2019

Using Software-Defined Networking and Openflow Switching to Reroute Network Traffic Dynamically Based on Traffic Volume Measurements

Ihab Al Shaikhli

LOYOLA UNIVERSITY CHICAGO

USING SOFTWARE-DEFINED NETWORKING AND OPENFLOW SWITCHING TO REROUTE NETWORK TRAFFIC DYNAMICALLY BASED ON TRAFFIC VOLUME MEASUREMENTS

A THESIS SUBMITTED TO

THE FACULTY OF THE GRADUATE SCHOOL

IN CANDIDACY FOR THE DEGREE OF

MASTER OF SCIENCE

PROGRAM IN COMPUTER SCIENCE

BY

IHAB AL SHAIKHLI

CHICAGO, IL

MAY 2020

ACKNOWLEDGEMENTS

I would like to express my special appreciation and thanks to my advisor Professor Dr. Peter Lars Dordal from Loyola University Chicago, Computer Science Department for the continuous support of my master's study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my master's study. It is my honor to be able to work with him, and I have learned more than I ever expected during my time at Loyola.

I would like to thank my thesis committee Dr. William L. Honig and Professor Corby B. Schmitz for their precious advice, encouragement, insightful comments, and the time spent on my thesis. Many thanks to Loyola University Chicago.

Special thanks to my friend Albayati, thank you for supporting me, listening and offering me advice during the entire master's program at Loyola University Chicago.

Last but not the least, A special thanks to my family. Nobody has been more important to me in the pursuit of this thesis than my family. Words cannot express how grateful I am to my mother Adawiya and my three brothers Ziad, Nabeel and Abdulrahman. They are always there for me. They supported me and they are supporting me until this moment with my life and my educational journey. Without their encouragement, support and motivation, I would never be where I am today.

I sincerely pray for mercy and forgiveness for my beloved father, who passed away during my master's study at Loyola. Who taught me, supported me during my whole life and encouraged me to pursue higher levels of education. Without him I would not be at this point of my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Traditional switching and routing have been very effective for network packet delivery but does create some constraints. For example, all packets from a given source to a given destination must always take the same path. Within a traditional Ethernet network, a tree topology must be used.

Software-Defined Networking (SDN) has the potential to bypass this tree-topology limitation by placing the control of the switches and their forwarding tables under a central device called a controller. SDN also allows for sets of controllers. The controller can identify individual network flows and issue commands to the switches to, in effect, assign individual flows to specific paths. This allows different flows between the same source and destination to take different paths.

In this project we use SDN to assign TCP connections to specific paths through a network. Different connections between the same pair of endpoints can be assigned different paths. Different directions of the same TCP connection (different TCP "flows") can be assigned different paths. Paths are chosen by the controller, with full knowledge of the network topology, so there is no need for restrictions on topological loops.

Unlike with Ethernet link aggregation, our approach does not require that the propagation delays on different links are equal, or even are similar. Each TCP flow gets a single path, which eliminates link-related packet reordering.

One application of this is to achieve static load balancing. We create a specific topology in which there are multiple "trunk lines" between two host clusters; we can then spread the traffic load between the two host clusters evenly over the trunk lines.

We are also able to achieve dynamic load balancing by periodically reassigning the TCP flows to different paths through the trunk lines. This distributes the traffic evenly over the trunk lines. For this portion of the project we assumed that individual TCP connections were rate-limited, with the rate varying with time, so we could measure the per-connection bandwidths and assume these values would remain in effect for a reasonable interval.

We create the networks and switches using the Mininet emulation environment.

CHAPTER I

INTRODUCTION

Traditional switching and routing have been very effective for network packet delivery but does create some constraints. For example, all packets from a given source to a given destination must always take the same path. Within a classic Ethernet network, a tree topology must be used.

Software-Defined Networking (SDN) has the potential to bypass this tree-topology limitation by placing the control of the switches and their forwarding tables under a central device called a controller. SDN also allows for sets of controllers. The controller can identify individual network flows and issue commands to the switches to, in effect, assign individual flows to specific paths. This allows different flows between the same source and destination to take different paths.

In this project we explore ways to arrange for traffic between two node clusters to be spread over multiple trunk lines, in which different connections are usually assigned different paths. The technique we use is to assign each individual flow between two nodes to a specific path, using OpenFlow software-defined networking. The OpenFlow specifications are currently managed by the Open Networking Foundation, opennetworking.org.

We create a network topology with three sections: an “upper” cluster of N hosts and switches, a similar “lower” cluster, and a set of K trunk lines connecting the two (see Figure 1).

This models two network clusters joined by multiple trunk lines, so that traffic between the clusters can potentially travel over any of the trunk lines. This topology made identification of alternative paths straightforward. One direction for future work is to identify “trunk path” candidates from an arbitrary network, and then to apply our multitrunk mechanism to those trunk paths.

Each host in our topology is connected directly to its own “host switch”. The upper host switches connect, via a complete interconnection graph, to the upper trunk switches, and similarly for the lower host switches and lower trunk switches. The upper and lower trunk switches are connected, in pairs, by the trunk lines.

We then use Software Defined Networking, and in particular Open Flow, to route individual TCP flows over a specific trunk line. This allows for different flows to take different paths and enables a static form of load balancing.

In our simplest version, flows are assigned to one of the available paths using a round-robin approach; flow 1 is assigned to path 1, flow 2 to path 2, etc. If there are, for example, three paths, then flow 3 will be assigned to path 3 and flow 4 will be assigned to path 1.
Reverse flows do not need to follow the same path as forward flows, and, in fact, we have all reverse flows use trunk 1 (path 1).

That the flows are traveling along the intended paths can be verified both by using tcpdump on intermediate switch nodes, to observe the traffic, and also by using the `ovs-ofctl` utility which lists, for each switch, the flows as seen by that switch. We give an example of interpreting `ovs-ofctl` output below.

In a later version, we monitor each flow for its average bandwidth utilization, and then, at time intervals, reassign flows to paths based on their most recent usage so as to divide traffic among the trunk lines into reasonably equal shares. For this portion of the project we assumed that individual TCP connections were rate-limited, with the rate varying with time, so we could measure the per-connection bandwidths and assume these values would remain in effect for a reasonable interval. We were then able to observe the reassignment of flows to paths as the flow rates varied with time.

We create the networks and switches using the Mininet emulation environment.

Overview of our Network

We choose a specific topology in which there are a fixed number of "trunk" paths (lines) between two clusters of nodes. While most of the techniques here apply to general topologies, it can be difficult to calculate the set of all paths between pairs of nodes, so we standardize on one particular topology.

Our topology consists of 2N hosts divided into two clusters of N hosts each, and K trunk lines between the two clusters. This is related to the "doublebell" topology in which two clusters of hosts are joined by a single link. Usually we chose N=5 and K=3, making 10 hosts, 16 switches and 3 trunks. We used only one controller. Five hosts and five host switches are in the upper side of the network, while another five hosts and five host switches are in the lower side of the network. There are six trunk switches connected to the trunks, three in each side.

There is no assumption that the trunk links are in any way similar. They need not have similar propagation delays, they need not have similar MTUs, and their endpoints (eg s1, s2 and s3) need not be geographically close. As long as OpenFlow can forward packets along the trunk

links, they need not all even be Ethernet. In particular, there is no assumption that the trunk links can be aggregated at the Ethernet layer.

Host names start with the letter h, followed by a number. Within Mininet, switch names start with the letter s followed by a number, known within Mininet as the switch “dpid”; in the diagram below, only the DPID number is shown. The controller is referred to by the letter c. The hosts names start from the first host which is h1; the situation is the same for the switches. Switch numbering starts with the six switches which are connected to the trunks from each. The switches names will depend on N and K in the lower and upper side of the trunks. For the upper side the host switches are 2K+1 to 2K+N and the trunk switches are 1 to K, for the lower side the host switches are 2K+N+1 to 2K+2N and the trunk switches are K+1 to 2K. Controller c is for all switches.

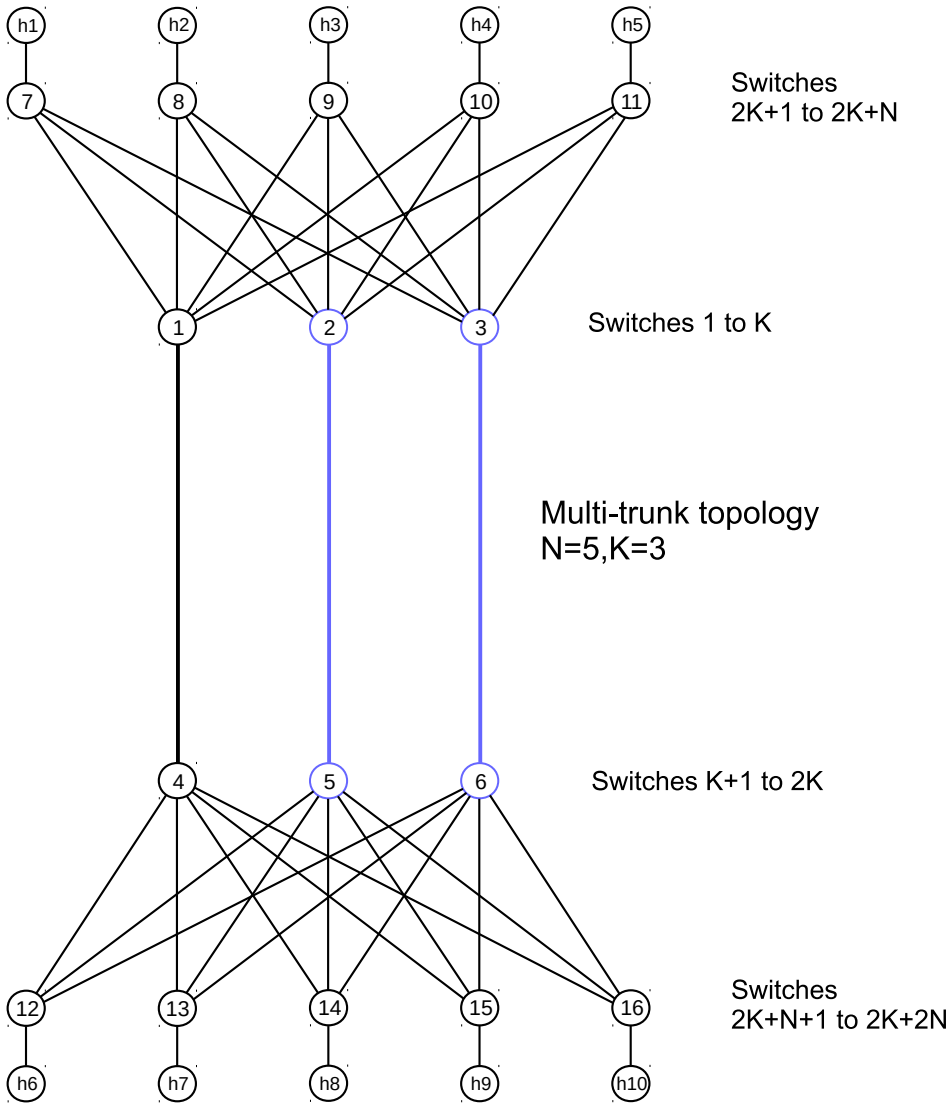

Figure 1. Multitrunk Topology.
Trunks are S1-S4, S2-S5 and S3-S6.
The blue trunks S2-S5 and S3-S6 are used only for assigned flows.
All management traffic (ARP and ICMP) takes S1-S4.

Some switches, in particular those shown with black circles in the diagram above, will behave at least partly as learning switches. The blue switches will receive traffic only when the controller c has assigned a flow to a path through that switch.

A TCP flow is one direction of a TCP connection.

Here is the basic strategy for handling a new TCP connection from the upper cluster, say, to the lower cluster. It is implemented in the SDN controller:

- Choose a trunk link, based on current traffic patterns. A simple choice is to select trunks according to round-robin scheduling.
- Determine the full path from the source node to the destination node, utilizing the selected trunk link. In our setting, knowing the host-facing switches and the trunk effectively determines the path.
- Create the reverse path, if necessary. In our work, we chose to have reverse traffic always use the s1−s4 trunk, partly because the reverse flow carried only ACKs, and partly to demonstrate that having the two flows of a TCP connection be routed separately was entirely feasible.
- Issue the necessary OpenFlow commands to the switches along the paths so that traffic for this particular new TCP connection is forwarded along these paths. We always issued these commands from the far end of the path towards the front, so, in principle, once a switch had the flow entries for a TCP connection, all downstream switches would already be configured. The idea was to avoid having switch A know about a connection and so forward the packets to switch B, which would not yet have any idea what to do with the packets. In practice, thread race conditions and communications delays meant this situation was not completely avoidable.

We are now ready to outline our multitrunk.py POX program, which assigns TCP flows to one of the available trunk lines in our topology. We will return to further details, below, in Section Basic multitrunk: Assigning Connections to Trunks. The trunk line assigned to a TCP

flow is chosen in round-robin fashion. Only flows from top to bottom are handled this way; the reverse flows from bottom to top all are sent through the trunk s4 − s1. This corresponds to the idea that bulk data traffic is from top to bottom; the reverse flows are small and may carry nothing but ACKs.

Each individual TCP flow will have an OpenFlow match rule specifying

- The source and destination Ethernet addresses
- The source and destination IP addresses
- The source and destination TCP port numbers

These OpenFlow rules will be installed in each switch on the path. Once a TCP connection is made, and the OpenFlow rules are installed, then traffic for that connection will not be looked at further by the controller.

Much of the code here is specific to our topology, but it generalizes in a straightforward way to other topologies with multiple “trunk routes” from one cluster of nodes to another. Matching on both source and destination is necessary so that each TCP flow can be routed individually; matching on destination only would force all flows to the same destination to take the same path. Including match rules for the Ethernet addresses is not strictly necessary.

One of the problems to be solved is the network-discovery problem. Switches discover their immediate neighbor switches, and the ports by which these immediate neighbors are reached, through a special discovery protocol modeled after the spanning-tree protocol. This runs as a separate phase, before data traffic is allowed. This does not help, however, with discovering the host neighbors a given switch has, and the ports, as hosts cannot be required to participate in a discovery protocol. We solved this problem by using ARP queries. Normally, ARP queries

simply identify the destination Ethernet address, given the IP address, but we were able to leverage that process to obtain information about the host relationship to the switches as well. Host ARP queries and responses are used to identify the switch to which the host connects. Because ARP traffic (and ICMP traffic) is not assigned a specific path, we created a spanning tree and restricted ARP and ICMP traffic to that spanning tree. ARP traffic cannot be assigned a path, the way TCP connections are, as the full information needed to construct such paths is not available until after the ARP exchange has completed.

The OpenFlow switches we are using can be programmed to forward packets based on any of the following:

- Destination MAC address
- Source MAC address
- Destination or source IP addresses
- Destination or source TCP ports

[10]. Classical Ethernet switches would forward traffic only on the destination MAC address. With the OpenFlow functionality above we can create separate forwarding rules for each individual TCP flow. The two flows making up a TCP connection can be forwarded to take different paths.

The three trunk lines of Figure 1 are visually parallel, and this may suggest that the trunks can be aggregated at the Ethernet layer. However, not only can the trunks have different propagation delays and different MTUs, as described above, but the trunks do not necessarily have to be “parallel” in any practical sense. For example, our basic techniques would apply to the

following network, with four host clusters and four trunk lines. TCP connections from cluster 1 to cluster 3 can be routed via trunk1−trunk2 or via trunk4−trunk3.

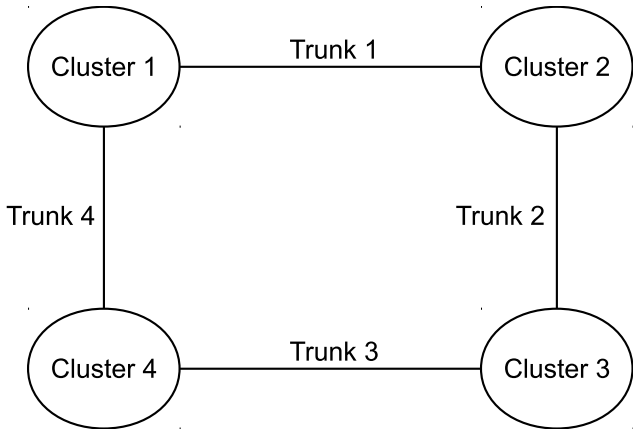

Figure 2. Example network in which trunk lines are not parallel. Traffic from Cluster 1 to Cluster 3 can still be divided along the Trunk1-Trunk2 and Trunk4-Trunk3 paths.

The changes necessary for our program to handle a new topology like this would be relatively straightforward. Our program does depend on having knowledge of the trunk topology. The program uses this knowledge to calculate paths, as in the first and second steps of our basic strategy, above. If the topology is changed, knowledge about the new topology would have to be incorporated in the controller to enable trunk and path selection. Once that is done, however, the basic mechanism for implementing the new paths using OpenFlow is largely the same.

Background on Ethernet

Datagram forwarding is the method used in most Ethernet Switches. These switches do that by establishing their forwarding database tables without the help of a network controller. Originally, Ethernet switches had to be able to work as drop-in replacements for hubs, and so could not rely for their operation on any control communication with the associated hosts, and so

cannot rely on any specialized protocol for operation. Instead, the forwarding is based on the signals coming from other switches. [1]

Normal switches will build a forwarding table of (destination Mac address, next_hop) pairs, where an arriving packet addressed to a given the MAC address is forwarded out the corresponding interface. For switches, the next_hop represents the output port of the current switch. Switches learn about destination Mac addresses gradually. They must fall back to flooding if they didn't have any entry in their database for a particular destination; otherwise they could not forward such packets. Unknown Unicast Flooding is a behavior where regular switches Fall back to flooding where switches send the packets that arrive at all the hosts on the network except for the host where the packet came from. Switches build their tables without the help of switch-to-host or switch-to-switch coordination through Fall back to Flooding.

Switches learn how to reach new destinations by noting the source address of arriving packets, and also the arrival interface. The switch then assumes that interface would be used to reach that address in the future, and adds that (address,interface) pair to its forwarding table. This helps the switch learn how to get to new destinations without flooding. For example, if a packet arrives via interface F from source address S, the table will get the entry (S, F).

If a packet addressed to destination D arrives at a switch S via interface I, the switch looks in its table for a record (D,J). If an entry (D, J) was present where J does not equal I, then D will be reachable through the Interface J. In that case the packet will be forwarded through interface J. If J equals I, meaning the packet came in to the switch via the same interface as the outbound interface, then the packet is not forwarded further. That might happen if the interface happened to be connected to the same Ethernet link used by another switch S2, which used that

link to get the packet closer to D. The packet is delivered to D, but also to S, which in this case represents the packet having traveled too far. The unknown-destination fallback to flooding will be in practice in the case of J where J does not equal I. When there is no entry is there for destination D, the packet will be flooded out of the switch through all of the interfaces J not equal to I. As switches find out the interfaces to use to reach active destinations, this need for the alternative to fall back will be needed less often after a short while.

Switch forwarding-table entries are wiped out after a short period of no use.

If the destination address is the broadcast address, each switch will always flood the packet. If the destination address is a multicast address, the switch may simply flood the packet. However, the switch may also attempt to keep track of multicast groups, via IGMP snooping, in which case the switch will forward the packet only out those interfaces that are part of the multicast tree.

Following the principles of layering, MAC addresses are generally invisible to the application layer; applications identify other hosts using their IP addresses. If a remote host is on a different LAN, the local host will not know the remote host's MAC address at all. For hosts on the same LAN, the correspondence between MAC and IP addresses is made via the ARP protocol; one consequence of ARP, however, is more broadcast traffic. Put in mind switches can’t find neighbors who are connected directly to them until they receive packets from those neighbors.

Once all the switches along a path have forwarding-table entries for a given destination, then the switches each directly forward the packet along the path. There is no more need for flooding. Each packet is directly forwarded, not sent to unnecessary links and not flooded now

that switches know where the destination host is located. Loops cannot be allowed in a switched-Ethernet, so paths must be unique. Endlessly forwarded packets will be the result of an existence of a loop in the network where the packet which is sent to unknown destination will be stuck in the loop till the end of time.

The learning algorithm also ensures some privacy. Once a packet is forwarded along the direct path from source to destination, with no flooding, then no other hosts can eavesdrop on the packet. In the past, unswitched Ethernets were known for cases where one host machine could eavesdrop on all the passwords of the network, Ethernets which are fully switched do not have this problem. Although switches have this characteristic of protection against spying, it can still be breached by the act of flooding the network with fake source addresses. This will force the switch to drop much of its learned forwarding table and go back to fallback to flooding.

Forwarding tables of large switches usually have a room for about 10,000 to 100,000 entries and it is usual that networks which are fully switched will have much less than 100,000 entries. Since the topology must be loop free, this results in this switch size limitation. Broadcast traffic will be a larger and larger portion of the traffic as the switched Ethernet gets bigger and bigger in size, as the broadcast traffic must travel everywhere but most unicast traffic does not. This makes the business or company which is using this network to change their network to include routers in its design. It is a good practice if you have one LAN or VLAN to have less than 1000 hosts for it.

The degree of support of internal parallelism is a major difference when it comes to how much an Ethernet switch will cost. The question here is can the switch send for example three packets simultaneously if in a scenario where three packets are concurrently arriving on ports 1,2

and 3 and they are going out via ports 4,5 and 6 respectively. This scenario in the case of a simple switch is that these packets will be bottlenecked as this switch has only one CPU and one memory bus. At most two concurrent transmissions can happen for a small five-ports switch; such a switch is able to handle this much of parallelism with ease.

CHAPTER II

BACKGROUND AND TOOLS

Spanning Tree

Early switches were designed so a packet would circulate forever if it was traveling along a loop in the network. Such a loop has a devastating effect on network throughput. But a positive side of loops is they provide redundancy so if a link goes down the connectivity will remain; this characteristic is great to have. Here where the spanning tree algorithm comes to play. [4] This is a switch-to-switch protocol where it creates a subgraph of the graph of the switches' connections, this subgraph allows reachability and it is in the same time has no loops. Connections that are not a part of tree are deactivated by the spanning-tree algorithm even if those links are along the shortest path between two hosts; this happens a soon as a spanning tree has been established. In case that a connection which is a part of a spanning tree goes down which may result into splitting the network into two then in that case some of the previously disabled links will again be used as a new spanning tree will be established. [11]

In a network in which all switches support the spanning-tree algorithm, a question arises whether the switches can create forwarding tables which are loop free and, at the same, make use of all the links. The problem here is not with the unicast traffic but with the forwarding of the Ethernet broadcast traffic. Broadcast packets could circulate in the network endlessly if there are loops in the topology, even if unicast packets are routed along the shortest paths to their destinations.

Historically, the topology of switch connections was potentially complicated by the existence of Ethernet multi-tap coaxial cable segments and by the existence of hubs. Both multi-tap cable and hubs are now considered obsolete. In designs that lack the presence of non-switched ethernets and hubs, the edges and nodes of the topology are simply the hosts and Ethernet links. If pieces of non-switched multi-host Ethernet exists, however then each of these pieces will be a topology node with a topology edge to each switch where it connects to directly. Note that switches that do not participate in the spanning-tree algorithm will be considered as hubs by the switches that are running the STP algorithm. However, almost all switches today do support the spanning-tree algorithm.

Every edge that connects to a switch will connect through an interface. The ID of a switch (every switch has a unique ID) is the smallest ethernet address. The goal of the spanning-tree algorithm is to be able to forward packets to any destination and in the same way is to stop paths that are unwanted and unnecessary or redundant.

The node with the smallest ID will be selected as the main node (the root node). Any time two switches have a connection between them through an ethernet cable while these same switches also have direct connections to the main node (root node), the direct link to the root node will be used. If there is more than one direct link, then the one used will be the one with the smallest ID. Although recently a cost factor is used for network implementations where this cost factor is a bandwidth inversely proportional meaning lower cost has a bigger bandwidth; the simplest path cost measure is the hops number. In case of an occurrence of an outage then a recalculating will occur for the spanning tree which means that this process is dynamic to be adjusted in case of unexpected events. Also, if it happened where an outage, for example,

divided the network into two networks, then the two pieces will each have their own spanning tree.

All switches send out a “Hello” message − technically known as a BPDU (bridge protocol data unit) − from all its interfaces. These are sent from the interface ethernet physical address to the ethernet multicast address 01:80:c2:00:00:00. Keep in mind that a unique physical address for each interface is not otherwise required for switches.

Bridge Protocol Data Units or BPDUs have the path cost, the root node and switch IDs where those messages (BPDUs) are not sent to unknown destination as they are known by switches where switches look for the short route to the root, the root switch (the switch with lowest ID) and finally a tie breaker (a path to the root via a close by switch with a lower ID. If there more than one port (let’s say two ports) connects to the switch, then the tie breaker will become the port number.

If all the connections have the same bandwidth will let us have a faster paths preference. BPDUs are sent out of all the interfaces of a switch in the case of that switch seeing a new root candidate. Distance will be indicated because of these sent BPDUs. Also, interface leading to the root will be included. The result of this will be that the switch learning which of its ports will be used for reaching the root, the neighbor switches connected to each port of his ports and its path to the root.

Now all or some of the interfaces of the switch can be pruned by the switch itself. All interfaces that are not enabled by the following rules will be disabled by the switch:

(1) Further-out switches ports where theses ports job is to reach the root will be enabled. Also, if the port which reaches the root will also be enabled.

(2) When segment and other segment neighbor switches connect through the remaining port, then this port will be enabled under a condition. The condition will be the switch cost has to be minimum to the root among the neighbors. In case of a tie between two ports it will be the port with the smallest ID, if the tie happens between neighbors in that case it will be the neighbor with the smallest ID.

(3) If there were no switch neighbors that are connected directly to the port, then the port will be enabled as well as it is probably connected to switch neighbors directly.

Rule 1 will establish the spanning tree. In case that the root will be reached by S3 through S2 then this rule will be certain that the port of S3 is open towards S2 also the corresponding port of S2 to S3 is also open (the rule that we are talking about is rule 1). All connected to multiple switches network segment will get a unique path to the root that is made sure by rule 2, for example if S3 and S2 segments-neighbors who are connected another segment for example segment N then in this case the segment with the smallest number “wins” meaning that S2 will have its directed to the root port opened (enabled) while S3 will not. Creating host nodes paths on N (segment N) will be the concern as S3 and S2 will use rule 1 for creating their paths. Retaining connectivity for any sub segment is ensured by rule 3 and this rule effect will extend to include all the hosts who are connected to switch ports directly.

Software-Defined Networking and OpenFlow

Large Ethernet networks using the spanning-tree algorithm generally have many pruned, or suspended, links. These suspended links provide redundancy or backups to the network when switched from being suspended to back in service. These suspended links will give a huge help in the cases of failing of any part of the main ethernet. Making first-class use of all the links, and

the avoidance of disabling links in a high-performance environment, is a question still to be answered.

The main difficult area here is the packets addressed to unknown destinations. Traditional switches rely on fallback to flooding. The use of broadcast (flooding) is difficult in a loop topology network.

SDN works by using a controller and avoids the common distributed learning algorithm. Using a controller helps providing redundancy and forwarding in the same time. The controller has a full view of all the network information that any participating switch has discovered. When a switch sees a packet with a new source or destination, that information is sent to the controller, which can in many cases assign the full path the packet is to take. [10]

This is done where each participating switch has the forwarding mechanism applied on under the authority of the controller. In this context the controller can be either set of distributed nodes or can be a single node on the network. The purpose of the controller is the management of the forwarding tables of the switches.

In the beginning the controller probes all the switches for information about links to adjacent switches. With this information, the controller can build a spanning tree for broadcast traffic through that network. Broadcast traffic can be forwarded on the constructed spanning tree links as it is ordered by the controller. Unlike traditional switches the links which are not a part of the constructed spanning tree can be still serving to deliver to known destinations. Broadcast traffic can be identified by destination address – the broadcast address – or by protocol: in basic IPv4 networks, the only broadcast traffic is normally ARP and DHCP. In our networks, the only broadcast traffic is ARP “who-has” queries, as DHCP is not used.

Reports of packets to unknown destinations are delivered to the controller by the switches, at which point it is the controller's job to determine what the next step will be. One way is by forwarding traffic to unknown destination along the same spanning tree which is used for broadcast traffic. This allows the use of loop topologies alongside with the safe existence of fallback to flooding.

When a switch discovers a previously unknown source, the controller can inform all other switches on the network the best route to the new source. The controller can also inform all switches at startup that it is their responsibility to report the addresses of any new source to the controller.

For security purposes SDN controllers can disallow forwarding packets between nodes; this behavior is like the work of firewalls. Example of that if there is a bunch of computers belonging to user A and there is another group of computers belongs to user B, controllers can be configured to a degree that no computer from user A group can forward packets to any computer from user B.

Controller software can be programmed locally allowing control that is considered very local of network functionality, but at many networks the controller's built rely on standardized modules. Combining Ethernet with loop topology alongside the network's functionality control is the most important feature of SDN.

OpenFlow Switches

The controller's ability to inform switches of how to forward packets is the most important characteristic of it. OpenFlow switches are a packet-forwarding architecture, created by Open Network Foundation and considered to be a specific standard of SDN. [10]

Flow tables are the components which OpenFlow forwarding is built from. Packet match fields and groups of packet-response actions, if the match was successful, are the main components of a flow-table. Forms of the packet-response actions are:

(1) Forwarding the packet out of a specified single interface.

(2) Dropping the packet.

(3) Forwarding the packet to the controller (usually so the controller can make a forwarding decision).

(4) Flooding the packet out a set of interfaces.

(5) Matching the packet at another (higher-numbered) flow table.

(6) Modifying some fields of the packet (not used in this project).

A single entry for the Ethernet address destination can be an example of match fields; this corresponds to traditional forwarding on destination only. However, OpenFlow match fields can also include the ingress interface number, or it can include any other packet bit-field. An example of this is the use of IP addresses where the forwarding can be performed partially or completely using the IP addresses instead of the Ethernet addresses. The case of forwarding by IP addresses (instead of Ethernet addresses) will make the OpenFlow switch act superficially as a layer 3 device, that is, a router. However, OpenFlow switches still act at the Ethernet layer (for example, the Ethernet destination address is not changed). Theoretically, OpenFlow switches may skip some header updates required by routers, such as decrementing the IPv4 time-to-live field; it is up to the OpenFlow programmer to enable these updates. Note, however, that failing to decrement the TTL field may cause severe network difficulties if the traffic ends up in a loop.

OpenFlow allows different TCP connections to be sent along different paths, using matching rules involving the destination TCP port and destination IP address. Splitting large volume flows and real-time traffic is done using policy-based-routing abilities, this is all done in SDN (software defined network) settings.

Forwarding packets to all the interfaces except the one that the packets came from is known as flooding and this flooding technique is how OpenFlow switches normally handle broadcast packets. There is an option that can be applied on interfaces which sets those interfaces into NO_FLOOD mode, meaning that packets designated for flooding will not be sent through theses interfaces (packets designated for flooding can be broadcast for anything else). Implementation of spanning trees for broadcast traffic is done through setting some interfaces to NO_FLOOD mode. Flow table entry is not a requirement for broadcast flooding and unicast traffic still can be performed using interfaces with NO_FLOOD mode enabled.

In the case of a packet matching two or more flow table entries, a priority value is assigned to match fields. The winning entry will be the entry with the highest value. Flow-table entries with no match fields are called the table miss entries; because they have no match fields, they are considered to match every packet. Their priority value is usually 0, so that any other, more-specific matching rule will be used if available. Packets that match no entries at all will be dropped or in some cases will be forwarded to the controller.

Common OpenFlow packet modifications include updating the IPv4 checksum and decrementing TTL, making the switch act more like a true router, or VLAN coloring. Packet-modification rules are all included in the flow table instructions which may contain managing or modifying packets. Flags, last_used time and counters are also included in flow tables beside the

match fields instructions which already a part of flow tables. In case of disappearing of no matching packets these packets will be all removed, and this function is done by last_time used. QoS (Quality of Service) is enabled on the OpenFlow switch using counters, examples of QoS is bandwidth limiting.

Learning Switches in OpenFlow

We start with a naive (and incorrect) approach to implementing a learning switch in OpenFlow. Match rules will be based on the destination MAC address only and will correspond to the normal destination-based forwarding table. If there is no match, the arriving packet will be reported to the controller; this is the default OpenFlow behavior. The controller will then create in the switch an appropriate forwarding match rule, if the next hop can be identified, or else flood the packet out all other interfaces, if the next hop cannot be identified.

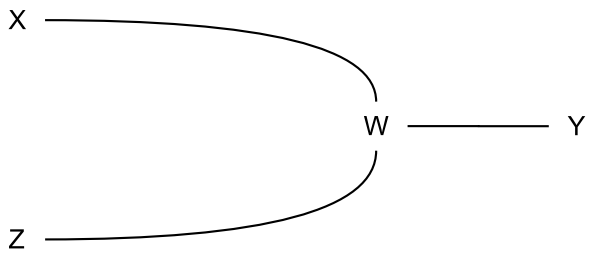

Figure 3**.** Ethernet Learning Example.

In the diagram above, the switch W monitors a packet going from host X to host Y. What the switch will do is that it will report this sent packet to the controller. We will assume that the controller knows how Y is connected to W − likely because Y has previously sent a packet − and so the controller will create a flow entry in the switch W matching all packets with destination address Y. Now consider the case of a packet arriving to switch W from another host Z. This

packet would be forwarded by the new rule since it would match the flow entry for destination-address equal to Y. Because there is a match, there will be no need for the packet to be forwarded to the controller. But this means that the controller will never install a flow entry for Z as a destination, as it is will never learn Z's address.

A better solution to implementing a learning switch in OpenFlow is to use pairs of destination and source addresses. [7, 8] In case of a packet traveling between hosts and using the switch W in his moving between the hosts, when this packet doesn't match any existing flow entry at switch W, it will be reported to the controller. Reporting it to the controller will help the controller to know that host X can be reached through the port which the packet arrived at W. We discuss the standard POX implementation of this approach below, in section Example 1: l2_pairs.py.

Imagine host X broadcasting a packet meant to be send to host Y or it sends it to host Y. Switch W will report this sent packet to the controller and the controller in return will send it back to W to be flooded, all of that because the switch W table is empty. This will enable the controller to learn that host X could be reached from switch W through port 1.

Let's continue the scenario by saying that host Y will respond to host X by sending another packet, when this packet from host Y reaches switch W, the switch W still doesn't have entries (flow table entries) so in this case the switch will report this packet to the controller. Here the controller already knows that the switch W can reach host X through port 1 and it knows (the controller) from the packet sent from host Y to host X through it the host Y could be reached via port 2. With all this knowledge the controller now can install in switch W two flow tables where:

dst=Y, src=X: forward out port 2
dst=X, src=Y: forward out port 1

Packet arriving from a third host (let's call it host Z) trying to get to host Y through the switch W will not be forwarded as it has a source address that is different than the source address of host X, even though the destination address is the same. Since it will not be forwarded it will be then sent to the controller. The controller will install rules for:
dst=Z,src=Y and dst=Y,src=Z

After host Y reply to Z. The controller will never learn how to reach the host Z and will never be able to install rules for host Z if packets from host Z were never reported to the controller. This situation can happen if flow rule for destination Y only (dst=Y) was set in the switch W.

In case of the availability of single flow entry table then the OpenFlow learning is optimal using the pairs method. Scale will be a problem if we went with this method because we will need 100 million pairs of flow table (destination,source) entries to represent all of the forwarding possibilities if there are 10 thousand network addresses in all.

It is possible to implement the pairs method, using the python programming language, Pox controller and Mininet network emulator, at least when the total number of network addresses is small.

An alternative OpenFlow learning algorithm is to have more than one table, one for the source and another for the destination address. This approach is much more scalable. In this scenario the controller doesn't have to remember forwarding information that is partial as it is on the previous version where we discussed after the controller receives the packet from the host X. This is called the multiple flow tables approach. We discuss the OpenFlow implementation of this approach in the section Example 2: l2_nx.py, below.

Any steps will be applied after the match against the first table happens when the packet arrives. A second group of steps may emerge as one of the steps in the first group requires that the packet to be matched against the second table. Let presume that A0 is the first table and A1 is the second table while keeping on repeating the previous example of E−>F and F−>E.

In the beginning rules that are low priority match is switch S are the rules which are to be installed before the arrival of any packets to the switch:

A0: match nothing: flood and send to A1
A1: match nothing and send to the controller

The purpose of low priority is to make sure of the use of better matching rules when they are available. Since there are no packet fields to match, these low priority rules are considered to be default rules as they do match all packets.

The previous rule of table A0 means the packet will be flooded to B when the packet goes from host X to host Y. The controller will install the rules below in switch S after knowing that the rule for table A1 means that the packet must be sent to the controller. The rules of the controller are:

A0: match dst=X: forward through port 1, send to A1
A1: match dst=X: do nothing

The first rule will match as host Y will replay to host X and the packet will be resubmitted to A1 as the packet will be forwarded by switch Z. Here the previous A1 rule will not match. The packet afterwards will be sent to the controller where the only match will happen in the default rule. This will result in the creation of two additional rules in the switch Z as the packet will be sent to the controller, these additional rules will be:

A0: match dst=Y: forward via port 2 and sent to A1
A1: match src=Y and do not do anything

A0 rules at this stage will make sure that correct forwarding is being carried on while A1 rules will make sure that no packets get sent to the controller from this flow of packets as host X and Y continue to talk to each other.

The addresses list in the two tables A0 and A1 will always be the same as the controller will always create in both table the same address. Source addresses will be matched by A1 while destination addresses by A0.

We can also use the concept of multiple flow tables to have switches that makes decisions of QoS (Quality of Service) prioritizing, where using the priority which is determined by the second table the packet will be forwarded out of a port basing on the determination made by the first table. This can be achieved by listing all the source and destination addresses pairs combinations within a single table, but this approach is considered to have a lot of entries.

The use of controllers as firewalls could be difficult to implement but with the new concept of the understanding of IP addresses through OpenFlow it became easier. It became easier to block traffic like what a router might do between two different IP subnets. Except for specific hosts pairs who are using specific protocols OpenFlow let us use it to block all other kind of traffic. There are a variety of examples on different situations where for example we can make a user contact a database for example while in the same time blocking all other traffic going to that database. This communication between the user and the database will be done using a TCP port.

Mininet

The Mininet application allows multiple “virtual” hosts and switches to run on a single host. This is achieved by using Linux network namespaces and virtual Ethernet links. Each

virtual host runs in its own network namespace, with its own network interfaces. Network interfaces can then be connected by point-to-point virtual Ethernet links. Mininet provides tools to allow for the easy configuration of such virtual networks, allowing for network experimentation. [9]

The consumption of resources by these virtual network hosts, even if the allocated resources to these virtual machines (nodes) are minimized, is still considered to be high to the degree it is only possible to have a limited number of nodes in the network that we created. The goal is to have nodes that use sufficiently few resources that we can have, on one host computer, as many as 1000 virtual network nodes running at the same time.

MININET was created for the purposes of SDN (software defined networking), but it is also works great for traditional networking modeling including the experimenting and demonstrating purposes. The MININET system help us achieve what we want meaning it helps to have a lot of virtual nodes running with the least of resources.

A Mininet node corresponds to a network namespace, or “container”. The Mininet software creates the namespace, and then one process within that namespace to represent the host CPU. Network namespaces are inherited so all subprocesses will inherit the view of the container it is within. Virtual Ethernet interfaces will be constructed to tie virtual container interfaces together according to the rules provided for the network layout.

The same filesystem by default is shared by all the Mininet containers for efficiency. Problems could arise sometimes with applications that require files in specified locations with individualized configurations. Mininet can be configured to accommodate individual filesystems, but in general the shared setup will be simplest and least resource intensive.

Protocol implementations and applications before they can be used will need to be ported to run in the simulator in the environments that use simulation. Emulation in networking environments has an advantage compared to simulation which is that any network software available on the host system is available to all the emulator nodes. It is worth mentioning that even with this advantage of emulation, there are other drawbacks; for example, we can't emulate a speed in a link that goes beyond the capabilities of the hardware that supports the link. Another drawback is a network can be slow due to the complexity and size of the network and the need for one CPU to manage all the elements, even before traffic volumes become significant.

Installing Mininet

Mininet must be run on Linux because the network-namespace rules it uses are specific to Linux. It is recommended to have a virtual machine system running on your operating system to install Linux on it. This idea is recommended even if your operating system on your laptop is a Linux operating system since it is a good idea to separate a Mininet project world from the host system's world. Mininet could influence the operations of the user's computer as for example Mininet virtual switches often interfere with the laptop suspend feature.

The Mininet site contains a virtual machine with Mininet preinstalled. The package will be downloaded as a Zip file and after it is being unzipped it will give us two files; a file with a size of 2GB (more or less) where this file is the virtual disk image and comes in a .vmdk extension. The other file represents the virtual machine specification and it will have an extension of .ovf

Two virtual-machine options are recommended to download and install Mininet, which are VMware Workstation Player and Virtual Box. Both free and supported. Creating a new

virtual machine (a Linux virtual machine) is easier, next select the option for it to use a virtual disk that is already existing, lastly assign disk for it which will be the downloaded .vmdk file.

The privileges needed to run Mininet can be gained through the sudo command which will obtain the root privileges. When the login happens using “Mininet” as the username and the password. Know that via command sudo bash you can stay logged on as root with keeping the terminal window open, and it is recommended to enter the commands one at a time. For example: `sudo python switchline.py`.

The use of the preinstalled version which comes bundled with the POX controller for Openflow who is a very useful software is considered the way to go about this, installing and setting up a virtual machine that runs Linux from scratch (Ubuntu distribution for example) and then having Mininet to be setup on it is another way to do this.

Graphical interface for the desktop does not come with so if a user wants it then he or she will have to put in mind the “cost” of having it which is represented not in money but in using an external 4GB just for it. The user will have to run `apt-get install ubuntu-desktop` to have the full ubuntu desktop. Also, the user must put in mind to apply these steps while he or she are logged in as **root**. Another way to have the user interface is by running the command `apt-get install xinit lxde` which will install lxde desktop environment. This second way is a Mininet site recommendation as it is half the size of ubuntu, so it is a lighter weight alternative.

Graphical text editors can be installed but with lxde comes the leafpad editor which is always bundled with it as the standard text editor.

For Linux systems that are Debian-based a recommendation before starting anything is to run these commands:

```
apt-get update
apt-get upgrade
```

The command startx must be executed if the user wants to have the graphical interface where it is executed right after the installation of the ubuntu desktop.

Virtual machines operating in one window on a host system often “capture” the mouse, thus disabling access to other host-system windows. This can be quite annoying, but luckily there are software packages that are designed to overcome such virtual machine behaviors. In the case of Linux guest systems using VirtualBox, there is a guest compatibility package that can be installed on the guest system. Using the command `mount /dev/cdrom /media/cdrom` with the mounting of a CD image is required in the virtualbox where doing these two previous steps will install the compatibility package. Alternatively, in VMWare the mouse can be released to the host system by hitting CTRL+ALT while the same thing is achieved on Virtual box by hitting right hand CTRL key.

Once the Mininet system is running, the Mininet software can be updated with the following:

```
cd /home/mininet/mininet
git fetch
git checkout master   # Or a specific version like 2.2.1
git pull
make install
```

The most convenient setting for users who have no experience with Mininet and/or are experiencing Mininet for the first time is to work with Mininet using the lxde (graphical desktop setting). The desktop setting will be more convenient as it allows for the copying and pasting

between the host and the virtual machine also the use of WireShark and the opening of xterm will be smoother and without any troubles or system bugs.

On the other hand, it is always possible to run the command `ssh -X -l username` mininet where this command has to have the feature of X windows forwarding enabled as it is require multiple SSH logins. The use of this command allows the user to have the program window to close or display properly with having the ability to open gedit mininet-demo or wireshark program. Meaning the ability to open in the SSH command line a graphical program. It is worth noting that the user cannot use or run Wireshark or xterm if that user decides to not to use X windows and access the Mininet via SSH terminal sessions only.

Using Mininet

There are several commands that we can use at the top-level Mininet command prompt. The command links will list the connections between nodes, the command intfs will list the interfaces and there is another command net which is the most useful of all the interfaces where it does all the work of the previous commands where it lists the connections, the nodes and the interfaces. Here is an example of the output of the “nodes” command. [9]

```
h1 h1-eth0: s1-eth1
h2 h2-eth0: s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
```

This means that there are two nodes, h1 and h2, and one switch s1. The h1 node has one interface, named h1-eth0, which connects to s1’s interface s1-eth1. Similarly, the h2 node has one interface, h2-eth0, which connects to s1’s s1-eth2 interface. The last line lists s1’s three interfaces: the loopback interface lo, and the previously mentioned s1-eth1 and s1-eth2.

We can run arbitrary linux commands on individual nodes. To do this, we include the node name first, and then the command, at the top-level Mininet prompt. Below is an example of how to run "ifconfig" on h1, and how to have h1 ping node h6:

```
h1 ifconfig
h1 ping h6
```

The first line will tell us the IP address of h1, for example 10.0.0.1. This command also will show what is the MAC address of h1 (more correctly the h1-eth0 interface of h1) which might be for example 51:33:cf:b0:73:a9. The second command sends a "ping" to h6, from node h1. If successful, we know there is network connectivity between h1 and h6. It is also worth mentioning that there is another command which between each pair of hosts will generate a ping; the name of the command is `pingall`.

There is a command that works on both switch nodes and on host nodes, the name of this command is Xterm; by using it we can open a full shell window. It is worth mentioning that root permissions are what the xterm command runs with. Example on how we can use it is as below:

```
Xterm h1
```

This will open a full shell window connected to host h1; the window will appear on the mininet desktop. After running the command `xterm` the `ping h6` command will not work because of the hostname h6 is not known (not recognized) to bash. To go about solving this problem we will have to add the entries for h6 and h1 to the /etc/hosts file, or ping using the IP address 10.0.0.2.

When h1 and h6 are entered into the /etc/hosts file of the mininet host system, that will mean within the mininet the names h1 and h6 are shared and defined. Put in mind confusion will

be certain if different addresses were previously assigned to h1 and h6 through the configuration of a different mininet.

We can see the ping packets by monitoring the interface h1-eth0 using wireshark. The command `h1 wireshark &` can be utilized to have wireshark up and running on the specified node. The wireshark command can be run from mininet> prompt.

Another possible option is that if h6 has an ssh server enabled on it we have the option to try to ssh to h6. For this to work we have to run the ssh server, sshd, on each node. This can be specified in the mininet configuration file.

Switches network systems are by default shared with the mininet host system by the switches themselves. Another option is that we can utilize xterm by running it on the switch (or switches) and starting WireShark on it. Running the command on both trunk switches who are in the way between the two hosts h1 and h6. The commands that we will run are:

s1 ifconfig
s4 ifconfig

Then we get the results of Mininet while it is running outside of the Mininet process and we compare these results with outputs of the previous two commands.

Although no interfaces for c0 were shown by the intfs commands and net commands the same interfaces will be on c0 (the node controller).

By default, the nodes of Mininet are not connected to the outside world, so it is a great practice to run WireShark on s1-eth1 to examine the flow of packet (traffic) on a network in idle state. Without the need of WireShark filtering nor any other traffic there is a way to examine the ARP exchange, the three-way handshake of TCP, the delivery of the packets and finally the

termination of the connection. This is done by running netcat I-5432 on h6 and then netcat 10.0.0.2 5432 on h1, all that after running xterm windows on h1 and h6.

Open vSwitch

Mininet nodes that represent switches (rather than hosts) run the Open vSwitch software package. This provides a switch platform with support of management interfaces. Open vSwitch is licensed under the Apache 2 open-source license. [6]

VirtualBox, Xen, KVM and XenServer are all Linux virtualization technologies who are supported by Open vSwitch. Also, the way Open vSwitch was designed is to support multiple physical server's distribution. Add to that it exposes visibility interfaces and standard control to the networking layer which is virtual. Lastly in VM environments Open VSwitch is considered the perfect to play the role of a virtual switch.

C and python bindings for transactional database configurations, policing, Quality of Service (QoS) configuration, With or without LACP on upstream switch NIC bonding, Tunneling (GRE, Geneve, STT, LISP and VXLAN), Bunch of Extensions most importantly OpenFlow 1.0, Connectivity fault management of 802.1ag, Mirroring for more visibility with sFlow(R) and NetFlow, Access ports and Standard 802.1Q VLAN with trunk, Uses Linux kernel module which leads to high performance forwarding, These features are all supported by Open VSwitch.

Without any help from kernel module, userspace implementation is another Open VSwitch implementation which is entirely independent. DPDK devices or Linux devices are accessible by OVS in userspace mode. Userspace OVS compared to a switch which is kernel-

based it is easier to port. Costly in performance and experimental is what the non DPDK userspace datapath Open VSwitch is.

It makes sense to wonder the reason of the use of Open vSwitch, as it is known how reliable and fast the Linux bridge (the built-inL2 switch) which is used to bridge the traffic between the outside world and VMs as this ability is needed by Hypervisors.
When the previous version did not provide what the users hoped for, comes the answer which is targeted to virtualization deployments of multi-server presented in the Open vSwitch. In some cases, the integration with or changing to switching devices that has special characteristics, using endpoints that are highly dynamic to characterize environments and finally logical abstraction maintenance.

Exporting access to control traffic is a method of OpenFlow which is supported by Open VSwitch. Link state traffic and the use of inspection of discovery which leads to the global network discovery are two of the uses of exporting remote access to control traffic.

Hardware Integration

In case it was in an NIC end-host or in a chassis or a hardware switch the housing was, Open vSwitch in-kernel datapath or forwarding path is manageable to do the packet processing to hardware chipset offloading. This means that pure software implementation or hardware switch can be controlled by the control path of the Open vSwitch.

Many platforms that are vendor specific, Marvell, Broadcom and other multiple silicon chipsets are trying to port Open vSwitch to hardware chipsets.

Automated network control mechanism could be used to manage hosting environments which are virtualized and the non-virtualized environments (bare metal environments), this can

be done if the control abstraction of the Open vSwitch was exposed by the physical switches. This is one of the great advantages of the hardware integration. Add to that the performance enhancing within virtual environments.

Summary:

In large-scale virtualization environments which are Linux-based, Open vSwitch allows both dynamic and automated network control, as compared to networks without Open vSwitch. Reusing of subsystems (like Quality-of-Service stack which already exists) when we can and working on having the program residing in the kernel as small as it can be (since doing that will be great to have a better performance) are the goals of using Open vSwitch. Example on the previous is the availability of packaging for user space utilities and including of Open vSwitch as a part of the kernel for Linux 3.3 model.

Simple Mininet example (our topology)

He At this point we are prepared to describe the network configuration we created. Recall from the section **Overview of our Network** that our experimental topology, with N=5 and K=3, is the following:

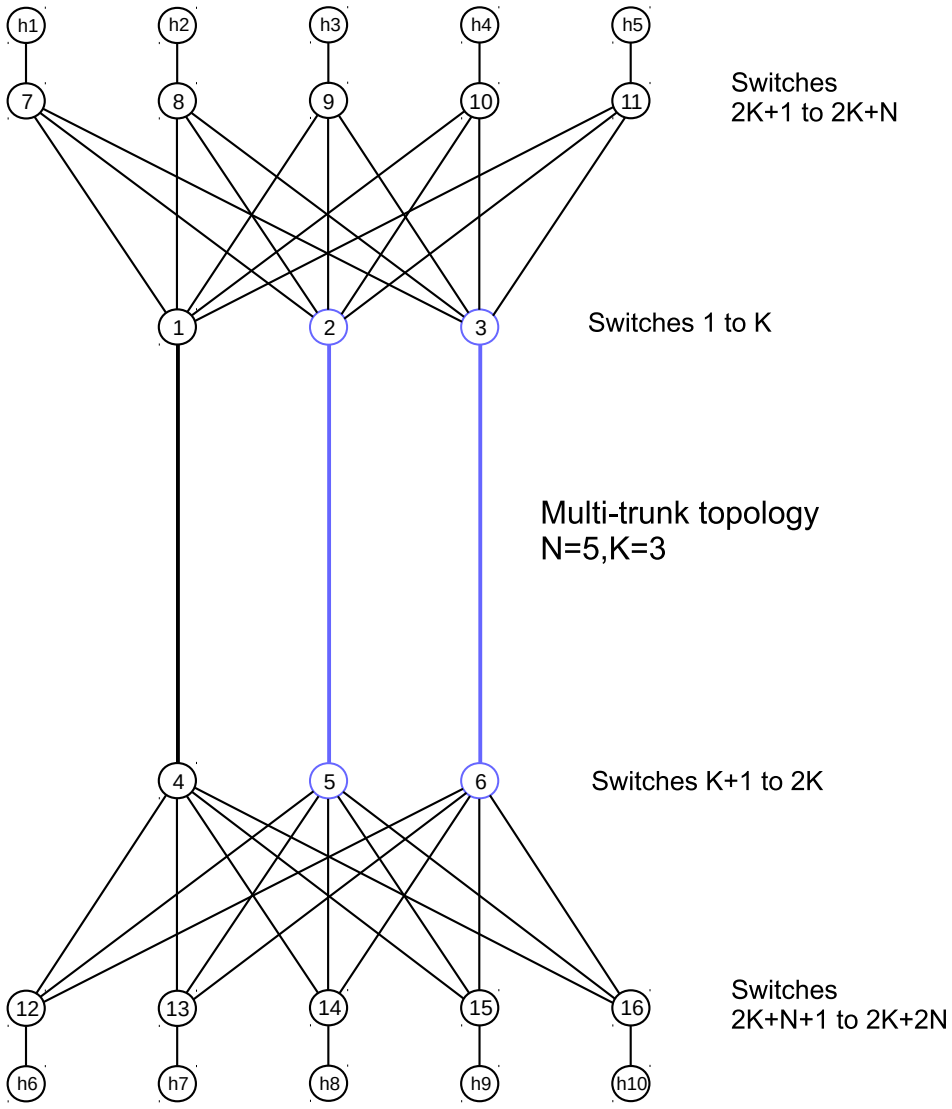

Figure 1. (repeated)

Our goal is to spread traffic between a node in the top cluster, h1-h5 above, and a node in the bottom cluster, h6-h10, over the three “trunk lines” 1-4, 2-5 and 3-6. Any individual TCP connection will be assigned to one trunk line; traffic sharing depends on having multiple TCP connections. TCP flows can be between the same pair of nodes or between two different pairs.

For example, two TCP connections between h3 and h8 can be assigned two different trunks, and two TCP connections h1—h10 and h2—h8 can also be assigned different trunks.

Trunk assignment is by flow, that is, by one direction of the TCP connection. Typically, the two flows (directions) of a single TCP connection will be assigned different trunks.

In our configuration, we have a single controller c0 for all the switches. This controller will be used to specify the forwarding of every packet at every switch. Switches will notify the controller of any new TCP connections (or any other previously unassigned traffic), and the controller can then send forwarding rules to the switches in response. The controller will typically send forwarding rules to the switch that reported the traffic and also to several other switches as well.

The controller communicates with the switches using TCP connections. In the Mininet default environment, the controller is accessible at the localhost address, which is the same for all the Mininet nodes. Therefore, there is never a need to find the controller, or to create a path to it. In more advanced settings, however, this may be necessary.

In examples 1 and 2 below, we show how to program c0 to make the switches behave like learning switches.

The POX Controller

The POX controller is preinstalled on the Mininet virtual machine. This is a general-purpose controller using the Python2 language. Example 1, below, shows one way to make all Open vSwitch switches act as Ethernet learning switches. Other forwarding strategies are also possible. The focus now will be the switch operation and the programming interface to the POX controller. [7]

The overall structure of a Pox program is a set of **handlers** that respond to **events**. The handlers may also call other code. Events originate at the switches, which then communicate the event to the controller, which in turn calls the appropriate event handler. Pox event types, for our purposes here, can be grouped into the following categories [7, 13]:

PacketIn: This event is triggered if a switch receives a packet that it forwards to the controller. This happens if the switch has not been programmed to handle the packet, or if the switch has been programmed specifically to forward the packet to the controller. For example, if a switch doesn't know how to forward a packet without flooding, and we want the switch to learn how to forward the packet, we will have the switch send the packet to the controller. Often, though not always, a PacketIn event will result in the controller giving new forwarding instructions to the switch.

ConnectionUP: This event is triggered when a switch connects initially to the controller. The initial instructions of packet handling given from the controller to the switch will happen in this point.

LinkEvent: Link availability reports are provided in this stage for the controller to be told by the switch about if a link is available or not. Link events are optional and are based on a modified version of the LLDP link-layer discovery protocol (used by ordinary Ethernet switches during the spanning-tree protocol). To activate link events, a special "discovery" module must be loaded by Pox which causes each switch to send link-discovery packets, also known as "Hello" messages, out each port. The result of a link event is an identification of the switch that sent the event, one of that switch's ports, and the switch and port to which that first port connects. By observing link events we can determine the exact network topology.

BarrierEvent: This event is triggered when a set of switch actions completes. We set the barrier, indicating a set of commands, and the switch triggers this event when those commands have completed. The responses of the switch to the messages received after the barrier and that is done after the switch is done with messages received before the barrier. In summary this stage is about the response of the switch to the OpenFlow Barrier message.

In the pox/forwarding and the pox/misc directories there are modules that explain how to program controllers. The POX Wiki also has definitions, explanations and examples of programming controllers.

OpenNetworking.org technical library has versions of the data structures of the POX which is connected to a big degree to the specification of the OpenFlow switch.

In mininet, the Pox controller lives on the host system. Each switch can then reach the controller via its own localhost interface. This means that switches do not have to perform any controller discovery, and do not have to set up paths to the controller. This is a great convenience. Alternative, more realistic approaches to switch-controller communication can also be implemented.

Switches communicate with the controller via TCP connections. The connections usually are made to port 6633 on the controller, though this can be configured in mininet.

An OpenFlow table consists of a set of OpenFLow entries, or rulesets. Packets start processing with Table 0.

Pox and OpenFlow Tables

A typical OpenFlow entry contains one or more packet-matching rules, and an action. The matching rules can match on any of the following packet attributes. [10]

- Ethernet destination address

- Ethernet source address
- Ethernet packet type (eg IP, IPX, ARP)
- IP destination address
- IP source address
- Transport protocol
- TCP destination port
- TCP source port

The action can be to forward the packet out a specific switch port, or to modify the packet (not used in this project), or to resubmit the packet to another OpenFLow table.

An incoming packet is matched against each of the OpenFlow entries in Table 0. If a match is found, the action is executed. The action can be to forward the packet, to forward the packet to the controller (thus triggering a PacketIn event at the controller), to flood the packet, or to process the packet by another, higher-numbered table.

The Pox programming interface correspond very directly to the OpenFlow rule specification.

Rules sent from the controller to a switch can be examined and verified using the command-line program `ovs_ofctl`. Rules can also be created or deleted using this command. We illustrate the creation of basic rule sets via the examples 0 - 2 below.

Example 0. hub.py

The simplest POX example can be found in hub.py. In this example, the controller instructs the switches to act as Ethernet hubs. This means that each arriving packet is flooded out all other switch ports. All the work is done by the ConnectionUp handler. [8]

```
def _handle_ConnectionUp (event):

    msg = of.ofp_flow_mod()
    msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
    event.connection.send(msg)
```

In this example, we create a handler for ConnectionUp events; these occur whenever a switch first connects to the controller. In the code above, when a switch connects to the controller then the controller

- Creates an `ofp_flow_mod` message it will send to the switch; this message corresponds to an OpenFlow flow entry. The entry is sent to Table 0, the default table.
- States that the action of the rule is to flood arriving traffic; that is, to send it out all ports other than the arrival port. There are no matching rules in this particular OpenFlow rule, meaning that it matches all traffic. (In general, if a packet matches more than one OpenFlow rule, then the rule to be applied is determined by rule priority values, not by rule order).
- Sends the message back to the switch, which then installs the entry.

At this point there are no PacketIn events because the switch has been told how to handle every packet.

Example 1. l2_pairs.py

The l2_pairs.py controller, written by James McCauley, is a simple example of how OpenFlow can be used to construct an Ethernet learning switch. The name comes from the fact that switching acts at layer 2, Ethernet logical layer, and that flows are identified by the pair (source_address, destination_address). This approach is not quite the same as what is done by native Ethernet switches, but it is slightly simpler to implement in OpenFlow. [8]

As packets arrive at the controller, sent by a switch, the controller records state information about what it has learned so far about the network, and installs new OpenFlow forwarding entries in the switch when it is able to do so.

Native Ethernet learning switches follow a two-step process, as outlined above:

- Record the source address and arrival interface, for forwarding of future packets back towards the sender.
- Look up the destination address, to see if we have a forwarding route for this packet (and flood if we do not).

This involves, in effect, two separate matches, which we cannot do with a single OpenFlow match table (we describe the two-table approach in the next example). Furthermore, if we were to have the controller tell a switch how to forward a packet to a given destination address, the switch will forward all future packets to that destination on its own, and not send those future packets to the controller. This makes it impossible for the controller to learn anything more about the source addresses of those future packets sent to that destination, and so the controller never learns how to forward to those source addresses.

The l2_pairs.py controller solves this problem by creating match rules that match on both the source and destination addresses. If a packet arrives at switch S from source A to destination B, and no matching rule for src=A,dst=B has yet been established, then S sends the packet to the controller.

The controller maintains, for each switch, a table of known (destination, interface) pairs. When the packet from host A to host B is forwarded by switch S to the controller C, then C will add address A, with the packet's arrival interface, to its table for S. The controller also looks up

the packet destination, B, in this table, to see if it has previously recorded a next_hop (forwarding interface) for that destination.

If not, the controller tells S to flood the packet, just as an Ethernet learning switch would do with a packet for which it did not have a forwarding entry for the destination. No new OpenFlow rule is created.

However, if the controller does have an entry for B, it is prepared to take action. Recall that, at this point, the controller knows how S should forward packets either to A or to B. The controller then installs two rules on S:

- src=A,dst=B: forward via the interface from the table
- src=B,dst=A: forward via the interface by which the packet arrived at S

As an example, suppose A sends a packet to B, and it arrives at S via interface 1. The controller does not yet know how to reach B, so it instructs S to flood the packet. The controller also enters (A,1) into its private forwarding table. Now suppose B sends a reply to A. When this arrives at S, via interface 2, the packet is sent to the controller. At this point the controller knows that S reaches A via interface 1 (from its table) and B via interface 2 (from this packet), and so installs the rule above.

Note that if D now sends a packet to B, arriving at S via interface 3, then S does not forward the packet, despite the fact that C knows that S should forward all packets to B out interface 2. The destination matches, but not the source. If C installed a forwarding rule at S for the destination alone, then S would not report the D-to-B packet to C, and so C would never discover how S can reach D.

We now look at the POX code. The controller has to handle only PacketIn events; all the work is done in that handler. The first step, when a packet arrives, is to put its switch and source into a Python dictionary named table[S,src]. Within POX, switches are identified by their so-called dpid values, which are obtained from the PacketIn event object's connection attribute: table[(dpid,packet.src)] = event.port

Dpid values generally match the switch numbers assigned by mininet; that is, s1 has a dpid of 1, etc. The value packet.src represents the source address; packet.dst is similar. The arrival interface is available in event.port.

The next step is to check to see if there is an entry in table for the destination, by looking up table[(dpid,packet.dst)]. If there is not an entry, then the packet gets flooded as in hub.py above: we create a packet-out message containing the to-be-flooded packet and send it back to the switch.

If the controller does find an entry in this lookup, that is, a switch port dst_port, it proceeds as follows. OpenFlow supports various types of messages from controller to switch; the first step is to create an empty flow modification message. We then fill in various msg.match fields. The available match attributes are in the following table [7]:

Attribute	Meaning
in_port	Switch port number the packet arrived on
dl_src	Ethernet source address
dl_dst	Ethernet destination address
dl_type	Ethertype / length (e.g. 0x0800 = IPv4)
nw_tos	IPv4 TOS/DS bits
nw_proto	IPv4 protocol (e.g., 6 = TCP), or lower 8 bits of ARP o
nw_src	IPv4 source address
nw_dst	IP destination address
tp_src	TCP/UDP source port
tp_dst	TCP/UDP destination port

Table 1: Available Match Attributes

For this example, dl_src and dl_dst will be used.

After the match fields are defined, we need to specify an action. The msg.actions field has the form of a list; we append to this the single action that the packet should be forwarded (output) via the appropriate port. After the action is specified, we send the message to the switch. We first create the reverse entry, forwarding from destination back to source.

```
msg = of.ofp_flow_mod()
msg.match.dl_dst = packet.src       # reversed dst and src
msg.match.dl_src = packet.dst       # reversed dst and src
msg.actions.append(of.ofp_action_output(port = event.port))
event.connection.send(msg)
```

The corresponding rule in hub.py had no match specifications and use the virtual port OFPP_FLOOD.

We next create the matching rule for the src-to-dst flow, by reversing src and dst, and using the port dst_port found in the controller's table[].

```
msg = of.ofp_flow_mod()
```

```
     msg.data = event.ofp                # Forward the incoming
     packet
     msg.match.dl_src = packet.src       # not reversed this
     time!
msg.match.dl_dst = packet.dst
msg.actions.append(of.ofp_action_output(port = dst_port))
event.connection.send(msg)
```

The second line, msg.data = event.ofp, includes the entire packet to be retransmitted in the message. The packet will be retransmitted after the new match/action rule is installed.

We can use the command-line utility ovs-ofctl to view the flow tables of each individual switch. We will use our mininet configuration program switchline_rc.py, which creates a topology of several switches in a row, each with an attached.

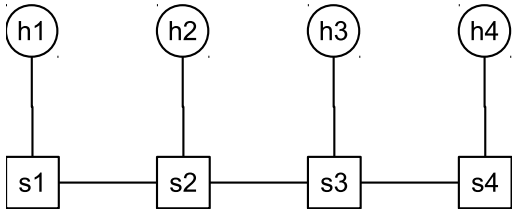

Figure 4**.** Linear arrangement of switches.

From within Mininet will be having h1 ping h4 and h2 ping h4 after we start switchline_rc.py and next l2_pairs.py module (the Pox module). So, we if we ran the command. [12]

ovs-ofctl dump-flows s2

We will get output like the following (with some entries deleted for simplicity). Note that MAC addresses in Mininet are assigned sequentially.

cookie=0x0, ...,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04 actions=output:3
cookie=0x0, ...,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:02 actions=output:1
cookie=0x0, ...,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04 actions=output:3
cookie=0x0, ...,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01 actions=output:2

Switch s2 has three ports; port 2 connects to s1, port 1 connects to h2, and port 3 connects to s3. The above ovs-ofctl output means that s2 forwards traffic from h1 to h4 via port 3, from h4 to h2 out via port 1, from h2 to h4 via port 3, and from h4 to h1 via port 2. There is no entry yet installed for packets from h1 to h3, or from h3 to anywhere, because we did not have h3 send any packets.

Example 2. l2_nx.py

The l2_pairs algorithm is quite serviceable, but if there are N hosts then each switch will have an OpenFlow match table of size approximately N2. The l2_nx.py example, also by James McCauley, addresses this by using two different OpenFlow matching tables, and achieves this using only O(N) space rather than O(N2). [2] The first table, table 0, matches the packet's source address, while table 1 matches the packet's destination address. The source code is found in the pox/pox/forwarding directory. [8]

Multiple flow tables use is enabled by Pox extensions Nicira. This Nicira extension to Pox is referred by nx.

If there is no match for an arriving packet in table 0, the packet is both sent to the controller and sent on to table 1. If there is a match, the packet is sent on to table 1 but not to the controller.

Table 1 is set up to match the destination address. If a match is found then the packet is forwarded to that destination. If there is no match then the packet is flooded.

As an example, let us suppose that A sends a packet to B, and B replies; the topology is A−−−S−−−B. A’s packet arrives at S, and S looks for a match on the packet’s source address, namely A. There is no match, so the packet is forwarded to the controller, which instructs S to

create an entry for address A in both table 0 and table 1. The table-1 entry contains an action that matching packets are to be forwarded via the port on S that connects to A.

The packet is also sent on to table 1, where a match on the destination address, B, is sought. There is no match there, either, so S floods the packet, in accordance with the associated action.

Now B replies. When this second packet arrives at S, table 0 is searched for a match on the source address. No match is found, so the packet is sent to the controller, which installs rules for matching B in both tables of S. In table 1, the action is to forward the packet via the port by which the packet from B just arrived.

The second packet is also matched against table 0, using the destination address A. Now a match is found, and so the packet is forwarded out the port that leads to A.
Any future packets between A and B will match both table 0 and table 1, so the controller will no longer be involved.

Because the controller always has S insert identical matching rules into table 0 and table 1, the two tables will always contain the same values. We could in theory achieve the same effect with a single table by submitting first the packet source address to table 0, and then the destination address to the same table. OpenFlow, however, always requires that any subsequent matches be made against a higher-numbered table, to avoid infinite loops, so a single table is not possible.

Using two OpenFlow tables in POX requires use of the so-called Nicira extensions. We load these with a command-line invocation such as the following:

./pox.py **openflow.nicira** −convert-packet-in forwarding.l2_nx

It is also necessary to create flow-modification messages with nx.nx_flow_mod() instead of of.ofp_flow_mod().

The path through the nx_l2.py code begins with the _handle_ConnectionUp() handler, invoked when the switch S first connects to the controller. The controller sends messages to S to implement the following:

- Enable multi-table support
- Create a default rule for table 0 that forwards the packet back to the controller
- Create a default rule for table 0 that floods the packet

Here is the code for the last step above:

```
msg = nx.nx_flow_mod()
msg.table_id = 1
msg.priority = 1 # Low priority
     msg.actions.append(of.ofp_action_output(port =
     of.OFPP_FLOOD))
event.connection.send(msg)
```

Recall that if there are multiple matches within a single table, then the match with the highest priority wins. Address-specific match rules will be installed with a priority of 32768 (the default), exceeding the default priority of 1.

When S reports arriving packets to the controller, in accordance with the table-0 default action, the controller's _handle_PacketIn() event handler is invoked. Here is the code that adds the packet's source address to table 0:

```
msg = nx.nx_flow_mod()
msg.match.of_eth_src = packet.src
     msg.actions.append(nx.nx_action_resubmit.resubmit_table(tab
     le = 1))
event.connection.send(msg)
```

The msg.table_id is, by default, 0. The packet's Ethernet source address is in packet.src; the `msg.match` field to match this against the source address of incoming packets is `of_eth_src`. The action that is appended to the match rule instructs S to resubmit the packet to table 1.

A consequence of this implementation is that, unlike in the l2_pairs.py example, the controller maintains no state about what addresses are reachable via what ports on switch S. In fact, the controller maintains no state whatsoever.

CHAPTER III

BASIC MULTITRUNK ASSIGNING CONNECTIONS TO TRUNKS

We are now ready to describe our multitrunk.py POX program, which assigns TCP flows to one of the available trunk lines in our topology. The trunk line assigned to a TCP flow is chosen in round-robin fashion. Only flows from top to bottom are handled this way; the reverse flows from bottom to top all are sent through the trunk s4 − s1. Each individual TCP flow will have a match rule specifying

- dl_src and dl_dst, the Ethernet addresses
- nw_src and nw_dst, the IP addresses
- tp_src and tp_dst, the TCP port numbers

These OpenFlow rules will be installed in each switch on the path. Once a TCP connection is made, and the OpenFlow rules are installed, then traffic for that connection will not be looked at further by the controller.

Much of the code here is specific to our topology, but it generalizes in a straightforward way to other topologies with multiple trunks from one cluster of nodes to another.

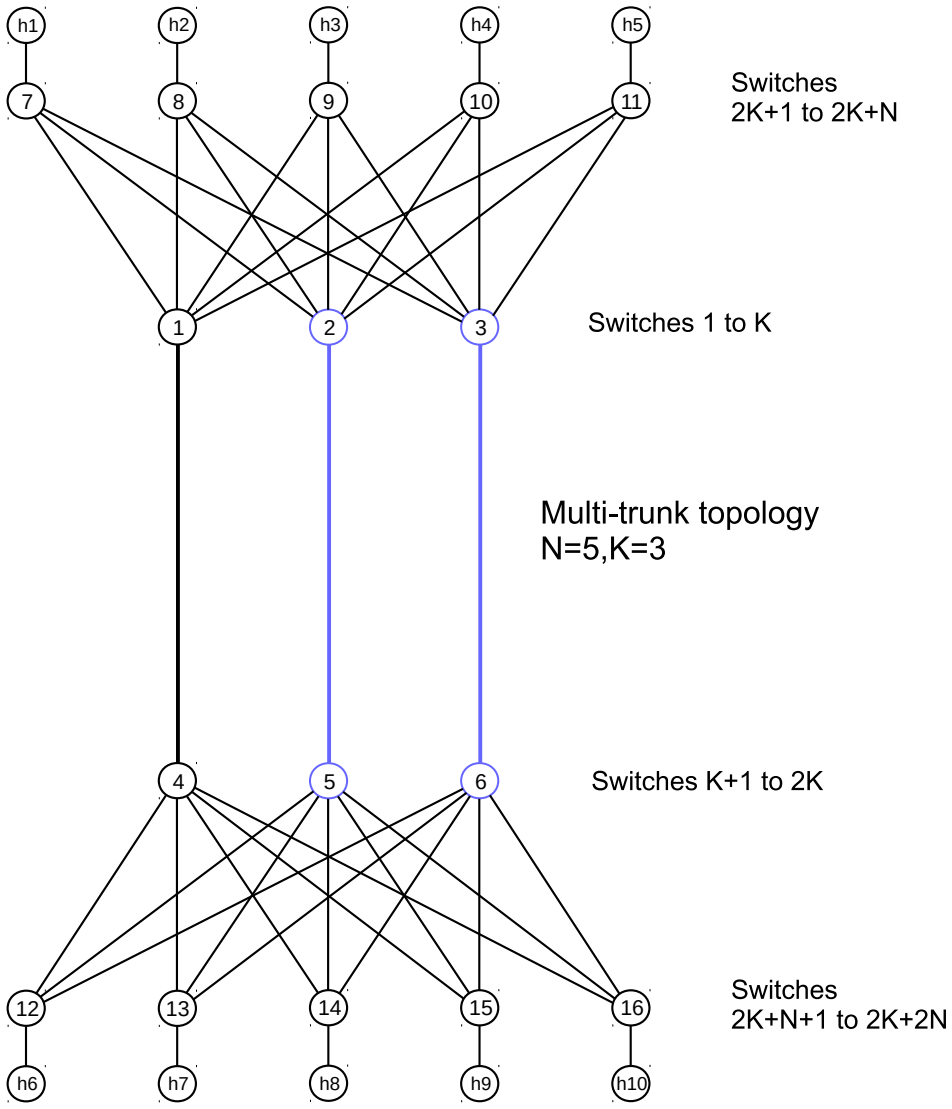

Figure 1. (repeated)

Matching on both source and destination is necessary so that each TCP flow can be routed individually; matching on destination only would force all flows to the same destination to take the same path. Including match rules for the Ethernet addresses is not strictly necessary. Switch-to-switch connections can be discovered during the processing of POX LinkEvent events, below. However, there is no automatic way to discover switch-to-host connections, such as s7-

to-h1. While these connections can be determined from the diagram, there is no way to determine from the diagram the port used by s7 to reach h1. We solve this problem by using ARP.

We rely on ARP messages to tell the controller where the hosts are, just as ARP traditionally tells senders where their destinations are. If h1 sends out an ARP “who-has h8” query, then s7 will discover that it is directly connected to h1, and via what port. S7 will report this to the controller. S7 will know that the arriving ARP packet comes from a host and not another switch because, at startup, each switch will be able to identify which of its ports connect to other switches (see the discussion on switch neighbor discovery, below). As the ARP packet is not arriving from one of those switch ports, it must represent a direct host connection.
We will have the controller tell s7 that the ARP packet should be flooded; we will use a spanning tree to avoid loops. The ARP packet will eventually be delivered to its destination, h8. This host will reply, and thereby inform switch s14 that it is directly connected to h8. This information will also be sent to the controller.

ARP packets will be delivered via forwarding rules built by ordinary Ethernet learning. In order to get this to work, we must identify a spanning tree, and we must implement an Ethernet learning algorithm in POX. We will use the algorithm of l2_nx.py, using two OpenFlow tables. The spanning tree will be determined manually; we will simply disable switches s2, s3 and s5, s6. This disables the blue trunk lines shown in the diagram, and also all links from the host switches to s2, s3, s5 and s6.

We will also allow ICMP traffic to be delivered over this same spanning tree, again using Ethernet learning. While ICMP traffic carries some important IP-layer error messages, these do

not appear in our emulation. Our only visible use of ICMP traffic is in host-to-host pings, which is what we use to establish that connectivity is working.

Matching of TCP flows is done entirely in flow table 0, following the style of the l2_pairs example. However, flow table 1 is needed for handling the ARP and ICMP traffic.

For this program the controller must learn what switches are connected to what other switches, and via what ports. POX includes a “discovery” module to achieve this. When this module is activated, the controller instructs each switch to send special messages out each of its ports. These messages are never forwarded. If the receiver of one of these messages sent by switch S1 is another switch S2, then S2 reports receipt of the message back to the controller. The controller then learns that S1 and S2 are neighbors, and also what port is needed to reach S2 from S1 and vice-versa. If the message is delivered to a host, nothing happens. The messages themselves are modeled after the HELLO messages used by the spanning-tree algorithm.

multitrunk.py

Different paths can be utilized to make a TCP connection between different hosts (two hosts); the purpose of multitrunk is to assign these paths so as to distribute traffic over all the available trunks.

The choices of using different and multiple routes between two hosts is why we want to use multitrunk. It explains how different paths (or trunks) can be used to route the TCP connections.

As in the figure above, between h1-h5 and h6-h10 there will be multiple trunk lines; these are the K connections s1−s4, s2−s5 and s3−s6.

Our Pox module that runs on the controller is multitrunkpox.py. The program that sets up our topology in Mininet is multitrunk12.py. The variable K decides the number of the trunks;

K=2 is the default but we use K=3 in all our examples. N represents the number of hosts at each end. The topology contains many loops, but, as long as we prevent flooding traffic from using the switches s2, s3, s5 and s6, and thus the blue trunk links, the broadcast-traffic looping will not happen. There are also loops local to each end of the topology that our approach prevents; because s2 does not forward routine packets, the loop s7−s2−s8−s1−s7 has no effect.

We are primarily concerned with connections between one of the top hosts, h1-h5 in the figure, and the bottom hosts, h6-h10. It is these flows that will be assigned one of the s1−s4, s2−s5 or s3−s6 trunks.

Each TCP connection consists of two flows, one for each direction of the connection. If we create a TCP connection from h1 to h6, from the top to the bottom of the picture, then we will have one flow going from h1 to h6 and one flow going from h6 to h1. We will vary the routing only for the top-to-bottom h1 to h6 flow; the left-hand trunk link s1-s4 will always be the path used for h6 to h1 flows.

If we create a TCP connection from h1 to h2, both at the top of the picture, then we will again have one flow going from h1 to h2 and one flow going from h2 to h1. In this case, the h2-to-h1 flow will be routed via s1, and we could in principle route the forward h1-to-h2 flow via any of s1, s2 or s3. In fact, we do not assign special routes for this top-to-top case; both flows will be routed via s1.

The forwarding information used by OpenFlow will have to contain the TCP ports; we need this to allow different TCP connections between the same pair of hosts (top and bottom) to be routed via different trunks. Such connections will in a round-robin way alternate among the

trunk lines. It is worth mentioning that in the beginning the s1 to s4 will be used for the first flow of TCP between h1 to h6.

The controller will know which hosts are directly connected through which switch ports and also the MAC addresses of these hosts will be learned by switches through PacketIn message of the ICMP and ARP packets. It is worth mentioning that the TCP traffic is differently handled.

Switch-to-Switch Link Discovery

The first phase of bringing up multitrunkpox.py is neighbor discovery among the switches. The names of adjacent switches could be supplied at compile time by reference to the diagram, but the switch ports used to reach these adjacent switches can only be discovered dynamically. [13, 7]

Switch discovery requires the Pox openflow.discovery module, requested on the command line. The complete command line is

```
./pox.py openflow.discovery openflow.nicira −convert-packet-in log.level –WARNING
forwarding.switchgraph${VER} −N=$N −K=$K
```

The discovery module, outlined above, has each switch send out special packets resembling spanning-tree HELLO messages, technically known as LLDP messages. When switches receive these from their neighbors, they forward them to the controller, which processes them via the `handle_linkEvent()` handler. A HELLO packet includes the switch port and the dpid of the switch who sent the packet. Along with the receiving switch port the dpid gets sent to the controller, this happens when a neighboring switch receives an LLDP packet so this packet get sent with what we said earlier the dpid and the port. A report will be generated after this point and get sent as a linkEvent to the multitrunkpox module; this report has the port

numbers and the switches at the sender side and also the receiver side after the controller learn them.

The learning of multitrunkpox module happens as the processing of the messages of LinkEvent takes place. The LinkEvent phase is so short in time to a degree it can be measured in seconds and the learning process is knowing the information which is what and how the connection to a neighboring switch will take place and at what ports are connected directly.

The neighbor hosts at this point we don't have any information about since we didn't get any packets from them but what we will say that we know is that the neighbor switches that are directly connected. After the linkEvent phase is completed, we can identify the switch ports that connect directly to hosts because these are the ports for which no information has yet been discovered.

The controller keeps all this information about switches and switch-to-switch links in a graph of Python SwitchNode objects, one per switch. Each SwitchNode has a list of neighboring SwitchNodes, and the ports used to reach them. There is a separate list, for each switch, of directly neighboring hosts and the ports used to reach them; these are discovered later. A Python map is maintained between switch numeric values − the so-called dpid values − and the corresponding SwitchNode object.

The exchange of link messages takes some time. As a result, we did not attempt to start TCP traffic until 10 seconds after the POX controller was started. The delay in this link-exchange protocol is of the same general order of magnitude as the delay in the Spanning Tree Protocol but can be somewhat faster as there is no need for “convergence”. All the switches are

learning is the links by which neighbors can be reached; there is no shortest-path algorithm involved.

ICMP and ARP Traffic

After the LinkEvent processing and the construction of the SwitchNodes, the switches connect to the controller, which triggers the handle_ConnectionUP() processing. It is at this stage that the switches are initialized to default rules to handle ICMP and APR traffic. The match/action rules for ICMP and ARP traffic are sent to the switches.

Only switches designated as “flooder” switches carry ARP and ICMP traffic; that is, s2-s3 and s5-s6 are excluded. The direct links from s7-s11 to s2 and s3 are not disabled; s7-s11 would therefore flood packets to s2 and s3. However, s2 and s3 would then drop the packets, so they would not propagate further. This is not exactly the same as the traditional spanning-tree algorithm, which does disable such links, but the end effect is the same. One way to look at this is to say that a spanning tree is constructed “manually”, and then OpenFlow rules are established to allow ARP and ICMP traffic to be forwarded in the usual way along this spanning tree. TCP traffic, on the other hand, will always be assigned specific routes by the controller; it does not use the spanning tree.

Table-initialization code had to include match rules restricting attention to ARP and ICMP traffic, as below. Match rules do not allow “or” conditions, so two sets of rules had to be created. In the code below, msgi is the message for ICMP match rules, and msga is the corresponding message for ARP match rules.

```
msgi.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.IP_TYPE)
msgi.match.append(nx.NXM_OF_IP_PROTO(pkt.ipv4.ICMP_PROTOCOL))
msga.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.ARP_TYPE))
```

Overall, the ConnectionUp processing resembles that of the l2_nx.py example, earlier, in that two tables are set up. However, rules for both ICMP and ARP must be installed. Because we use two OpenFlow tables for the ICMP/ARP traffic, following the technique in Example 2: l2_nx.py, the table sizes need at most O(N) entries (versus the l2_pairs example, in which tables could need O(N2) entries).

The flooding versus non-flooding switches are handled as follows:

```
if flooder(connection.dpid):
       msgi.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
       msga.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
```

For non-flooding switches, no flood rule is installed.

Handling TCP Connections

The next step is to handle TCP connections. Recall that the switches are configured to report packets to the controller whenever there is no matching rule. No switch forwards TCP packets by default. So, when the first TCP packet of a connection (the SYN packet of the three-way handshake) arrived at the frontline switch adjacent to the initiating host, the packet would be forwarded to the controller, which then generates a PacketIn event for the packet.

In the PacketIn event handler, the controller had to distinguish between TCP traffic and ARP/ICMP traffic (UDP is not delivered). This is done using the POX find() method, as in the following:

- icmp = packet.find('icmp')
- arp = packet.find('arp')
- ipv4= packet.find('ipv4')
- tcp = packet.find('tcp')

These variables are set to “None” if the packet is not of the proposed type.

Once a TCP packet from a new connection is identified, we identify the source host ha and destination host `hb` and then create a `Connection` object conn for it; this represents the one-way flow ha→hb. The next step is to choose a trunk line (by choosing an upper trunk switch; one of S1-S3 when K=3). This is done by `picktrunk(conn)`, and is usually done on a round-robin basis, eg selecting S1, then S2, then S3, then S1 again and so on. The trunk switch chosen does not depend on the connection, provided the two endpoints are at opposite ends of the network. However, the process still works if both endpoints are at the same end of the network, in which case the path will lead from host to selected trunk switch and back to the other host, without traversing the trunk link itself.

At this point, with the Connection conn and trunk switch `trunkswitch`, we can calculate the unique path from ha to hb that goes through `trunkswitch`. We use here our knowledge of our specific topology, though finding this path can be done for general topologies also. The path is created by findpath(conn, trunkswitch), and returns a python list of host and switch nodes, for example [h2, s8, s2, s5, s15, h9].

We are then ready to configure the necessary forwarding entries in each switch along the path; this is done by `create_path_entries(conn, path)`. The entries are made in reverse order; for example, if the path is [h2, s8, s2, s5, s15, h9] then we configure the switches in order s15, s5, s2 and s8. This is done so that, if a packet of the connection reaches one switch, the remaining switches will likely already have been configured for the connection, and there will be no more communication with the controller. This is not guaranteed, as configuration is done by having the sender send a `flow_mod` message to the switch and then having the switch process that message; these steps are not synchronous. It is possible that an earlier switch on the

path will be configured before a later switch, even though we sent the `flow_mod` message to the later switch first. This proved not to be a problem in practice, however, and, in any event, the only problem would be the duplication of the path setup.

In order to make it easier for the controller to handle receiving two TCP packets from the same connection − most likely from two different switches, when the controller processes the first packet it records the association between conn and the new path in the dictionary `conn_to_path`. In the event that the controller later sees another packet from the same connection, it can easily determine that the path has already been defined and can assign the same path during processing of the second packet. In actual experience, we did regularly − though not frequently − see the controller process two packets for one TCP connection, indicating that this race condition was a real possibility. The `conn_to_path` dictionary also ensures that flow entries which time out (an OpenFlow option we did not make use of) will be re-established using the same path.

There is one potential problem that can come up in the `create_path_entries(conn, path)` process. When creating flow_mod entries for a switch S in the path, we need to know not only the next node − which we can determine from the path − but also the port on S used to reach that next node. If the next node is also a switch, then this port is available in the SwitchNode map. The initial switch-to-switch link-discovery protocol provides information about switch ports that connect to other switches; immediately following the end of that protocol, however, switch ports that lead directly to hosts have their neighbors left unspecified. So, potentially, with path [h2, s8, s2, s5, s15, h9], we might not know the port that s15 uses to reach h9.

This situation − that s15 does not know how to reach h9 − corresponds to the traditional learning-Ethernet switch situation where a switch does not have a forwarding entry to a given destination. In that case, the switch floods the packet. Our solution here is to rely on this port discovery having been completed during the earlier ARP exchange.

If host h2 has not previously sent traffic to h9, then h2 will not have h9's IP address, and will send an ARP "who-has" request. This request will likely be flooded, using the spanning tree we previously established that contains only the S1−S4 trunk. The ARP message will be flooded by s15 in particular and will reach h9. Host h9 will then send an ARP reply; it will be sent to h2's unicast address rather than to the broadcast address but flooding might still potentially occur. When h9's reply reaches its immediate-neighbor front-line switch s15, the switch will learn what port it uses to reach h9. This port will, of course, be the port by which the packet from h9 just arrived.

As part of the PacketIn process, whenever a packet arrives at a switch S and is forwarded to the controller, the controller checks to see if the arrival port at S was known to connect to another switch. If the arrival port is not known to be connected to a switch, it is assumed to be connected to a host, and so the controller records in the SwitchMap that that port of S connects directly to the host that originated the packet. In this case the switch S must be one of the front-line switches, as these are the only switches directly connected to hosts. It is this step that completes the SwitchMap process.

It is possible that the ARP processing has not finished by the time the first TCP packet (SYN packet) is sent. If this happens, then the `create_path_entries()` step will fail, because the final switch will not know what port connects to the destination host. If this happens,

it is likely to succeed when the process is repeated when the SYN packet reaches the next host. In the worst case, the SYN packet will be lost, and by the time it is retransmitted the earlier ARP processing will almost certainly have completed.

Assuming `create_path_entries()` is successful, forwarding rules will be created for each switch on the path.

At the time we create the forward connection, conn, we can also set up the path for the reverse connection. The reverse path, rpath, can either use the same trunk link as the original path (in which rpath is the Python list reversal of path), or it can use a different trunk link. After defining rpath, we proceed as follows:

```
rconn = conn.reverse()
create_path_entries(rconn, rpath)
conn_to_path[rconn] = rpath
```

Using the ovs-ofctl command

We can view the path entries using the `ovs-ofctl` command on the Mininet host system. For example, the output below was obtained from an instance of our network with K=2 (two trunk lines) and N=1 (one host at each end). [12]

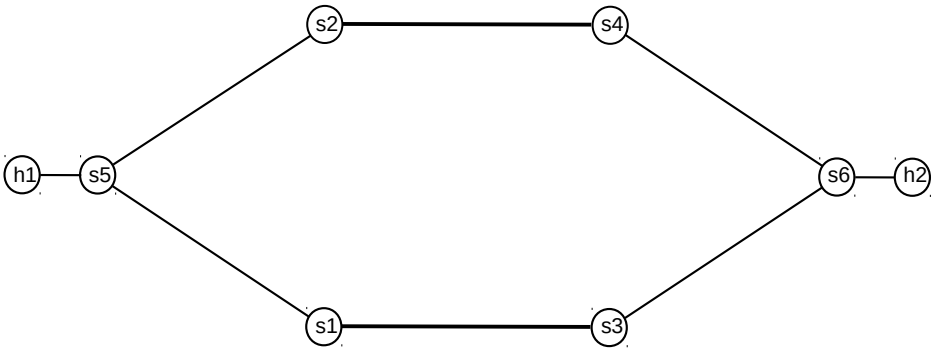

Figure 5. Two paths between two hosts
(Multitrunk with N=1 and K=2)

We created two different ssh connections between h1 and h2, and then ran `ovs-ofctl dump-flows s5`. Only entries relating to TCP connections are shown below:

cookie=0x0, ...,
tcp,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.0.0.1,nw_dst=10.0.0.2,tp_src=59404,tp_dst=22
actions=output:1

cookie=0x0, ...,
tcp,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.0.0.1,nw_dst=10.0.0.2,tp_src=59526,tp_dst=22
actions=output:2

cookie=0x0, ...,
tcp,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01,nw_src=10.0.0.2,nw_dst=10.0.0.1,tp_src=22,tp_dst=59404
actions=output:3

cookie=0x0, ...,
tcp,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01,nw_src=10.0.0.2,nw_dst=10.0.0.1,tp_src=22,tp_dst=59526
actions=output:3

The h1-to-h2 flows are represented by the first two entries. h1 has MAC address 00:00:00:00:00:01 and IP address 10.0.0.1; similarly for h2. TCP source port 59404 and the s1-s3 trunk is used for the first h1→h2 flow. The port of s5 that connects to s1 is 1, hence the output:1 field. The Mininet link command output states s5-eth1<->s1-eth2; the eth1 interface corresponds to port 1.

The second entry represents the second h1→h2 flow, using the s2-s4 trunk and having source port 59526. From s5, the output port is 2.

The third and fourth entries are the reverse flows, h2→h1. The source is h2, the destination is h1, and the port on s5 that leads to h1 is 3.

In general, OpenFlow allows idle TCP connections to time out; the flow entries are then deleted. If traffic resumes, the first switch to see the new traffic again contacts the controller; the `conn_to_path` dictionary ensures that the same path is re-established. Because setting up the ovs-ofctl command takes a modest amount of time, it was convenient to use an infinite timeout to avoid having to hurry. However, a timeout should have a limit in a production environment.

If OpenFlow timeouts are used, then connections can be deleted when the final TCP FIN packet is seen, or a TCP RST. This would involve ensuring that such packets are always reported to the controller; we did not implement this.

Comparison to ECMP

The OpenFlow approach here can be compared to Equal-Cost MultiPath routing, or ECMP, in which traffic to a given destination is also divided up among multiple paths. [3] In ECMP, each path has the same "cost" value, which is also the case in all our examples although it is not an explicit requirement. ECMP is supported by most large commercial switches and routers.

ECMP can, in principle, be configured to divide between the two trunks on a per-packet level, using round-robin scheduling: one packet is sent via trunk 1 and the second is sent via trunk 2, and alternating continuously.

When used with TCP, however, ECMP is almost always configured so it routes packets of any one flow over the same path. The path is chosen based on a hash of the TCP connection information (source IPaddr, source port, destination IPaddr, destination port). This is done in part to minimize packet reordering, which is a serious problem for TCP connections as out-of-order packets are often interpreted as an indication of packet loss. Keeping each flow on the same path also avoids the possibility that different packets could face different MTUs, and thus different fragmentation rules. Therefore, in ECMP, as with our OpenFlow approach, each TCP flow takes the same path.

This same-path behavior means that the ECMP approach is effectively identical to our approach, except that our approach allows us to choose that path, while ECMP does not.

This lack of support for path choice puts ECMP at a potentially significant disadvantage. With three flows and three paths, the probability that all three flows would be assigned different paths is 1/6. For our OpenFlow approach, the probability is 1.

For moderately large numbers of TCP flows, ECMP is likely to assign roughly equal numbers of flows to each path. Even here, though, the OpenFlow approach allows redistributing flows among the paths based on current utilization; ECMP does not permit this. For very large numbers of TCP flows, ECMP is likely to assign roughly equal volumes of traffic to each path, but, for more modest numbers of flows, our approach gives the system managers much more control.

Packet-by-packet ECMP can be implemented in OpenFlow by forwarding packets not to a port but to an OpenFlow group. The OpenFlow switch has to then be configured to use round-robin selection as part of its group action. This was not available to us as POX offers very limited support for group actions. In any event, problems with TCP packet reordering make this option unattractive.

One drawback of the OpenFlow approach, compared to ECMP, is the need to maintain state information about each flow in every applicable switch (in particular the "front-line" switches). Each switch has more processing to do, and so the OpenFlow approach may be slower for very large traffic volumes. However, for smaller traffic volumes the OpenFlow overhead is small, and this is the case where the OpenFlow advantages over ECMP are strongest.

Comparison to LAG and LACP

Our technique can also be compared to Link Aggregation Groups (LAG) and Link Aggregation Control Protocol. [5] In LAG and LACP, multiple links between a pair of switches, or between a switch and a multi-port server, can be bonded together to serve as one.

In the sense that our technique is “bonding” together multiple trunk lines, LAG/LACP is similar. However, LAG/LACP requires that all the bonded links start and end at the same pair of devices. Our arrangement has no such requirement; in fact, each trunk line in our standard diagram connects to its own pair of switches, not connected to by any other trunk line.

LAG/LACP also requires that each path is an individual link, and, furthermore, that the propagation delays on each link are approximately the same. While we do not demonstrate this case, our technique also works with trunk lines that may have multiple of hops, rather than being simple point-to-point links. Our technique definitely does not care if propagation delays are widely different. Finally, our technique would work if there were three clusters of hosts, clusterA, clusterB and clusterC, and we wanted to spread traffic from clusterA to clusterB over some trunk lines directly from A to B and some trunk lines from A to C to B.

CHAPTER IV

ADVANCED MULTITRUNK REASSIGNING CONNECTIONS TO NEW TRUNKS

In the final version of our POX program, we introduced a process in which flows were rerouted at periodic intervals; that is, the flows were reassigned to new trunks. Rerouting was done on the basis of the recent bandwidth usage history of each flow, in order to maximize the overall utilization of the three trunk lines.

For the recent-bandwidth strategy to make sense, we had to assume that each flow's bandwidth fluctuated with time under the direction of the application creating the connection. That is, flows were rate-limited, based on the needs of their creating application. We also assumed that the typical timescale for rate changes was at least as long as the interval at which we updated the flow statistics.

At regular intervals, typically of length 1 second, a Python timer fired and caused the controller to call request_flow_stats(), which sent `ofp_flow_stats_request` messages to all front-line switches. These statistics requests were answered asynchronously, creating `FlowStatsReceived` events. The handler for these was `handle_flow_stats()`. From these responses we obtained information about each connection's throughput during the previous interval. These were stored in `FlowInfo` objects, in the map `flowdict` indexed by Connection (flow) objects.

At slightly longer intervals, typically 2 to 5 seconds, the function `reshuffle_flows()` was called, which reassigned each top-to-bottom flow to a possibly

different trunk line. We did this reassignment so as to distribute the flows as fairly as possible among the trunk links.

In early versions, flow reshuffling was initiated manually by Unix signals received by the POX process, in particular SIGUSR1 and SIGUSR2.

We created a `FlowInfo` class that held information about recent throughput history, and a dictionary `flowdict` that mapped connections to `FlowInfo` objects. These `FlowInfo` objects were updated asynchronously by the `FlowStatsReceived` event handler.
We also modified the Mininet configuration to assign a fixed bandwidth to each of the trunk links. This involved adding the bw=BANDWIDTH option to each Mininet call to addLink(). Typically, we used a fixed bandwidth of 20 Mbps.

We can now describe the `reshuffle_flows()` algorithm. For each connection (that is, one-way connection, or flow), we calculated its recent bandwidth, using the `FlowInfo` data. The next step was to sort the connections in decreasing order of recent bandwidth; that is, with the highest-bandwidth users first. We then went through this sorted list and reassigned each connection in turn to the trunk that had the largest remaining amount of unassigned bandwidth. For example, if the flow bandwidths were 11, 10, 8, 5, 2, the flows would be distributed over the trunks as follows:

- 11: trunk 1
- 10: trunk 2 (trunks 2 and 3 each have 20 Mbps to allocate; trunk 2 comes first)
- 8: trunk 3 (at this point trunk 3 has 20 Mbps while trunks 1 and 2 have 9 and 10 Mbps respectively)
- 5: trunk 3 (which has 12 Mbps free, versus 9 and 10 for trunks 1 and 2 respectively)

- 2: trunk 2

This can be described as a “greedy” algorithm. Each trunk was initially allocated bandwidth of the BANDWIDTH value set in our Mininet setup. Because recent bandwidth history was also limited by the trunk-line BANDWIDTH constraint, the sum of the bandwidth-history values could not exceed K * BANDWIDTH, where K is as usual the number of trunk links.

Flows were transferred to their new trunk using the function `move_flow(conn, newtrunk)`. If a path from h3, with front-line switch s9, was being moved from trunk s1−s4 to s2−s5, then creating the new path entry in s9 would need to overwrite the old path entry in s9, so that the flow’s traffic would now be forwarded from s9 to s2 instead of to s1. The default `flow_mod` command action, however, was OFPFC_ADD. While using OFPFC_ADD does allow overwriting a switch’s flow entry, it resets all the traffic statistics to 0. However, it was essential that we retained each flow’s traffic statistics going forward, so we could properly take into account its bandwidth history.

To handle this, we wrote a separate function `create_path_entries2(conn, path, pivotswitch)`, where `pivotswitch` represents the last switch that remains in the new path; that is, the switch that must forward the flow out a new port. This would be s9 in the example above. We then wrote `modTCPrule()` (where `create_path_entries()` used `addTCPrule()`) which used the OFPFC_**MODIFY** command instead of OFPFC_ADD. When applied to an OpenFlow switch flow rule, this updates the forwarding while preserving the past traffic statistics. The `modTCPrule()` function was used to modify the pivotswitch flow table.

We also had to delete the flow entries that were no longer used at all; in our example above these would be the old entries in s1 and s4. To delete flow entries we created `delTCPrule()` which invoked the OFPFC_DELETE command.

The final step was to create trafficgen.py, which generated rate-limited traffic where the rate could be configured to evolve with time. For example, the command trafficgen.py h10 5432 5 99 99 20 20 20 50 50 20 generated traffic to host h10, on port 5432, with traffic changing at 5-second intervals according to the bandwidth pattern 99%, 99%, 20% 20%, 20%, 50%, 50%, 20%, where each percentage is the percentage of the maximum available bandwidth.

Our greedy algorithm here is known not to be optimal, in that in some cases it cannot find a trunk line to which to assign the last connection, while at the same time preserving the desired goal of having the sums of the past bandwidths of the flows on each link be less than or equal to the total available bandwidth on that link. However, this goal is not essential, in that if a trunk line is oversubscribed then the bandwidth of each flow assigned to that trunk will be proportionally reduced.

We are not aware of any existing Ethernet mechanisms that assign flows to paths based on bandwidth-usage history.

In one demonstration experiment, we created five flows, each with a varying rate limit. Here is a graph of how those flows were allocated to the trunk lines:

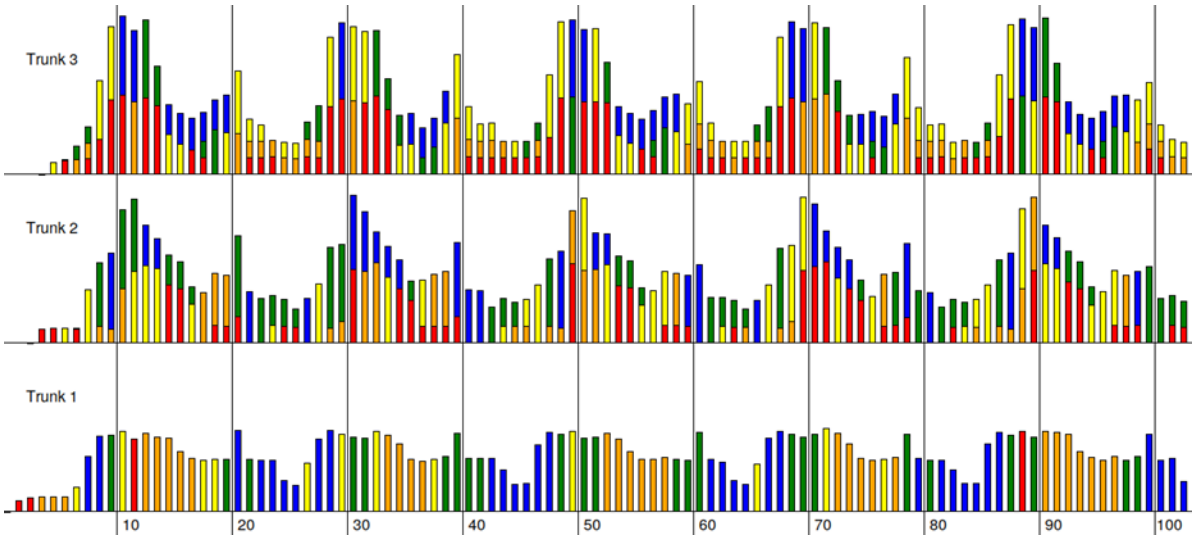

Figure 6**.** Dynamic flow rebalancing

The five flows are red, orange, yellow, green and blue. The flow rates vary with a period of about 20 seconds; this periodicity is visible in the graph. Our algorithm always assigns the flow with the biggest previous bandwidth history to trunk 1; the algorithm will then always assign the next two flows to trunks 2 and 3 respectively, and then the final two flows also to the set of trunks 2 and three (though the final two flows may be assigned to the same trunk).
We were unable to start all five flows at the same time. The beginning of the graph shows this, with new flows added over seven 2.0-second cycles.

Note the significant increase in the blue flow's bandwidth history between 19 and 20, resulting in the blue flow moving from trunk 3 to trunk 1. Similarly, while the green flow was the highest at time 9, it was just slightly outpaced at time 10 by the yellow flow, and so green moved to trunk 2.

REFERENCE LIST

[1] Peter Dordal, “Software-Defined Networking,” *An Introduction to Computer Networks*, [E-Book], 2016. Available: http://intronetworks.cs.luc.edu/current/html/ethernet.html#software-defined-networking/. [Accessed July 2018].

[2] Denys Haryachyy, “Learning POX OpenFlow controller: L2 Switch using Multiple Tables,” June 7, 2014. [Online]. Available: https://haryachyy.wordpress.com/2014/06/07/learning-pox-openflow-controller-l2-switch-using-multiple-tables/. [Accessed March 07, 2018].

[3] IEEE, “IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks - Amendment: Equal Cost Multiple Paths (ECMP),” Mar 27, 2014.

[4] IEEE, “IEEE 802.1D Standard: Media Access Control (MAC) Bridges,” 1990.

[5] IEEE, “IEEE 802.3ad Standard: Link Aggregation,” 2000.

[6] Linux Foundation, “Open vSwitch Documentation,” 2016. [Online]. Available: http://docs.openvswitch.org/. [Accessed Feb 03, 2018].

[7] Murphy McCauley, “POX Wiki,” in *A Dictionary of the Internet*. Stanford University, [Online], 2018. Available: Stanford Online, https://openflow.stanford.edu/display/ONL/POX+Wiki/. [Accessed: Apr 03, 2018].

[8] Murphy McCauley, “The POX network software platform,” *github.com*, 2013. [Online]. Available: https://github.com/noxrepo/pox/. [Accessed: March 12, 2018].

[9] Mininet Team, “Mininet Walkthrough,” Mininet Team, 2018. [Online]. Available: http://mininet.org/walkthrough/. [Accessed: May. 12, 2018].

[10] ONF, “OpenFlow Switch Specification version 1.5.1,” *Open Networking Foundation*, March 26, 2015.

[11] Radia Perlman, “An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN,” *ACM SIGCOMM Computer Communication Review 15(4),* September 1985.

[12] Pica8 Inc, “OVS Commands Reference version 15,” *pica8.com*, January 2015.

[13] Xu Yansen, “The Event System in POX,” July 29, 2015. [Online]. Available: http://xuyansen.work/the-event-system-in-pox/. [Accessed April. 29, 2018].

APPENDIX A

SWITCHGRAPH12A.PY

```
# original learning-algorithm code copyright 2012 James McCauley
#
# pld: this version runs all connections through s1/sK
# but allows a connection to be migrated to an si/sK+i
# This version ASSUMES the N,K-double-bell
# problem: given an sj host switch, how do we figure out how to
# forward to si?

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# Primary dictionaries:
#
# smap: map of switches dpid -> SwitchNode
# flowdict:  dictionary of <Connection, FlowInfo> pairs
# conn_to_path: map of (one-way) Connections to paths: lists of hosts/switches traversed.

# switchgraph1: first table looks at dest, second table looks at src

# switchgraph7:
# * support for SIGUSR1, SIGUSR2
# * fixed handle_flow_stat() so a flow goes into flowdict only if it crosses from one side to the
other
# * added support for idle_timeout
# changed Connection.__str__ output format

# switchgraph8:
# * make addTCProute() UNIdirectional, so a separate route must be entered for the reversed path.
# * separate finding the path of a connection from setting up the connection
# * improved printing in handle_linkEvent()
# * deleted connections if they don't show up in a handle_flow_stat report
# *     because the reporting switch has timed the connection out for exceeding idle_timeout

# switchgraph12:
# fixed an error in install_icmp_entry() (wrong packet type for ARP), and removed the spurious
default clause in set_default_action() (there can only be one default clause!)
```

```
"""
This learning switch requires Nicira extensions as found in Open vSwitch.
Furthermore, you must enable packet-in conversion.  Run with something like:

  ./pox.py openflow.discovery openflow.nicira –convert-packet-in log.level –WARNING
forwarding.switchgraph2 –N=5 –K=3

       h1–s7   ...                          ...  s12–h6
       h2–s8   ...  s1----------s4  ...  s13–h7
       h3–s9   ...  s2----------s5  ...  s14–h8
       h4–s10  ...  s3----------s6  ...  s15–h9
       h5–s11  ...                          ...  s16–h10


This forwards based on ethernet source and destination addresses.
This component uses two tables on the switch – one for source addresses
and one for destination addresses.  Specifically, we use tables 0 and 1
on the switch to implement the following logic:
0. Is this dest address known?
   NO:  flood
   YES: forward
   Either case: send to Table 1
1. Is this source address known?
   YES:  do nothing  (= drop)
   NO: send to controller

Table initialization is done by _handle_ConnectionUp,
including a rule to forward unknown packets to the controller.
We then add new entries for ARP/ICMP traffic when such packets arrive.

  'OFPP_MAX'          : 65280,
  'OFPP_FLOOD'        : 65531,
  'OFPP_CONTROLLER' : 65533,

"""

from pox.core import core
from pox.lib.addresses import EthAddr
import pox.openflow.libopenflow_01 as of
import pox.openflow.nicira as nx
import pox.lib.packet as pkt
from pox.lib.revent import EventRemove
import pox.lib.util as util
import time
import threading
import signal

TCPstarted = False    # flag used to identify start of TCP traffic, which generally comes AFTER
all hosts have been identified

BROADCAST = EthAddr('ff:ff:ff:ff:ff:ff')

SLEEPTIME = 2    #10           # time in seconds

ICMP_IDLE_TIMEOUT = 0
TCP_IDLE_TIMEOUT  = 0     # 10301  # 10301

# pld: should get N and K from the command line

N=5
K=3

# two SwitchNodes should be the same if their dpidvals are the same
class SwitchNode:
       # dpid, nmap, hmap
       def __init__(self, id, connection = None):
       self.dpidval = id
       self.nmap = {}   # map of (port, SwitchNode)
```

```
        self.hmap = {}   # map of (port, EthAddr)
        self.flagval = 0
        self.connexion = connection
        self.reservedports = []        # when all is done, this should be []

        def __str__(self):
        return "s" + str(self.dpidval)

        def __repl__(self):
        return "s" + str(self.dpidval)

        def dpid(self):
        return self.dpidval

        def switchPorts(self):    # returns list of ports that lead to other switches
        return self.nmap.keys()

        def hostPorts(self):    # returns list of ports that lead to hosts
        return self.hmap.keys()

        def setReservedPorts(self, portlist): # list of switch's ports, without knowing what they
connect to
        self.reservedports = []
        for p in portlist:
        if not (p in self.nmap) and not (p in self.hmap):
                self.reservedports.append(p)

        def switchNeighbors(self):  # returns list of all switch neighbors
        return self.nmap.values()

        def hostNeighbors(self):  # returns list of all switch neighbors
        return self.hmap.values()

        def addSwitchNeighbor(self, port, n):    # n is the neighbor SwitchNode
        if not port in self.reservedports:
                print "{}.addSwitchNeighbor({},{}): port {} not reserved".format(self, port, n,
port)
        self.nmap[port] = n
        if port in self.reservedports: self.reservedports.remove(port)

        def addHostNeighbor(self, port, n):           # n is an EthAddr
        if not port in self.reservedports:
                print "{}.addHostNeighbor({},{}): port {} not reserved".format(self, port, n,
port)
        self.hmap[port] = n
        if port in self.reservedports: self.reservedports.remove(port)

        def switchNeighbor(self, port):    # returns the switch reached by that port
        if port in self.nmap:
                return self.nmap[port]
        else: return None

        def hostNeighbor(self, port):    # returns the host reached by that port
        if port in self.hmap:
                return self.hmap[port]
        else: return None

        # returns the port needed to reach the given switch (by SwitchNode), or None
        def portToSwitchNeighbor(self, switch):
        for p in self.nmap:
        if self.switchNeighbor(p) == switch: return p
        return None

        # returns the port needed to reach the given host (by EthAddr), or None
        def portToHostNeighbor(self, host):
        for p in self.hmap:
        if self.hostNeighbor(p) == host: return p
        return None
```

```
        def setFlag(self, val):
        self.flagval = val

        def setFlag0(self):
        self.flagval = 0

        def flag(self):
        return self.flagval

        def connection(self):
        return self.connexion

        def setConnection(self, conn):
        self.connexion = conn

        def __hash__(self):
        return self.dpidval

        def __eq__(self, other):
        if isinstance(other, self.__class__):
                return self.dpidval == other.dpidval
        else:
                return False


class Connection:

        def __init__(self, ethsrc, ethdst, srcip, dstip, srcport, dstport):
        self.ethsrc = ethsrc
        self.ethdst = ethdst
        self.srcip  = srcip
        self.dstip  = dstip
        self.srcport= srcport
        self.dstport= dstport

        def reverse(self):
        return Connection(self.ethdst, self.ethsrc, self.dstip, self.srcip, self.dstport,
self.srcport)

        def __str__(self):
        return '(h{}->h{},{},{},{},{})'.format(hostnum(self.ethsrc), hostnum(self.ethdst),
self.srcip, self.dstip, self.srcport, self.dstport)
        #return '({},{},{},{},{},{})'.format(self.ethsrc, self.ethdst,self.srcip, self.dstip,
self.srcport, self.dstport)

        def __hash__(self):
        return hash((self.ethdst, self.ethsrc, self.dstip, self.srcip, self.dstport,
self.srcport))

        def __eq__(self, other):
        return self.ethsrc == other.ethsrc and self.ethdst == other.ethdst and self.srcip ==
other.srcip and self.dstip == other.dstip and self.srcport == other.srcport

        def crosses(self):
        if hostnum(self.ethsrc) <= N and hostnum(self.ethdst) > N: return True
        if hostnum(self.ethsrc) > N and hostnum(self.ethdst) <= N: return True
        return False

        def top_to_bottom(self):
        if hostnum(self.ethsrc) <= N and hostnum(self.ethdst) > N: return True
        return False

smap = {}  # map of all switches, by dpid: <dpid, SwitchNode>

conn_to_path = {}   # map of (one-way) Connections to list from host to host.

# sets the flag on all switches reachable from s, port p,
```

```
# *without* going through a switch in the list "excluded".
# s itself may be in the excluded list
# excluded is a list of switch dpids, not SwitchNodes.
# flag is set to k
# returns a list of all nodes marked with k

def markall (startsw, p, k, excluded):
   print "starting markall at", startsw
   markedlist = []
   firstnode = startsw.switchNeighbor(p)
   if firstnode == None:
        print "{}[{}] goes nowhere".format(startsw, p)
   if firstnode.dpid() in excluded: return
   todo = [firstnode]     # list of SwitchNode objects
   while todo != []:
        newtodo = []
        for s in todo:
        s.setFlag(k)
        markedlist.append(s)
        # print "marking", s
        for n in s.switchNeighbors():
               if n.flag() == k: break    # already marked
               if n.dpid() in excluded: break
               if n in newtodo: break
               newtodo.append(n)
        #
        todo = newtodo
   return markedlist

def floodtest():
        print "starting floodtest"
        s1 = smap[1]
        exclude = range(1,K+1)
        markedlist = markall(s1, 2, 37, exclude)
        print "mark test at s1:", strlist(markedlist)
        print "done with floodtest"

def pathtest():
        print "starting pathtest"
        printpath(2*K+1,2*K+N+4,2)
        printpath(2*K+5,2*K+N+1,3)
        print "done with pathtest"

# prints switches with ports from dpid1 to dpid2 via dpidtrunk
def printpath(dpid1, dpid2, dpidtrunk):
        if dpid1 <= 2*K or dpid1 > 2*K+N:
        print "bad dpid1:", dpid1
        return
        if dpid2 <= 2*K +N or dpid2 > 2*K + 2*N:
        print "bad dpid2:", dpid2
        return
        if dpidtrunk > K:
        print "bad trunk switch:", dpidtrunk
        return
        s1 = smap[dpid1]                    # starting switch, at "left"
        st1= smap[dpidtrunk]                # "left" trunk switch
        st2= smap[switchpeer(dpidtrunk)]   # "right" trunk switch
        s2 = smap[dpid2]                    # ending switch, at "right"
        p1   =  s1.portToSwitchNeighbor(st1)
        pt1a = st1.portToSwitchNeighbor(s1)
        pt1b = st1.portToSwitchNeighbor(st2)
        pt2a = st2.portToSwitchNeighbor(st1)
        pt2b = st2.portToSwitchNeighbor(s2)
        p2   =  s2.portToSwitchNeighbor(st2);
        print "{}[{}]--[{}]{}[{}]--------[{}]{}[{}]--[{}]{}".format(
        s1,p1,
        pt1a,st1,pt1b,
        pt2a,st2,pt2b,
```

```
        p2,s2
        )


def strlist(x):
        if x==[]: return '[]'
        res='[' + str(x[0])
        for s in x[1:]:
        res += ', ' + str(s)
        res += ']'
        return res

def switchpeer(i):
        if i<=K: return i+K
        return i-K


# dpid to monitor
#mon_dpid = 4

# Even a simple usage of the logger is much nicer than print!
log = core.getLogger()

def flooder(dpid):
  if dpid == 1 or dpid == K+1:
        return True
  # if dpid == 1: return True
  if dpid <= 2*K:
        #print "non-flooder: dpid=", dpid
        return False
  return True

# frontline switches are the entry switches, eg s7-s16
def frontline(dpid):
        if dpid > 2*K: return True
        return False

########  _handle_PacketIn  ######################

# PacketIn should tell us what switch ports connect to HOSTS

def _handle_PacketIn (event):
        global TCPstarted, conn_to_path
        packet = event.parsed
        packet_in = event.ofp     # The actual ofp_packet_in message.
        psrc = packet.src
        pdst = packet.dst
        inport = packet_in.in_port    # is this the same as event.port?
        assert inport == event.port, "inport {} not equal to event.port {}".format(inport,
event.port)
        dpid = event.connection.dpid
        if event.port > of.OFPP_MAX:
        log.debug("Ignoring special port %s", event.port)
        return

        # handle_PacketIn ignores the trunk switches except for s[1]-s[K+1]
        # the other trunk paths are used ONLY when paths are created.
        if not flooder(dpid): return

        if isdhcp(packet): return      # pld: ignore DHCP traffic

        # see if this packet came from a known switch
        if dpid in smap.keys():
        switch = smap[dpid]
        else:
        switch = None
        print "unknown switch s{}".format(dpid)       # no point continuing?
```

```
        # was this packet forwarded FROM another switch?
        isFromSwitch = (inport in switch.switchPorts())

        # if this port isn't in the switchport list, assume it's a direct host connection
        if not isFromSwitch:
        # if we have NOT seen psrc before as a host
        if not psrc in switch.hostNeighbors():
                if ishost(psrc):
                print "{}.{} <-> h{} [type {}]".format(switch, inport, hostnum(psrc),
format(packet.type,'04x'))
                switch.addHostNeighbor(inport, psrc)
                else:  # from a host port but not a normal host
                udp = packet.find('udp')
                if udp is not None and udp.srcport != 67 and udp.dstport != 67:
                        print "{}: sees weird packet from {} via port {}".format(switch,psrc,
inport)
                #switch.addHostNeighbor(inport, psrc)

        icmp = packet.find('icmp')
        arp = packet.find('arp')
        ipv4= packet.find('ipv4')
        tcp = packet.find('tcp')

        if (icmp is not None or arp is not None):
        install_icmp_entry(event, psrc)
        return

        if tcp is None:
        if packet.type != 0x800:
                print "unknown packet type:", packet.type
        else:
                udp = packet.find('udp')
                if udp is None:
                print 'unknown packet, not UDP or TCP'
                elif udp.srcport == 67 or udp.dstport == 67:    # dhcp
                pass
                else:
                print "unknown udp packet from ({},{}) to ({},{})".format(
                        ipv4.srcip, udp.srcport, ip.dstip, udp.dstport)
        return

        # now we know it's a TCP packet
        if not TCPstarted and tcp is not None:
        TCPstarted = True
        TCPstart()
        if tcp is not None:
        #if hostnum(psrc) <= N and hostnum(pdst) >= N:
        conn = Connection(psrc, pdst, ipv4.srcip, ipv4.dstip, tcp.srcport, tcp.dstport)
        addTCProute(conn, smap[2])
        # shortcut to creating the reverse path:
        """
        rpath = revlist(conn_to_path[conn])
        rconn = conn.reverse()
        create_path_entries(rconn, rpath)
        conn_to_path[rconn] = rpath
        """
        addTCProute(conn.reverse(), smap[3])    # try having the reverse traffic take a different
path?

# this installs entries for ICMP traffic, and also ARP

def install_icmp_entry(event, psrc):
        #packet = event.parsed
        #psrc = packet.src
        #pdst = packet.dst
        # FINALLY add to the tables. Only ICMP and ARP packets should get here.
        # First, source table. pld: this is now table 1
        msg = nx.nx_flow_mod()
```

```
        #msg.match = of.ofp_match()    # pld: does this even exist? or use nx_match()?
        msg.match = nx.nx_match()   # pld: see pox dox "Using nx_match"
        msg.table_id = 1
        #msg.match.of_eth_src = psrc
        #msg.match.dl_type  = pkt.ethernet.IP_TYPE                  # IPv4
        #msg.match.nw_proto = pkt.ipv4.ICMP_PROTOCOL         # ICMP
        msg.match.append(nx.NXM_OF_ETH_SRC(psrc))
        msg.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.IP_TYPE))
        msg.match.append(nx.NXM_OF_IP_PROTO(pkt.ipv4.ICMP_PROTOCOL))
        # empty action list here!
        event.connection.send(msg)

        # Add to destination table. pld: this is now table 0
        # add flow for ICMP
        msg = nx.nx_flow_mod()    # pld: was "lmsg": woe
        msg.match = nx.nx_match()   # pld: see pox dox "Using nx_match"
        msg.table_id = 0
        msg.match.append(nx.NXM_OF_ETH_DST(psrc))
        msg.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.IP_TYPE))
        msg.match.append(nx.NXM_OF_IP_PROTO(pkt.ipv4.ICMP_PROTOCOL))

        msg.actions.append(of.ofp_action_output(port = event.port))
        msg.actions.append(nx.nx_action_resubmit.resubmit_table(table = 1))
        event.connection.send(msg)

        # now add entries for ARP
        msg = nx.nx_flow_mod()
        msg.match = nx.nx_match()   # pld: see pox dox "Using nx_match"
        msg.table_id = 1
        msg.match.append(nx.NXM_OF_ETH_SRC(psrc))
        msg.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.ARP_TYPE))
        event.connection.send(msg)

        msg = nx.nx_flow_mod()
        msg.match = nx.nx_match()   # pld: see pox dox "Using nx_match"
        msg.table_id = 0
        msg.match.append(nx.NXM_OF_ETH_DST(psrc))
        msg.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.ARP_TYPE))
        msg.actions.append(of.ofp_action_output(port = event.port))
        msg.actions.append(nx.nx_action_resubmit.resubmit_table(table = 1))
        # again, action list is empty
        event.connection.send(msg)

        log.info("Learning %s on port %s of %s"
        % (psrc, event.port, event.connection))
        #req_flow_stats1(event.connection)

        ##if dpid == mon_dpid and ishost(packet.src):
        ##   print "s"+str(dpid) +":", "packet from", packet.src, "arriving on port", event.port

########  addTCProute(tcppacket,ts)  ########################

# creates a ONE-WAY route for TCP traffic from ha to hb vi trunk switch ts
# pld CHANGE: replace ha/hb parameters with packet. Then ha = packet.src, hb = packet.dst
# AND we can also extract other packet attributes.
# ha, hb are EthAddr; ts1 is a SwitchNode.
# The forwarding entries created here just use table 0

def addTCProute(conn,ts1):
        global conn_to_path
        ha = conn.ethsrc
        hb = conn.ethdst
        ida = hostnum(ha)
        idb = hostnum(hb)
        print "adding TCP route h{} -> h{} via {}".format(ida, idb, ts1)
        #print "TCP/IP: {}.{} <-> {}.{}".format(conn.srcip, conn.srcport, conn.dstip,
conn.dstport)
        path = findpath(conn, ts1)
```

```
        create_path_entries(conn, path)
        conn_to_path[conn] = path
        return

def move_s2():
        """
        move flows from h1-h5 to h6-h10 that *were* through s2 to s3
        """
        s2 = smap[2]
        s3 = smap[3]
        for c in conn_to_path:
        if hostnum(c.ethsrc) > 5: continue
        if hostnum(c.ethdst) <= 5: continue
        path = conn_to_path[c]
        if not (s2 in path): continue
        print 'moving connection from s2 to s3: {}'.format(c)
        print 'old path:', path
        addTCProute(c, s3)
        # delete flow from s2, s5
        s5 = smap[5]
        for s in [s2, s5]:
        fm = of.ofp_flow_mod()
        fm.xid = None
        fm.command = of.OFPFC_DELETE
        s.connection().send(fm)

def move_flow(conn, newtrunk):
   """move Connection conn, if it exists, to newtrunk.
   If the connection already goes through newtrunk, do nothing
   newtrunk is a dpid (that is, is 1, 2, or 3.
   Initially assume that conn connects one end to the other
   """
   newtrunksw = smap[newtrunk]
   if not conn in conn_to_path: return
   path = conn_to_path[conn]
   if firsttrunk(path) == newtrunk: return
   start_host = hostnum(path[0])
   end_host = hostnum(path[len(path)-1])
   oldtrunk = gettrunk(path)
   if oldtrunk == newtrunk:
        return
   if start_host <= N and end_host <= N:
        print "can't move non-crossing connection {}".format(conn)
        return
   if start_host > N and end_host > N:
        print "can't move non-crossing connection {}".format(conn)
        newpath = findpath(conn, newtrunksw)
        create_path_entries(conn, newpath)
        conn_to_path[conn] = newpath
        # delete old entries on path
        unused_switches = path[2:4]
        for s in unused_switches:
        delTCPrule(conn, s)
        #fm = of.ofp_flow_mod()
        #fm.xid = None
        #fm.command = of.OFPFC_DELETE
        #s.connection().send(fm)
        return
   if start_host <= N and end_host > N:
        print 'moving connection {} from s{} to s{}'.format(conn, oldtrunk, newtrunk)
        pivotswitch = path[1]
        newpath = findpath(conn, newtrunksw)
        create_path_entries2(conn, newpath, pivotswitch)
        conn_to_path[conn] = newpath
        # delete old entries on path
        unused_switches = path[2:4]
        for s in unused_switches:
        delTCPrule(conn,s)
```

```
        #fm = of.ofp_flow_mod()
        #fm.xid = None
        #fm.command = of.OFPFC_DELETE
        #s.connection().send(fm)
        #pass
   if start_host > N and end_host <= N:
        print 'moving connection {} from s{} to s{}'.format(conn, oldtrunk, newtrunk)
        pivotswitch = path[4]
        newpath = findpath(conn, newtrunksw)
        create_path_entries2(conn, newpath, pivotswitch)
        conn_to_path[conn] = newpath
        # delete old entries on path
        unused_switches = path[2:4]
        for s in unused_switches:
        delTCPrule(conn,s)
        #fm = of.ofp_flow_mod()
        #fm.xid = None
        #fm.command = of.OFPFC_DELETE
        #s.connection().send(fm)
        #print 'moving connection from s5 to s6: {}'.format(c)
        #pass

# pld: this was REALLY incomplete originally!
# it mirrors the original creation of the entry in addTCPrule()
def delTCPrule(c,s):
        msg = nx.nx_flow_mod()
        msg.match = nx.nx_match()   # pld: see pox dox "Using nx_match"
        msg.table_id = 0
        #msg.idle_timeout = TCP_IDLE_TIMEOUT
        #msg.match.of_eth_dst = psrc
        msg.match.append(nx.NXM_OF_ETH_SRC(c.ethsrc))
        msg.match.append(nx.NXM_OF_ETH_DST(c.ethdst))
        msg.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.IP_TYPE))
        msg.match.append(nx.NXM_OF_IP_PROTO(pkt.ipv4.TCP_PROTOCOL))
        msg.match.append(nx.NXM_OF_IP_SRC(c.srcip))
        msg.match.append(nx.NXM_OF_IP_DST(c.dstip))
        msg.match.append(nx.NXM_OF_TCP_SRC(c.srcport))
        msg.match.append(nx.NXM_OF_TCP_DST(c.dstport))
        msg.command = of.OFPFC_DELETE
        s.connection().send(msg)


def firsttrunk(path):
   """ returns 1, 2 or 3 depending on whether the path goes through s1, s2 or s3
   """
   for i in range(1,K+1):
        s = smap[i]
        if s in path: return i
   return -1


# ts here is one of s1,s2,s3
def findpath(conn, ts):
        ha = conn.ethsrc
        hb = conn.ethdst
        ida = hostnum(ha)
        idb = hostnum(hb)
        if ida <= N and idb <= N:   # both on upper side
        return path_both_upper(conn, ts)
        if ida > N and idb > N:       # both on lower side
        ts2 = smap[switchpeer(ts.dpid())]
        return path_both_lower(conn, ts2)
        if ida > N and idb <=N:       # lower to upper: just reverse the direction
        #print "swapping", ida, "and", idb
        return revlist(path_upper_to_lower(conn.reverse(),ts))
        return path_upper_to_lower(conn,ts)
```

```
# creates TCP rules at each switch in the path.
# The first and last entries of the path must be hosts.
# switch entries on the path are of type SwitchNode, not dpids
# path must contain at least one switch!
# ha->s1->s2->...->sM->hb

# option: have the LAST switch in the path (lastswitch, below) add a rule to delete the flow
# if the TCP FIN bit is set. That's not right; shouldn't delete flow until *both* sides have sent
FIN.
def create_path_entries(conn, path):
        plen = len(path)
        lastswitch = path[plen-2]
        addTCPrule(lastswitch, conn, lastswitch.portToHostNeighbor(path[plen-1]))
        i = plen-3
        while i > 0:
        sw = path[i]
        nsw = sw.portToSwitchNeighbor(path[i+1])    # neighbor switch
        if nsw == None:
                print "bad path to create_path_entries: {} and {} not connected".format(sw, nsw)
        addTCPrule(sw, conn, nsw)
        i -= 1
        return

# like the above but with provision for pivotswitch, where an existing entry is *modified*
def create_path_entries2(conn, path, pivotswitch):
        plen = len(path)
        lastswitch = path[plen-2]
        addTCPrule(lastswitch, conn, lastswitch.portToHostNeighbor(path[plen-1]))
        i = plen-3
        while i > 0:
        sw = path[i]
        nsw = sw.portToSwitchNeighbor(path[i+1])    # neighbor switch
        if nsw == None:
                print "bad path to create_path_entries: {} and {} not connected".format(sw, nsw)
        if sw == pivotswitch:
                print 'Modifying switch {} for connection {}'.format(sw, conn)
                modTCPrule(sw, conn, nsw)
        else:
                addTCPrule(sw, conn, nsw)
        i -= 1
        return


# the following should move a connection to go through trunk switch ts:
# maybe we need the entire connection path as a parameter?
def moveTCProute(conn, ts):
        return

# ts here is one of s1,s2,s3
def path_both_upper(conn, ts):
        ha = conn.ethsrc
        hb = conn.ethdst
        ida = hostnum(conn.ethsrc)
        idb = hostnum(conn.ethdst)
        print 'calling path_both_upper(h{}->h{},{},{},{},{},{})'.format(ida, idb, conn.srcip,
conn.dstip,conn.srcport, conn.dstport, ts)
        assert ida <= N, "source host {} not on LHS".format(ha)
        assert idb <= N, "dest host {} not on RHS".format(hb)
        sa = smap[hostswitch(ida)] # switch ha connects to
        sb = smap[hostswitch(idb)]
        #ts2 = smap[switchpeer(ts1.dpid())]
        path = [ha, sa, ts, sb, hb]
        return path
        #conn_to_path[conn] = route
        #conn_to_path[conn.reverse()] = route[::-1]
        # install (ha->hb route)
        # ha-sa---ts1---sb-hb
```

```
        #sb:  forward out port sb.portToHostNeighbor(hb)
        #addTCPrule(sb,  conn, sb.portToHostNeighbor(hb) )
        #ts1: forward out port ts1.portToSwitchNeighbor(sb)
        #addTCPrule(ts1, conn, ts1.portToSwitchNeighbor(sb) )
        #sa:  forward out port sa.portToSwitchNeighbor(ts1)
        #addTCPrule(sa,  conn, sa.portToSwitchNeighbor(ts1) )

# ts2 here is one of s4,s5,s6
def path_both_lower(conn, ts2):
        ha = conn.ethsrc
        hb = conn.ethdst
        ida = hostnum(conn.ethsrc)
        idb = hostnum(conn.ethdst)
        print 'calling path_both_upper(h{}->h{},{},{},{},{},{})'.format(ida, idb, conn.srcip,
conn.dstip,conn.srcport, conn.dstport, ts2)
        assert ida > N, "source host {} not on LHS".format(ha)
        sa = smap[hostswitch(ida)] # switch ha connects to
        sb = smap[hostswitch(idb)]
        assert idb > N, "dest host {} not on RHS".format(hb)
        route = [ha, sa, ts2, sb, hb]
        return route
        #conn_to_path[conn] = route
        #conn_to_path[conn.reverse()] = route[::-1]
        # install (ha-->hb route)
        # ha-sa---ts2---sb-hb
        #sb:  forward out port sb.portToHostNeighbor(hb)
        #addTCPrule(sb,  conn, sb.portToHostNeighbor(hb) )
        #ts2: forward out port ts2.portToSwitchNeighbor(sb)
        #addTCPrule(ts2, conn, ts2.portToSwitchNeighbor(sb) )
        #sa:  forward out port sa.portToSwitchNeighbor(ts2)
        #addTCPrule(sa,  conn, sa.portToSwitchNeighbor(ts2) )
        #return

# upper_to_lower takes a Connection conn, and a trunk switch ts1, and creates
# creates routes for BOTH conn and conn.reverse().
# each route goes through trunk switch ts1.

def path_upper_to_lower(conn, ts1):
        ha = conn.ethsrc
        hb = conn.ethdst
        ida = hostnum(conn.ethsrc)
        idb = hostnum(conn.ethdst)
        # print 'calling upper_to_lower(h{}->h{},{},{},{},{},{})'.format(ida, idb, conn.srcip,
conn.dstip, conn.srcport, conn.dstport, ts1)
        assert ida <= N, "source host {} not on LHS".format(ha)
        sa = smap[hostswitch(ida)] # switch ha connects to
        sb = smap[hostswitch(idb)]
        assert idb > N, "dest host {} not on RHS".format(hb)
        ts2 = smap[switchpeer(ts1.dpid())]
        # install (ha->hb route)
        # ha-sa-ts1----ts2-sb-hb
        route = [ha, sa, ts1, ts2, sb, hb]
        return route
        #conn_to_path[conn] = route
        #sb:  forward out port sb.portToHostNeighbor(hb)
        #addTCPrule(sb,  conn, sb.portToHostNeighbor(hb) )
        #ts2: forward out port ts2.portToSwitchNeighbor(sb)
        #addTCPrule(ts2, conn, ts2.portToSwitchNeighbor(sb) )
        #ts1: forward out port ts1.portToSwitchNeighbor(ts2)
        #addTCPrule(ts1, conn, ts1.portToSwitchNeighbor(ts2) )
        #sa:  forward out port sa.portToSwitchNeighbor(ts1)
        #addTCPrule(sa,  conn, sa.portToSwitchNeighbor(ts1) )

        #rconn = conn.reverse()
        #conn_to_path[rconn] = route[::-1]
        # reverse:
        #addTCPrule(sa,  rconn,  sa.portToHostNeighbor(ha) )
        #addTCPrule(ts1, rconn, ts1.portToSwitchNeighbor(sa) )
```

```
        #addTCPrule(ts2, rconn, ts2.portToSwitchNeighbor(ts1) )
        #addTCPrule(sb,  rconn,  sb.portToSwitchNeighbor(ts2) )
        #conn_to_path[rconn] = route[::-1]



def addTCPrule(switch, conn, port):
        assert (port in switch.hmap) or (port in switch.nmap), "{}: unknown port
{}".format(switch, port)
        psrc = conn.ethsrc
        pdst = conn.ethdst
        msg = nx.nx_flow_mod()
        msg.match = nx.nx_match()   # pld: see pox dox "Using nx_match"
        msg.table_id = 0
        msg.idle_timeout = TCP_IDLE_TIMEOUT
        #msg.match.of_eth_dst = psrc
        msg.match.append(nx.NXM_OF_ETH_SRC(conn.ethsrc))
        msg.match.append(nx.NXM_OF_ETH_DST(conn.ethdst))
        msg.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.IP_TYPE))
        msg.match.append(nx.NXM_OF_IP_PROTO(pkt.ipv4.TCP_PROTOCOL))
        msg.match.append(nx.NXM_OF_IP_SRC(conn.srcip))
        msg.match.append(nx.NXM_OF_IP_DST(conn.dstip))
        msg.match.append(nx.NXM_OF_TCP_SRC(conn.srcport))
        msg.match.append(nx.NXM_OF_TCP_DST(conn.dstport))
        msg.actions.append(of.ofp_action_output(port = port))
        switch.connection().send(msg)
        # other match options, if IPv4 addrs or TCP ports are passed in:
        # NXM_OF_IP_SRC, NXM_OF_IP_DST
        # NXM_OF_TCP_SRC, NXM_OF_TCP_DST

# like the above but just modifying the port
# HOW DO WE DO THIS??? See Modify Actions in Existing Flow Entries, and OFPFC_MODIFY
def modTCPrule(switch, conn, port):
        assert (port in switch.hmap) or (port in switch.nmap), "{}: unknown port
{}".format(switch, port)
        psrc = conn.ethsrc
        pdst = conn.ethdst
        # msg = nx.nx_flow_mod(command=of.OFPFC_DELETE, table_id = 1)
        msg = nx.nx_flow_mod()         # (command=of.OFPFC_MODIFY, table_id=0)
        msg.command = of.OFPFC_MODIFY
        msg.table_id = 0
        #msg.idle_timeout = TCP_IDLE_TIMEOUT
        #msg.match.of_eth_dst = pdst
        msg.match = nx.nx_match()   # pld: see pox dox "Using nx_match"
        msg.match.append(nx.NXM_OF_ETH_SRC(conn.ethsrc))
        msg.match.append(nx.NXM_OF_ETH_DST(conn.ethdst))
        msg.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.IP_TYPE))
        msg.match.append(nx.NXM_OF_IP_PROTO(pkt.ipv4.TCP_PROTOCOL))
        msg.match.append(nx.NXM_OF_IP_SRC(conn.srcip))
        msg.match.append(nx.NXM_OF_IP_DST(conn.dstip))
        msg.match.append(nx.NXM_OF_TCP_SRC(conn.srcport))
        msg.match.append(nx.NXM_OF_TCP_DST(conn.dstport))
        msg.actions.append(of.ofp_action_output(port = port))
        switch.connection().send(msg)
        # other match options, if IPv4 addrs or TCP ports are passed in:
        # NXM_OF_IP_SRC, NXM_OF_IP_DST
        # NXM_OF_TCP_SRC, NXM_OF_TCP_DST


########  _handle_LinkEvent  ######################

# pld: LinkEvents don't need the "barrier" trick that PacketIn events do.
# Although we really would like to know when all the LinkEvents are received.

def _handle_LinkEvent(event):
        l = event.link
        #print l
        sw1 = l.dpid1
```

```
        sw2 = l.dpid2
        pt1 = l.port1
        pt2 = l.port2
        #  link from (sw1,pt1) to (sw2,pt2); may or may not be new
        # if sw2 < sw1: sw1,pt1,sw2,pt2 = sw2,pt2,sw1,pt1
        if sw2 < sw1: return
        sw1s = str(sw1)
        pt1s = str(pt1)
        sw2s = str(sw2)
        pt2s = str(pt2)
        #print 'link added is %s'%event.added
        #print 'link removed is %s' %event.removed
        if event.added:
                change = 'added'
        else:
                change = 'removed'
                return          # pld: TEMPORARILY don't do anything when links go down
        #print change+':', 's'+sw1s+'.'+pt1s, '<->', 's'+sw2s+'.'+pt2s
        # look up switches in smap (or install them)
        if sw1 in smap:
                s1 = smap[sw1]
        else:
                s1 = SwitchNode(sw1)
                smap[sw1] = s1
        if sw2 in smap:
                s2 = smap[sw2]
        else:
                s2 = SwitchNode(sw2)
                smap[sw2] = s2
        s1pt1 = s1.switchNeighbor(pt1)   # old neighbor s1[pt1]

        # this is a new report if s1pt1 == None
        if s1pt1 != None and s1pt1 != s2:
                print "warning: switch {} changed port {} neighbor from {} to {}".format(s1, pt1,
s1pt1, s2)
        else:
                pass
                #print "switch {} gets port {} neighbor {}".format(s1,pt1,s2)

        # if s1pt1 != None and s1pt1 == s2: do nothing
        if s1pt1 != s2:
                s1.addSwitchNeighbor(pt1, s2)

        s2pt2 = s2.switchNeighbor(pt2)   #old neighbor s2[pt2]
        if s2pt2 != None and s2pt2 != s1:
                print "warning: switch {} changed port {} neighbor from {} to {}".format(s2, pt2,
s2pt2, s1)
        else:
                pass
                #print "switch {} gets port {} neighbor {}".format(s2,pt2,s1)

        if s2pt2 != s1:
                s2.addSwitchNeighbor(pt2, s1)

        if s1pt1 == None and s2pt2 == None:
                print 'adding switch connection:', 's'+sw1s+'.'+pt1s, '<->', 's'+sw2s+'.'+pt2s
        elif s1pt1==None:    # but s2 has s1 as a neighbor already
                print 'adding switch connection:', 's'+sw1s+'.'+pt1s, '->', 's'+sw2s+'.'+pt2s
        elif s2pt2==None:
                print 'adding switch connection:', 's'+sw2s+'.'+pt2s, '->', 's'+sw1s+'.'+pt1s

########### _handle_ConnectionUp  ##################

def _handle_ConnectionUp (event):
        # Initialize the forwarding rules for this switch.
        # After setting up, we send a barrier and wait for the response
        # before starting to listen to packet_ins for this switch - before
        # the switch is set up, the packet_ins may not be what we expect,
```

```
        # and our responses may not work!

        print "handle_ConnectionUP from dpid", event.connection.dpid,
util.dpid_to_str(event.connection.dpid)
        portlist = event.connection.ports.values()
        # get port_no of each item in portlist
        portlist = map(lambda x: x.port_no, portlist)
        portlist = filter(lambda x: x < of.OFPP_MAX, portlist)
        # print "portlist:", portlist
        dpid = event.connection.dpid
        connection = event.connection
        # Turn on Nicira packet_ins
        msg = nx.nx_packet_in_format()
        #event.connection.send(msg)
        connection.send(msg)

        # Turn on this switch's ability to specify tables in flow_mods
        msg = nx.nx_flow_mod_table_id()
        connection.send(msg)

        # Clear second table
        msg = nx.nx_flow_mod(command=of.OFPFC_DELETE, table_id = 1)
        connection.send(msg)


        # this version sets default flooding actions only for ICMP and ARP packets
        # (though there IS a rule to send unknown packets to the controller)
        def set_default_action(connection):
        # pld fallthrough rule for table 0: flood (IF a flooder) and send to table 1
        # CHANGE jan 27, 2017: only create flood rules for ICMP and ARP packets
        # match on packet type, but not on source
        msgi = nx.nx_flow_mod()  # icmp msg
        msga = nx.nx_flow_mod()  # arp msg
        msgi.table_id = msga.table_id = 0
        msgi.priority = msga.priority = 1 # Low priority
        msgi.idle_timeout = msga.idle_timeout = ICMP_IDLE_TIMEOUT

        msgi.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.IP_TYPE))
        msgi.match.append(nx.NXM_OF_IP_PROTO(pkt.ipv4.ICMP_PROTOCOL))
        msga.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.ARP_TYPE))

        if flooder(connection.dpid):
                msgi.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
                msga.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
        msgi.actions.append(nx.nx_action_resubmit.resubmit_table(table = 1))
        msga.actions.append(nx.nx_action_resubmit.resubmit_table(table = 1))
        connection.send(msgi)
        connection.send(msga)

        # here we create a default rule to send ANY packet to the controller
        # May 29: only do this for "frontline" (entry) switches, not trunk switches
        if frontline(connection.dpid):
                msg = nx.nx_flow_mod()
                msg.table_id = 0
                msg.priority = 0    # rules for ARP/ICMP should be higher priority
                msg.idle_timeout = ICMP_IDLE_TIMEOUT
                msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
                connection.send(msg)

        # pld fallthrough rule for table 1: send to controller
        msgi = nx.nx_flow_mod()  # icmp msg
        msga = nx.nx_flow_mod()  # arp msg
        msgi.table_id = msga.table_id = 1
        msgi.priority = msga.priority = 1 # Low priority
        msgi.idle_timeout = msga.idle_timeout = ICMP_IDLE_TIMEOUT

        msgi.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.IP_TYPE))
        msgi.match.append(nx.NXM_OF_IP_PROTO(pkt.ipv4.ICMP_PROTOCOL))
```

```
        msga.match.append(nx.NXM_OF_ETH_TYPE(pkt.ethernet.ARP_TYPE))

        msgi.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
        msga.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
        connection.send(msgi)
        connection.send(msga)

        # set default action
        set_default_action(connection)

        def ready (event):      # called right below, as parameter
        if event.ofp.xid != 0x80000000:
                # Not the right barrier
                return
        log.info("%s ready", event.connection)
        event.connection.addListenerByName("PacketIn", _handle_PacketIn)
        return EventRemove

        connection.send(of.ofp_barrier_request(xid=0x80000000))
        connection.addListenerByName("BarrierIn", ready)

        # now install switch
        if dpid in smap:
        sw = smap[dpid]
        if sw.connection() is None:
                sw.setConnection(connection)
        else:
        sw = SwitchNode(dpid, connection)
        smap[dpid] = sw
        # now add empty port list
        sw.setReservedPorts(portlist)

######################  flow stats  ############################
# When we get flow stats, put it in this dictionary
# identify the connection object

BWERROR = 0.0123456789

class FlowInfo:

        def __init__(self, lasttime, bytes, packets, idle_timeout, dpid):
        self.lasttime = lasttime
        self.prevbytes = 0
        self.bytes = bytes
        self.packets = packets
        self.idle_timeout = idle_timeout
        self.bw = None                  # not known yet
        self.dpid = dpid

        def update(self, lasttime, bytes, packets, idle_timeout, dpid):
        interval = lasttime - self.lasttime
        if bytes >= self.bytes:
                self.bw = (bytes - self.bytes)/interval
        else:
                self.bw = BWERROR
        self.lasttime = lasttime
        self.bytes = bytes
        self.packets=packets
        self.idle_timeout = idle_timeout
        if dpid != self.dpid:
                print 'flowinfo changed from s{} to s{}'.format(self.dpid, dpid)

        def __str__(self):
        if self.bw == None: bw=0
        else: bw = self.bw
        return '({},{} B,{} B/s,{} via s{})'.format(self.lasttime, self.bytes, bw, self.dpid) #,
self.idle_timeout)
```

```
# dictionary of <Connection, FlowInfo> pairs
# ONLY for connections passing through one of s1,s2,s3
# this is because request_flow_stats only requests info for the trunk switches s1,s2,s3
# PROBLEM: this means that the flow stats are reset when a connection changes trunk!

flowdict = {}

def printflowdict():
       i=1
       for c in flowdict:
       print "{}: connection {}".format(i, c)
       print flowdict[c]
       i+=1

# only switches in switchlist, below in request_flow_stats(), report.
# currently this is s1,s2,s3
# CHANGED TO s7,s8,s9,s10,s11, because a connection never changes its switch on this list
# we only report flows that actually use a trunk link;
# that is, NOT h1-s7-s2-s10-h4

# if we get an event from switch si (event.dpid == i),
#  and there's a connection in flowdict with si in its path,
# but that connection doesn't show up in the event list, REMOVE THE CONNECTION
# That connection was removed by a switch because of idle_timeout.
# Think hard about the setting of TCP_IDLE_TIMEOUT!

def handle_flow_stat (event):
  global flowdict
  switchlist = range(2*K+1, 2*K+N+1)
  conn = event.connection
  packets = 0
  byte_count = 0
  tcp_flow_count = 0
  dpid = event.dpid     # the switch that answered
  if dpid not in switchlist:
       print 'warning: handle_flow_stat received message from switch s{}'.format(dpid)
  sw = smap[dpid]
  connlist = []
  for f in event.stats:   # f is one of the switch's connection objects.
       fm = f.match
       # openflow 1.5.1 p 129
       # what is in wildcarded field? Apparently it is None
       if fm.tp_src == None: continue
       c = Connection(fm.dl_src, fm.dl_dst, fm.nw_src, fm.nw_dst, fm.tp_src, fm.tp_dst)
       # ignore c if fm.dl_src and fm.dl_dst are not on opposite sides!
       if not c.top_to_bottom(): continue    # pld: was c.crosses()
       #src = hostnum(fm.dl_src)
       #dst = hostnum(fm.dl_dst)
       #print '********connection from h{} to h{}'.format(src,dst)
       # ignore connections that don't go from one side to the other
       #if src <= N and dst <=N: continue
       #if src > N and dst > N: continue
       connlist.append(c)
       # the following are written out mostly to document the attribute names
       byte_count = f.byte_count      # was +=, which is probably wrong
       packets = f.packet_count
       duration_sec = f.duration_sec
       duration_nsec = f.duration_nsec
       idle_timeout = f.idle_timeout
       hard_timeout = f.hard_timeout
       priority = f.priority
       table_id = f.table_id
       show = f.show
       pack = f.pack
       unpack = f.unpack
       cookie = f.cookie
       if c in flowdict:
       #print 'updating connection {} to {}'.format(c, flowdict[c])
```

```
        oldbytes = flowdict[c].bytes
        if oldbytes > byte_count:
                print('WARNING: connection {} decreased byte_count from {} to {}'.format(c,
oldbytes, byte_count))
        flowdict[c].update(time.time(), byte_count, packets, idle_timeout, dpid)
        else:
        #print 'new connection: flowdict is'
        #dumpdict(flowdict)
        flowdict[c] = FlowInfo(time.time(), byte_count, packets, idle_timeout, dpid)
        #print 'handle_flow_stat: connection from s{}: {}, info={}'.format(conn.dpid, c,
flowdict[c])
        #web_flows += 1
        #print "dumping f"
        #print dir(f)
        # print "flow:", f.match, "bytes:", f.byte_count
  # now go through connections in flowdict.keys.
  # If a connection c has sw in its path conn_to_path[c], but c is NOT in connlist, delete it
  delete_list = []
  for c in flowdict:
        if not (c in conn_to_path):
        print 'handle_flow_stat: connection {} has no path!'.format(c)
        delete_list.append(c)
        continue    # c has no path!
        path = conn_to_path[c]
        if sw in path and not (c in connlist):     # connection *should* therefore be in connlist
        print 'handle_flow_stat: deleting Connection {} through {} from flowdict'.format(c, sw)
        delete_list.append(c)
  for c in delete_list:
        del flowdict[c]

  log.info("Traffic: %s bytes over %s flows", byte_count, tcp_flow_count)
  #print "Traffic:", byte_count, "bytes over", tcp_flow_count, "flows from", event.connection

def dumpdict(d):
   for k in d:
        print '({}->{})'.format(k, d[k])
   print

core.openflow.addListenerByName("FlowStatsReceived", handle_flow_stat)

shufflecount = 10000000    # max number of reshuffle() calls

# executed at intervals of SLEEPTIME > 1
def statsthread():
        global stopThread
        global shufflecount
        while True:
        for i in range(SLEEPTIME-1):
                time.sleep(1)
                if stopThread: exit(0)
        # print ('thread waking up')
        request_flow_stats()
        clean_expired_entries()
        time.sleep(1)
        if TCPstarted and shufflecount > 0:
                shufflecount -= 1
                reshuffle_flows()

def clean_expired_entries():
        pass

"""
Reshuffling plan:

1. Start with flowdict
2. convert to a LIST of (Connection,FlowInfo) pairs.
3. eliminate connections that do not go between the top (h1-hN) and the bottom (h[N+1]-h[2N])
4. order in decreasing order by f.bw
```

```
5. Create a table for trunk1 ... trunkK, of bw_allocations, each initialized to zero
6: the algorithm
    for each (c,f) in the list
        find the trunk ti with the smallest bw_allocation (or first such trunk, if there are
ties)
        assign connection c to trunk ti
        update the trunk's bw_allocation by adding f.bw
"""

def conn_bw(x):
       (c,f) = x
       if f.bw == None: return 0
       return -f.bw

def reshuffle_flows():
       cflist = flowdict.items()
       for (c,f) in cflist:
       if not c.top_to_bottom():    # pld: was c.crosses()
              cflist.delete((c,f))
       # sort by f.bw, in descending order
       # FINISH
       # cflist.sort(key=___w)   # lambda (c,f) : -f.bw)
       cflist.sort(key=conn_bw)   # lambda (c,f) : f.bw)
       trunk_usage={}   # map from trunk numbers 1,2,3 to assigned utilization
       for i in range(1,K+1):
       trunk_usage[i] = 0
       print "reshuffling starting"
       for (c,f) in cflist:
       #find the trunk  with the smallest bw_allocation (or first such trunk, if there are ties)
       trunk = find_min_trunk(trunk_usage)
        #assign connection c to trunk
       move_flow (c, trunk)
        #update the trunk's bw_allocation by adding f.bw
       if f.bw is None: bandwidth = 0
       else: bandwidth = f.bw
       trunk_usage[trunk] += bandwidth
       if c.top_to_bottom():
              #print('moving connection {} to trunk {}'.format(c,trunk))
              pass
       print ("trunk usage: 1: {}; 2: {}; 3: {}".format(trunk_usage[1], trunk_usage[2],
trunk_usage[3]))

# returns, eg, 2 if the trunk through s2 has the smallest trunk_usage value.
def find_min_trunk(trunk_usage):
   i = 1
   index = i
   min_so_far = trunk_usage[i]
   i += 1
   while i <= K:
        if trunk_usage[i] < min_so_far:
        index=i
        min_so_far = trunk_usage[i]
        i += 1
   return index

# pld: can you tell we were having trouble with this one?
# results of request are processed by handle_flow_stat()
def request_flow_stats():
  #global core
  switchlist = range(1,K+1)   # [1, 2, 3]    # use s7-s11?
  switchlist = range(2*K+1, 2*K+N+1)
  #print 'printing core.openflow.connections'
  #print type(core.openflow.connections)
  #print dir(core.openflow.connections)
  #print core.openflow.connections
  #for con in core.openflow.connections: # make this _connections.keys() for pre-betta
  #  #print "connection:", con.dpid
  #  switchlist = [1, 2, 3]    # send request only to these switches
```

```
  #  if con.dpid in switchlist:
  #     con.send(of.ofp_stats_request(body=of.ofp_flow_stats_request()))
  # end for
  #threading.Timer(10,request_flow_stats).start()
  for dpid in switchlist:
        if dpid in core.openflow.connections:
        con = core.openflow.connections[dpid]
        assert dpid == con.dpid, 'request_flow_stats(): connection mismatch: connections[{}]
=s{}'.format(dpid, con.dpid)
        con.send(of.ofp_stats_request(body=of.ofp_flow_stats_request()))
        else:
        print "warning: request_flow_stats() cannot reach s{}".format(dpid)


st=None
cv=None
stopThread = False

def intr_handler(sig, frame):
        global st, stopThread
        stopThread=True
        print 'CNTL-C received'
        exit(0)


# to invoke this, find the pox pid (eg using 'ps axuww|grep pox' at the command line)
# and then use this:
#       kill -SIGUSR1  pid

def usr1_handler(sig, frame):
        print 'SIGUSR1 received'
        printflowdict()

def usr2_handler(sig, frame):
        print 'SIGUSR2 received'
        move_s2()
        pass


def launch (N=3,K=1):
  global st, cv
  signal.signal(signal.SIGINT, intr_handler)
  signal.signal(signal.SIGUSR1, usr1_handler)
  signal.signal(signal.SIGUSR2, usr2_handler)
  n = int(N)
  k = int(K)
  NKsetter(n,k)
  print "N=", N, "K=", K
  st=threading.Thread(target=statsthread, name="statsthread")
  st.start()
  # cv.acquire()
  print 'statsthread started'
  msg = nx.nx_flow_mod()
  #print msg
  #print dir(msg)
  def start ():
        if not core.NX.convert_packet_in:
        log.error("PacketIn conversion required")
        return
        core.openflow.addListenerByName("ConnectionUp", _handle_ConnectionUp)
        log.info("Simple NX switch running.")
  core.call_when_ready(start, ['NX','openflow'])
  # Listener below added by pld: is this in the right place?
  core.openflow_discovery.addListenerByName("LinkEvent", _handle_LinkEvent)
  #tf = threading.Timer(10,floodtest)
  tp = threading.Timer(10, pathtest)
  tp.start()
```

```
# action to be taken on receipt of first TCP packet:
# calculate routes, or print host destinations, etc

def TCPstart():
        print "TCP traffic starting"
        foundreservedport = False
        for s in smap.values():
        #print "switch {}".format(s)
        if s.reservedports != []:
                print "{}: reserved port list is {}".format(s, s.reservedports)
                foundreservedport = True
        if not foundreservedport:
        print "all reserved ports were assigned!"
        return

def NKsetter(n,k):
  global N,K
  (N,K) = (n,k)

def hostswitch(i):    # host is hi
        return i+2*K

def switchpeer(i):
        if i<=K: return i+K
        return i-K

# pld utility about strange dhcp packets
def isdhcp(packet):
        dhcp = packet.find('dhcp')     # pld: doesn't work?
        if dhcp is None: return False
        return True


def hostnum(addr):   # returns, eg, x for 00:00:00:00:00:0x, 0 for other formats
   addr = addr.toStr()
   if addr[:14] == '00:00:00:00:00':
        return int(addr[15:],16)       # pld: this is a 2-byte hex string
   else:
        return 0

def ishost(addr):   # returns true for, eg, 00:00:00:00:00:0x
   addr = addr.toStr()
   if addr[:14] == '00:00:00:00:00':
        return True
   return False

def revlist(lis):
        return lis[::-1]

def gettrunk(path):
        trunklist = range(1,K+1)
        for s in path:
        if isinstance(s, SwitchNode) and s.dpid() in trunklist: return s.dpid()
        return -1
```

VITA

Ihab Al shaikhli earned his Bachelor of Computer Engineering from University of Technology, Baghdad, Iraq, in 2010. He is currently pursuing a Master's degree in Computer Science at Loyola University Chicago.