OPTIMIZING THE POX CONTROLLER WITH DATABASE SYSTEMS

BY

FAN YANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisor:

Associate Professor Philip Brighten Godfrey

Abstract

Software-defined networking (SDN) brings new opportunities and challenges to the current design of networks and how networks can be managed. In comparison to the traditional network architecture, SDN simplifies the control of the network by using a logically centralized controller and a set of OpenFlow switches. Network administrators can program OpenFlow switches to behave like routers, firewalls, load balancers, etc. by building different kinds of applications on top of the controller. On one hand, the controller maintains the connection with each OpenFlow switch and pushes down any instructions specified by the control application to the network. On the other hand, the controller provides a centralized view of the network state to control applications, and control applications can make intelligent decisions based on the overall state of the network. POX [4] is one of the early popular SDN controllers. In this project, we studied the key design decisions made in POX and optimized the current implementation by building a new database module. Topology dependent applications are now supported by both in-memory and persistent storage. Since the database component updates the changes of the network in the database in real time, it also servers as the key step to connect the Ravel project to the actual software OpenFlow switches. Ravel [14] is a database based controller platform. After starting Ravel, all the network elements on Mininet need to be mapped into the PostgreSQL database and constantly being monitored. This can be easily achieved using POX with the database component enabled. Once the connection between the network and the database is established, a SDN can be managed within the database by Ravel. We tested the database functionalities and evaluated its performance on Mininet. From the test result, we conclude that the overhead of retrieving link and switch information from the database is reasonably small.

To Father and Mother.

Acknowledgments

This project would not have been possible without the support of many people. I would like to gratefully and sincerely thank my advisor, Dr. Brighten Godfrey, for his guidance, understanding, and continuous support during my study at UIUC. I would also like to thank Anduo Wang, Chi-Yao Hong, and Mengjia Yan for sharing their expertise, great insights, and many valuable advice on this project. Finally, I would like to thank my parents for their love, support, and unwavering belief in me.

Table of Contents

Chapter 1

Introduction

Computer Networks are complex and difficult to manage. They are composed by all different types of equipments each running different sets of distributed protocols. Network administrators often face the problem to configure and maintain each individual network device with its own standard, and adjust them to work together through different interfaces [8]. Software-defined networking (SDN) aims to abstract away the low-level details that are specific to each different network device and provides an uniformed control over different network components [6]. SDN has two distinct characteristics. First, SDN explicitly separates the control plane (which decides how to handle the traffic) from the data plane (which forwards the packets with respect to the decisions made by the control plane). Second, SDN consolidates the control plane by having a centralized controller over multiple data plane elements. The controller communicates with network devices via a well-defined API called OpenFlow [13]. And by installing rules sent by control applications, the controller is capable to instruct the OpenFlow switch to behave as a router, a switch, or a firewall as needed. SDN inspires more modular and extensible design of the network architecture and provides opportunities to invent new control applications.

The controller serves as the brain of SDN, and many efforts (e.g POX [4], Floodlight [1], Beacon [7], and Ravel [14]) have been made in designing a good SDN controller. In this project, we studied the architecture of POX and extended the current design by adding a new database component to support topology dependent applications and to facilitate Ravel's network monitoring. In the current implementation, each POX module needs to maintains its own copy of the network state via data structures such as lists and maps in memory. The network state is replicated and less efficiently maintained at multiple places. Developers are forced to start by searching for every corresponding event in order to handle the topology update coming from different network elements, rather than directly focusing on application logic. Moreover, if the network state can be made available within the database, users can choose to use Ravel's vertical and horizontal orchestration services to manage SDN with standard SQL queries. To improve the existing POX framework and to support Ravel features, we proposed to include a database component in the POX controller. The database component is developed as a separate module and can be easily plugged into the POX controller. With the database functionality enabled, POX will be more resilient to controller failure and richer database based management schemes can be applied to manage SDN.

Chapter 2

POX Design Overview

A large part of POX's development is based on NOX [9], which proposed the idea of an operating system for networks. NOX used the analogy to argue that managing networks using a centralized controller is like providing an operating system for running different programs and applications. In the past, programs were written in assembly languages provided with no abstraction to memory, storage, or communication. Programmers had to handle every low level detail in order to use system resources. Like in today's network, devices are vendor specific and each device requires separate configuration to plug them into the network. What NOX offers for the network is similar to what an operating system offers for a personal computer. NOX defines a network API for developers to program those low-level network devices using high-level policies.

POX is a python wrapper for the original NOX controller. It implements a SDN controller and provides a well-defined API for programmers to develop control applications on top of the controller. This chapter explores the key design decisions made in POX and the basic concepts of the OpenFlow protocol.

2.1 Architecture

POX provides an extremely flexible and extensible framework, which allows new user defined applications to be easily built on top of it and be compatible with other existing modules. Figure 2.1 illustrates the basic design of the POX controller. POX does not manage the network itself, instead it provides the connection with all the OpenFlow switches and serves as a network OS. Developers have the flexibility to define their own control applications such as a traffic monitor or a load balancer on top of POX.

2.2 POX core object and event handling

To support asynchronous communication among different components, POX has a special group of classes that are commonly known as the core object. Together with the feature of raising events and setting up event handlers, POX provides a convenient way for different components to interact with each other in POX.

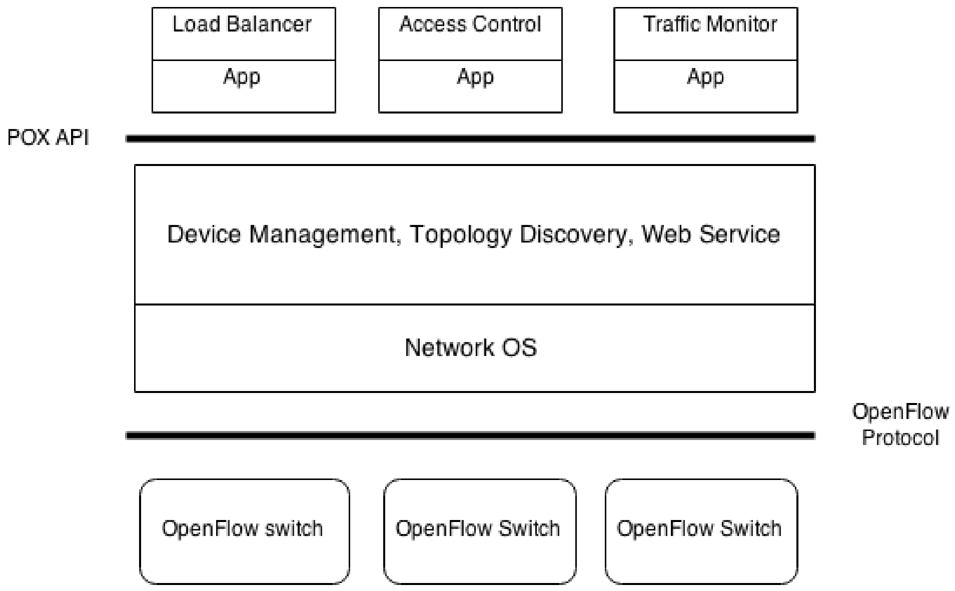


Figure 2.1: The POX controller architecture

This design is also known as publisher and subscriber scheme.

Instead of using import statements to have one component import another so that they can interact, components will ”register” themselves on the core object, and other components can query the core object if an interesting event happens. This allows any old or new module to be easily added or removed without affecting all other components. For instance, a control application can include the following to send and receive notifications from other modules:

```
# Listen to dependencies
def startup ():
      core.openflow.addListeners(self, priority=0)
      core.openflow_discovery.addListeners(self)
      core.host_tracker.addListeners(self)
core.call_when_ready(startup, (’openflow’,’openflow_discovery’))


# Attach to the core
def launch ():
      core.registerNew(db)
```

If any of the listed dependencies cannot not be resolved during POX's startup phase, an error message will be displayed and POX will be waiting for the missing component.

2.3 Communicating with OpenFlow switches

One of the primary purposes of POX is to provide a platform that can support network developers to easily build OpenFlow control applications. In order to facilitate programmers to focus on the application logic rather than low level details of message passing, POX designed the `openflow` component to provide the abstraction for control applications. Applications can then set up event handlers or send messages directly through POX without knowing the details about how to talk to any of the switches.

2.3.1 Datapaths

OpenFlow uses DPID, which stands for data path identifiers, to identity each unique connection with an OpenFlow switch. Internally, Mininet assigns a DPID to each switch and is communicated to the controller during handshaking by `ofp_switch_features`. We converted DPIDs to integer identifiers to store information about OpenFlow switch in the Postgres database.

Each datapath between the controller and the switch is a bi-direction channel represented by a `connection` object . When messages coming from the switch and received at the controller, they show up as events. The application sitting on top of the controller should set up event handler to specify what to do with this type of event. When application wants to send instructions to the switches, they typically construct a `of.ofp_flow_mod` message and invoke the `connection.send` method to send control message to the network.

2.3.2 OpenFlow events

An OpenFlow event will be raised in response to any updates on the network, or more specifically, in response to OpenFlow switches. It contains many different attributes that developers can use to retrieve the root cause or sender information about this event. For instance, if we want to map the host status into the database, we can set up event handlers like the following to monitor hosts on the network in real time:

```
def _handle_HostEvent (self, event):
    dpid = dpid_to_str(event.entry.dpid)
    port = str(event.entry.port)
    macaddr = str(event.entry.macaddr)
```

```
if event.join:
     # add host in db
if event.leave:
    # remove host in db
if event.move:
    # update host in db
```

2.3.3 Sending control messages

OpenFlow messages allow control applications to program how the switches should react with each incoming packets. For instance, in POX's routing module, after the path computation is finished, the routing module will construct a `ofp_flow_mod` message, fill in the payload of the message, and call `connection.send(msg)` to install the forwarding rules at each switch. In the payload of the message, users can specify if a flow should be added, deleted, or modified in the forwarding information base, sets how long this rule is valid, define how packets are matched against a group, etc.

Chapter 3

Customizing Topology in Mininet

We used Mininet to test the performance of the database module on simulated networks. To better understand and interpret the experimental results, we studied the advantages and limitations of this tool, and summarized our key findings in this chapter.

3.1 Design principles

Mininet [2] is a lightweight SDN prototyping environment. In contrast to testbed or virtual machines, Mininet only requires local resources, i.e. a laptop, to simulate a SDN environment. As discussed in [8], this tool has been performed surprisingly well since the early days for researchers and developers to invent and test new network protocols. The underlying principle utilizes the two linux features: network namespaces and virtual Ethernet pairs [12]. Mininet creates a virtual network by placing host processes in network namespaces so that each host would have its own IP, ports and interface. Mininet connects the simulated controller, switch, and hosts by virtual Ethernet paris. If a packet is sent from a host $h1$ to the other host $h2$, in the Mininent simulation, this is equivalent to an ICMP echo request was sent out from $h1$'s private $eth0$ network interface and enters the kernel through a virtual Ethernet pair. The request is then processed by some switch in the root namespace and exits to $h2$ through a different virtual Ethernet pair [11]. Mininet is packaged into a virtual machine and supports both a command line interface (CLI) and an application programming interface (API).

3.2 Pros and cons

The main advantage for Mininet is that it supports rapid prototyping SDN applications on a single laptop. Mininet is distributed as a virtual machine which comes with all dependencies already installed. Mininet also provides great flexibility in creating and designing the network environment. Specifically, Mininet includes many commonly used network topologies, supports external controllers, and allows users to adjust link bandwidth and other network parameters.

However, since Mininet only provides the lightweight virtualization that runs on a single host, it can support up to a few hundred of nodes. The size of the network is limited so performance evaluation on large networks will be affected. The salability issue is discussed in detail in both [12] and [5], where the author pointed out that the setup time can go up to 70+ seconds for a linear topology with 100 nodes. In our experiments, we noticed a significant long delay after the number of nodes is above 80 for a linear topology. Furthermore, Mininet does not provide guarantee that a packet will be scheduled promptly when issued from a host when it is under high load. This could be a reason that caused some of the `ping` test returned unreasonably large round trip time after the network size goes up to 80+ nodes. Mininet also has minor issues where it's simulated forwarding in software requires $O(n)$ lookup time, but in hardware, this can be done within $O(1)$ time complexity.

3.3 Creating the testing environment

Users can interact with Mininet by the CLI in an ad-hoc way to create and to test the applications on a SDN network. The command line interface allows user to compactly specify which controller the network will be connected with, which topology the user is intreated in using, and the size of the network in the virtual machines terminal. The following commands creates a linear topology with 3 nodes in Mininet using the POX controller:

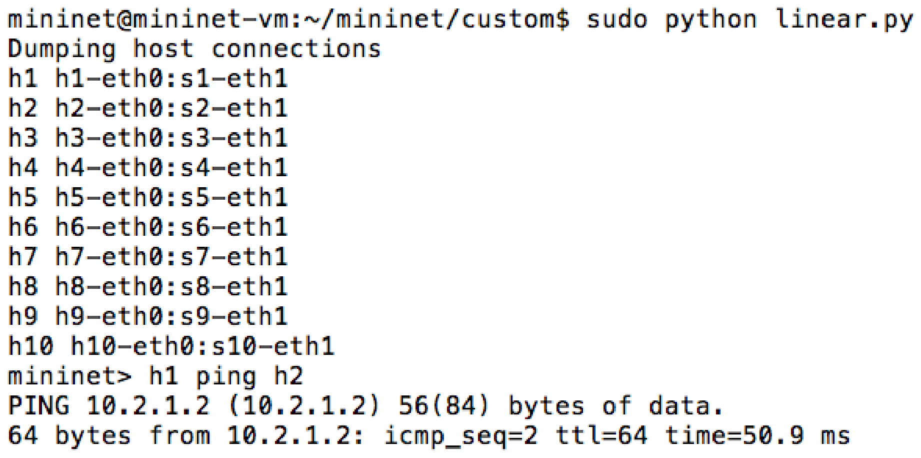
```
sudo mn --controller=remote,ip=192.17.160.85,port=6633 --mac --topo=linear,3
```

Mininet also supports python API to build customized networks. In the virtual machine that contains Mininet, there is folder named `custom` that contains examples of customized topologies. The above linear topology can be created by following code. Users have more freedom to express what actions should be performed by the network elements with the python API.

```
from mininet.net import Mininet
from mininet.topo import LinearTopo
from mininet.node import RemoteController
from mininet.cli import CLI
from mininet.util import dumpNodeConnections

myTopo = LinearTopo(10)
net = Mininet(topo=myTopo,controller=None)
```

```
net.addController('remote', controller=RemoteController, ip='192.17.160.85', port=6633)
hosts = [None]*10
for i in range(10):
        hosts[i] = net.hosts[i]
        hosts[i].setIP('10.'+str(i+1)+'.1.'+str(i+1))
        #print "Host", hosts[i].name, "has IP address", hosts[i].IP()
net.start()
print "Dumping host connections"
dumpNodeConnections(net.hosts)
CLI(net)
net.stop()
```

The above code creates a linear topology with 10 hosts and assigned IP addresses to them. We created customized testing environment for evaluation in similar ways. Figure 3.1 shows the output network created by the above code followed by a `ping` test after the routing rules are installed.



```
mininet@mininet-vm:~/mininet/custom$ sudo python linear.py
Dumping host connections
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
h4 h4-eth0:s4-eth1
h5 h5-eth0:s5-eth1
h6 h6-eth0:s6-eth1
h7 h7-eth0:s7-eth1
h8 h8-eth0:s8-eth1
h9 h9-eth0:s9-eth1
h10 h10-eth0:s10-eth1
mininet> h1 ping h2
PING 10.2.1.2 (10.2.1.2) 56(84) bytes of data.
64 bytes from 10.2.1.2: icmp_seq=2 ttl=64 time=50.9 ms
```

Figure 3.1: A customized network with 10 hosts

Chapter 4

Connecting POX with Database

In this section, we focus on building the connection between the POX controller and the database. We identified several modules, such as `forwarding.topo_proactive`, `forwarding.l2multi`, and `misc.gephi_topo`, etc. that all need to refer to the topology of the network. Currently, each of these modules needs to start by creating maps and lists to maintain switches, links, and hosts information. This resulted in code duplication and developers are forced to start by searching for each corresponding event to handle topology updates rather than directly focusing on the application logic. Furthermore, if the controller is abruptly shut down, the entire network state will be lost. We developed a new database module to provide a persistent storage for POX to keep track of the current state of the network. To illustrate how topology dependent modules can use the database module, we modified the original `forwarding.topo_proactive` module to read topology information from the database instead and then computes the routes for any given two end points. We measured the overhead introduced by the database module by recording time spent on each database operation and detailed the result in the evaluation section.

4.1 Current limitations

There are mainly two potential drawbacks without using a database as a backend storage in POX. The first problem is that the network state needs to be stored separately in every module that depend on it. In other words, each new module that relies on topology updates has to first create it's own copy before implementing any module specific code. A more space-efficient and modular scheme would be to store the network state at one place in a persistent storage, and all the other modules that need to use topology information can then retrieve it using commonly used database operations. Having the network topology available in the database also enabled richer database based management schemes to be applied in SDN. Specifically, the Ravel system can be used here to coordinate SDN applications with standard SQL queries.

Furthermore, it can be problematic when the controller is abruptly shutdown, in which case the entire network state is lost. To better handle controller failures, one strategy is to keep a checkpoint and constantly update it to reflect the latest state of the network. If unexpected failure occurs, the controller can retrieve the most recent information from the database and resume to learn new updates. The database module is able to constantly detect topology change by listening to specific events from POX, and updates the corresponding information in database.

4.2 Design overview

Since POX already provides an extensible framework to monitor and control OpenFlow switches, the database functionality can be added into the controller by using the API. We also want to keep the database module compatible with all existing components so topology dependent applications can choose which implementation suits better for their needs. At first glance, we can embed the code into `openflow.discovery`, `openflow.topology`, and `host_tracker.host_tracker` wherever it logs the link, switch, or host changes of the network and prints it to the console. For example, we can change the log operations in `openflow.discovery` to database updates.

```
if link not in self.adjacency:
      self.adjacency[link] = time.time()
      log.info('link detected: %s', link)
      # change to insert
      cursor.execute("INSERT INTO link(in_switch,in_port,out_switch,out_port) VALUES
      (%s,%s,%s,%s)",(l.dpid1,l.port1,l.dpid2,l.port2))
      self.raiseEventNoErrors(LinkEvent, True, link)
else:
      # Just update timestamp
      self.adjacency[link] = time.time()
```

Another approach is to bundle all the database operations into a new module and plug the module into POX. Having a separate module has several advantages. First of all, it leaves the original implementation of the POX modules untouched. Since we are building a separate module on top of POX using its standard API, other modules who want to use the database component only need to include the module in the command line when starting the controller. Moreover, developers can switch between the two according to different requirements. Due to better modularity and maintainability of this approach, we implemented a database module named `ext.db` and placed it under the `ext` directory which contains all the user defined extensions in POX. The design overview is illustrated in figure 4.1. We now inserted an other module between the application and the POX core layer to implement the database functionality. Network events will be propagate to the database module through the POX core, and existing applications can then refer to them by performing SQL queries.

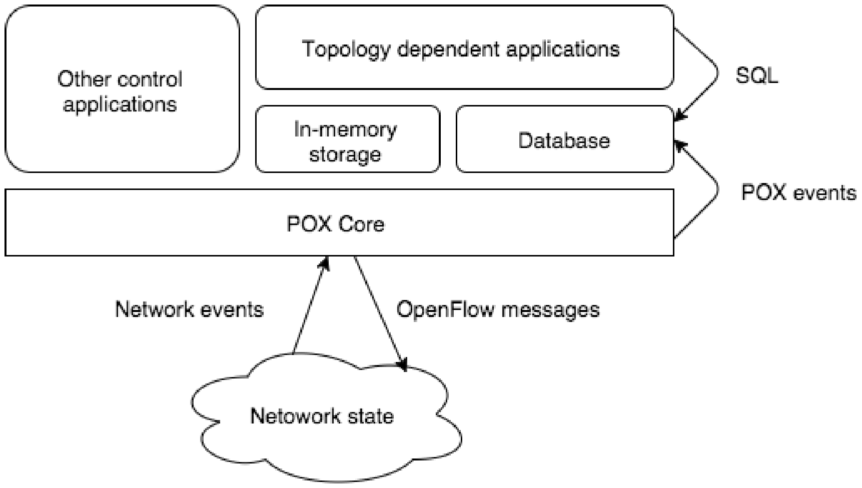


Figure 4.1: Design overview

4.3 Implementation details

In this section, we present the implementation of the database component by first looking at a demo with a simple network to check the basic functionalities. Then, we will discuss how other applications in POX, such as a proactive routing module, can make use of the database module and tested the database overhead on Mininet.

4.3.1 Mapping a simple network

The following topology shown in figure 4.2 is used to demonstrate the basic functionalities of the database component. The simple network consists of 3 hosts each connecting to a switch in a liner fashion.

We begin by connecting to the Postgres database and create tables for the three basic network elements: switch, host, and link. Those operations: obtaining a cursor, setting automatic commit, and table creation will need to be completed at the time when the controller starts. We illustrated the initialization of the database module in the following code:

```
class db ():
```

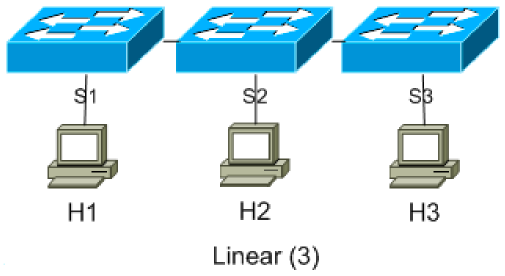


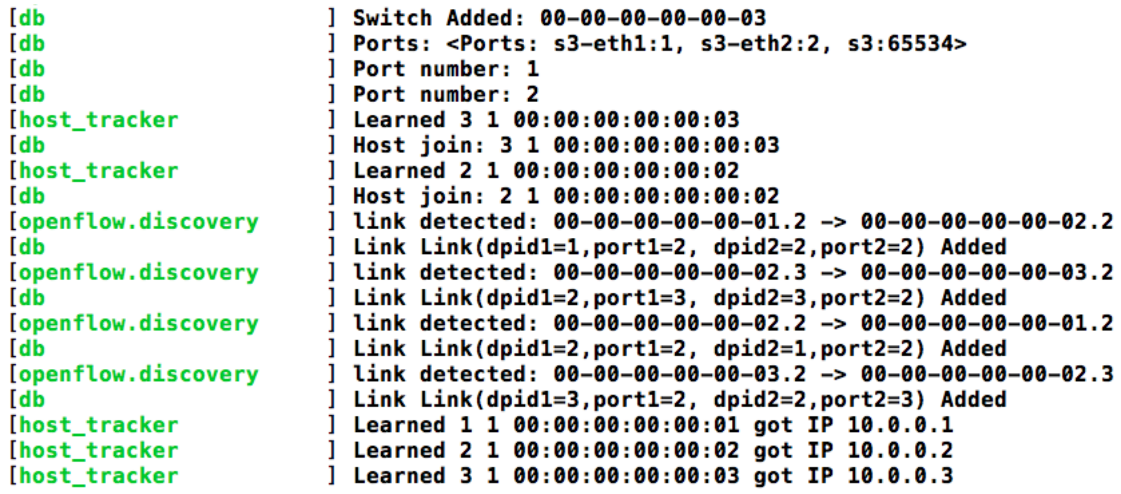
Figure 4.2: Example topology

```
   def __init__ (self):
      #Connect to Postgres
      conn_string = "host='localhost' dbname='testdb' user='fanyang' password=''"
      conn = psycopg2.connect(conn_string)
      conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
      global cursor
      cursor = conn.cursor()
      log.info("Connected to Database!\n")
      cursor.execute("CREATE TABLE switch(id serial primary key, switch int, port int);")
...
```

Once the tables are created, all the network elements will be stored in Postgres and users can easily lookup their states from the database.

Since POX uses publisher and subscriber scheme to manage the interaction among components, we need to subscribe the database module to POX's core object, and setup event handlers to receive the topology updates after they are detected by the controller. Specifically, we are interested in listening to `ConnectionUp`, `ConnectionDown`, `LinkEvent`, and `HostEvent`. The following code segment shows how to set up a handler to detect newly joined switches and save their status into the database. Other event handlers can be set up in a similar way.

```
def _handle_ConnectionUp (self, event):
  s_dpid = dpid_to_str(event.dpid)
  log.info("Switch Added: " + s_dpid)
  ports = event.connection.ports.__str__();
  log.info("Ports: " + ports)
```



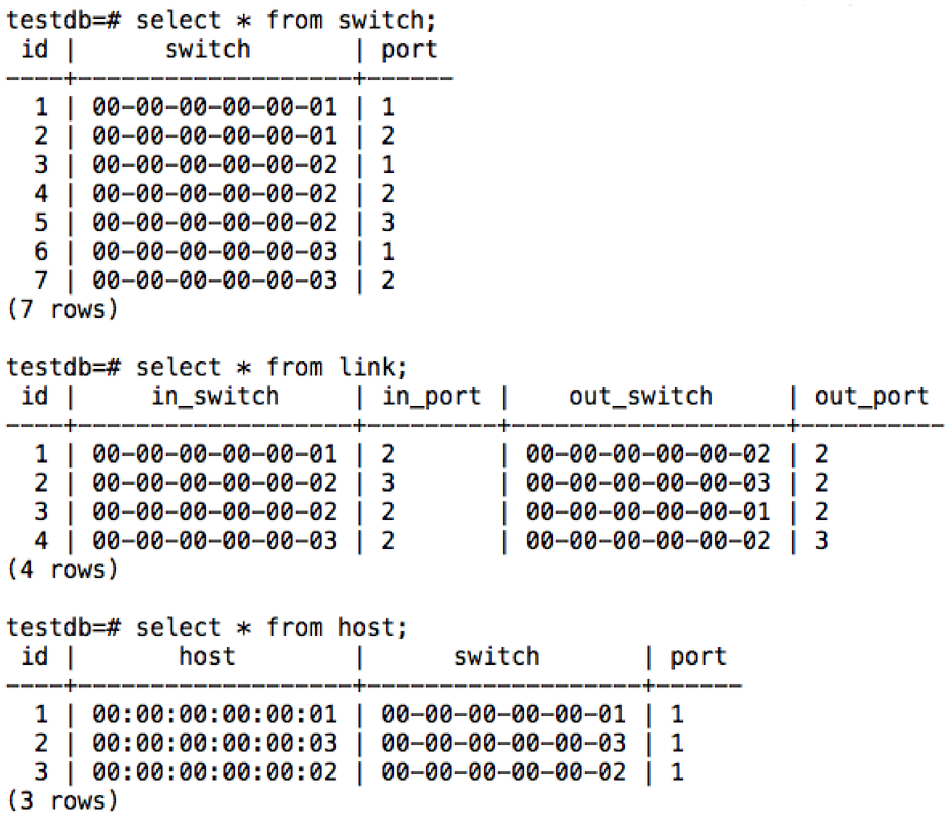
```
[db                      ] Switch Added: 00-00-00-00-00-03
[db                      ] Ports: <Ports: s3-eth1:1, s3-eth2:2, s3:65534>
[db                      ] Port number: 1
[db                      ] Port number: 2
[host_tracker            ] Learned 3 1 00:00:00:00:00:03
[db                      ] Host join: 3 1 00:00:00:00:00:03
[host_tracker            ] Learned 2 1 00:00:00:00:00:02
[db                      ] Host join: 2 1 00:00:00:00:00:02
[openflow.discovery      ] link detected: 00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
[db                      ] Link Link(dpid1=1,port1=2, dpid2=2,port2=2) Added
[openflow.discovery      ] link detected: 00-00-00-00-00-02.3 -> 00-00-00-00-00-03.2
[db                      ] Link Link(dpid1=2,port1=3, dpid2=3,port2=2) Added
[openflow.discovery      ] link detected: 00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
[db                      ] Link Link(dpid1=2,port1=2, dpid2=1,port2=2) Added
[openflow.discovery      ] link detected: 00-00-00-00-00-03.2 -> 00-00-00-00-00-02.3
[db                      ] Link Link(dpid1=3,port1=2, dpid2=2,port2=3) Added
[host_tracker            ] Learned 1 1 00:00:00:00:00:01 got IP 10.0.0.1
[host_tracker            ] Learned 2 1 00:00:00:00:00:02 got IP 10.0.0.2
[host_tracker            ] Learned 3 1 00:00:00:00:00:03 got IP 10.0.0.3
```

Figure 4.3: The initial topology in pox

```
#Add all ports attached to this switch
for val in event.connection.ports.itervalues():
if "-eth" in val.name:
  log.info("Port number: " + str(val.port_no))
  port_no = str(val.port_no)
  cursor.execute("INSERT INTO switch (switch,port) VALUES (%s,%s)",(s_dpid,val.port_no))
```

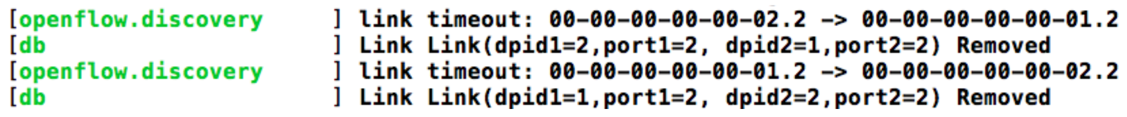
After finishing setting up different event handlers for monitoring different events on the network, we can test the database module with the sample network. We expect to see that the corresponding state is mapped into the database after the network has been created by Mininet. Furthermore, if there is any changes in the network, such as a link goes down or a new switch shows up, that event should be immediately reflected in the database as well.

Let us verify the functionalities with the simple network topology showed in figure 4.2. Given the topology, we can expect to see 3 switches and 3 hosts are stored in the switch and host table respectively, and 6 links (each line represents two directions, and there are 3 in total) stored in the link table. Part of the log information for the initial topology from the POX controller is shown in figure 4.3. The green column shows which source module gives the output that is shown in black. We can see that `db` detects switch, link, and host updates of the network in the same way as `open_flow.discovery` and `host_tracker` detects the changes of the network. The output of the database module for the initial topology is shown in Figure 4.4.



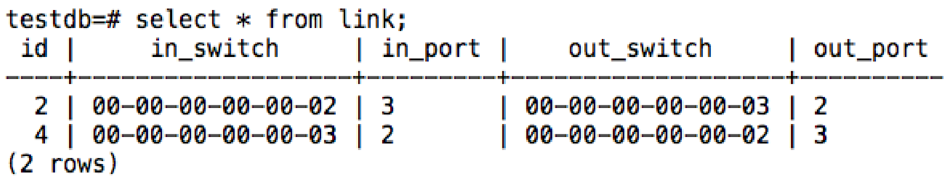
```
testdb=# select * from switch;
 id |       switch        | port
----+---------------------+------
  1 | 00-00-00-00-00-01 | 1
  2 | 00-00-00-00-00-01 | 2
  3 | 00-00-00-00-00-02 | 1
  4 | 00-00-00-00-00-02 | 2
  5 | 00-00-00-00-00-02 | 3
  6 | 00-00-00-00-00-03 | 1
  7 | 00-00-00-00-00-03 | 2
(7 rows)

testdb=# select * from link;
 id |     in_switch      | in_port |     out_switch     | out_port
----+--------------------+---------+--------------------+----------
  1 | 00-00-00-00-00-01 | 2       | 00-00-00-00-00-02 | 2
  2 | 00-00-00-00-00-02 | 3       | 00-00-00-00-00-03 | 2
  3 | 00-00-00-00-00-02 | 2       | 00-00-00-00-00-01 | 2
  4 | 00-00-00-00-00-03 | 2       | 00-00-00-00-00-02 | 3
(4 rows)

testdb=# select * from host;
 id |        host        |       switch       | port
----+--------------------+--------------------+------
  1 | 00:00:00:00:00:01 | 00-00-00-00-00-01 | 1
  2 | 00:00:00:00:00:03 | 00-00-00-00-00-03 | 1
  3 | 00:00:00:00:00:02 | 00-00-00-00-00-02 | 1
(3 rows)
```

Figure 4.4: The initial topology in database



```
[openflow.discovery      ] link timeout: 00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
[db                      ] Link Link(dpid1=2,port1=2, dpid2=1,port2=2) Removed
[openflow.discovery      ] link timeout: 00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
[db                      ] Link Link(dpid1=1,port1=2, dpid2=2,port2=2) Removed
```

Figure 4.5: Link failure in pox



```
testdb=# select * from link;
 id |     in_switch      | in_port |     out_switch     | out_port
----+--------------------+---------+--------------------+----------
  2 | 00-00-00-00-00-02 | 3       | 00-00-00-00-00-03 | 2
  4 | 00-00-00-00-00-03 | 2       | 00-00-00-00-00-02 | 3
(2 rows)
```

Figure 4.6: Link failure in database

From the three tables, we can see that all the information is captured correctly into the database.

Let us move on to examine what will happen if there is a change in the network. If we turned down the link between $s1$ and $s2$ to simulate a link failure, we can see that this change is detected immediately by the controller as shown in figure 4.5. So now if we examine the link table in the database, which is shown in figure 4.6, the record for that link should also be deleted. After comparing the outputs of both situations, we can conclude that the network is correctly mapped into the database by the `db` module.

4.3.2 Adapting the routing module

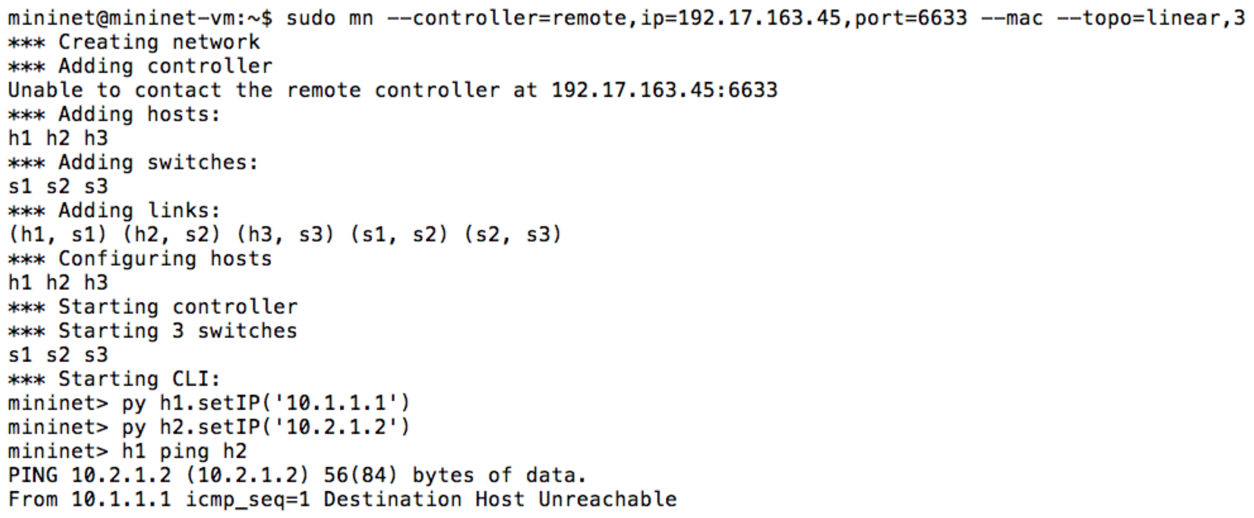
There are several modules, e.g. `forwarding.l2_multi`, `forwarding.topo_proactive`, etc, in POX that are topology dependent. We adapted the routing module `forwarding.topo_proactive` to demonstrate the use of the database component, and other applications can be modified in similar ways.

The routing module implements a proactive policy to route packets. Initially, the openFlow switches are not aware of any routing rules. Upon start, as the controller discovers new links and switches on the network, the shortest path between any given pair on the current topology will be calculated. Subsequent updates from the network will trigger re-computes of all the routes. Once route computation is done, the forwarding rules will be sent by the routing module to the switches by using `_ofp_flow_mod` method with the proper message payload.

The major changes we made in the routing module are primarily in `_calc_paths()` and `_get_raw_path()` function. The `_get_raw_path()` will call `_calc_paths()` and recursively build a path between a given pair of `src` and `dst`. The modified `_calc_paths()` function will read information about the switches, ports, and links from the database instead.

4.4 Evaluation

To evaluate the performance the database module, we tested if the adapted module can correctly routes a packet given a source and destination pair. Then, we measured the overhead of retrieving link and swich information from the database by timing all the SQL queries made in the routing module.



```
mininet@mininet-vm:~$ sudo mn --controller=remote,ip=192.17.163.45,port=6633 --mac --topo=linear,3
*** Creating network
*** Adding controller
Unable to contact the remote controller at 192.17.163.45:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s1, s2) (s2, s3)
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 3 switches
s1 s2 s3
*** Starting CLI:
mininet> py h1.setIP('10.1.1.1')
mininet> py h2.setIP('10.2.1.2')
mininet> h1 ping h2
PING 10.2.1.2 (10.2.1.2) 56(84) bytes of data.
From 10.1.1.1 icmp_seq=1 Destination Host Unreachable
```

Figure 4.7: Initially, switches are unaware of any routing rules

4.4.1 Testing connectivity

This test is done on the simple network shown in figure 4.2 by running the `ping` test between two hosts $H1$ and $H3$. If we start POX without the routing component, we can expect to see no messages can be delivered between the two hosts, which is shown in figure 4.7.

After adding the routing module, we can see that both implementation successfully computes the routes and the round trip time (RTT) is returned as below:

```
Without database: rtt min/avg/max/mdev = 0.054/0.085/0.202/0.048 ms
With database: rtt min/avg/max/mdev = 0.055/0.380/2.309/0.787 ms
```

4.4.2 Testing the database overhead

One of the biggest concerns of moving the topology information into the database is how much delay the database operation may cause as compare to in-memory implementations. We measured the overhead of querying the database by inserting a timing method around each query and averaged over all queries. Figure 4.8 shows the execution time of the select query on a 20 nodes (10 hosts and 10 switches) network. There are two places where we changed the in-memory access to database select operations, and after the 10 switches all discovered by the controller, we plotted the graph of the 20 query execution time. We can see that the latency is fairly stable around 0.00282 seconds, and since the algorithm computes the routes each time a link is discovered, we also observed that the latency does not vary much across different iterations. The overall overhead introduced by including the database component in this routing module is therefore given by the total number of operations times the average latency per operation, which is approximately 0.00564 seconds.

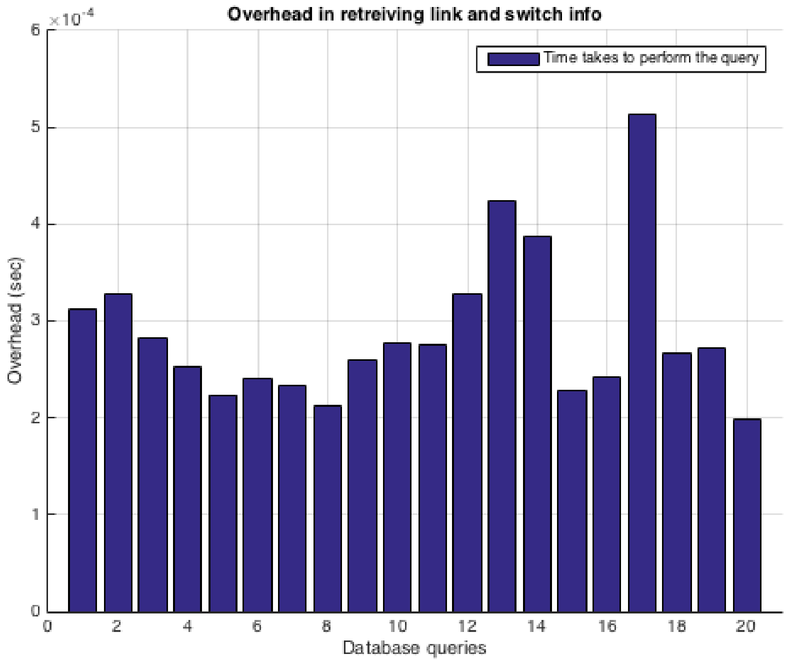


Figure 4.8: Overhead of database operations

4.5 Discussion

Prior to evaluate the overhead of the database operations, we ran the `ping` test on different size of networks provided with a randomly selected a source and a destination. This helped us to verify if indeed both implementations have correctly installed routing rules at the switches. What we found was that the RTT changes with respect to the given pair of nodes and the network size. In most cases, the test returns a RTT ranging from 0.064 ms to 350 ms. Approximately up to more than 80 nodes, the `ping` test returned unreasonably large RTT or gives `Destination Host Unreachable`. Our conclusion is that the overhead of including the database component in the routing module is fairly small given the the size of the network is under 80 nodes, and we would expect the change in general case is still reasonable given that per operation cost is about 0.00282 seconds.

Chapter 5

Related Work

There has been many research papers focusing on improving the SDN controller platform by having the database as their component. In this chapter, we review the core ideas from the most relevant work and discuss how they relate to this project.

5.1 Database used in Ravel

Ravel [14] presents a database-centered SDN design to orchestrate various kinds of control applications. In Ravel, the SDN controller is implemented using the PostgresSQL database. Network configurations are stored as base tables, and the control is achieved by querying or updating virtual views derived from the base tables. Ravel supports: (1) vertical orchestration which synchronizes the database view with the actual network state; (2) horizontal orchestration which coordinates multiple control applications with user specified priorities. Ravel demonstrates that using the relational database representation can lead to an efficient solution for coordinating SDN applications with the average per-operation overhead less than 10ms.

Building the database component in POX is the key step to connect the Ravel project to the actual software OpenFlow switches. During the setup phase of Ravel, all the network elements on Mininet need to be mapped into Postgres. And while Ravel is running, the network state needs to be constantly monitored. Both can be easily achieved using POX with the database component enabled. After the network state is made available within the database, the Ravel system can behave as the SDN controller. Applications and users can manage the network state through standard SQL queries with the vertical and horizontal orchestration services provided in Ravel.

5.2 Database used in Onix

Using relational database to store network states was also explored in Onix. Onix [10] aims to build a SDN platform as a distributed system to address the scalability issue of SDN. The controller platform consists of several Onix instances each maintaining a Network Information Base (NIB) to keep track of the network state. Onix implements a persistent SQL store in order to provide strong consistency and durability of the network state among those instances. The updates from one NIB are disseminated to other NIBs via database triggers. Onix shows the effectiveness of using the transactional database to maintain a consistent view of the network state across many control applications.

The database module we developed applies similar strategies to the POX controller. With the database component, different controller applications can refer to the same datastore for topology updates. More importantly, the database module also provides the connection to Ravel, which can support rich application orchestration features that Onix does not support.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

SDN architecture decoupled the control plane from the data plan and enabled more extensible and modular design of networks. POX implements a SDN controller, which provides developers with an simple and open interface to build new applications. In this project, we studied the key design decisions made in POX and optimized the controller by extending a database component to maintain the network state in a persistent storage. Previously, developers had to begin by searching for all relevant event handlers in order to maintain network state information. After mapping the network state into the database, the topology information can be accessed all from one place and is stored persistently. Network administrators can use SQL queries to manage which applications are allowed to access which part of the topology in the database. The basic database functionalities are adapted in the routing module and are tested with Mininet. From the test results, we can conclude that the database module can correctly map the network into the database and the overhead of including this functionality is reasonably small.

6.2 Future work

Future optimizations can be added to directly manage the OpenFlow switches and computing routes inside the database.

6.2.1 Changing flow table by SQL queries

To manage the topology information in a finer grain, we can explore the potential of using the POX web API and the Postgres triggers to allow users to input SQL queries and change the flow table of the switches. Besides switch, link, and topology table, we could also main a flow table for each switch and use Postgres triggers to asynchronously pass user queries to those switches. This feature will allow users to have finer control over the network within the database.

6.2.2 Computing routes inside the database

Another optimization we can do is to compute the routes within the database itself by using Postgres extensions. Since the entire topology information is available in the database, we could use pgRouting to further simply the task for topology dependent applications. pgRouting [3] supports A*, shortest path, and many other well known routing algorithms. With the support of the database module, we can run pgRouting over the topology table. The routing application only need to specify two end-points, all the path computation is done within the database, and the path and be queried by SQL queries.

References

[1] Floodlight Project. `http://www.projectfloodlight.org/floodlight/`.

[2] Mininet API. `http://mininet.org/`.

[3] pgRouting Project. `http://pgrouting.org/`.

[4] POX API. `https://openflow.stanford.edu/display/ONL/POX+Wiki`.

[5] Teaching computer networking with Mininet. `http://conferences.sigcomm.org/sigcomm/2014/doc/slides/mininet-intro.pdf`.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12. ACM, 2007.

[7] D. Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2013.

[8] N. Feamster, J. Rexford, and E. Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[10] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[11] B. Lantz, B. Heller, N. Handigol, V. Jeyakumar, and B. OConnor. Mininet-an instant virtual network on your laptop (or other pc), 2015.

[12] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[14] A. Wang, B. Godfrey, and M. Caesar. Ravel: Orchestrating software-defined networks. In *SOSR Demo*, 2015.