ECMP Based on Traffic Monitoring

|  |  |  |
| --- | --- | --- |
|  | Yi-Sung Chiu National Chiao Tung University Taiwan steven30801@gmail.com |  |

ABSTRACT

With the limitation of load balancing with hash function or other algorithm according to packet header fields, in this paper we present a dynamically scheduling mechanism based on traffic monitoring. Each switch or router determine next-hop of flow by choosing the least loaded link. In the case of multiple elephant flows between two end devices and with same protocol stacks, such as one user requests for large files from a FTP server, our method could distribute flows to ports in a more efficient way.

CCS CONCEPTS

• **Networks** → Network algorithms

KEYWORDS

SDN, ECMP, load balancing

1 INTRODUCTION

Nowadays, significant usage and demand to the Internet results in enormous throughput of the network devices. Therefore, the network was built with loops and multi-paths to increase the throughput, accessibility and fault tolerance as well. In legacy network, there are several approaches to determine a path of flow from multi-paths. The key idea of all approaches is to balance the load of each path and expects to spread flows evenly on the entire network. Load balancing is even more crucial in datacenters and networks with servers.

There are several methods to balance load on multi-paths. For simple instance, we can split flows based on different packet categories, such as separate TCP and UDP packets by transport layer protocols. Another method is splitting flows with different destination IP address, like separating flows sent to 140.113.1.0/24 and to 140.113.2.0/24 by subnets.

In RFC 2991 [1], IETF discusses multi-path issues and three algorithms are provided to determine next-hops for network devices in multi-paths scenario:

* + *Module-N Hash*: To select a next-hop from the list of N next-hops, the router performs a modulo-N hash over the packet header fields that identify a flow.
  + *Hash-Threshold*: The router first selects a key by performing a hash over the packet header fields that identify the flow. The N next-hops is assigned unique regions in the hash function's output space. By comparing the hash value against region boundaries the router can determine which region the hash value belongs to and thus which next-hop to use.
  + *Highest Random Weight (HRW)*: The router computes a key for EACH next-hop by performing a hash over the packet header fields that identify the flow, as well as over the address of the next-hop. The router then chooses the next-hop with the highest resulting key value

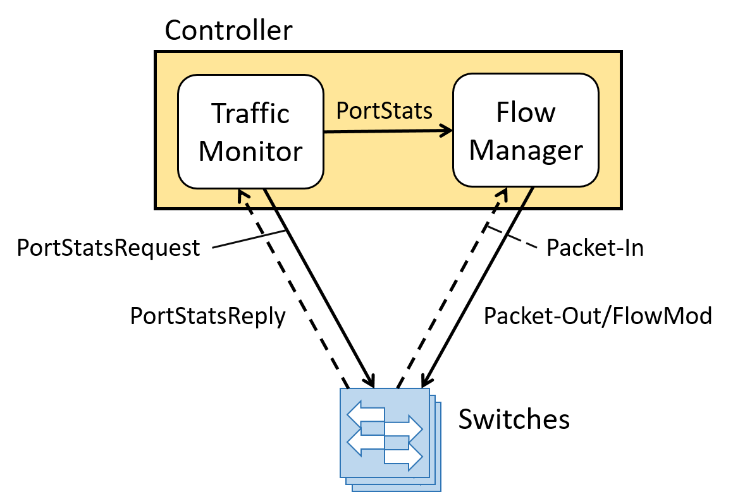
In software-defined network, such as prevalently used OpenFlow protocol, load balancing is mostly implemented with select type of group table, which selects action buckets through a particular algorithm depending on switch implementation. Take Open vSwitch (OvS) as an example [2]. For version 2.3 and earlier, OvS used destination Ethernet address to choose an action bucket in a select group. For version 2.4 and later, by default OvS hashes source and destination Ethernet address, VLAN ID, Ethernet type, source and destination IPv4/v6 address and protocol, and for TCP and SCTP only, the source and destination ports.

We can observe that currently used methods for load balancing are related to hash functions. The defect of hash function is apparently the possibility of collision. Especially in the small-scale network such as datacenters, there are only several paths between any two nodes, and thus it is probably multiple data flows are assigned to the same path, leading to unbalancing load. Another negative case of load balancing with hash functions is that, if there exists only two or three trunk links that any flow between two subnets must go through one of these links, then elephant flows between subnets may concentrate in one trunk link.

Since hash algorithm is nearly random distribution, there is no way to take the actual condition of network into consideration. Taking advantage of software-defined network, we are able to access traffic statistics of network devices easily. Our approach is to monitor the ongoing network traffic, consider whether links of each switch are congested or not and select the least loaded link to transmit data. Therefore, we can distribute the flow between links evenly and balance network load efficiently.

2 DESIGN AND ARCHITECTURE

Our controller architecture can be split into two parts, Traffic Monitor and Flow Manager. The whole architecture and control messages are depicted in Fig. 1.



**Figure 1: Architecture and control messages**.

2.1 Traffic Monitor

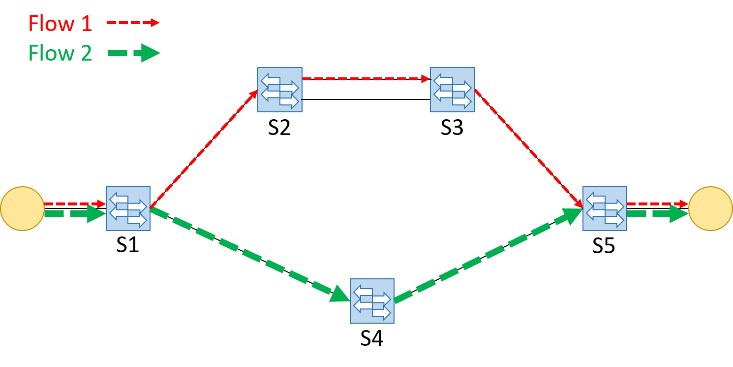
We use Traffic Monitor to monitor transmitted bytes and packets at all ports of each switch. Since we are going to select ports to deliver packets, transmission statistics are used. Traffic Monitor is also implemented with a timer, and thus polls every switch for port status in a period of time. After switches reply with port status such as port Ethernet address, transmitted and received packets and bytes, only transmission statistics are recorded and passed to Flow Manager.

2.2 Flow Manager

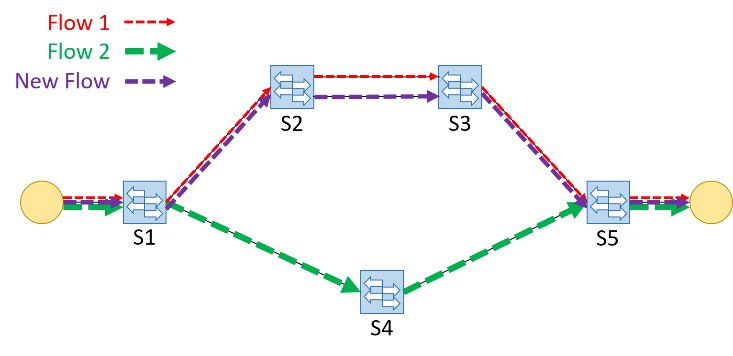
Flow Manager is used to control flow tables of all switches. It deals with packet-in packets from switches and sends back packet-out and flow modify packets. Flow Manager implements learning bridge protocol. Mappings between Ethernet addresses and ports are recorded when receiving packet-in. Besides, it also receives port statistics from Traffic Monitor and uses those to determine the port to which flows are output if multi-path exists.

3 ALGORITHM

In our design, we implement per-switch based algorithm so that we can balance the load against each link on switches along the path. Besides, any new flow first passing through a switch will trigger packet-in at the switch. For example, in Fig. 2(a) there is a simple network domain with two existing flows between two hosts connected to switch *S1* and *S5*, and *Flow 1* has smaller throughput than *Flow 2*. Suppose there is another new flow starting to transmit. When the new flow arrives at switch *S1*, *S1* will send a packet-in to the controller. By observing the traffic statistics, the controller can determine to output the new flow to the link connecting to *S2*, because *Flow 1* is smaller than *Flow 2*. Similarly at switch *S2*, the controller will find out that there are two links to *S3* where one link is used by *Flow 2* and the other is idle. Therefore, the new flow is sent to the idle link certainly. Fig. 2(b) shows the ultimate network status.



**(a)**



**(b)**

Figure 2: Path of new flow is determined in network with the existing Flow 1 (Red) and Flow 2 (Green). And Flow 1 has smaller throughput than Flow 2.

3.1 Traffic Monitor

Our algorithm for Traffic Monitor is to periodically query for traffic statistics and thus quite simple. When replies come from switches, those will trigger the handler function with signals. Traffic Monitor then calculate the throughput in previous period of time by subtracting the current number of transmitted packets and bytes with the latest values.

3.2 Flow Manager

Since every flow first passes through a switch will trigger packet-in on that switch, our load balancing algorithm is basically implemented in packet-in handler function. In our practice, we use load balancing with traffic monitoring only on TCP and UDP flows when multi-path exists for the flow, and flows with other protocols are executed with select group type in OpenFlow, which selects ports by the hash function. Otherwise, when multi-path does not exist, then learning bridge protocol is simply used to forward flows with mappings between destination Ethernet addresses and ports. Fig. 3 shows the algorithm flow chart.

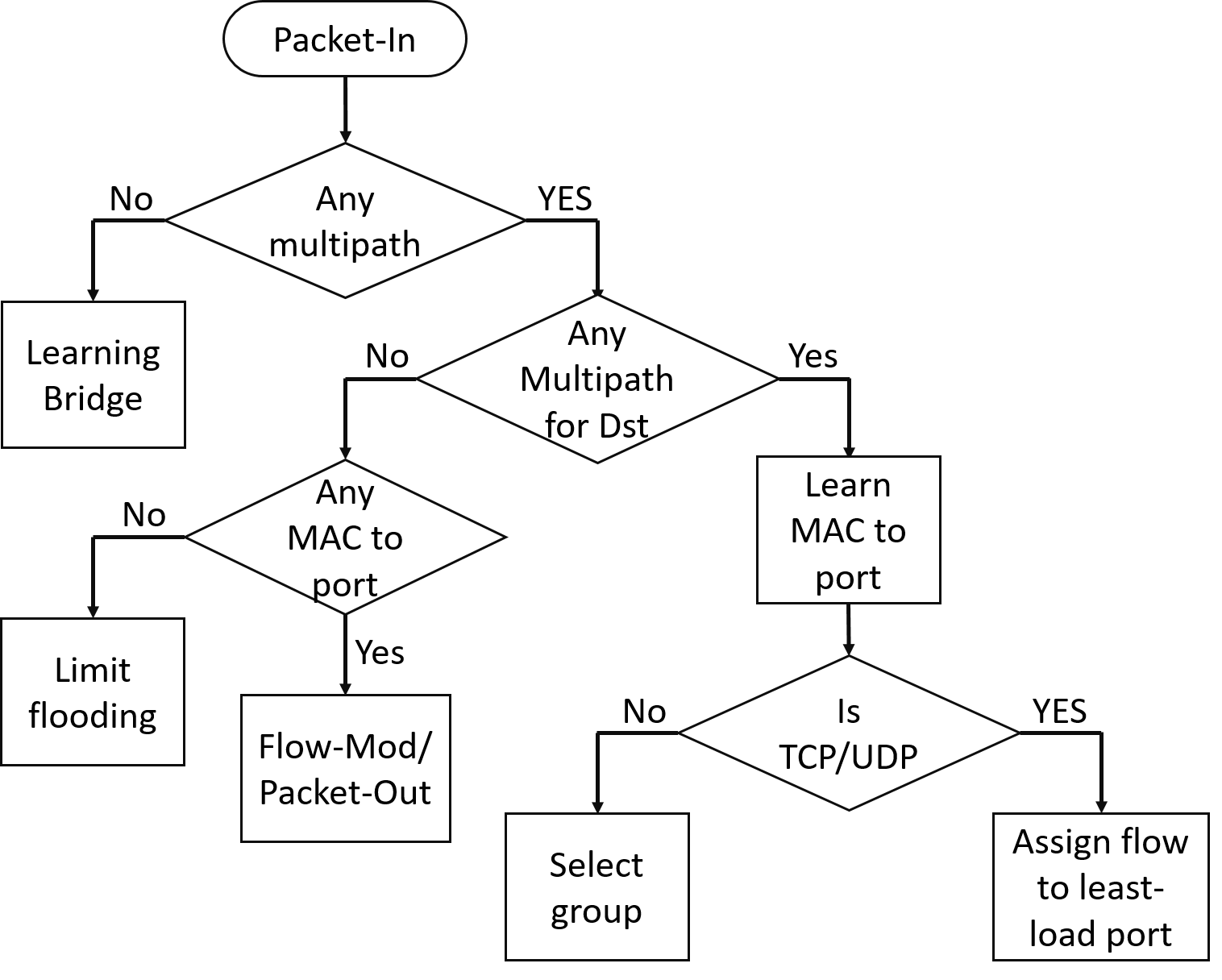


Figure 3: Algorithm flow chart of Flow Manager.

When receiving packet-in packets, the controller will first determine whether there is any multi-path on the switch sending packet-in, said *S0*. If negative, then simple learning bridge protocol is performed.

If positive, then the controller will further determine if such multi-path exists for the flow which triggers packet-in, said *F0*. If negative, the modified learning bridge protocol is implemented. The reason why we use modified version is that the original learning bridge will flood the packet to all ports on *S0* if there is no mapping between the destination Ethernet address and any port. However, from two previous conditional statements we know that there is multi-path existing although not for *F0*, which means there exists a loop and *S0* is one of nodes on the loop but the destination host is not connected to that loop. Therefore, if flooding the packet on the loop, the packets will travel through the loop and turn back to *S0* since none of nodes are able to manage the packets. *S0* will be triggered to flood packets again, turning out flooding packets is going to linger in the network and occupy the bandwidth permanently. The solution to such problem is to flood packets restrictedly. Only ports without multi-path connected will be flooded.

Otherwise, if there exists multi-path for *F0*, then first the mapping between source Ethernet address and port is recorded. The controller then judge if the flow is either TCP or UDP because in this practice we only take TCP and UDP flows into our load balancing consideration. If negative, then we use select group type to select a port with the hash algorithm. On the contrary, we perform our load balancing module with traffic monitoring to select the least loaded port.

4 IMPLEMENTATION

In our implementation, there are three type of flow rules applied on switches. The table of flow rules is shown in Table 1.

Table 1: **Flow modify fields**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flow Rule** | **Hard Timeout** | **Priority** | **Match Field** | **Actions** |
| Learning Bridge | 0 | 3 | *in\_port, eth\_dst, eth\_src* | Output |
| Select Group | 1000 | 1 | *in\_port, ethertype* | Group |
| TCP/UDP Load Balancing | 1000 | 5 | *ethertype, ipv4\_dst, ip\_proto, tcp\_dst*/*udp\_dst* | Output |

The first one is learning bridge flows, which are the mappings between destination Ethernet addresses and ports. These flows have the intermediate priority and the actions are outputting to corresponding ports. We assign these flows permanent lifetime and suppose all the devices are at fixed locations. The second one is select group flows, which are select group type method. The match fields are defined as input ports (*in\_port*) and packet Ethernet type (*ethertype*). These two match fields are not the only choice. You can define your own criteria. The select group flows have the minimal priority and 1000 seconds hard timeout. The actions are mentioned to execute Group entries. The reason why these flows have the lower priority than learning bridge flow is to support layer two forwarding. The last flows are our TCP/UDP load balancing with traffic monitoring, which have 1000 seconds hard timeout and the maximal priority. The match fields are designed as Ethernet type (*ethertype*), destination Ipv4 address (*ipv4\_dst*), Transport layer protocol (*ip\_proto*) and destination port number (*tcp\_dst*/*udp\_dst*). Since we want to preserve session consistent that packets of same sessions are forwarding along same paths, but however, we do not take source IPv4 address and source port number into consideration to simplify match fields, thus the remaining are these four fields. The actions to these flows are outputting to the least loaded links.

5 EXPERIMENT AND EVALUATION

We use 2x2 leaf-spine topology in the experiment. The topology and network configuration is shown in Fig. 4. The datapath IDs and port numbers are shown because these two configurations are required.

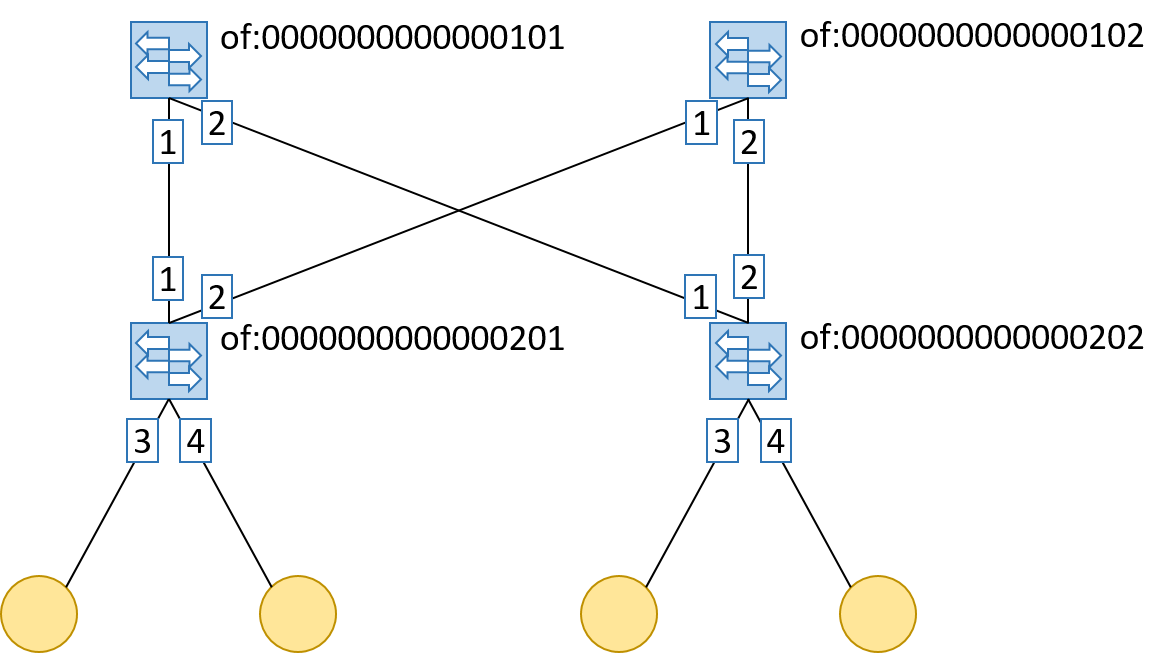
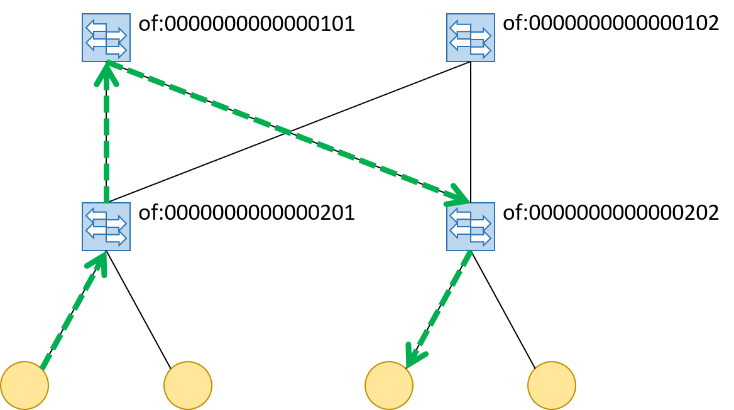
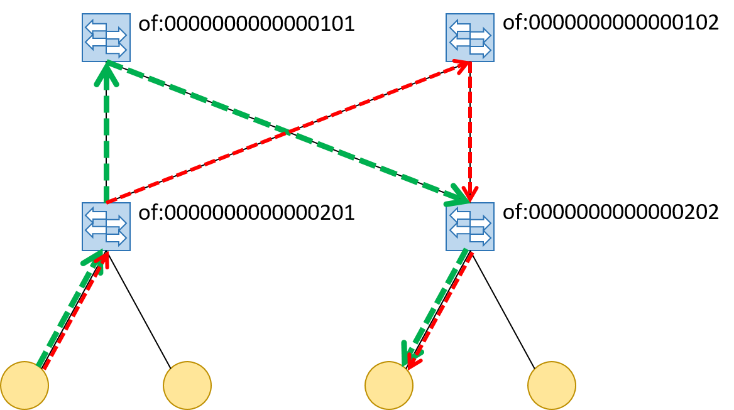


Figure 4: Experiment topology and network configuration.

We performed the experiments with an elephant flow already exiting on the 2x2 leaf-spine topology, as Fig. 5(a). Then we tried to start another flow from the same two end hosts, Ethernet type and Transport layer protocol, and check if the new flow was routed to the idle link, as Fig. 5(b).



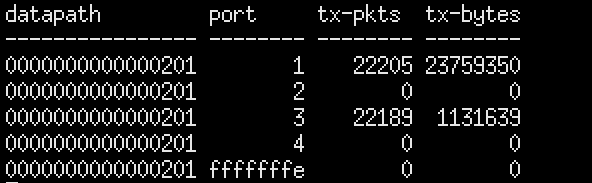
**(a)**



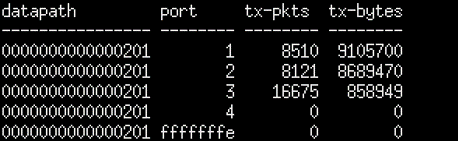
**(b)**

Figure 5: The existing elephant flow (Green) and the expected new flow (Red) in the experiment.

We periodically monitored transmission statistics for ports at the leaf switch to which the sender connected and observed the paths of flows. Initially the elephant flow existed and was output from the port 1 at the leaf switch, as Fig. 6(a). The transmission throughput at port 3 was caused by the acknowledge packets. After the new flow started, the port statistics were shown in Fig. 6(b), and there was extra transmission throughput at port 3, which should result from the new flow.



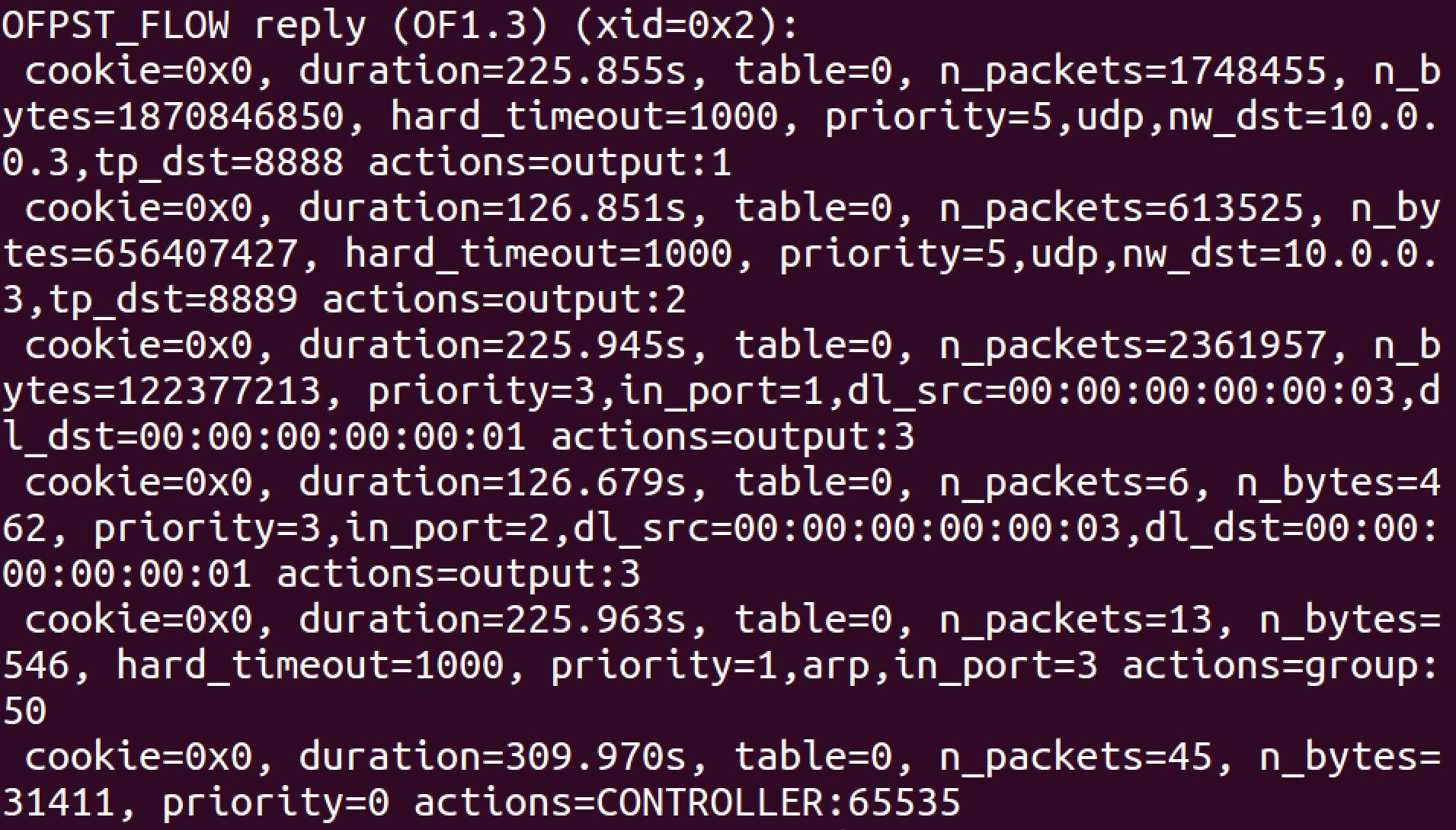
**(a)**



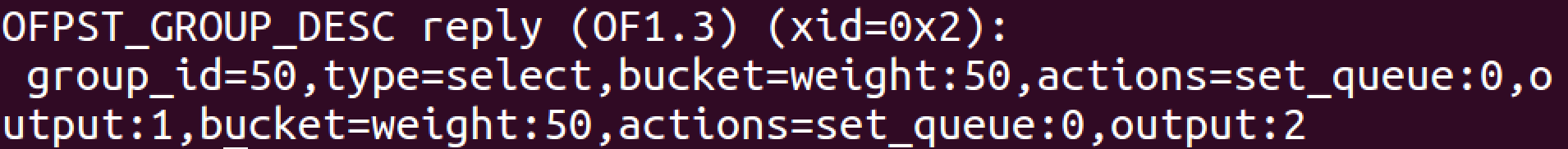
**(b)**

Figure 6: Transmission statistics for ports at datapath **of:0000000000000201 before (a) and after (b) the new flow.**

Moreover, we checked the flow table at the leaf switch, as shown in Fig. 7(a), to see if the flow entries were install correctly. The first and second flow entries were installed by our load balancing method, and the match fields were exactly what we introduce previously. Besides, the third and fourth flow entries were installed by the learning bridge protocol, which simply forwarded packets to the hosts directly connected to. The fifth flow entry was installed by the select group type method and was used for ARP packets. Since we did not implement dedicated functions to deal with ARP packets, those were forwarded on the path selected by hash functions. Fig. 7(b) shows the group table and the only entry for select group type.



**(a)**



**(b)**

Figure 7: Flow table (a) and group table (b) at datapath **of:0000000000000201**.

Hence we could conclude that the path of the new flow was determined by monitoring the traffic statistics and selecting the least loaded link, which is more efficient to balance the load for each link.

6 CONCLUSION

In our work, we present a method to distribute the flow evenly on links connected to each switch. This per-switched based load balancing approach is to periodically monitor the traffic statistics, and assign flows to the link which has least traffic at the moment. Our simple experiment show this method can efficiently solve the problem of the defect of currently used load balancing with hash functions, which is the possibility of collision. The method should be tested under the reality and more complex network topologies and this is also our future works.

REFERENCES

|  |  |
| --- | --- |
| [1] | Dave Thaler, Christian E. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, IETF, 2000. |
| [2] | GitHub - Netronome/agilio-ovs-hooks https://github.com/Netronome/agilio-ovs-hooks/blob/master/FAQ.md#q-how-does-ovs-divide-flows-among-buckets-in-a  n-openflow-select-group |