SDN Flow Caching

N.B.U.S. Nanayakkara, R.M.L.S. Bandara, N.B. Weerasinghe, S,N, Karunarathna
Department of Computer Engineering, Faculty of Engineeirng
University of Peradeniya, Sri Lanka

Abstract—**Software-Defined Networking (SDN) allows control applications to install fine-grained forwarding policies in the underlying switches, using a standard API like OpenFlow. By allowing the switches to store flow rules it can perform a parallel lookup to quickly identify the highest-priority match for each packet. The cost and power requirements limit the number of rules the switches can support. To make matters worse, these hardware switches cannot sustain a high rate of updates to the rule table. In this paper, we show how to give applications the illusion of high-speed forwarding, large rule tables, and fast updates.**

Keywords—Software Defined Networking, Flow Caching

I. Introduction

SOFTWARE-DEFINED NETWORKING (SDN) allows control applications to install fine-grained forwarding policies in the underlying switches, using a standard API like OpenFlow. By allowing the switches to store flow rules it can perform a parallel lookup to quickly identify the highest-priority match for each packet. The cost and power requirements limit the number of rules the switches can support. To make matters worse, these hardware switches cannot sustain a high rate of updates to the rule table. Ideally, enterprise administrators could specify fine-grain policies that drive how the underlying switches forward, drop, and measure traffic.

The existing techniques for flow-based networking rely too heavily on centralized controller software that installs rules reactively, based on the first packet of each flow. On the surface, the simplest approach to flow-based management is to install all of the low-level rules in the switches in advance. However, pre-installing the rules does not scale well in networks with mobile hosts, since the same rules would need to be installed in multiple locations. In addition, the controller would need to update many switches whenever rules change. Even in the absence of mobile devices, a network with many rules might not have enough table space in the switches to store all the rules, particularly as the network grows or its policies become more complex. Instead, the system should install rules on demand or allow caching the rules in a separate cache.

In this paper, we show how to give applications the illusion of high-speed forwarding, large rule tables, and fast updates.

It is not possible to blindly apply existing cache-replacement algorithms, because of dependencies between rules with overlapping patterns. These rules can match on a wide variety of packet-header fields, and perform simple actions as forwarding, flooding, modifying the headers, and directing packets to the controller. This flexibility allows SDN enabled switches to behave as firewalls, server load balancers, network address translators, Ethernet switches, routers, or anything in between.

II. Background

The traditional network devices have control and data-flow functions established the same device. The control to a network administrator is only from the network management plane, which can be configure each network node separately. This behavior of current network devices does not permit detailed control-plane configuration. This is the reason for introducing software-defined networking. The target of SDN is to "provide open user-controlled management of the forwarding hardware of a network element." As mentioned in [8]. SDN has the idea of centralizing control-plane, but keeping the data plane separate. So the network hardware devices keep their data plane, but hand over their switching and routing functions to the controller. This enables the administrator to configure the network hardware directly from the controller. This centralized control of the entire network makes the network highly flexible [9, 10].

A. *Proactive And Reactive Modes*

SDN Controller uses two modes of operations to fill the flow tables of connected switches. Namely they are Proactive mode and Reactive mode. Controller uses only one mode on a SDN network.

In Proactive mode SDN controller places all the rules according to the topology of the network once the switch is initiated. The rules are then relatively static.

In contrast reactive mode, rules are relatively dynamic. Initially the flow table is empty for a given Switch. Switches forwards the packets which do not have an entry in the flow table (OPF packet in). Then the controller examines the packet, decides the flow entry and send both packet (OPF packet out) and flow rule to the switch back. So each and every packet which does not have a flow entry in the switch are forwarded to the controller introducing more delays.

This paper focuses on developing a flow cache for reactive mode. So once a new switch is connected some of the flow rules can be initially taken from this cache. So that the number of packets forwarding to the controller is minimized, and the delay introduced by forwarding packets to the controller is eliminated

III. Related Work

The research paper [5] shows how Software Defined Network allows control applications to install forwarding rules in switches, using a standard API like OpenFlow. It gives information on modern switches, those rules are stored in Ternary Content Addressable Memory (TCAM). It performs a parallel lookup. That is essential to quickly identify the highest-priority match for each packet. Even though it is efficient, the cost and power consumption limit the number of rules the switches can support. And also these hardware switches cannot sustain a high rate of updates to the rule table. Furthermore it shows how to give applications high speed forwarding, large rule tables, and fast updates. It is done by combining the best of hardware and software processing. The Cache Flow system caches the most recently used rules in the small TCAM, while relying on software to handle the small amount of traffic. But existing cache-replacement algorithms cannot be applied easily because of relationships between rules with overlapping patterns. So rather than caching large chains of dependent rules, long dependency chains are joined to cache smaller groups of rules while protecting the semantics of the policy.

Research paper [6] convinces about data traffic in mobile networks due to the massive adoption of smartphones and tablets, which creates new problems in network. While it always requires more Bandwidth and Data Consumption, cost of network infrastructure is increasing rapidly in mobile networks which will point to an "end of profit" as well. So mobile operators require some methodology that allows to increase the network capacity within low network costs. So to optimize the network capacity usage, the best way is to place caches in strategic locations within network. However, the modern architecture of LTE network does not provide a suitable environment to place the caches in the most optimal locations. This paper proposes to use a SDN approach to remove the need of establishing tunnels. With SDN, packets are forwarded according to rules installed on switches through a centralized controller. The forwarding rules which are installed will be used in routing algorithms to permit scaling as signaling messages are not flooded in the network, but sent directly to the appropriate switches. Those switches use OpenFlow to permit to a centralized controller to install specific forwarding rules. It was proposed to use a caching relocation system based on simple, hence implementable, feedback loop algorithm.

OpenFlow is used to communicate with switches and routers by controllers. OpenFlow is designed to implement network policies. [8] Most of the details regarding OpenFlow packets and Controller architecture are referred from [9], [10] and [11].

IV. Methodology

A. *Caching Flow Entries*

Fig. 1 shows the separate application which works as a cache in the SDN controller. Other applications running on top of SDN controller could be routing algorithms etc. SDN controller has 2 APIs as Northbound API and Southbound API. The SDN cache is implemented using the Northbound API. The southbound API is used to communicate between SDN controller and SDN switches using OpenFlow protocol.

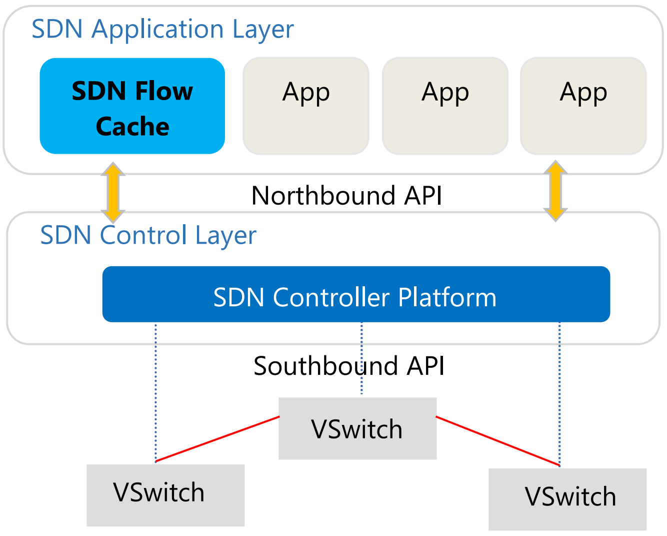

Fig. 1. Application Architecture of the Flow Cache

When a switch is connected to a Controller, after exchanging *hello* messages, the controller sends *featureReq* message to the switch to get information about the switch. Then the switch replies using a *featureRes* message which has all the features of the switch. Then the switch will send LLDP (Link Layer Discovery Protocol) packets to discover devices connected to each port of the switch. These discovered MAC addresses will be used to determine where to forward a packet in a network if the flow tables are not updated.

As an example, first consider a network with two hosts connected to two L2 Switches separately (Fig 2). When the two switches are connected the controller will know the link between S1 and S2 and the links between hosts H1 and H2 with S1 and S2 (H-S1, H2-S2, S1-S2). When H1 pings H2, the first ICMP packet will be forwarded to the Controller as S1 has no flow entry for H2 MAC address. Then the controller will send a flow to the S1 switch with the forwarding entry for H2 MAC address.

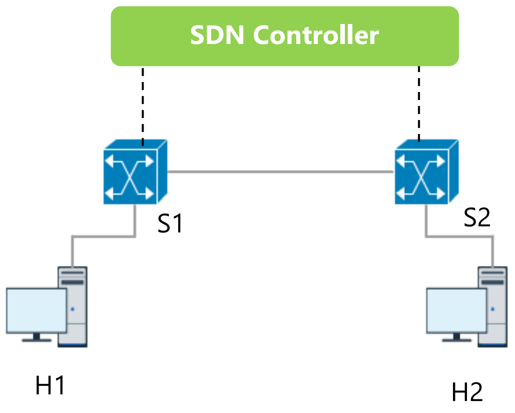

Fig. 2. Example Network I

Suppose that another Switch S3 with host H3 is connected to the network (Fig. 3). This will send LLDP packets to controller and inform the controller about H3 and S3-S2 link. When H3 pings to H2, it will also send the first packet to the controller and get the forwarding flow entry for H3. In this scenario, if the flow cache is implemented, whenever a switch connects to the network it will send flow entries for Hosts and Switches in the other section of the current network topology. So that the new switch will not have to flood the controller with packets in order to update its entries in the flow table. This will have significant impact on the networks with large number of hosts and controllers which are dynamically changes rapidly, such as Data Centers.

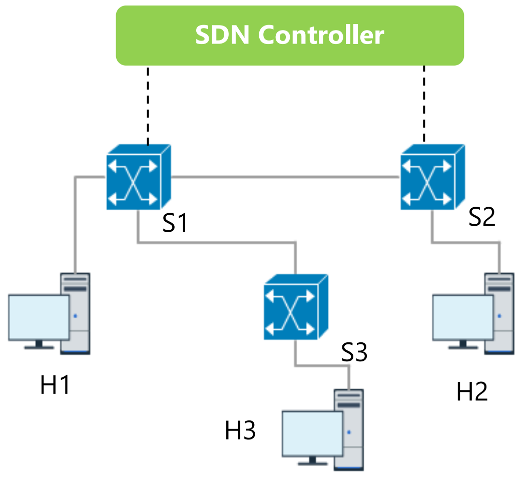

Fig. 3. Example Network II

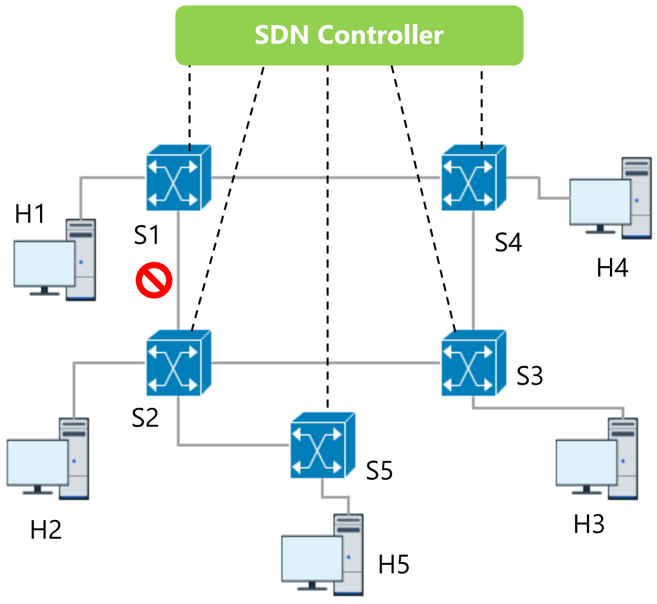

Fig. 4. Example Network III

In the next scenario, consider a network with four switches. The network with S1, S2, S3, S4 switches was the previous network before switch S5 with H5 was introduced. Suppose that the link S1-S2 is restricted for packets from other ports and it is used only for host H1 and H2. In this scenario, the new connected switch S5 should forward its packets via S3 and S4 to S1. This maneuver should be implemented in the flow installing algorithm with validation.

B. Caching Algorithms and Validation

The caching of each flow will be identified from its MAC address. Therefore it can be used as a key for a Hash Table. The object model of the Hash Table will contain an Array of Flow IDs that are created by in the controller. The Cached flows will be stored in another Hash Table and each key would be the Flow ID and object model will be a flow. When LLDP packets are identified by the controller application it will look at the Hash Table and get the flow entries and process it in an algorithm to build new flow entries. If the MAC is new it will make a new Flow Entry object and insert it to the stack install flows for links that are connected to ports.

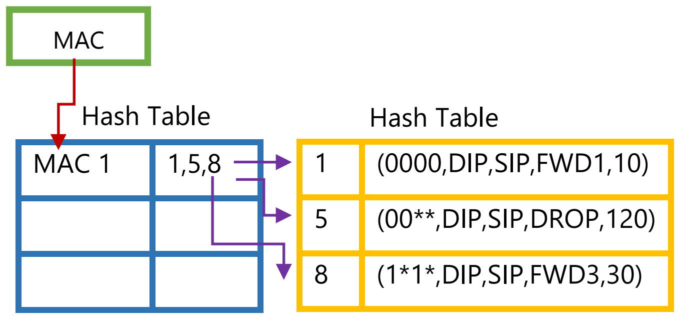

Fig. 5. Data Structure for the Flow Cache

When the user modifies the network with restrictions, the Hash table should be updated to have the restrictions and priorities in the network. This will send only the allowed connections for a new switch in the network. Algorithm 1 can be used to send only the allowed flows to a new switch.

Algorithm 1: Send Flows for Discovered Connection

```
1  func Send_Flows(conn) begin
2    rule_list = PriorityQueue(Flows ϵ conn.MAC)
3    restricted = [ ]
4    for each rule in rule_list
5      if rule.action == DROP then
6        restricted.add(rule)
7    GenerateFlows(flows - restricted)
```

The caching algorithms for SDN rules that can be used are described in the research paper [14] and any applicable algorithm can be used with caching flows.

V. Implementation and Evaluation

As described in the methodology, the application will be an instance of a modified L2Switch application in the controller. L2Switch application has all the functions necessary for a physical VSwitch device that access the controller to update flows or network configurations. A referring code base for the switch is in the dev list of OpenDaylight Controller. This development environment was used to implement the caching method for the flows.

OpenDaylight code base is in java and the application implementation is in OSGI framework in the controller. This framework also uses RESTCONF, NETCONF and YANG (RFC6020) models to communicate between network layers and controller layer. This architecture is overall known as MD SAL (Model Driven Service Abstraction Layer) which is defined by OMG[9] (Object Management Group) globally. The controller used OSGI as it can load components in run-time. RESTCONF provide the programming interfaces for the Northbound API that actually runs the SDN network manipulation in the controller. NETCONF is a network management protocol that provides configuration and operational conceptual data stores and perform CRUD operation for these data. Basically it is responsible for running configurations in network devices.

The MD SAL adaptation layers performs the communication between NETCONF and RESTCONF so that the network devices can access the network control plane in the controller.

When the application is informed about a link from the underlying applications, it will update the controller to have a Flow Entry for it. When H1-S1 link is informed it will not wait for a packet from H1 to insert flow entries. Instead controller will update the Flow Cache and send a Flow to the switch as well.

The cache will use a Hash Table to relate each MAC address that the SDN Controller receives. When this is applied in the controller it will be optimized in with relevance to other two modes in the controller as shown in the Fig. 1.

Testing was done using Mininet and it is a network emulator. It is more precisely a network emulation arrangement system. It has a collection of end-hosts, switches, routers, and links on a single Linux kernel. We can make a single system look like a complete network which is running in the same kernel, system, and user code by using a lightweight virtualization, A Mininet host behaves just like a real machine and we can send SSH requests into it and run random programs. The programs we run can send packets through the interface which seems like a real Ethernet interface, with a given link speed and delay. Packets get processed by that Ethernet switch or router, with a given amount of queuing. When two programs, communicate through Mininet, the measured performance should match that of two native machines.

To test the developed system, a new switch connected is connected to an existing network (initially with 10 nodes/ switches). Then a new host is connected to the newly added switch. Now this host pings to a host which is already there in the network (identified by the controller). The time for the 1st ping is recorded. Then the number of switches (Nodes) is increased and the same process of adding a new switch and pining is repeated. The times are recorded.

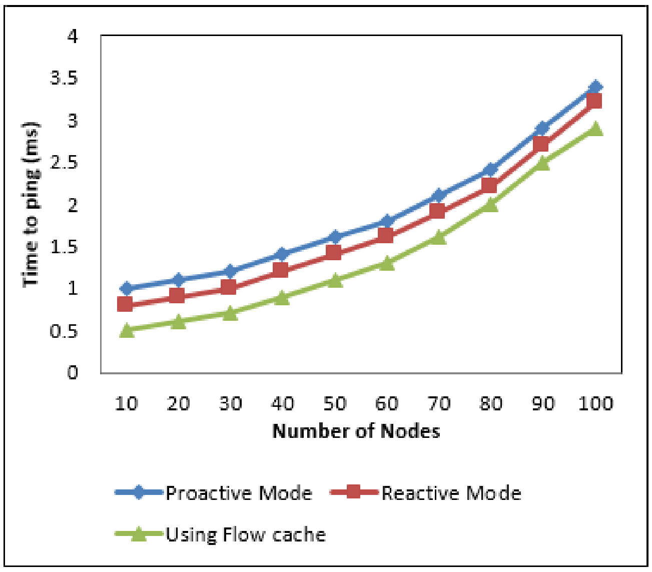

Fig. 6 Time to Ping VS Number of Nodes

VI. Conclusion

As in the work covered in this paper, a flow cache using a hash map was designed and implemented in OpenDaylight SDN controller and tested using mininet. Proactive mode SDN networks can be optimized by using a separate flow cache. Flow table of a newly connected switch can be updated using the cache. So that, newly connected switches won't forward the no match packets to the controller. This will reduce the network added delay. Furthermore it will reduce unnecessary packet processing at the controller.

The result statistics showed a performance upgrade with the use of cache than the usual proactive or reactive modes. The performance is up in milliseconds scale but it is really vital in applications such as transaction processing streaming.

References

[1] Duan, Q., Yan, Y.H., Vasilakos, A.V., "A Survey on Service-Oriented Network Virtualization toward Convergence of Networking and Cloud Computing," IEEE Transactions on Network and Service Management, vol.9, no.4, pp.373–392, December 2012.

[2] Big Switch Networks, The Open SDN Architecture, http://www.bigswitch.com/sites/default/files/sdn_overview.pdf, 2012.

[3] Shin, M.K., Ki-Hyuk Nam, K.H., Kim, H.J., "Software-Defined Networking (SDN): A Reference Architecture and Open APIs," Proceedings, 2012 International Conference on ICT Convergence (ICTC), pp.360–361, 15–17 October 2012.

[4] https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

[5] Infinite Cache Flow in SDN by Princeton University
ftp://ftp.cs.princeton.edu/techreports/2013/966.pdf

[6] Caching Using Software-Defined Networking in LTE Networks
https://hal.inria.fr/hal-01117447/file/caching-software-defined.pdf

[7] LRU-K Page Replacement Algorithm ppt by Prof. Shahram Ghandeharizadeh
https://www.google.lk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwjW55ni_aHMAhVpIJoKHYwGBFYQFggaMAA&url=http%3A%2F%2Fdblab.usc.edu%2Fcsci485%2FLRU-K485.ppt&usg=AFQjCNEfoOMn2dN1Syfk7qvAZ7Au5mU_ww&sig2=Ad09dRTucn9zNu6PGs-pHQ&bvm=bv.119745492,d.bGs&cad=rja

[8] Software-Defined Networking: Why We Like It and How We Are Building On It
http://www.cisco.com/c/dam/en_us/solutions/industries/docs/gov/cis13090_sdn_sled_white_paper.pdf

[9] Openflow Protocol Standards and researches.
http://flowgrammable.org/sdn/openflow/

[10] SDN Resources at Open Networking Foundation
https://www.opennetworking.org/sdn-resources/openflow

[11] Open Cache Project by Lancaster University, United Kingdom
https://opencache-project.github.io/research.html

[12] SDNHub OpenDaylight Application Developer's tutorial
http://sdnhub.org/tutorials/opendaylight/

[13] OMG Object Management Group
http://www.omg.org/

[14] rule-caching algorithms for software-defined networks
http://www.cs.princeton.edu/~nkatta/papers/cacheflow-long14.pdf