SDN Project: Load Balancer with Space Efficient Rules

By Dekel Auster and Michal Shagam

Abstract:

The goal is to create a load balancer with a space efficient set of rules (linear to the amount of ranges/hosts) that redirects and splits traffic between several hosts of a certain resource.

General Terms:

TCAM – ternary content addressable memory – high speed memory used for searching. Data can be stored and queried based on three types of input: zero, one and don't care (wild card).

LCP – Longest Common bit Prefix.

ELCP 0/1 – extended LCP. Adds a single differentiating bit and completed by wildcards.

Wildcard bits (displayed in diagrams as '*') were implemented using masks (zeros for wildcard bits and ones for the rest).

Openflow switch table - TCAM tables with values for the packet fields (wildcards for the rest) and rules/instructions to perform on a matching packet.

Openflow Packet Metadata – packet section that can be set and matched, allows values to be passed between tables in a pipeline.

Rule priority – matching priority between rules placed in the same table. Rules with higher priority checked first.

Important instructions and actions:

Goto action – instructs the switch to pass the packet matched to the specified table, continuing the search and creating a pipeline. This instruction can only be specified when the table is not empty.

Set Field action – sets the value of a field. For example, this action can be used to replace the addresses.

Output action – sends the packet.

Write metadata instruction – sets the value of the metadata field.

Introduction:

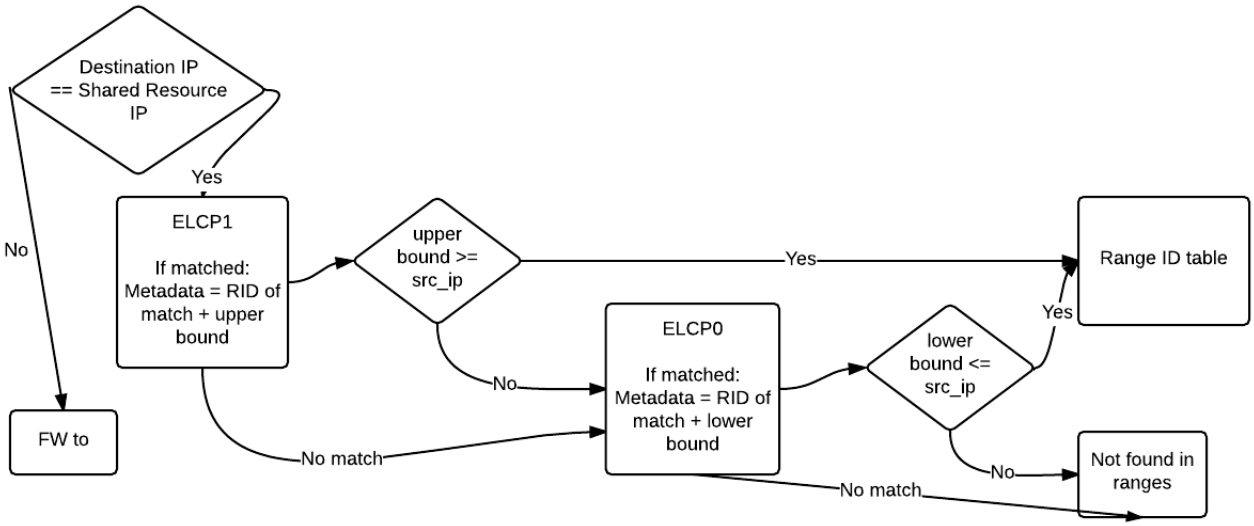


Figure 1: Diagram of the match process

The load balancer directs traffic based on IP packets (eth_type=0x800). We use a common destination IP to define the shared resource while the ranges are based on the IPv4 source address. Each packet with the shared resource's IP is redirected to one of the resource hosts based on the range to which the packet's source address belongs. For our project, the range span covers the entire IPv4 address space although different ranges can be defined (non-overlapping).

The controller cannot be used to redirect the packets since the switches' controller forwarding is limited to 200 packets per second [1]. Therefore, we must use rule defined redirection. Finding the packet range is done by matching and comparisons. A naïve implementation would be to create a rule for each packet combining rules whenever we can. In this case the rules are not linear to the amount of resource hosts.

The match process:

The first table, the first decision module checks whether the packets were sent to the shared resource (Figure 1) and passes those packets to the following tables. This filtering method utilizes the table pipeline as a priority control method.

The ELCP1 table is a list of the ELCP1's defined for the ranges – each rule's priority being of size LCP length + 1. When a packet matches the rule, the range ID and the range's upper bound are placed in the packet's metadata section, each of size 32bits and the section size being 64bits. The packet is then forwarded to the next table which implements a comparator.

The comparator checks that the IP meets the boundary requirement, not only the LCP. Since our ranges span the entire address space, satisfying either one of the match and boundary pairs suffices. When the packet reaches the RIDs table, the range ID is then matched to one of the entries. When a match is

found, the packet's destination IP and mac addresses are replaced by the appropriate host's addresses. The packet is then passed on to the common resource switch.

A Detailed look at parts of the process:

ELCP Creation and Comparator Implementation:

The upper and lower bound 32bit IPv4 addresses for each range are used to define the ELCPs. The LCP is found and used to define ELCP0 and ELCP1. ELCP0 = LCP + 0 + * (completion to 32 bits with wildcards). ELCP1 = LCP + 1 + * (completion to 32 bits with wildcards).

The comparator can't be defined as one simple rule. Matches do not offer comparisons as atomic actions. We can perform a bit (*i*th bit from each value) comparison indirectly with two rules (3 for a full comparison). For two numbers, in order to implement the comparator, we can repeat the bit comparison for each of the bits taking into consideration the bit order giving more significant bits higher priority.

Comparison for the ***i***th bit:

A	B
$**** 1 *****,$	$**** 0 ***** \rightarrow A\ is\ larger\ than\ B$
$**** 0 *****,$	$**** 1 ***** \rightarrow B\ is\ larger\ than\ A$
$**** X *****,$	$**** X ***** \rightarrow continue\ to\ following\ bit$

The comparator is implemented using rule priority comparing the more significant bits first. The comparator searches for the first bit that differs in order to decide which value is larger. If none differ, they are equal. The comparator is created in the beginning of the program and doesn't change later on. The values it compares change but remain in the same packet sections – the lower half of the metadata and the IPv4 source address.

For each bit index, the bit comparison (shown above) is performed. In the last case, the comparison continues to a rule of lesser priority until all the bits are checked. For a comparator of 32 bits – we have 65 rules, the last being the values are equal.

Creating the first set of rules:

The first set of ranges is created based on the server list given in the configuration file (also described under usage instructions). The list includes the IP and MAC addresses of the resource hosts used including the chosen weight distribution (each host is given a weight between 0 and 1. The sum of all the weights equals 1). The upper and lower boundaries are created based on these weights. The tables are then created using the first set of boundaries.

Updating Ranges:

The number of packets sent to each of the resource hosts was counted. Every interval, the counters are checked (their previous values kept – used to determine the previous interval packet count). When the ratio of the ranges with maximum and minimum packet count is two or more, the ranges are repartitioned.

We used a probabilistic method of repartitioning. A portion, randomly generated, of the range with the maximum packet count was transferred to the ranges above and below leaving the rest of the ranges unchanged. The fraction of the range added to the range above and below were determined by generating a uniform random number ([0,1)) and the distribution between above and below was also determined by a generated random number.

The maximum and minimum count ranges are not expected to be adjoining ranges. After a certain amount of intervals, the counters are expected to reach equilibrium in terms of the packet count ratio.

For example, if the max count is 100 and the min count is 30, the range will be repartitioned. If for example, the repartition function generated the numbers .35 and .73, 35% of the max count range will be transferred, 25.55% to the range above (0.73 · 35%) and 9.45% to the range below.

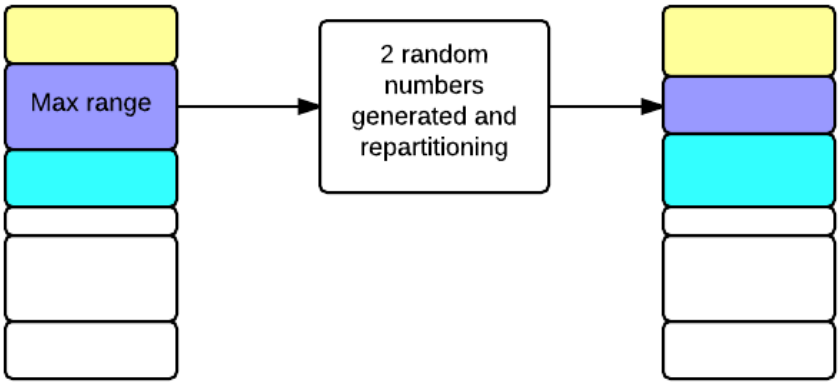


Figure 2: repartitioning of the maximum packet count range and the ranges below and above

The rules that depend on metadata matching – the values matched are located in the metadata (the comparator tables and the RIDs table) - don't need to be changed since they are not range boundary dependent. This leaves the ELCP1/0 tables that require updating when the ranges change.

Topology:

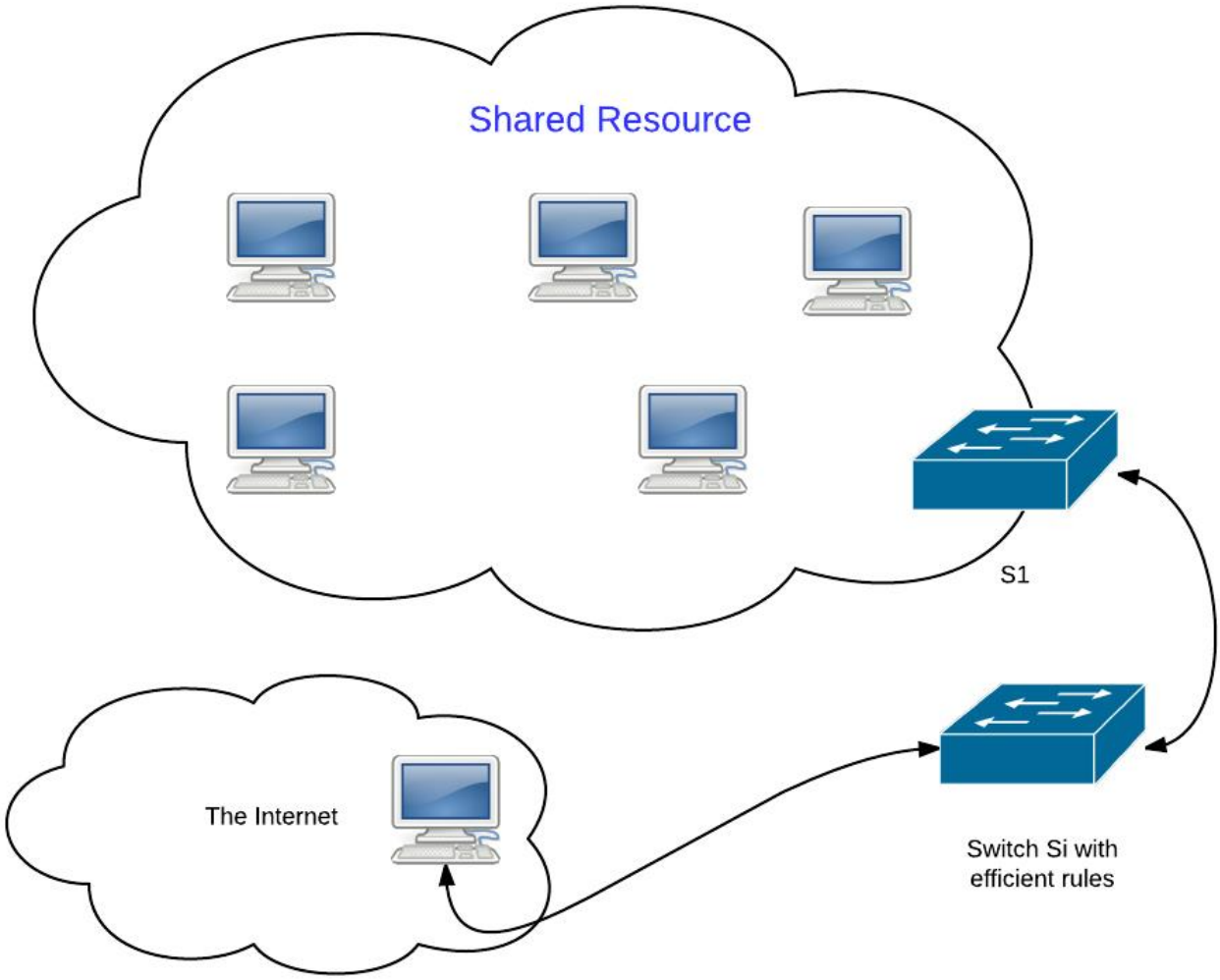


Figure 3: shared resource topology

The topology depicted in the figure above was used in order to test the load balancer. Two switches were used in order to simplify the rule set and handling of packets. Each host in the resource has its own internal IP, however externally; the entire resource has one shared IP, the load balancer IP.

The virtual server IP address and MAC address are required to be the addresses of a host connected to S1 directly. Otherwise, S1 won't receive the packet. This simplifies transferring the packet between the hosts (the exact port linking S1 and Si do not need to be used in this case).

Usage Instructions:

The project is based on the Openflow1.3 protocol.

In order to create the switch with the custom topology:

- sudo mn --custom ./ryu/ryu/app/loadtopo.py --topo load,[number of resource hosts],[number of outside hosts] --mac --switch ovsk --controller remote

or simply use the default parameters (4 resource hosts and 3 outside hosts):

- sudo mn --custom ./ryu/ryu/app/loadtopo.py --topo load --mac --switch ovsk --controller remote

For each switch, define the correct Openflow protocol:

- sudo ovs-vsctl set bridge s1 protocols=OpenFlow13
- sudo ovs-vsctl set bridge s2 protocols=OpenFlow13

Configuration:

Before starting the controller, please check that the configuration in ryu/ryu/app/loadbalancerconfig.py is correct. These are later imported into the controller module.

We've added the toolset module, confighelper.py, for creating randomly distributed or even weights.

Parameters:

- virtual_server – a tuple with the virtual MAC and virtual IP for the shared resource.
  For example:
  virtual_server = ("00:00:00:00:00:4", "10.0.0.4")
  Note that the virtual server needs to have the MAC and IP address of one of the hosts connected to s1 for the load balancer to work correctly.
- servers – A list of tuples of the MAC address, IP address and the weight. The weights for all servers must be smaller than 1 and add up to 1.
  For example:
  servers = [("00:00:00:00:00:01", "10.0.0.1", 0.33), ("00:00:00:00:00:02", "10.0.0.2", 0.33), ("00:00:00:00:00:03", "10.0.0.3", 0.34)]
  The confighelper functions create_even_weight_servers and create_random_weight_servers create these types of lists assuming the --mac option was used.

Starting the controller:

- ryu-manager ryu/ryu/app/micdekload.py

Use the following instruction to see the switch rule set:

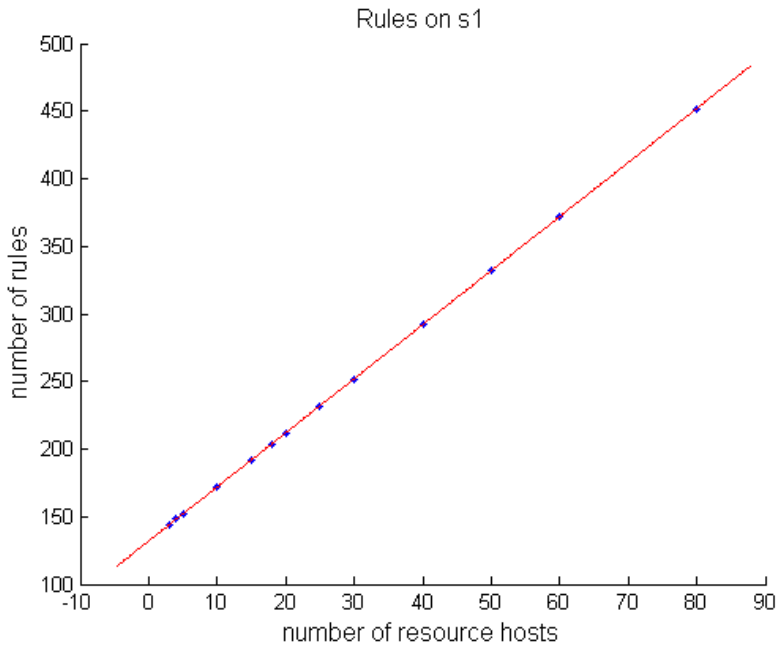
- sudo ovs-ofctl dump-flows s2 –O OpenFlow13

The shared resource can offer access to a website for example:

- h1 python –m SimpleHTTPServer 80 &
- h6 wget [virtual IP]

Statistics:

The number of rules as a function of the number of resource hosts does not depend on the weight distribution.



As expected, the rule count y acts as a linear function of the number of resource hosts x:

(1) $y = 132 + 4x$

4 rules are added for every new host. This rule count does not change after updates since the range boundaries are simply replaced.

The rule count for a naïve load balancer can be much higher, in the worst case up to 2*w-4 entries per range.[1] Meaning the constant is much greater and depends on the range length. On average the rule count is closer to w entries per range.

The space efficient rule count beats the worst case naïve rule count after only 3 resource hosts and increases much faster as can be seen in Figure 4.

[1] W is the length of the range boundaries. For a 32 bit address space, this would be 32.

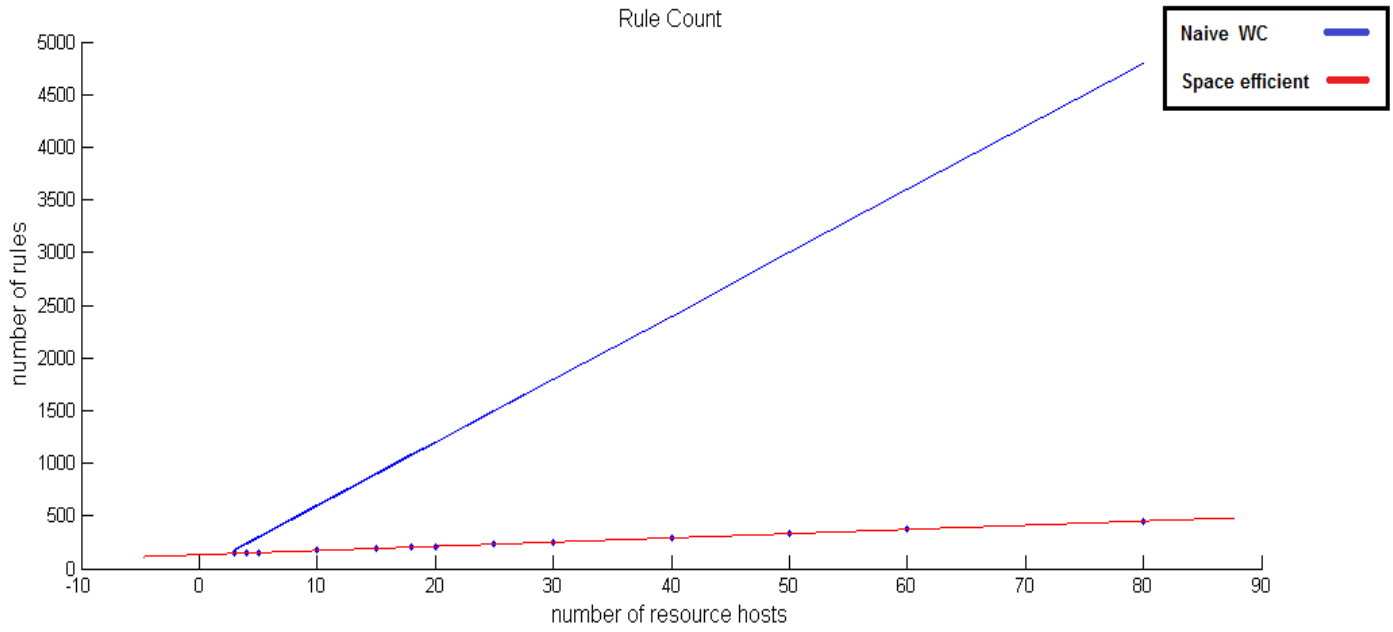


Figure 4: Rule count for Naive WC and space efficient rules

Rule updating statistics:

Statically, 3 resource hosts is the amount of hosts where the rule count for the space efficient load balancer and the naïve load balancer in the worst case almost intercept. Therefore, we will perform a few tests in this mode to identify and single out the dynamic component of efficiency (rule updating).

The first test checks a worst case scenario where a specific IP sends all of the requests and the imbalance is at a maximum. Out of the 3 hosts (starting with evenly distributed weights), we setup a simple http server for one of them. In the period of a minute, we sent 32 requests, causing 7 rule updates amounting to 42 changed rules. In 25 of the requests we reached the host with the site server while in 7 other times, we reached on of the other two servers. The average (of the 25 successful requests) byte transfer rate was 177.2 MB/s.

$$\frac{updates}{requests} = .21875$$

Another worst case scenario depends on the starting weights. Instead of having an imbalance in requests, we can create an imbalance in the weight distribution. These two cases are equivalent.

In order to model the average case, we will average several random weight distributions. In order to normalize the results (to that of the evenly distributed weights), we will send requests in the period of a minute and compare the ratio of updates/requests. The standard deviation of this ratio will be taken as the ratio error.

(2) $$\frac{updates}{requests} = .1509 \pm .0023$$

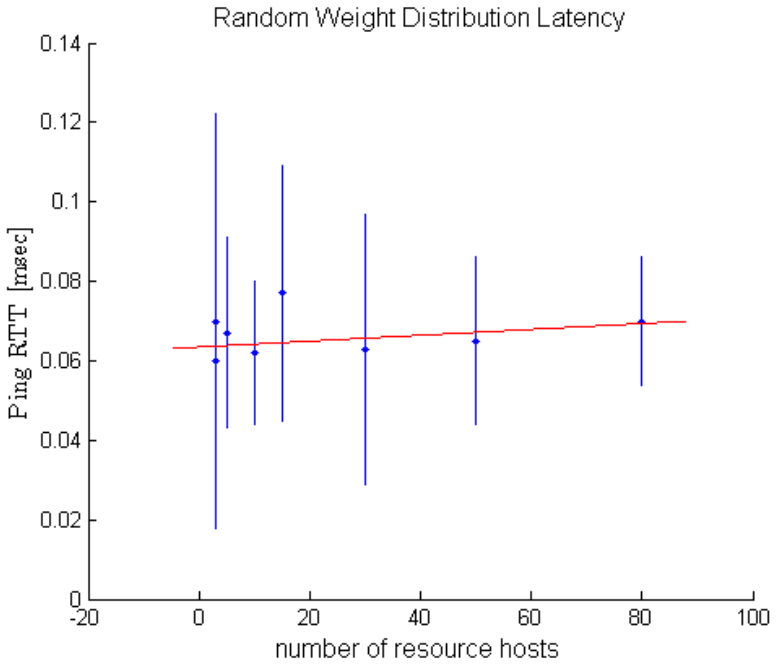
Latency check:

We wish to check the affects of both a pipeline in general and the number of hosts on the latency. In order to check this, we performed the ping test in several cases. The response time was recorded in each case and the standard deviation was taken as its error.

Influence of the number of hosts with random weights:

The response time is expected to be independent of the number of hosts. Graph 1 shows the response time for a random weight distribution of host ranges.

A linear fit was performed ($RTT = a_1 + a_2 \cdot \#(hosts_{resource})$).



Graph 1: Random Weight Range Distribution Fit

The linear fit results:

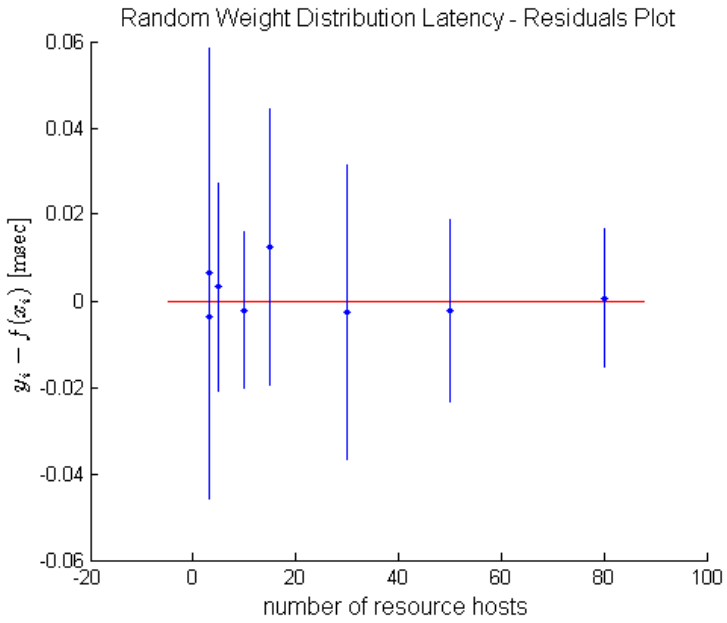
$a_1 = 0.0635 \pm 0.0080\ msec$

$a_2 = 0.00007 \pm 0.00018\ msec$

$\chi^2 = 0.25\ , \chi^2_{red} = 0.041\ , Pvalue = 0.00029$

The results show us that the latency does not depend on the number of hosts. The slope, a2, is effectively zero since it is smaller than its error. Visually, it does seem like the slope is small but there. A

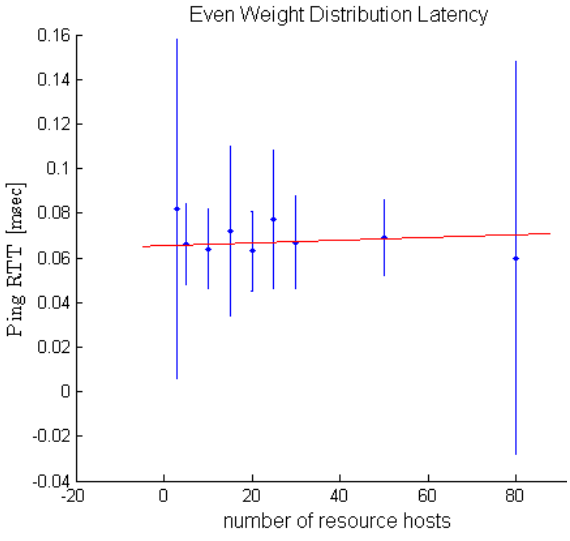
feasible explanation would be the linear match in each table. The rules are each checked linearly by priority and the more ranges, the more rules.



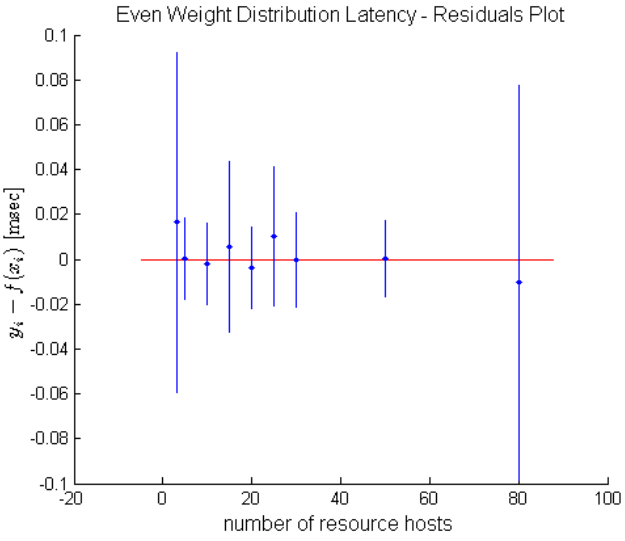
Graph 2: residuals plot for random weight distribution

From the residuals plot above, we can see that a linear fit was a correct choice -the residuals do not show us a specific pattern or distribution. The errors are very large and this may be the reason the slope is effectively zero.

Influence of the number of hosts with even weights:



Graph 3: response time for even weight distribution fit



Graph 4: even weights distribution residuals plot

The results for an even weight distribution are very similar to those of the random weight distribution. The average response time is larger but not by much, less than the standard deviation.

The linear fit results:

$a_1 = 0.0654 \pm 0.0076\ msec$

$a_2 = 0.00006 \pm 0.00027\ msec$

$\chi^2 = 0.24\ , \chi^2_{red} = 0.035\ , Pvalue = 0.000049$

Latency of pipeline:

Using the single topology (a single switch and 1 table in pipeline), the average latency was checked. Since the latency is checked between two hosts, the host count was also constant. The response time found here can be compared to that of the previous results taken for a pipeline of 6 tables.

The response time found was $0.064 \pm .030\ msec$. Both response times, with even and random weights, are well within one standard deviation of this value. Meaning, we cannot say the pipeline creates a latency issue as has been suggested. More so, the random weight response time was actually smaller than the one found here.

Discussion:

For a small number of hosts, the space efficient load balancer might not seem like the best choice but the rule count increases at a smaller pace and the simplicity of rule updating make it a better choice in the long run. The rule definition and most importantly, the amount of rules used for the space efficient load balancer do not depend on the range partition at all. Although the average amount of rule updates is large, the physical space taken at a certain moment is still very efficient and allows us to use a switch with less memory.

The space efficient load balancer beats the naïve load balancer with a smaller number of entries with 3 hosts in the WC and after 5 hosts on average.

In our implementation, the amount of ranges is kept constant and only the range distribution changes. Keeping the range count constant allows us to keep the tables that do not actually depend on the range boundaries (upper and lower) constant. This way, the amount of rules updated in each interval is either 6 or 0 depending on whether the ranges are repartitioned. A constant number of ranges, updating by repartitioning instead of merging and splitting, eliminates the need to update most of the tables. We also dodge possible bugs that can be caused by shifting the ranges.

When repartitioning, we chose to use a probabilistic method and not a deterministic one. Repartitioning based on a constant value does not always represent the actual load. Assuming the traffic in the network itself is stable - we expect the load balancer to reach equilibrium with probabilistic repartitioning. Determining the correct constant for repartitioning the ranges assumes previous

knowledge of the network's traffic distribution. Since our updates depend on the traffic during certain intervals, randomly choosing the ratio between the ranges merged with the upper and lower ranges was chosen instead of repartitioning all of the ranges. It is expected that for non-varying traffic, the result will be the same and if the traffic does vary, extreme repartitioning may not be the best choice. Both random aspects of the repartition help reduce the possibility of repetitive back and forth updates such as in a situation where the same ranges are always updated since a particular IP address in the range transferred between ranges is the source of most of the traffic.

The results for the comparison between the evenly distributed weights and the randomly distributed weights show us that on average, the random distribution and subsequent rule updating are more likely to fit the load balancer state. Evenly distributed weights will usually not represent the load balancer state and several rule updates are needed before the load balancer reaches a state of balance. In this project, we assumed the connections were short and therefore, the randomization of weights was the better choice bringing the state of the load balancer to balance quickly handling the changes fast.

The response times were found to be constant, not dependent on the number of hosts. In reality, the match process (linearly going over each rule) means there is a small dependency. We can see this visually although the results show the errors were too large to determine whether there actually is a dependency. The errors were chosen as the standard deviation and not the standard error (division by the $\sqrt{N}$ where N is the size of the sample) due to the way we defined a sample. This choice is debatable and we can see that the conclusions would be very different if the other type of error was chosen. The standard deviation was chosen as the error since the samples are not completely independent and don't constitute separate samples.

When checking for latency caused by the pipeline, the measurements did not suggest that the pipeline length affects the latency. A better way to check this would be to create several pipelines of varying orders that simply forward the packet. This would allow us to fit a function with more than one degree of freedom. The results would be more accurate, less dependent on the errors chosen, and we would be able to determine whether we missed a pattern/correlation from the residuals plot.

In summation, the load balancer was created with a space efficient rule set which is effective even after 3 or 5 resource hosts. The rule update count was minimal since most of the tables were unaffected by the boundary updates. More importantly, the possible side effects were found to be minimal or no larger than the expected errors.

References:

[1] http://conferences.sigcomm.org/sigcomm/2013/papers/sigcomm/p541.pdf
[2] https://www.sdncentral.com/technology/sdn-openflow-tcam-need-to-know/2012/07/
[3] http://sdnhub.org/resources/useful-mininet-setups/
[4] http://sdnhub.org/tutorials/opendaylight/
[5] http://www.cs.tau.ac.il/~schiffli/SDNX-Full.pdf
[6] http://pages.cs.wisc.edu/~agember/sdn/session4
[7] http://www.routereflector.com/2013/11/mininet-as-an-sdn-test-platform/

[8] http://openvswitch.org/pipermail/discuss/2013-November/011948.html
[9] http://web.engr.illinois.edu/~caesar/courses/CS498.S12/labs/lab7.pdf
[10]http://www.cs.tau.ac.il/~schiffli/sdn/ryu_docs/
[11]http://flowgrammable.org/sdn/openflow/message-layer/#tab_ofp_1_3_3
[12]https://github.com/mininet/mininet/blob/master/INSTALL
[13]For openflow13 patch:
https://groups.google.com/a/openflowhub.org/forum/#!topic/floodlight-dev/Z5l7-qgSuVc

Code:

The controller – micdekload.py

```
import math
import thread
import time
from random import random

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import set_ev_cls, CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.ofp_event import EventOFPFlowStatsReply
from ryu.lib.packet import ethernet, packet
from ryu.ofproto import ofproto_v1_3, ether

import confighelper
import loadbalancerconfig


class MicDekLoad(app_manager.RyuApp):
    """
    Rule-based Load Balancer for OpenFlow.
    Based on multiple articles by Liron Schiff et al, Tel-Aviv University.
    Written by Michal Shagam and Dekel Auster.
    """
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        super(MicDekLoad, self).__init__(*args, **kwargs)
        self.mac_to_port = {} # Used by the learning switch, for packets when not handled by the LB.
        self.numberOfRules = 0
        self.changedRules = 0
        self.virtual_mac = loadbalancerconfig.virtual_server[0] # Virtual MAC Address of the LB.
        self.virtual_ip = loadbalancerconfig.virtual_server[1] # Virtual IP of the LB.
```

```
    self.servers = loadbalancerconfig.servers # List of tuples: (MAC, IP, weight) of the servers.
    self.number_of_servers = len(loadbalancerconfig.servers) # Number of servers.
    self.num_packets = [0] * self.number_of_servers # Number of packets processed by each server.
    self.ranges = [] # Range separation of the servers.


  def create_first_ranges(self):
    """
    Returns a preliminary range separation of the servers.
    """
    ranges = []
    start = 0
    end = 0

    # Create (start, end) tuple for each server.
    for i in xrange(self.number_of_servers):
      weight = self.servers[i][2]
      end = start + int(math.floor((2**32 - 1) * weight))
      ranges += [(start, end)]
      start = end + 1
    # Fix rounding errors, by making sure that the last server's last IP is 255.255.255.255.
    ranges[self.number_of_servers - 1] = (ranges[self.number_of_servers - 1][0], 2**32-1)
    return ranges


  def create_comparator_metas(self):
    """
    Creates metadata information and masks for the comparator tables (m<=end and m>start).
    """
    # This holds tuples of: metadata to match, a string representation of IP to match,
    # and the mask for this match ("000...1...000"). The last parameter is a boolean, whether the
    # metadata in this index should be bigger or smaller than the IP.
    metas = []

    for i in xrange(32):
      #          METADATA                IP                  MASK                M>=P
      metas += [(int("0"*i + "1" + "0"*(32-i-1),2), "0"*i + "0" + "0"*(32-i-1), "0"*i + "1" + "0"*(32-i-1),
True)]
      metas += [(int("0"*i + "0" + "0"*(32-i-1),2), "0"*i + "1" + "0"*(32-i-1), "0"*i + "1" + "0"*(32-i-1),
False)]
    # Wildcard rule.
    metas += [(int("0"*32,2), "0"*32, "0"*32, True)]
```

```
        return metas

    def create_rule_set(self, datapath):
        """
        Creates all rules for switches that should support the load balancing feature.

        The tables are as follows:
        0 - basic operations on requests to/from the LB.
        1, 3 - add metadata to the packet, and forward to comparator tables.
        2, 4 - comparators. receive the metadata from tables 1 and 3 (respectively), and if the
             compare is successful, forwards to the given range.
        5 - changes destination IP and MAC to the requested server, according to the given range ID.
        """
        self.build_table_0(datapath)
        self.build_table_1_3(datapath)
        self.build_table_2_4(datapath)
        self.build_table_5(datapath)
        print "The number of rules is: %d" %(self.numberOfRules, )


    def build_table_0(self, datapath):
        """
        Builds table 0, as defined in create_rule_set doc.
        """
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # Create first rule - if the dest is the LB, move forward.
        self.add_flow_to_table(datapath, 1, parser.OFPMatch(eth_type=ether.ETH_TYPE_IP,
ipv4_dst=self.virtual_ip), [], [parser.OFPInstructionGotoTable(1)], 0)
        self.numberOfRules +=1
        # Create rule to change src for each server to the LB IP and MAC.
        for i in xrange(self.number_of_servers):
            actions = [parser.OFPActionSetField(eth_src=self.virtual_mac),
                    parser.OFPActionSetField(ipv4_src=self.virtual_ip),
                    parser.OFPActionOutput(ofproto.OFPP_NORMAL, )]
            self.add_flow_to_table(datapath, 2, parser.OFPMatch(eth_type = ether.ETH_TYPE_IP,
ipv4_src=self.servers[i][1]), actions, [], 0)
            self.numberOfRules +=1


    def build_table_1_3(self, datapath):
```

```
    """
    Builds tables 1 and 3, as defined in create_rule_set doc.

    This creates first-level separation of each server, and adds rules so that
    each candidate is forwarded to comparators 2 and 4 (respectively). By adding
    metadata, we allow the comparators to decide if the packet should be forwarded to
    the destination that we determine here by the range.
    """
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser


    self.ranges = self.create_first_ranges()


    # Add one rule (of the LCP+1) for each server, for tables 1 and 3.

    for i in xrange(self.number_of_servers):

        self.add_rules_to_1_3(datapath, self.ranges[i][0], self.ranges[i][1], i)

        self.numberOfRules +=2


    # If no match was found in table 1, move to table 3.

    self.add_flow_to_table(datapath, 0, parser.OFPMatch(), [], [parser.OFPInstructionGotoTable(3)], 1)

    self.numberOfRules +=1


def build_table_2_4(self, datapath):
    """
    Builds tables 2 and 4, as defined in create_rule_set doc.
    Table 2 compares m >= p. Table 4 compares m < p. If successful, forward to table 5.
    """
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    metas = self.create_comparator_metas()

    comparator_priority = 65
```

```
        for meta in metas:
            if meta[3]:
                # m >= p - in table 2, found a match. in table 4, discard
                instructions2 = [parser.OFPInstructionGotoTable(5)]
                instructions4 = [] # This should never happen! We missed a rule.
                if meta == metas[-1]: # If in the wildcard meta, send also from table 4.
                    instructions4 = instructions2
            else:
                # m < p - in table 2, no match, move to table 3. in table 4, found a match.
                instructions2 = [parser.OFPInstructionGotoTable(3)]
                instructions4 = [parser.OFPInstructionGotoTable(5)]

            # Create a string representation of the source IP.
            vals = [ str(int(meta[1][i*8:i*8+8], 2)) for i in xrange(4)]
            src_ipv4 = ".".join(vals)
            # Create the masks for the metadata (as a number) and the IP (as a string).
            vals = [ str(int(meta[2][i*8:i*8+8], 2)) for i in xrange(4)]
            meta_mask = int(meta[2], 2)
            src_mask = ".".join(vals)

            # Add rule to table 2.
            self.add_flow_to_table(datapath, comparator_priority, parser.OFPMatch(metadata=(meta[0],
meta_mask), eth_type=ether.ETH_TYPE_IP, ipv4_src=(src_ipv4, src_mask)), [], instructions2, 2)
            # Add rule to table 4.
            self.add_flow_to_table(datapath, comparator_priority, parser.OFPMatch(metadata=(meta[0],
meta_mask), eth_type=ether.ETH_TYPE_IP, ipv4_src=(src_ipv4, src_mask)), [], instructions4, 4)
            comparator_priority-=1
            self.numberOfRules +=2


    def build_table_5(self, datapath):
        """
        Builds table 5, as defined in create_rule_set doc.
        """
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        rid_meta_mask = (2**32-1)*(2**32) # only the RID part of the metadata - the most significant 32
bits.

        for i in xrange(self.number_of_servers):
            actions = [parser.OFPActionSetField(eth_dst=self.servers[i][0]),
                       parser.OFPActionSetField(ipv4_dst=self.servers[i][1]),
```

```
                parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
        self.add_flow_to_table(datapath, 0, parser.OFPMatch(eth_type=ether.ETH_TYPE_IP,
metadata=(i*(2**32), rid_meta_mask)), actions, [], 5)
        self.numberOfRules +=1


    def convert_lcp_to_ipv4(self, elcp, pref_length):
        """
        Converts the given ELCP and prefix length into a representation that is acceptable as src_ipv4.
        """
        # Create string representation of the ELCP.
        vals = [str(int(elcp[i * 8:i * 8 + 8], 2)) for i in xrange(4)]
        ipv4 = ".".join(vals)
        # Create string representation of its prefix mask.
        mask = "1" * pref_length + "0" * (32 - pref_length)
        maskvals = [str(int(mask[i * 8:i * 8 + 8], 2)) for i in xrange(4)]
        mask_str = ".".join(maskvals)
        return (ipv4, mask_str)


    def add_rules_to_1_3(self, datapath, lower, upper, range_id):
        """
        Add rules to tables 1 and 3 for the given range ID and its limits.
        """
        (lcp, elcp0, elcp1, priority) = self.find_lcp(lower, upper)

        # Table 1 uses the upper bound and ELCP1.
        self.add_rules_to_1_3_for_table(datapath, elcp1, upper, priority, range_id, 1)
        # Table 3 uses the lower bound and ELCP0.
        self.add_rules_to_1_3_for_table(datapath, elcp0, lower, priority, range_id, 3)


    def add_rules_to_1_3_for_table(self, datapath, elcp, range_limit, priority, range_id, table_id):
        """
        Helper method for add_rules_to_1_3, for each table.
        """
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        (ip, wildcard_mask) = self.convert_lcp_to_ipv4(elcp, priority + 1)
        match = parser.OFPMatch(eth_type=ether.ETH_TYPE_IP, ipv4_src=(ip, wildcard_mask))
        meta = range_id * (2**32) + range_limit
```

```
        meta_mask = 2**64 - 1 # Write the entire metadata. The 32 MSBs are for the RID, the others for the comparators.
        write_meta = parser.OFPInstructionWriteMetadata(meta, meta_mask)
        self.add_flow_to_table(datapath, priority, match, [], [write_meta, parser.OFPInstructionGotoTable(table_id + 1)], table_id)


    def rewrite_rules_in_1_3(self, datapath, old_lower, old_upper, new_lower, new_upper, range_id):
        """
        Rewrite rules in tables 1 and 3, when the lower and upper values of a range have changed.
        """
        (old_lcp, old_elcp0, old_elcp1, old_priority) = self.find_lcp(old_lower, old_upper)
        (new_lcp, new_elcp0, new_elcp1, new_priority) = self.find_lcp(new_lower, new_upper)

        # Rewrite rules in table 1, using ELCP1 and upper bound.
        self.rewrite_rules_in_1_3_for_table(datapath, old_elcp1, new_elcp1, old_upper, new_upper, old_priority, new_priority, range_id, 1)
        # Rewrite rules in table 3, using ELCP0 and lower bound.
        self.rewrite_rules_in_1_3_for_table(datapath, old_elcp0, new_elcp0, old_lower, new_lower, old_priority, new_priority, range_id, 3)
        self.changedRules+=2

    def rewrite_rules_in_1_3_for_table(self, datapath, old_elcp, new_elcp, old_range, new_range, old_priority, new_priority, range_id, table_id):
        """
        Helper method for rewrite_rules_in_1_3, for each table.
        """
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        meta_mask = 2**64 - 1 # Write the entire metadata. The 32 MSBs are for the RID, the others for the comparators.

        (old_ip, old_wildcard_mask) = self.convert_lcp_to_ipv4(old_elcp, old_priority + 1)
        (new_ip, new_wildcard_mask) = self.convert_lcp_to_ipv4(new_elcp, new_priority + 1)
        old_match = parser.OFPMatch(eth_type=ether.ETH_TYPE_IP, ipv4_src=(old_ip, old_wildcard_mask))
        new_match = parser.OFPMatch(eth_type=ether.ETH_TYPE_IP, ipv4_src=(new_ip, new_wildcard_mask))
        old_meta = range_id * (2**32) + old_range
        new_meta = range_id * (2**32) + new_range
        old_write_meta = parser.OFPInstructionWriteMetadata(old_meta, meta_mask)
        new_write_meta = parser.OFPInstructionWriteMetadata(new_meta, meta_mask)
```

```
        # If any part of the rule is different, replace the old rule by a new one.
        if old_ip != new_ip or old_wildcard_mask != new_wildcard_mask or old_meta != new_meta or
old_priority != new_priority:
            # We first delete the old rule, since otherwise, we cannot tell which rule will be
            # deleted since not all field are matched.
            self.delete_flow_from_table(datapath, old_priority, old_match, [], [old_write_meta,
parser.OFPInstructionGotoTable(table_id + 1)], table_id)
            self.add_flow_to_table(datapath, new_priority, new_match, [], [new_write_meta,
parser.OFPInstructionGotoTable(table_id + 1)], table_id)


    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        """
        Event that is dispatched in the beginning of the run of the controller.
        For all switches except 1 (the inner switch of the servers), add all rules.
        Also start monitoring statistics, for re-partitioning of the ranges.
        """
        datapath = ev.msg.datapath
        # Do the LB process except in the inner switch of the LB (it should know its real servers!).
        if ev.msg.datapath_id != 1:
          self.create_rule_set(datapath)
          thread.start_new_thread(self.send_flow_stats_request, (datapath,))

    def add_flow_to_table(self, datapath, priority, match, actions, instructions, table):
        """
        Adds a flow with the given parameters to the table.
        """
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = instructions

        # Add an apply_actions instruction for the actions.
        if actions != []:
            inst += [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                     actions)]

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                            match=match, instructions=inst, table_id=table)
        datapath.send_msg(mod)
```

```
    def delete_flow_from_table(self, datapath, priority, match, actions, instructions, table):
        """
        Deletes a flow with the given parameter from the table, strictly.
        """
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        inst = instructions

        # Add an apply_actions instruction for the actions.
        if actions != []:
            inst += [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                                  actions)]

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority, out_port=ofproto.OFPP_ANY,
out_group=ofproto.OFPG_ANY, command=ofproto.OFPFC_DELETE_STRICT, match=match,
instructions=inst, table_id=table)

        datapath.send_msg(mod)


    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        """
        This function (taken from simple_switch) handles all packets, except the ones directed
        to the load balancer. Also, after the load balances does its job, the packets arrive here
        to get to the actual servers.
        """
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]

        dst = eth.dst
        src = eth.src

        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})
```

```
    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port
    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [datapath.ofproto_parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow_to_table(datapath, 1, match, actions, [], 0)

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)


def find_lcp(self, lower, upper):
    """
    Finds the LCP (longest common prefix) of the given IP ranges,
    in binary format.
    """
    lower_bin = bin(lower)[2:]
    upper_bin = bin(upper)[2:]
    prefix_length = 0
    lcp = ''
    elcp0 = ''
    elcp1 = ''

    # Pad to 32 characters.
    if (len(lower_bin) < 32):
        lower_bin = '0' * (32 - len(lower_bin)) + lower_bin
    if (len(upper_bin) < 32):
        upper_bin = '0' * (32 - len(upper_bin)) + upper_bin

    # If they are the same, no requests can be retrieved to the server.
```

```
        if (lower_bin.startswith(upper_bin)): # they have the same length
            return ('0'*32, '0'*32, '0'*32, 0)

        # Calculate LCP.
        for i in xrange(32):
            if (upper_bin[i] == lower_bin[i]):
                lcp = lcp + upper_bin[i]
                prefix_length += 1
            else:
                # Found a different bit - return.
                elcp0 = lcp + '0'
                elcp1 = lcp + '1'
                num_wild = 32 - len(elcp0)
                elcp0 += '0' * num_wild
                elcp1 += '0' * num_wild

                return (lcp, elcp0, elcp1, prefix_length)


    def send_flow_stats_request(self, datapath):
        """
        Send stat requests from a different thread, and handle its result - infinitely.
        """
        ofp = datapath.ofproto
        ofp_parser = datapath.ofproto_parser
        last_packets = [0] * self.number_of_servers # Keep the last packet results, because we compare
deltas.
        self.skip_partition = True # Whether to skip re-partitioning in the next run.

        while True:
            # Send request to get connection stats.
            req = ofp_parser.OFPFlowStatsRequest(datapath)
            datapath.send_msg(req)
            self.received_reply = False

            # Wait until the results are all done.
            while self.received_reply == False:
                time.sleep(1)

            # Read results, fix the partitioning accordingly.
            # If the difference between the busiest and most relieved server are big, re-partition.
            min_delta = 0
```

```
        min_delta_id = 0
        max_delta = 0
        max_delta_id = 0
        # Find the busiest and most relieved servers.
        for i in xrange(self.number_of_servers):
            # Calculate a weighted delta.
            delta = (self.num_packets[i] - last_packets[i]) * self.servers[i][2]
            if i == 0 or min_delta > delta:
                min_delta = delta
                min_delta_id = i
                  if i == 0 or max_delta < delta:
                max_delta = delta
                max_delta_id = i
            last_packets[i] = self.num_packets[i]

        # Re-partition if needed.
        if not self.skip_partition and min_delta * 2 < max_delta:
            self.repartition(datapath, min_delta_id, max_delta_id)
            # Skip partition, so that next time we will have the refreshed deltas of the
            # new rules.
            self.skip_partition = True

        else:
            self.skip_partition = False

        # Sleep until next iteration.
        time.sleep(3)

    @set_ev_cls(EventOFPFlowStatsReply, MAIN_DISPATCHER)
    def flow_stats_reply_handler(self, ev):
        """
        Handle stats reply. Set self.num_packets accordingly.
        """
        # Reset number of packets
        for i in range(self.number_of_servers):
            self.num_packets[i] = 0

        # Calculate number of packets that went through table 5, by the dest MAC address.
        for stat in ev.msg.body:
            # Only care about valid rules in table 5.
            if stat.table_id != 5 or len(stat.instructions) != 1 or len(stat.instructions[0].actions) != 3:
                continue
```

```
        # Convert MAC address to string (e.g. "0000000000").
        mac = str(stat.instructions[0].actions[0].field.value).encode('hex').lower()
        # Find the server with the matching MAC address.
        server = -1
        for i in xrange(self.number_of_servers):
          # Convert server's MAC to the same format.
          current_mac = self.servers[i][0].replace(":", "").lower()
          if current_mac == mac:
            server = i

        # If found, add the packet count to this server's counting.
        if server >= 0 and server < self.number_of_servers:
          self.num_packets[server] = self.num_packets[server] + stat.packet_count

      # Allow other thread to continue.
      self.received_reply = True

  def repartition(self, datapath, min_id, max_id):
      """
      Re-partition the ranges, in order to make the busiest server more relieved.
      The current logic is decreasing the range of the busiest server, without modifying the min. server.
      """
      percent_to_take = random() / 2 # How much % to take from the busiest server.
      total_max_size = self.ranges[max_id][1] - self.ranges[max_id][0] # Total size of the busiest server.
      # if we have two sides - how much to give to each side?
      if max_id != 0 and max_id != self.number_of_servers:
        up = random()
      # we only have 1 side
      elif max_id == 0:
        up = 1
      else:
        up = 0
      percent_up = percent_to_take * up
      percent_down = percent_to_take * (1 - up)
      amount_up = int(math.floor(total_max_size * percent_up)) # Amount to give to max_id+1
      amount_down = int(math.floor(total_max_size * percent_down)) # Amount to give to max_id-1

      for i in xrange(self.number_of_servers):
        old_range = self.ranges[i]
        # Increase top of max_id-1
        if i == max_id - 1:
          self.ranges[i] = (old_range[0], old_range[1] + amount_down)
```

```
        # Increase bottom of max_id and decrease its top.
        elif i == max_id:
            self.ranges[i] = (old_range[0] + amount_down, old_range[1] - amount_up)
        # Decrease bottom of max_id - 1.
        elif i == max_id + 1:
            self.ranges[i] = (old_range[0] - amount_up, old_range[1])

        self.rewrite_rules_in_1_3(datapath, old_range[0], old_range[1], self.ranges[i][0],
self.ranges[i][1], i)
      print "The number of updated rules is: %d" %(self.changedRules, )
```

Custom Topology – loadtopo.py

```
#!/usr/bin/python

# usage: mn --custom <path to loadtopo.py> --topo load ...

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.cli import CLI
import os

class LoadTopo(Topo):
    def __init__(self, res_hosts = 4, out_hosts = 3, **opts):
            Topo.__init__(self, **opts)
            self.switchNum = 2
            sws = [self.addSwitch('s%d' %(i+1, )) for i in xrange(self.switchNum)]
            rhosts = [self.addHost('h%d' % (i + 1)) for i in xrange(res_hosts)]
            ohosts = [self.addHost('h%d' % (i + 1+res_hosts)) for i in xrange(out_hosts)]
            for i in xrange(res_hosts):
                    self.addLink(sws[0], rhosts[i])

            for sw in xrange(1, self.switchNum):
                    for i in xrange(out_hosts):
                            self.addLink(sws[sw], ohosts[i])

            for sw in xrange(1, self.switchNum):
                    self.addLink(sws[sw], sws[0])

topos = { 'load': LoadTopo }
```

Configuration – loadbalancerconfig.py

```
# Add a list of servers, each represented by (MAC, IP, weight).
# The weights should sum up to 1.0.
servers = [
   ("00:00:00:00:00:01", "10.0.0.1", 0.33),
   ("00:00:00:00:00:02", "10.0.0.2", 0.33),
   ("00:00:00:00:00:03", "10.0.0.3", 0.34)
   ]

#For testing, we can use the utility functions in confighelper

#import confighelper
#servers = confighelper.create_even_weight_servers(50)

# The virtual MAC and IP for the load balancer.
# Due to some limitations in this exercise, this must represent
# a real host in the same switch as the servers. This host will be
# disconnected from the outer world.
virtual_server = ("00:00:00:00:00:4", "10.0.0.4")
```

Configuration Utilities – confighelper.py

```
import random

def create_even_weights(num_hosts):
   """
   Creates host list with even weights
   """
   even = [1.0/num_hosts for i in xrange(num_hosts)]
   return even

def create_even_weight_servers(num_hosts):
   """
   Creates host list with even weights - under 255 hosts
   --mac option is assumed to have been used
   """
   even = [("00:00:00:00:00:%0.2x" % (i+1, ), "10.0.0.%d" % (i+1, ), 1.0/num_hosts) for i in
xrange(num_hosts)]
   return even
```

```
def create_random_weights(num_hosts):
    """
    Creates host list with random weights
    """
    sum_weights = 1.0
    #random returns [0.0, 1.0) so there won't be zero weights
    weights = []
    for i in xrange(num_hosts-1):
        r = random.random()
        if r < sum_weights:
            weights += [r]
            sum_weights -= r
        else:
            weights += [r * sum_weights]
            sum_weights -= r * sum_weights
    weights += [sum_weights]
    return weights

def create_random_weight_servers(num_hosts):
    """
    Creates host list with random weights - under 255 hosts
    --mac option is assumed to have been used
    """
    rand_w = create_random_weights(num_hosts)
    servers = [("00:00:00:00:00:%0.2x" % (i+1, ), "10.0.0.%d" % (i+1, ), rand_w[i] for i in
xrange(num_hosts)]
    return servers
```