Extending Open vSwitch to Facilitate Creation of Stateful SDN Applications

Timothy Adam Hoff
Advisor: Tim Nelson
Reader: Shriram Krishnamurthi

1 Introduction

Software-Defined Networks (SDNs) have become increasingly popular as complements to existing networks. SDNs can be very helpful in certain tasks such as debugging because they have a global view of the network. Although SDN is an abstract concept, a concrete controller application protocol called OpenFlow has dominated recent developmental efforts. OpenFlow enables controller applications to specify a set of criteria to match a packet and perform a specific action (e.g., dropping a packet).

Open vSwitch (OVS) is an open-source switching software that supports OpenFlow. The OVS community is very active and routinely proposes and implements new functionality. Although it is possible to implement arbitrarily complex features, and migrate these changes to a switch, switches have relatively limited memory and computational resources. The limited resources, and the fundamental requirement that switches process packets quickly, remove the viability of resource-intensive applications. Furthermore, many developers and network administrators expect switches to be "dumb" devices and adding more complicated features may be perceived as antithetical to the simplicity requirement.

This paper details an addition to Open vSwitch that allows for certain applications to run with reduced controller interaction. Reducing this communication reduces overall network load, and can make applications faster by reducing communication time. After establishing the need for reduced controller interaction, this paper proposes a set of changes to Open vSwitch.

2 Background: Open vSwitch

Although OpenFlow is a relatively recent creation, there has been a lot of work using it as a way to innovate in the network domain. For instance, one common problem in networks is debugging. Traditional tools like ping are primarily concerned with connectivity between two hosts. Existing tools are generally limited in scope, and could be made more powerful by taking advantage of the programmability permitted by OpenFlow.

The OpenFlow protocol follows the match-action paradigm. That is, when a given packet arrives at the switch, the switch tests the packet for a match against the installed rules; if the criteria are satisfied (e.g., MAC source address matches a specific value), then the switch executes the matching rule's action list.

The standard implementation of Open vSwitch contains actions that primarily fall into three categories: packet modification, routing actions, and OpenFlow-specific actions. The first group of actions can modify packet headers, such as changing the IP source field. The second group performs tasks such as dropping a packet, or emitting the packet on a given port. The OpenFlow actions include sending the packet to a different switch table, or sending an OpenFlow message to the controller, which can then instruct the switch on how to handle the packet.

The aforementioned actions have one thing in common: the modifications done to packets only persist while the current packet is being processed, and these changes are isolated from other packets, which may be processed at the same time. This problem, among others, has led developers and companies to extend the base protocol. Of the many additions, Nicira has created one of the most widely used extensions. Until the Nicira extensions were created, no existing actions could create persistent changes to the flow tables. The Nicira extensions included an action called the `learn` action, which could install a new rule based on a combination of packet data and fixed data. The `learn` action appears to be a promising tool because of its ability to create switch rules, and therefore to modify the switch state.

Using these actions, very basic middlebox functionality can already be implemented on OVS switches without modification, including static and stateful firewalls.

The rule to create a static firewall blocking traffic from cs.brown.edu is:

```
Rule for static firewall:
  table=0, nw_src=128.148.32.12, actions=drop
```

To create a stateful firewall, all external packets are dropped except packets from hosts contacted by an internal host. Assuming that switch port 1 is for external traffic and port 2 is for internal traffic, a stateful firewall needs the rules displayed below. Once an out-bound packet reaches the switch, the learn actions execute, creating rules to correctly route each packet in the connection.

```
Rules for stateful firewall:
  table=0, priority=1, in_port=1, actions=drop
  table=0, priority=1, in_port=2, actions=learn(table=0, priority=2, in_port=1,
     NXM_OF_IP_DST[]=NXM_OF_IP_SRC[],NXM_OF_IP_SRC[]=NXM_OF_IP_DST[]
     ,NXM_OF_TCP_DST[]=NXM_OF_TCP_SRC[],NXM_OF_TCP_DST[]=NXM_OF_TCP_SRC[]),learn
     (table=0, priority=2, in_port=2, actions=output:1
```

3 Related Work

SIMON [5] (Scriptable Interactive Monitoring) enables network administrators to run scripts on all the traffic, and potentially find errors in network functionality. This style of monitoring is sometimes called a "hoarder" because all packets must be forwarded to the controller. Hoarders are very resource intensive, especially in data-centers, because they require large amounts of traffic to be sent to the controller. Scaling a system such as SIMON would increase the ability to use it in a production setting.

SIMON is an example of an exciting application that can debug OpenFlow applications; however, the need to see every packet makes SIMON infeasible in most settings. This issue inspires the need for an update or alternative system that can support similar stateful operations, but doesnt have the same bottleneck as SIMON.

FAST [3] (Flow-Level State Transitions) is a system that models the state machines of various programs through switch rules in a specific table. This table contains the state-machine for a given application, such as a stateful firewall. A stated goal of FAST is to reduce the amount of controller interaction in order to increase performance. One issue is that FAST doesn't support state transitions with negation (i.e., transitioning when no matching packet has been seen). Furthermore, FAST only supports a single state machine, which would hinder network monitoring because multiple state machines would have to be combined by taking a cross product of their states and combining their actions. This design decision isn't a fundamental limitation; however, it eliminates the ease of creating two small state machines running in parallel.

P4 [1] (Programming Protocol-Independent Packet Processors) allows users to write specifications defining the method by which packets are processed (including the fields in the packet header, and actions to take on each packet) rather than writing OpenFlow-style rules. This configuration prevents the need for version updates to handle additional packet headers, thereby helping to make a future-proof system. P4's persistent state includes registers, counters, and meters; however, it doesn't allow dynamic reconfiguration of packet processing.

In addition to network monitoring, another very active research area in Software-Defined Networking surrounds middlebox policy routing. This policy enforcement ensures that packets get routed through a specific sequence of devices such as a firewall and then a load balancer. The number of middleboxes grows very large as the size of the network increases; for instance, the number of middleboxes in one enterprise network is 636 with an additional 900 routers [9]. The sheer numbers of appliances in these networks make middlebox policy enforcement very complicated; furthermore, the number of devices makes configuration and maintenance complicated. These issues create the desire for some changes to a networking platform (such as OVS) to alleviate some of the issues present in current networks.

4 Implementation

The state modification permitted by the `learn` action could help alleviate the scalability issues of controller-dominated OpenFlow applications, because certain rules can now be installed without involvement of the controller program. Unfortunately, the rules installed by `learn` actions can only perform a limited set of actions, including emitting the packet on a given port and dropping the packet.

The stateful firewall example demonstrated that very simple stateful applications can be already be written in Open vSwitch. However, without involving the controller, the only existing stateful action is `learn`. `learn` allows for a single state transition by adding a new rule, which doesn't cover many needed cases including a state machine with more than one consecutive transition. The newly implemented actions aim to extend the stateful operations occuring on the switch, thereby reducing the controller interaction for certain applications such as temporal network monitoring. The `learn_learn` action permits consecutive state transitions by containing a `learn_learn` action within another. The additional actions, such as `learn_delete`, enhance the functionality and power of `learn_learn`.

4.1 Learn_Learn

A beneficial modification of the `learn` action would be to enable a `learn` action to install rules with an arbitrary action list. This means a `learn` action can install a rule which would also contain a `learn` action (which is the origin of the name `learn_learn`). By including a `learn_learn` within another one, multiple state transitions are possible without interacting with the controller, because the execution of the outer `learn_learn` installs a rule which contains a `learn_learn` action.

The obvious benefit of this action is that a sequence of stateful changes can be initiated with the installation of a single `learn_learn` action, with separate packets triggering successive state transitions. Because this action doesn't allow reference loops within its action list, each time a `learn_learn` action is executed, the action list in the newly installed rule is a smaller action set than the original `learn_learn`.

```
execute_learn_learn(ll):
  Create a Flow Mod object, fm
  For each spec in ll.specs:
    Populate fm with the spec value
  fm.actions = ll.actions
  execute fm
```

Fig. 1: `learn_learn` execution pseudocode

As mentioned before, one of the benefits of the `learn` action is that it can use both packet values and fixed values when creating the new rule. The values are called flow specs, and are used to create the match criteria for the learned rule. The above pseudocode illustrates that execution is divided into 3 primary steps: populating the flow specs, copying actions into the flow mod object, and then actually executing the flow mod to create a new rule.

Although populating a flow mod message via a `learn` action is straightforward, if a `learn_learn` action contains another `learn_learn`, and the inner action requires information from the packet that triggered the outer action, there is no way to perform such a reference. Furthermore, because the first packet has already been processed, it is no longer available. The straw-man solution to this problem would be to keep every packet processed by the switch and number the packets so that future actions could refer to the packets. This approach has obvious scalability issues, especially due to the limited resources available on switches.

The solution to this problem, named deferral, requires both extra data in each flow spec and computation in `execute_learn_learn()`. For each spec contained within a `learn`, `learn_learn`, or `learn_delete` action, a counter is present which indicates the number of packets until the values need to be populated. Allowing for a maximum depth of 255, each time a new level of the `learn_learn` is executed, the counter for these specs is decremented (pseudocode below). Once the counter is 0 for a given spec (e.g., MAC source address), the spec is populated based on the value from the currently processed packet.

```
do_deferral(action_list ls):
  For each action a in ls:
    if a.type is learn_learn or learn_delete or learn:
      Populate all specs in a which haven't already been populated
    if a.type is learn_learn:
      populate_inner_values(a.actions)

populate_inner_values(action_list ls):
  For each action a in ls:
    if a.type is learn_learn or learn_delete or learn:
      For spec s in a.specs:
        if s.deferral_count == 0:
          populate_spec(s)
        else:
          s.deferral_count--
    if a.type is learn_learn:
      do_deferral(a.actions)
```

Fig. 2: Deferral Pseudocode

Adding deferral and the ability to create a new rule with a more complex set of actions greatly expands the number of applications that can be created without involving the controller application because multiple state transitions can be included within a single `learn_learn`.

```
execute_learn_learn(ll):
  Create a Flow Mod object, fm
  For each spec in ll.specs:
    Populate fm with the spec value
  do_deferral(ll.actions)
  fm.actions = ll.actions
  execute fm
```

Fig. 3: Updated `learn_learn` execution pseudocode

4.2 Learn_Delete

In the standard implementation of Open vSwitch, there are three ways to delete rules: rules can time out, an OpenFlow-enabled application can delete rules, or an administrator can explicitly delete rules manually using the ovs-ofctl utility. However, the ability to delete rules through the execution of an action allows for some reduction in the number of rules in the switch's tables, allows additional state transitions within a state machine (described in section 5), and removes transitions which are no longer needed.

The `learn_delete` action functions similar to the `learn` action, except that it deletes rules whereas the `learn` action modifies or creates a new rule. `learn_delete` uses a combination of packet data and fixed data specified by the rule creator. Importantly, the `learn_delete` action specifies a single table to delete the rules from. If no priority is specified, then all matching rules will be deleted, although if a priority is specified, then all rules matching that priority will be deleted.

```
execute_learn_delete(ld):
  Create a Flow Mod object, fm
  if ld.priority = OVS_default_priority
    fm.command = DELETE
  else:
    fm.command = DELETE_STRICT
  for each spec in ld.specs:
    Populate fm with the spec values
  execute fm
```

Fig. 4: `learn_delete` execution pseudocode

As seen in the deferral pseudocode, `learn_delete` also uses deferral of values in order to use a previously seen packet's data. This allows a single `learn_learn` to install a pair of rules, one rule containing a `learn_learn` to transition forward in a given state machine, and a `learn_delete` to remove the transition forward, and transition backwards. Because both of the newly installed rules allowed deferral, they can use the same packet information for their match criteria.

4.3 Timeout_Act

Any query which involves negation must be of the form "within *n* seconds"; otherwise, the negation is true at every moment except the arrival of the packet, thus only range-satisfiability is useful. If there is a transition after the negative query is satisfied, an action which executes upon rule timeout is needed. The default implementation of Open vSwitch allows for rules to be deleted after a timeout occurs and the controller is notified of the timeout, but no other actions are executed. The `timeout_act` action addresses this concern by simply containing a list of actions, which are executed upon timeout.

An important distinction in the execution of this action is that there's no packet to access during its execution. This means that many actions shouldn't be used (e.g., modifying IP source address), and the actions that are used cannot rely on current packet data.

```
execute_timeout_act(t):
  for action a in t.actions
    a.execute()
```

Fig. 5: `timeout_act` execution pseudocode (upon rule timeout)

4.4 Increment_Table_Id

The following query motivates the need for an additional action:

```
see packet A
see packet B within 5 seconds of seeing packet A
```

The above query is very simple; however, if two packet As arrive and then a packet B arrives, then it would be preferable to inform the controller that the arrival of packet B has matched both of the packet As received. A single rule could not do this, unless it contained multiple `controller` actions.

The final action added as part of this work, `increment_table_id`, involves a shared switch-wide variable, namely a shared-atomic counter. Open vSwitch didn't have any actions wherein a global variable would be updated upon execution (with the exception of rule counters and learning rules).

By modifying a group of switch tables, such that a packet is automatically resubmitted through the tables, the above query can be turned into switch rules, and notify the controller upon the completion of each set of satisfying packets. The `atomic_table_id`, which is incremented when this action executes, can be used by `learn` actions in order to recognize multiple events of the same form (i.e., See packet A from above).

Using the tables 0 through 149 for the auto-resubmit section, the arrival of the first packet A results in a rule learned into table 0. Similarly the second packet A results in a rule learned into table 1. When packet B arrives, it executes the rules in both tables, thereby notifying the controller that the query was satisfied twice. The query demonstrates the need for the final update to the `learn_learn` pseudocode, which allows the rule to use either the atomic counter or a fixed number as the table_id.

```
execute_increment_table_id(i):
  increment_global_atomic_counter()
```

Fig. 6: `increment_table_id` execution pseudocode

```
execute_learn_learn(ll):
  Create a Flow Mod object, fm
  if ll.use_atomic_counter:
    fm.table_id = get_atomic_counter_value()
  else:
    fm.table_id = ll.table_id
  For each spec in ll.specs:
    Populate fm with the spec value
  fm.actions = ll.actions
  execute fm
```

Fig. 7: Updated `learn_learn` execution pseudocode including atomic_counter

5 Applications

The actions described above allow for a wide variety of applications to be implemented. Although the application motivating this work was monitoring, other applications can be implemented using these primitives, such as port knocking, without any interaction of the controller application. This section reviews these potential applications, and provides an analysis of potential applications of this system performing the same functionality as middlebox.

5.1 Monitoring

The match-action paradigm can be very useful but there's a strong underlying limit to OpenFlow: it can specify a set of actions that a given packet will follow, but cannot match sequences of packets without controller involvement, due to its lack of stateful operations. Furthermore, OpenFlow is primarily concerned with connectivity and functionality of a network, rather than enabling a broad set of tools capable of network testing and debugging.

Nelson et al. [4] used the changes outlined in the previous section to help tackle the network-monitoring problem. The authors implemented a compiler that converted queries into switch rules, which exist on switches just as any other traffic routing rule. These queries could specify a sequence of packets subject to some matching criteria, and the rules would inform the controller upon the completion of a query.

```
Query 1:
  see packet a          | a.dl_src = 1:1:1:1:1:1
  see packet b          | b.dl_src = 2:2:2:2:2:2
  see packet c          | c.dl_src = 2:2:2:2:2:2
```

In the above query, the compiler produces rules using `learn_learn`. The initial rule simply matches on dl_src = 1:1:1:1:1:1, and once the packet arrives it executes a `learn_learn` in order to enable the next transitions to occur. Once the third event completes, the controller is notified that the sequence has been completed.

As mentioned above, the atomic counter can be used instead of the table number to account for receiving multiple As followed by a single B and then a single C.

```
Rule for query 1:
  table_id=150, dl_src=1:1:1:1:1:1, actions=increment_table_id(), learn_learn(
     table_spec=LEARN_USING_ INGRESS_ATOMIC_TABLE,  dl_src=2:2:2:2:2:2, actions=
     {increment_table_id(),learn_learn(table_spec=LEARN_USING_
     INGRESS_ATOMIC_TABLE, dl_src=3:3:3:3:3:3, actions={controller})})
```

```
Query 2:
  see packet a                              | a.dl_src = 1:1:1:1:1:1
  not see packet b within 5 seconds         | b.dl_src = 2:2:2:2:2:2
  not see packet c within 5 seconds         | c.dl_src = 2:2:2:2:2:2
```

The above query differs from the first query due to its use of negation, and therefore timers. Query 2 requires a `learn_learn` to allow for the later transitions, the second rule has two components, a `learn_delete` to delete the rule if packet B arrives, and then a `learn_learn` inside of a `timeout_act` that creates the final rule once the timeout occurs. The rule simply contains a controller action as part of a `timeout_act`.

```
Rule for query 2:
  table=150,  dl_src=1:1:1:1:1:1 actions=increment_table_id(),learn_learn(
     table_spec=LEARN_USING_ INGRESS_ATOMIC_TABLE, hard_timeout=5, dl_src=2
     :2:2:2:2:2, actions=learn_delete(table_spec=DELETE_USING_RULE_TABLE,
     dl_src=2:2:2:2:2:2),timeout_act(increment_table_id(),learn_learn(
     table_spec=LEARN_USING_ INGRESS_ATOMIC_TABLE hard_timeout=5, dl_src=3
     :3:3:3:3:3, actions=learn_delete(table =USE_RULE_TABLE_ID, dl_src=3
     :3:3:3:3:3),timeout_act(controller)))))
```

5.2 Port Knocking

Port Knocking is a security technique in which a connection (e.g., SSH connection) can be established only after a sequence of packets is sent with a specific sequence of TCP port numbers. Port knocking typically uses iptables [2]; however, this is a node-specific solution and it is subject to configuration errors. Fortunately, the above primitives now allow rules to be installed that allow for the same protection without requiring individual configuration, and the application doesn't involve the controller.

The below figure depicts the state machine representing the establishment of a connection. The correct sequence of ports in this example is 5000, 6000, and then 7000, and any incorrect packet brings the state back to the start.

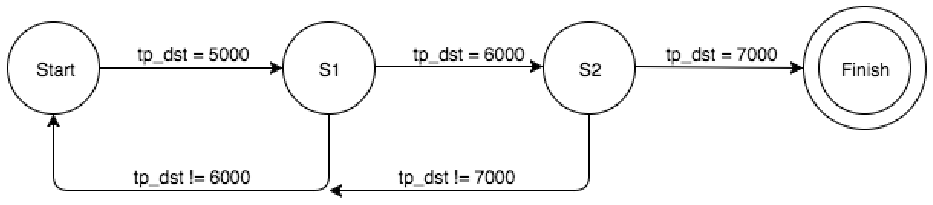


Fig. 8: Port Knocking State Machine

Each packet arrival triggers the execution of a rule's action list; this transitions the state within the port knocking state machine. Specifically, each transition forward (except the final one) executes a `learn_learn` in order to allow for the next forward transition, and another `learn_learn` is executed to allow for the backward transitions seen at the bottom of the diagram. Using `learn_learn` to transition forward and `learn_delete` to transition backwards and remove previous transitions, ensures that only the transitions explicitly depicted in the figure are possible.

Because this application doesn't involve the controller, the initial rule contains all of the future transitions are located within the initial rule (seen below).

```
Port Knocking Rule:
  table=150, tcp, priority=0, tp_dst=5000, actions=learn_learn(
     table=150,dl_type=0x0800,nw_proto=6,tp_dst=6000,priority=2,NXM_OF_IP_SRC[]
     =NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]=NXM_OF_IP_DST[](defer=0),
     actions={learn_learn(table=150, dl_type=0x800, priority=4, nw_proto=6,
     tp_dst=7000, NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]
     =NXM_OF_IP_DST[](defer=0), actions={learn_learn(table=150, priority=8,
     dl_type=0x0800, NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]
     =NXM_OF_IP_DST[](defer=0), actions={NORMAL}), learn_delete(table=150,
     priority=3, dl_type=0x0800, NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0),
     NXM_OF_IP_DST[]=NXM_OF_IP_DST[](defer=0)), learn_delete(table=150,
     priority=4, dl_type=0x800, nw_proto=6, tp_dst=7000, NXM_OF_IP_SRC[]
     =NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]=NXM_OF_IP_DST[](defer=0))}),
     learn_delete(table=150, dl_type=0x800, priority=2, nw_proto=6, tp_dst=6000,
      NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]=NXM_OF_IP_DST[](
     defer=0)), learn_delete(table=150, priority=1, dl_type=0x0800,
     NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]=NXM_OF_IP_DST[](
     defer=0)), learn_learn(table=150, dl_type=0x0800, priority=3, NXM_OF_IP_SRC
     []=NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]=NXM_OF_IP_DST[](defer=0),
     actions={learn_delete(table=150, priority=4, dl_type=0x0800, nw_proto=6,
     tp_dst=7000, NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]
     =NXM_OF_IP_DST[](defer=0)), learn_delete(table=150, priority=3,
     dl_type=0x0800, NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]
     =NXM_OF_IP_DST[](defer=0))})}),learn_learn(
     table=150,dl_type=0x0800,priority=1,NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0
     ), NXM_OF_IP_DST[]=NXM_OF_IP_DST[](defer=0),actions={learn_delete(
     table=150, priority=2, dl_type=0x0800, nw_proto=6, tp_dst=6000,
     NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]=NXM_OF_IP_DST[](
     defer=0)), learn_delete(table=150, priority=1, dl_type=0x0800,
     NXM_OF_IP_SRC[]=NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[]=NXM_OF_IP_DST[](
     defer=0))})"
```

The appendix contains the state of the flow tables at each state in the state machine.

5.3 Middlebox Applications

Recently, there has been a large volume of SDN research centered on middleboxes. Much of this work surrounds middlebox policy enforcement, and systems such as SIMPLE [8] have been created to streamline the task of ensuring that packets are routed through the correct sequence of middleboxes. Although this thesis doesn't discuss middlebox policy enforcement further, the combination of packet tagging and the monitoring described above could be used to alert network admins if packets aren't being routed through the sequence of middleboxes that they expected.

A common complaint of middleboxes is that they remain closed (i.e., black boxes), meaning that the exact processing of packets by a single middlebox can be difficult or impossible to determine. This fact makes network management in the presence of many middleboxes even more complicated. However, switches are increasingly able to perform similar functionality as middleboxes with the substantial benefit of switches being more open (open in the sense of configurable and easily accessible), especially if they are using Open vSwitch.

Adding arbitrarily many additions to OVS to support every possible middlebox isn't viable on switches due to the space and performance constraints. Perhaps future work could focus on adding optional extensions to OVS that would add additional actions to enable various middlebox capabilities without increasing the binary size for all users.

The latest versions of Open vSwitch have been exploring the benefit of storing additional state to enable more complex applications. OVS version 2.5 includes an action called conntrack [7]. This action is meant to help identify which packets are related to previously established connections, initializing new connections, etc. Note that the conntrack action actually uses Linux kernel connection tracking functionality. Furthermore, this connection tracking has allowed for easy implementation of the stateful applications, such as: Stateful Firewall and NAT.

6 Performance Analysis

The focus of this work is on the feasibility of modifying OVS to allow for the creation of additional stateful applications that don't rely on interaction with the controller application. However, a note on system performance may help inspire future work. An evaluation of this system by Nelson et al. [4] explains:

> For a single connection, a `learn` action has an average latency of 1-2 ms; for 128 simultaneous connections, the latency increases to an average 39 ms. For nested `learn` actions [i.e., a `learn_learn` action], the latencies are similar. This indicates that the addition of nested `learn`s has little additional impact on latency. Adding rules to a new table requires incrementing an atomic counter [i.e., executing an `increment_table_id` action], which incurs additional latency, taking an average of 124 ms for 128 connections. Both our extensions and the base Open vSwitch `learn` action incur latencies slower than line-rate.

Importantly, this delay impacts the type of applications that could run as switch rules. If the application requires line-rate processing speed, then a proprietary middlebox may be the only solution. However applications that don't have strict performance requirements (e.g., applications dominated by controller interaction) would benefit from this system, as it would reduce the number of messages to and from the controller. Furthermore, depending on communication time and performance of the controller, running the application on the switch would likely be faster as well.

7 Conclusion and Future Work

7.1 Future Work

As previously mentioned, the system described in this paper has areas for future improvements. In order for Open vSwitch to provide the same functionality as any middlebox with configurable packet processesing, an extensive effort would be required on the part of the OVS community to create a framework for such additons. This framework would help future efforts to expand the processing done on switches and prevent deterioration of the quality of the code base. Thus this change might not be feasible, unless more people support the currently contested notions of having a switch perform many more of the middlebox functions.

An additional, and reasonable, change is to modify the atomic counter to represent another field instead of table values. The number of tables available in OVS is 255, meaning that only short-lived queries are currently supported, which doesn't scale to datacenter-sized networks or queries that involve many packets. In order to prevent this, a table could be reserved and its operation could change, similar to FAST's State Table. Specifically, cookie or metadata values could be used to define groups. Each group would then be separately evaluated. Both cookie and metadata values in Open vSwitch are 64-bits; thus the switch would run out of memory for rules before all of the cookie-values had been used.

7.2 Conclusions

Open vSwitch is a very promising platform that emphasizes the importance and promise of Software-Defined Networking. OVS's match-action paradigm provides an easily understood framework for future additions. The creation of the `learn` action provided a glimpse into OVS's potential for stateful applications. By making the modifications described in this paper, the promise of stateful applications running on switches is further explored. Importantly, these applications are created by rules that still obey the match-action paradigm that Open vSwitch emphasizes.

8 Acknowledgements

I would like to thank my parents, Drs. Rainer and Martha Hoff, as this work wouldn't have been possible without their continued motivation and support.

This thesis would not have been possible with the collaboration of Professor Shriram Krishnamurthi, Professor Rodrigo Fonseca, Dr. Tim Nelson, and Nicholas DeMarinis. Working in collaboration with this group provided me guidance, and enabled me to learn a great deal.

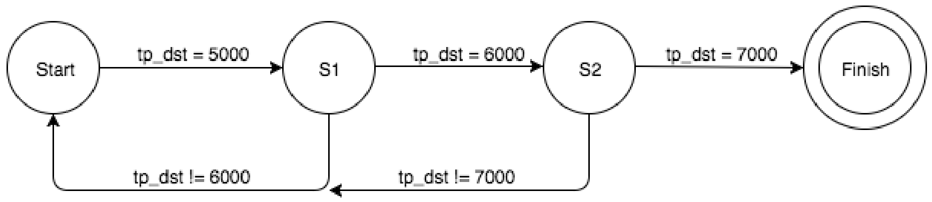
I would also like to thank Jason Teh for the many occasions where we were working late into the night, and continually encouraging me throughout this process.

References

1. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D.: P4: Programming protocol-independent packet processors. ACM Computer Communication Review (2014)
2. DigitalOcean: How To Use Port Knocking to Hide your SSH Daemon from Attackers on Ubuntu. `https://www.digitalocean.com/community/tutorials/how-to-use-port-knocking-to-hide-your-ssh-daemon-from-attackers-on-ubuntu` (2014), accessed April 9th, 2016
3. Moshref, M., Bhargava, A., Gupta, A., Yu, M., Govindan, R.: Flow-level state transition as a new switch primitive for SDN. In: Workshop on Hot Topics in Software Defined Networking (2014)
4. Nelson, T., DeMarinis, N., Hoff, T., Fonseca, R., Krishnamurthi, S.: Compiling Stateful Network Properties for Runtime Monitoring (2016)
5. Nelson, T., Yu, D., Li, Y., Fonseca, R., Krishnamurthi, S.: Simon: Scriptable interactive monitoring for sdns. In: Symposium on SDN Research (SOSR) (2015)
6. Open vSwitch `http://openvswitch.org/`
7. Open vSwitch: Stateful Connection Tracking & Stateful NAT. `http://openvswitch.org/support/ovscon2014/17/1030-conntrack_nat.pdf`, accessed April 9th, 2016
8. Qazi, Z., Tu, C., Chiang, L., Miao, R., Sekar, V., Yu, M.: SIMPLE-fying Middlebox Policy Enforcement Using SDN. In: Conference on Communications Architectures, Protocols and Applications (SIGCOMM) (2013)
9. Sekar, V., Egi, N., Ratnasamy, S., Reiter, M., Shi, G.: Design and Implementation of a Consolidated Middlebox Architecture. In: Symposium on Networked Systems Design and Implementation (NSDI) (2012)

Appendix

Port Knocking Rules



The rules for each state in the above state machine are enumerated below. The external host has MAC address: 11:22:33:44:55:66 and IP address: 1.2.3.4, whereas the internal host has MAC address 11:22:33:33:33:33 and IP address: 4.5.6.7.

```
Starting State:
  table=150, priority=0, tcp, tp_dst=5000 actions=learn_learn(table=150,
   priority=2, eth_type=0x800, nw_proto=6, tcp_dst=6000, NXM_OF_IP_SRC[](
   defer=0), NXM_OF_IP_DST[](defer=0), actions=learn_learn(table=150,
   priority=4, eth_type=0x800, nw_proto=6, tcp_dst=7000, NXM_OF_IP_SRC[](
   defer=0), NXM_OF_IP_DST[](defer=0), actions=learn_learn(table=150,
   priority=8, eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](
   defer=0), actions=NORMAL,), learn_delete(table=150, priority=3,
   eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),
   learn_delete(table=150, priority=4, eth_type=0x800, nw_proto=6,
   tcp_dst=7000, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),),
   learn_delete(table=150, priority=2, eth_type=0x800, nw_proto=6,
   tcp_dst=6000, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),
   learn_delete(table=150, priority=1, eth_type=0x800, NXM_OF_IP_SRC[](defer=0
   ), NXM_OF_IP_DST[](defer=0)), learn_learn(table=150, priority=3,
   eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0),
   actions=learn_delete(table=150, priority=4, eth_type=0x800, NXM_OF_IP_SRC
   [](defer=0), NXM_OF_IP_DST[](defer=0)),),), learn_learn(table=150,
   priority=1, eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](
   defer=0), actions=learn_delete(table=150, priority=2, eth_type=0x800,
   NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),)
```

```
S1:
  table=150, priority=1, ip,nw_src=1.2.3.4, nw_dst=4.5.6.7
   actions=learn_delete(table=150, priority=2, eth_type=0x800, nw_proto=6,
   tcp_dst=6000, ip_src=1.2.3.4, ip_dst=4.5.6.7), learn_delete(table=150,
   priority=1, eth_type=0x800, ip_src=1.2.3.4, ip_dst=4.5.6.7)

  table=150, priority=2, tcp, nw_src=1.2.3.4, nw_dst=4.5.6.7, tp_dst=6000
   actions=learn_learn(table=150, priority=4, eth_type=0x800, nw_proto=6,
   tcp_dst=7000, ip_src=1.2.3.4, ip_dst=4.5.6.7, actions=learn_learn(
   table=150, priority=8, eth_type=0x800, ip_src=1.2.3.4, ip_dst=4.5.6.7,
   actions=NORMAL,), learn_delete(table=150, priority=3, eth_type=0x800,
   ip_src=1.2.3.4, ip_dst=4.5.6.7), learn_delete(table=150, priority=4,
   eth_type=0x800, nw_proto=6, tcp_dst=7000, ip_src=1.2.3.4, ip_dst=4.5.6.7),)
   , learn_delete(table=150, priority=2, eth_type=0x800, nw_proto=6,
   tcp_dst=6000, ip_src=1.2.3.4, ip_dst=4.5.6.7), learn_delete(table=150,
   priority=1, eth_type=0x800, ip_src=1.2.3.4, ip_dst=4.5.6.7), learn_learn(
   table=150, priority=3, eth_type=0x800, ip_src=1.2.3.4, ip_dst=4.5.6.7,
   actions=learn_delete(table=150, priority=4, eth_type=0x800, nw_proto=6,
   tcp_dst=7000, ip_src=1.2.3.4, ip_dst=4.5.6.7), learn_delete(table=150,
   priority=3, eth_type=0x800, ip_src=1.2.3.4, ip_dst=4.5.6.7),)

  table=150, priority=0, tcp, tp_dst=5000 actions=learn_learn(table=150,
   priority=2, eth_type=0x800, nw_proto=6, tcp_dst=6000, NXM_OF_IP_SRC[](
   defer=0), NXM_OF_IP_DST[](defer=0), actions=learn_learn(table=150,
   priority=4, eth_type=0x800, nw_proto=6, tcp_dst=7000, NXM_OF_IP_SRC[](
   defer=0), NXM_OF_IP_DST[](defer=0), actions=learn_learn(table=150,
   priority=8, eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](
   defer=0), actions=NORMAL,), learn_delete(table=150, priority=3,
   eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),
   learn_delete(table=150, priority=4, eth_type=0x800, nw_proto=6,
   tcp_dst=7000, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),),
   learn_delete(table=150, priority=2, eth_type=0x800, nw_proto=6,
   tcp_dst=6000, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),
   learn_delete(table=150, priority=1, eth_type=0x800, NXM_OF_IP_SRC[](defer=0
   ), NXM_OF_IP_DST[](defer=0)), learn_learn(table=150, priority=3,
   eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0),
   actions=learn_delete(table=150, priority=4, eth_type=0x800, NXM_OF_IP_SRC
   [](defer=0), NXM_OF_IP_DST[](defer=0)),),), learn_learn(table=150,
   priority=1, eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](
   defer=0), actions=learn_delete(table=150, priority=2, eth_type=0x800,
   NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),)
```

```
 S2:
   table=150, priority=3,ip,nw_src=1.2.3.4,nw_dst=4.5.6.7 actions=learn_delete(
    table=150, priority=4, eth_type=0x800, nw_proto=6, tcp_dst=7000,
    ip_src=1.2.3.4, ip_dst=4.5.6.7),learn_delete(table=150, priority=3,
    eth_type=0x800, ip_src=1.2.3.4, ip_dst=4.5.6.7)

   table=150, priority=4, tcp, nw_src=1.2.3.4, nw_dst=4.5.6.7, tp_dst=7000
    actions=learn_learn(table=150, priority=8, eth_type=0x800, ip_src=1.2.3.4,
    ip_dst=4.5.6.7, actions=NORMAL,), learn_delete(table=150, priority=3,
    eth_type=0x800, ip_src=1.2.3.4, ip_dst=4.5.6.7), learn_delete(table=150,
    priority=4, eth_type=0x800, nw_proto=6, tcp_dst=7000, ip_src=1.2.3.4,
    ip_dst=4.5.6.7)

   table=150, priority=0, tcp, tp_dst=5000 actions=learn_learn(table=150,
    priority=2, eth_type=0x800, nw_proto=6, tcp_dst=6000, NXM_OF_IP_SRC[](
    defer=0), NXM_OF_IP_DST[](defer=0), actions=learn_learn(table=150,
    priority=4, eth_type=0x800, nw_proto=6, tcp_dst=7000, NXM_OF_IP_SRC[](
    defer=0), NXM_OF_IP_DST[](defer=0), actions=learn_learn(table=150,
    priority=8, eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](
    defer=0), actions=NORMAL,), learn_delete(table=150, priority=3,
    eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),
    learn_delete(table=150, priority=4, eth_type=0x800, nw_proto=6,
    tcp_dst=7000, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),),
    learn_delete(table=150, priority=2, eth_type=0x800, nw_proto=6,
    tcp_dst=6000, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),
    learn_delete(table=150, priority=1, eth_type=0x800, NXM_OF_IP_SRC[](defer=0
    ), NXM_OF_IP_DST[](defer=0)), learn_learn(table=150, priority=3,
    eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0),
    actions=learn_delete(table=150, priority=4, eth_type=0x800, NXM_OF_IP_SRC
    [](defer=0), NXM_OF_IP_DST[](defer=0)),),), learn_learn(table=150,
    priority=1, eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](
    defer=0), actions=learn_delete(table=150, priority=2, eth_type=0x800,
    NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),)
```

```
 Accepting State:
   table=150, priority=8,ip,nw_src=1.2.3.4,nw_dst=4.5.6.7 actions=NORMAL

   table=150, priority=0, tcp, tp_dst=5000 actions=learn_learn(table=150,
    priority=2, eth_type=0x800, nw_proto=6, tcp_dst=6000, NXM_OF_IP_SRC[](
    defer=0), NXM_OF_IP_DST[](defer=0), actions=learn_learn(table=150,
    priority=4, eth_type=0x800, nw_proto=6, tcp_dst=7000, NXM_OF_IP_SRC[](
    defer=0), NXM_OF_IP_DST[](defer=0), actions=learn_learn(table=150,
    priority=8, eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](
    defer=0), actions=NORMAL,), learn_delete(table=150, priority=3,
    eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),
    learn_delete(table=150, priority=4, eth_type=0x800, nw_proto=6,
    tcp_dst=7000, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),),
    learn_delete(table=150, priority=2, eth_type=0x800, nw_proto=6,
    tcp_dst=6000, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),
learn_delete(table=150, priority=1, eth_type=0x800, NXM_OF_IP_SRC[](defer=0
), NXM_OF_IP_DST[](defer=0)), learn_learn(table=150, priority=3,
eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0),
actions=learn_delete(table=150, priority=4, eth_type=0x800, NXM_OF_IP_SRC
[](defer=0), NXM_OF_IP_DST[](defer=0)),),), learn_learn(table=150,
priority=1, eth_type=0x800, NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](
defer=0), actions=learn_delete(table=150, priority=2, eth_type=0x800,
NXM_OF_IP_SRC[](defer=0), NXM_OF_IP_DST[](defer=0)),)
```