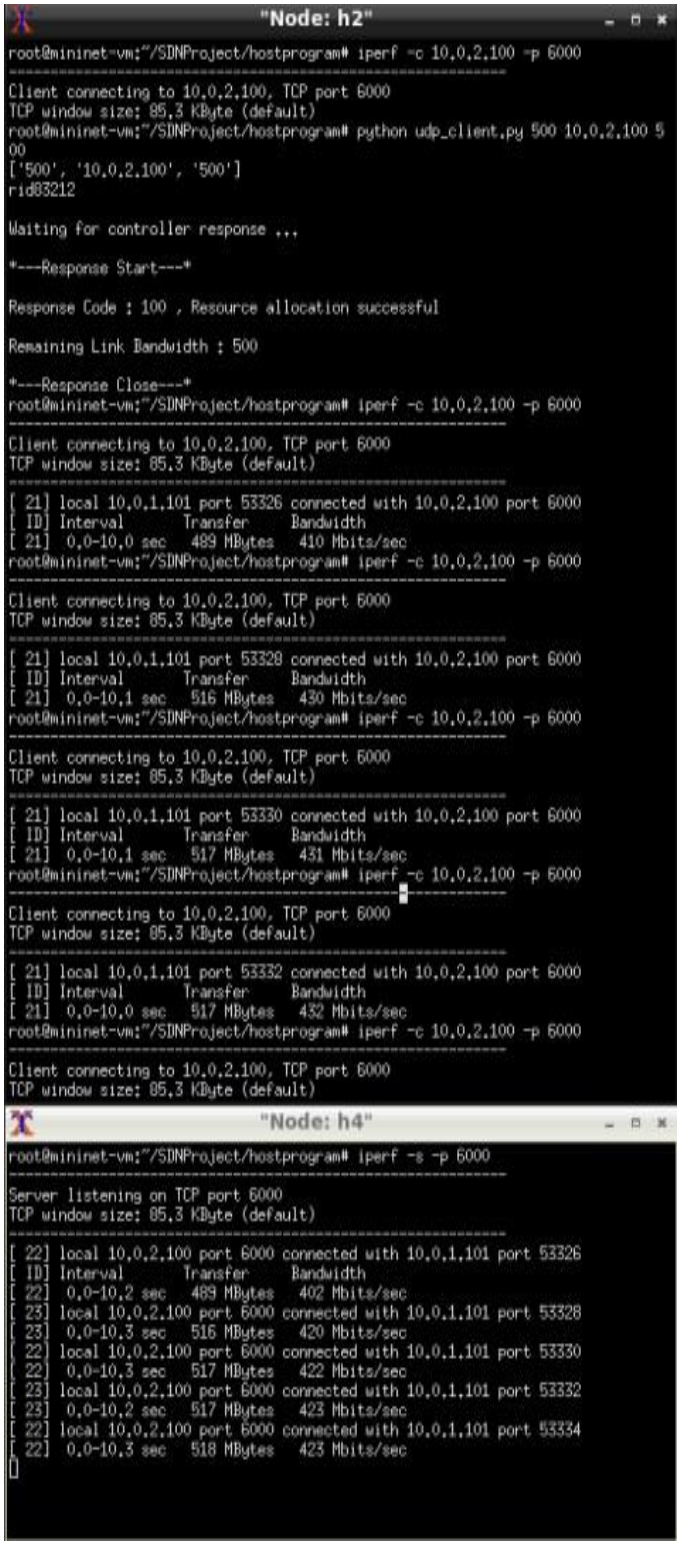

Fig. 15. Extensive testing of allocated bandwidth

Fig 16 shows iperf testing for connection from host h1 and h2 to host h4 where the hosts have requested a bandwidth of 500 Mbps and 200 Mbps respectively.

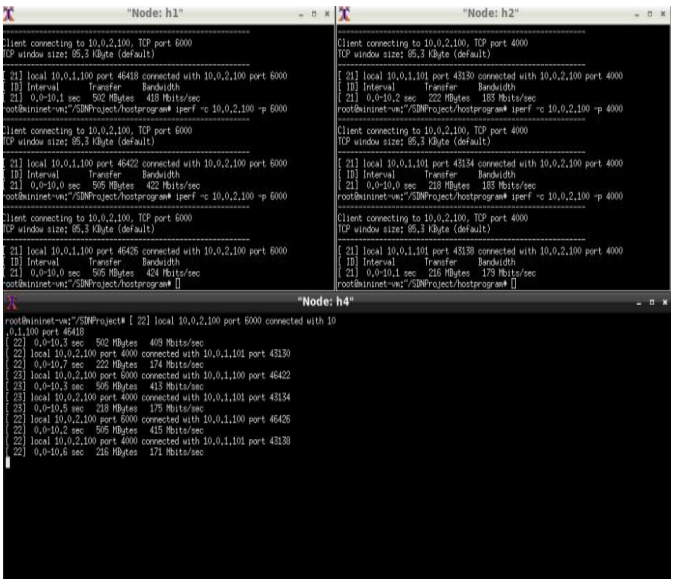

Fig. 16. h1 and h2 simultaneous iperf testing with h4 as destination for both

2) Scenario B - Bandwidth Allocation Failure : Scenario B(a) - Insufficient Bandwidth on a Link: Fig 12 shows the SDN Controller detecting insufficient bandwidth on a link which results in insufficient bandwidth exception.

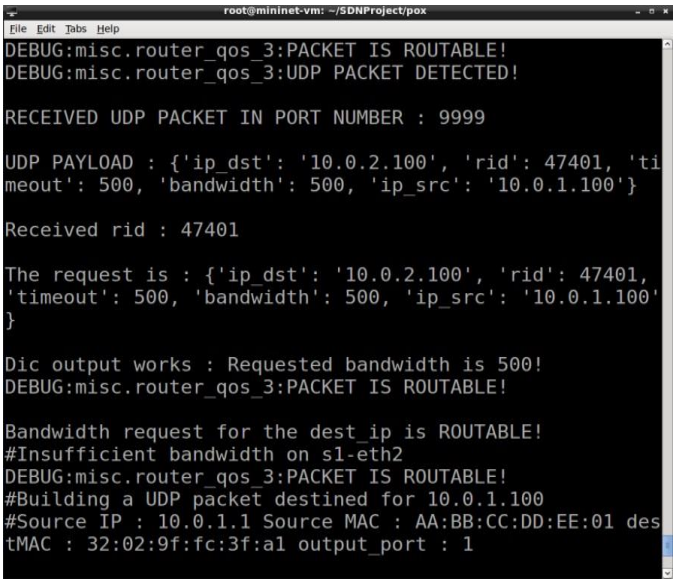

Fig. 17. SDN Controller : Insufficient bandwidth on a link

Fig 13 shows the client h1's receipt of the SDN Controller's failure response code because of insufficient bandwidth on a link. The screenshot is supported with h2's allocation of 750 Mbps of speed over a 1 Gbps link. h1's request of 500 Mbps of bandwidth would hence result in a failure. Note that the response to h1 also contains additional information regarding maximum available bandwidth.

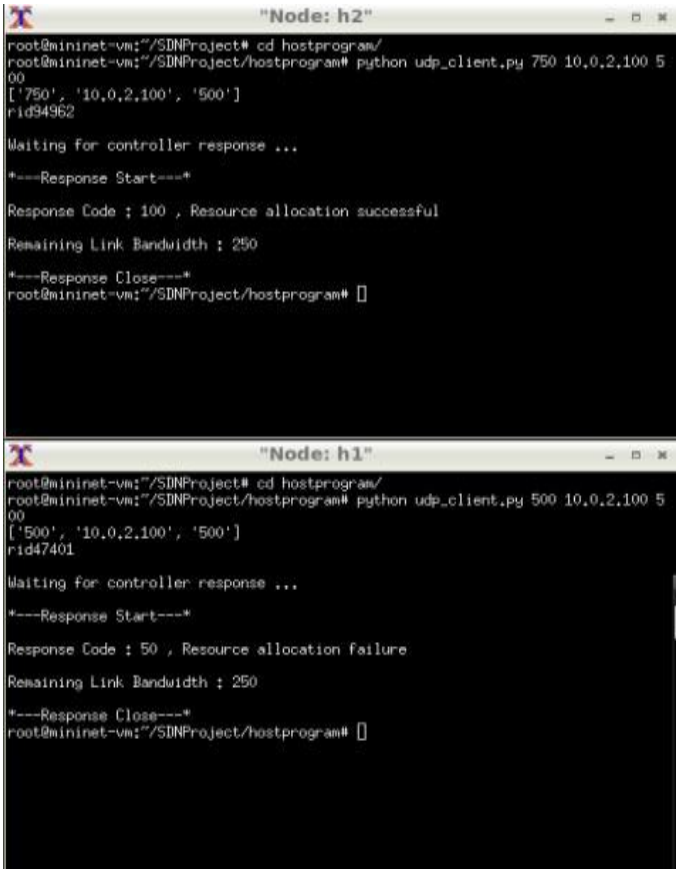

Fig. 18. Client's receipt of failure response code

V. PRACTICAL APPLICATIONS OF THE BANDWIDTH ALLOCATION PROGRAM

Looking into the future, the bandwidth allocating network application has a lot of scope when it comes to dynamic network requirements for end users. We believe this application could be embedded into the code of a modern operating system to provide the computer full capability of dynamic bandwidth requests. When this application is built into a standard PC's OS, the OS can presumably take care of bandwidth request negotiations with the SDN controller by using this network application built into it(when the PC itself is a part of an SDN). In such a scenario, no user intervention will be required to specify bandwidth requirements and timeouts. The user applications could relay their requests through this bandwidth request application by default. This network application would automatically detect required bandwidth for the type of application used by the user and would query the controller for the necessary bandwidth needed for certain amount of time negotiated with the source application. This will lead to seamless transition in bandwidth without the user being aware of any networking changes happening in the background. Another useful advantage we can derive using this network application is that, we can modify this application to specify *different bandwidths for different applications running on the same*

computer, thereby providing, say. simultaneous high speed video streaming and low speed background updates on the same PC! Implementing this network application in enterprise grade networks would drastically reduce the companies' bills, as most of the time, 100% of the allocated bandwidth is not used, but the charge is the same regardless of this fact. This network application also saves bandwidth in the network by a big margin because the app detects low bandwidth requirements and requests for minimum bandwidth. This is good for the network as it creates more available space for connections from other users using the network links.

VI. Conclusion

In the modern world where software advances at an unimaginably fast rate, it is difficult for systems programmed at hardware level to keep up. Any change needed to the system would require the replacing of the hardware itself, or worse, it would require the rebuilding of the entire architecture of system in which the hardware is deployed. The exact scenario is seen in computer networks where the control and data plane of devices are locked into one device that runs proprietary code. Configuration and manipulation of these devices requires knowledge of operating these devices using the vendor specified console commands. Software Defined Networking is a revolutionary solution to this problem which not only separates the data and control plane of a networking component thereby making it programmable, but also opens up options for programmers to bring the advancements of software to computer networks which would accelerate the evolution of computer networks to the evolution of software. The use of software defined networks would allow programmers and network administrators to implement various solutions within computer networks which was once not possible in an architecture where the data and control plane were coupled into one single device.

In this paper, we demonstrated the successful deployment of a network application that allocates bandwidth using QoS flow rules and OpenFlow. This application was developed using POX, which is a python module used to program SDN Controllers, and custom modules used in development of the client application. The building of this network application consisted of creating a scalable topology, programming the SDN controller and programming the client application for the end hosts. The application was created, deployed and tested using a software network emulator called mininet. We summarize various conclusions obtained from our results in this section.

In scenario A(a), the bandwidth value of 200 Mbps was requested and the recorded allocated bandwidth stayed constantly at around 191 Mbps. The maximum deviation recorded was 182 Mbps. Hence, we infer from the analysis of this scenario that our network has a maximum deviation of less than 10% of the requested bandwidth when there is a single requesting host in the network. This maximum bandwidth deviation is caused because of network virtualization and can be reduced. The constancy obtained is more efficient than the current variation in bandwidth allocated by ISPs to users. This constancy of bandwidth will be of great benefit to the user and saves bandwidth for the ISP as well.

In scenario A(b), we demonstrate constant bandwidth values recorded when two hosts within the network topology request different bandwidth values. The host requesting 200 Mbps had a maximum deviation of 175 Mbps which is less than 13% of the requested bandwidth. We infer from this observation that the network application can seamlessly allocate bandwidth to multiple hosts requesting different bandwidth values simultaneously and provide nearly same efficiency to the requesting clients.

In scenario B(a), a link in the network that does not have enough bandwidth (because of previously allocated requests) which leads the network application to throw an exception to the user stating there is not enough bandwidth resources hence rejects the request. We demonstrate using this scenario, the network applications ability to keep track of allocated bandwidths and send failure response codes to the client application.

The results obtained within the virtualized network gave nearly constant bandwidth values when bandwidth was allocated from a source host to destination host within the scalable custom topology. The dynamic control over bandwidth provided by this application and its constancy in bandwidth allocation is a prime reason for this applications to be used by ISPs to provide bandwidth plans for their customers for mutual benefit and better networking.

The project is available for use in github : https://github.com/abhilashvenky/SDNProject

KEYWORDS

SDN - Software Defined Network/Networking; OVS - Open vSwitch; API - Application Program Interface; QoS - Quality of Service; MAC - Media Access Control; BW - Bandwidth

Acknowledgment

The authors would like to thank their esteemed institution Amrita School of Engineering for its constant support and encouragement to research and innovate in the field of information technology. We would like to thank our teachers who helped us gain in depth knowledge in the field of computer networks which lead us to write this paper. We also would like to thank our friends and family who encouraged and supported us from the beginning.

References

[1] W. Xia, Y. Wen, C. H. Foh, D. Niyato and H. Xie, "A Survey on Software-Defined Networking," in IEEE Communications Surveys & Tutorials, vol. 17, no. 1, pp. 27-51, Firstquarter 2015. doi: 10.1109/COMST.2014.2330903

[2] H. Kim, N. Feamster, "Improving network management with software defined networking", IEEE Commun. Mag., vol. 51, no. 2, pp. 114-119, 2013.

[3] Cote, David, William Webb, and Sergio SLOBODRIAN. "Dynamic bandwidth control systems and methods in software defined networking." U.S. Patent No. 10,187,318. 22 Jan. 2019.

[4] N. Feamster et al., "The Case for Separating Routing from Routers," ACM SIGCOMM Wksp. Future Directions in Network Architecture, Portland, OR, Sept. 2004

[5] Salih, Mohammed,"Intelligent network bandwidth allocation using SDN (Software Defined Networking)",International Journal of Engineering and Technical Research, ISSN: 2278-0181, Vol. 4 Issue 07, July-2015
[6] A. Vishnu Priya, N. Radhika, "Performance comparison of SDN OpenFlow controllers", International Journal of Computer Aided Engineering and Technology, 4-5, 467-479, Nov 2019
[7] Jisha MS, Dr. N Radhika, "Software Defined Networking: A New Frontier in Networking." International Journal of Advanced Research in Computer Science and Software Engineering. Volume 5, Issue 7, July 2015 .
[8] N. Saritakumar, Adarsh V Srinivasan, Elfreda Albert S, Subha Rani, "Performance Evaluation of Pox Controller for Software Defined Networks", International Journal of Innovative Technology and Exploring, ISSN: 2278-3075, Volume-8, Issue-9S2, July 2019