**NETWORK PROGRAMMING AND APPLICATION (CMPE 207)**

**Project Report on**

**FLOODLIGHT SDN CONTROLLER**

**PART-I**

**ANALYSIS OF LOAD BALANCER MODULE**

**PART-II**

**IMPLEMENTATION OF RANDOM ALGORITHM AND STATISTICS DISPLAY IN LOAD BALANCER**

**Sneha Shridhar**

**SID:009463636**

**Submitted to**

**Dr. Younghee Park, Ph.D.**

**(Professor, Computer Engineering Department)**

**Department of Computer Engineering**

**San Jose State University**

**December 7th, 2015**

**Table of Contents**

|  |  |  |  |
| --- | --- | --- | --- |
| **Part-I Analysis of Floodlight Load Balancer module** | | | |
| 1 | Introduction to Floodlight | | 3 |
| 2 | Dependencies / List of Interfaces with Load Balancer Module | | 4 |
| 3 | Important APIs in Load Balancer | | 6 |
|  | 3.1 | Initialization APIs | 6 |
|  | 3.2 | APIs Used for Packet Flow | 7 |
| 4 | Scenario: How a ping works | | 12 |
| **Part-II Implementation of Random Algorithm and Statistics Display in Load Balancer** | | | |
| 5 | Load Balancer working mechanism | | 14 |
|  | 5.1 | Objective | 14 |
|  | 5.2 | Application Development Process | 14 |
|  | 5.3 | Packet Flow in case of a VIP address | 16 |
| 6 | Project Part-II Implementation | | 18 |
|  | 6.1 | Implementation of Random Algorithm in Load Balancer | 18 |
|  | 6.2 | Implementation of Load Balancer Statistics Display | 19 |
|  | 6.3 | Demonstration | 21 |
| 7 | Problems faced during the project and the how we overcame | | 23 |
| 8 | Conclusion | | 23 |
| 9 | References | | 23 |

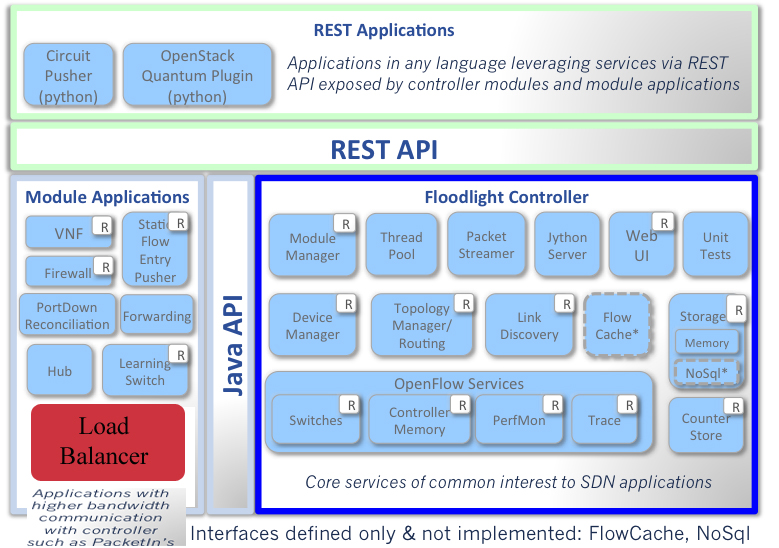
**PART I**

**FLOODLIGHT – ANALYSIS OF LOAD BALANCER MODULE**

**Chapter 1**

**Introduction to Floodlight**

The Floodlight Open SDN Controller is an enterprise-class, Apache-licensed, Java-based OpenFlow Controller. It is supported by a community of developers including a number of engineers from Big Switch Networks.

****

**Architecture of Floodlight SDN Controller**

# [Module Descriptions](https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Module+Descriptions+and+Javadoc)

Floodlight adopts a modular architecture to implement its controller features and some applications. **Module Loading System** describes the Java **IFloodlightModule** interface that realizes this framework.

Functionally, Floodlight consists of [**controller modules**](https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Controller+Modules) that implement core network services a software defined network would expose to applications, and [**application modules**](https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Module+Applications) that implement solutions for different purposes.

**Chapter 2**

**Dependencies / List of Interfaces with Load Balancer Module**

1.IRestApiService

The REST API Server allows modules to expose REST APIs over HTTP. Modules along with load balancer that have the REST server as a dependency expose APIs through adding a class that implements RestletRoutable. Each RestletRoutable contains a Router that attaches Restlet resources. Users will attach their own classes that extend a Restlet resource in order to handle requests for specific URLs. Inside the resource annotations such as @Get, @Put, etc are to select which method will be used for HTTP requests.

2.IDebugCounterService, IDebugCounter , IDeviceService :- Debug Module

The Device Manager learns about devices through PacketIn requests. It takes information from the PacketIn and classifies the device according to how the entity classifier is setup. By default the entity classifier uses MAC address and VLAN to identify a device. These two properties will define what is unique as a device. The Device Manager will learn about other properties such as IP addresses as well. One important piece of information is the device attachment points. If a PacketIn is received on a switch an attachment point will be created for that device. A device can have as many as one attachment point per OpenFlow island, where an island is defined as a strongly connected set of OpenFlow switches talking to the same Floodlight controller. The Device Manager will also age out attachment points, IPs, and devices themselves. Last seen timestamps are used to keep control of the aging process.

­­3.IRoutingService, ITopologyService : - TOPOLOGY MODULE

The Topology Service computes topologies based on link information it learns from the ILinkDiscoveryService. An important concept that the TopologyService keeps is the idea of an OpenFlow 'island'. An island is defined as a group of strongly connected OpenFlow switches under the same instance of Floodlight. Islands can be interconnected using non-OpenFlow switches on the same layer 2 domain.

All the information about the current topology is stored in an immutable data structure called the topology instance. If there is any change in the topology, a new instance is created and the topology changed notification message is called. If other modules want to listen for changes in topology they can implement the ITopologyListener interface.

4.IStaticFlowEntryPusherService

OpenFlow supports two methods of flow insertion: proactive and reactive. Reactive flow insertion occurs when a packet reaches an OpenFlow switch without a matching flow. The packet is sent to the controller, which evaluates it, adds the appropriate flows, and lets the switch continue its forwarding. Alternatively, flows can be inserted proactively by the controller in switches before packets arrive. In the case of proactive flow insertion, the arriving packet will never be sent to the controller for evaluation since it matches the proactively-inserted flow.

Floodlight supports both mechanisms of flow insertion. The Static Flow Pusher is generally useful for proactive flow insertion.

By default, Floodlight loads the Forwarding module which does reactive flow pushing. If you would like to exclusively use static flows, you must remove Forwarding from the floodlight.properties file.

5.IOFSwitchService

OFSwitchManager is a Floodlight module designed to manage all the OpenFlow switches connected to the Floodlight controller. It can be used to get references to and interact with switches such as send OFMessages like OFFlowMods and OFPacketOuts. Modules can leverage the OFSwitchManager by requesting a reference to the IOFSwitchService.

The OFSwitchManager uses the Netty library to handle threading and connections to switches. Each OpenFlow message will be processed by a Netty thread and will execute all logic associated with with the message across all modules.

6.IFloodlightModule

The FloodlightProvider provides two main pieces of functionality. It handles the connections to switches and turns OpenFlow messages into events that other modules can listen for. The second big function that it provides is decides the order in which specific OpenFlow messages (i.e. PacketIn, FlowRemoved, PortStatus, etc) are dispatched to the modules that listen for the messages. Modules can then decide to allow the processing of the message to go onto the next listener or to stop processing the message.

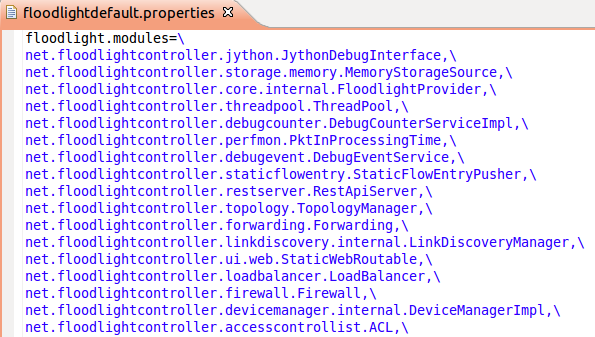
The FloodlightProvider handles module listener registrations and dispatches events to registered modules. Other modules can register for specific events like switches connecting or disconnecting and port status notifications. The FloodlightProvider will turn these wire protocol notifications into java based messages that other modules can handle. In order for modules to register for OpenFlow messages they must implement the IOFMessageListener interface.

**Chapter 3**

**Important APIs in Load Balancer**

3.1 Initialization APIs

When the floodlight controller starts, it loads the Main.java class. This class will check the floodlightdefault.properties files to check which all core modules and module applications exists in the controller and starts them one by one.



A load balancer object will be created and the below APIs which exists in all the core modules will be called.

1. getModuleServices()

Add Load balancer module to the list of modules in controller

1. getModuleServices()

Add Load balancer service to the list of services in controller.

1. init()

Create and initialize the variables required like the vips, pools, members which remain global. Add details in the object FloodlightModuleContext.

1. startup()

Start the listener to listen for PACKETIN messages

3.2 APIs Used for Packet Flow

High level flow chart for a packet flow in Load Balancer module is shown below.

If Multicast or Broadcast

No

Yes

IF VIP

No

IF VIP

No

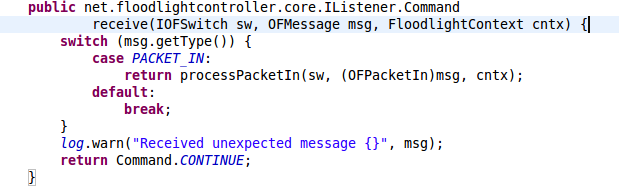
Forwarding Module

Yes

Yes

We can observe from the flowchart that all the packets in the controller will first come to the Load Balancer module. If Virtual IP address is not used as destination address, the packet is send to the forwarding module. Only PACKET\_IN messages sent to a Virtual IP address can be handled in Load Balancer module.

1. receive()



Inputs:

🡪IOFSwitch: Contains the details of the switch from which request is sent.

🡪OFMessage: Contains details like version of OF.

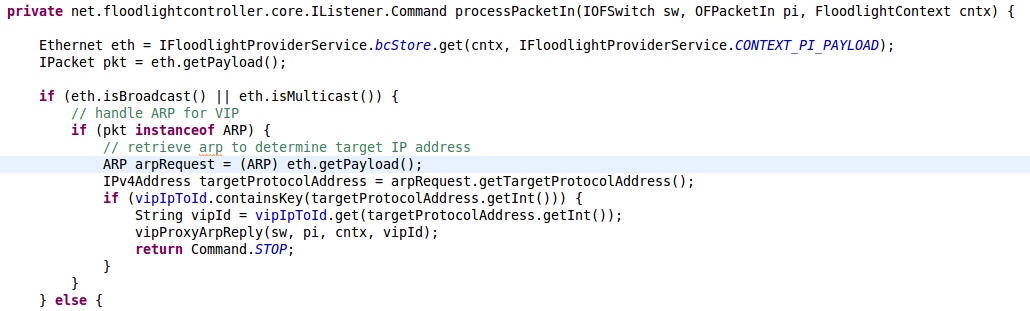
🡪FloodlightContext: Object where listeners can register and retrieve events.

Output:

🡪Command: Determines whether flow is to be stopped or transferred to forwarding module.

This is an entry point for all packets to the Load Balancer module. Only PACKET\_IN messages will be handled by Load balancer. If it is a PACKET\_IN message, it is send to processPacketIn() API.

1. processPacketIn()



Inputs:

🡪IOFSwitch: Contains the details of the switch from which request is sent.

🡪OFPacketIn: Contains details like version of OF and the packet details.

🡪FloodlightContext: Object where listeners can register and retrieve events.

Output:

🡪Command: Determines whether flow is to be stopped or transferred to forwarding module.

This API determines what needs to be done with the packet. First it checks whether the packet is for a broadcast or multicast. If it is broadcast or multicast packet it will be using ARP. Check if the ARP packet uses Virtual IP using getTargetProtocolAddress(), and then vipProxyArpReply() API will be called if it is a VIP.

If it is not broadcast or multicast packet, we need to check if it is a IPv4 packet. Again we will check if uses a VIP address. If it is VIP then we need to find the source address and the protocol used. TCP, UDP and ICMP packets can be handled by the Load Balancer. Then, based on the protocol used, we need to find the source port and destination port. We also need to find the pool and member the VIP belongs to. Finally pushBidirectionalVipRoutes() API is called.

1. vipProxyArpReply()



Inputs:

🡪IOFSwitch: Contains the details of the switch from which request is sent.

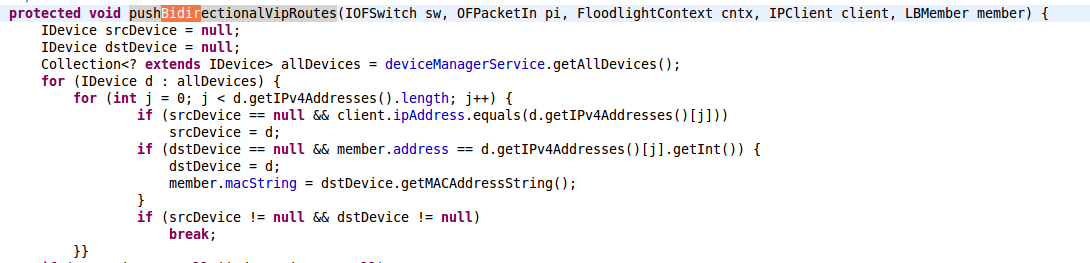
🡪OFPacketIn: Contains details like version of OF and the packet details.

🡪FloodlightContext: Object where listeners can register and retrieve events.

🡪 String: String which has the VIP address

This API will extract the ARP request from the payload using getPayload() function. This request is used to create an IP packet containing the ARP response. Then the pushPacket() API will be called.

1. pushBidirectionalVipRoutes()



Inputs:

🡪IOFSwitch: Contains the details of the switch from which request is sent.

🡪OFPacketIn: Contains details like version of OF and the packet details.

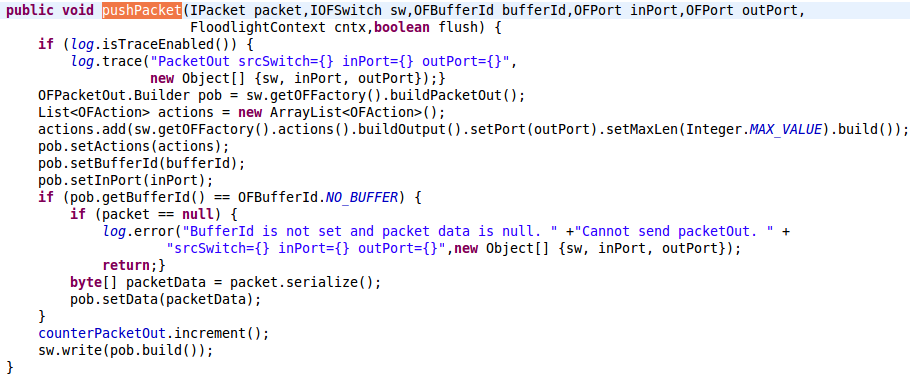
🡪FloodlightContext: Object where listeners can register and retrieve events.

🡪 IPClient: Object containing IP address, source port, Protocol and destination port.

🡪 LBMember: It contains the destination member details.

This API is used to find the route to be taken for sending the packet from source host to destination host. Source and destination device entities are retrieved. Validates that source and destination are not null and they are on same island. It makes sure that the source and destination not on same switchport, then it installs all the routes where both source and destination have attachment points(attachment points are recorded when we ping pair or pingall on the mininet).

1. pushpacket()



Inputs:

🡪IPacket: Object of the interface packet. Contains methods to get payload, checksum, etc

🡪IOFSwitch: Contains the details of the switch from which request is sent.

🡪OFBufferId: Contains details of buffer to be used for sending data.

🡪OFPort: Two OFports are used for source and destination.

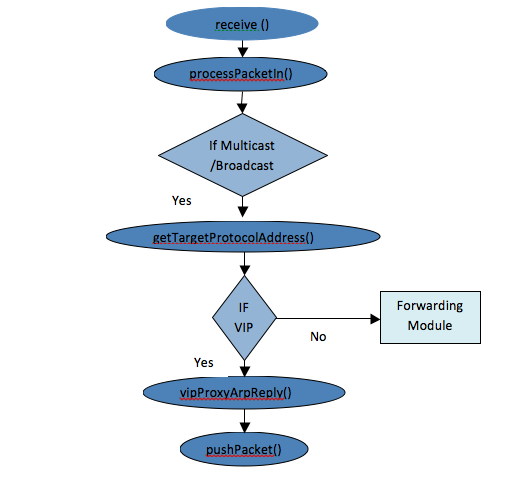
🡪FloodlightContext: Object where listeners can register and retrieve events.

This API is common to all the packets handles by Load Balancer irrespective of whether it is a broadcast, multicast or normal IPv4 messages. It creates a PACKET\_OUT message using the IPacket received and write sit to the buffer to send it.

**Chapter 4**

**Scenario: How a ping works.**

Suppose we just created a topology and want to ping one host from another. When two hosts communicate with each other for the first time, the destination MAC address is unknown. In this case, we need to use ARP to find the MAC address of the destination. Therefore, a ARP request message is created and send as a PACKET\_IN message to the Load Balancer receive() API.



The receive() API forwards the packet to processPAcket() API where it will satisfy the check for broadcast/multicast message. Again it will satisfy the check for ARP. The destination address is checked whether it is a VIP. If it is not a VIP it is send to forwarding module. Else it will be send to vipProxyArpReply() API. In the vipProxyArpReply() API, a ARP response message is created using the ARP request received. Then the packet is send from Load Balancer module by pushPacket() API.

The ARP request should reach the destination, which will generate an ARP reply. This will be sent as a packet-in to the controller, which will pass through the load balancer and go to forwarding, which will send the packet out to the destination and insert flows for all subsequent ARP packets. At this point, the source host should send an ICMP request to the destination. This will arrive to the controller as a packet-in message and be handed to the load balancer. Unless the destination IP is the VIP IP, the load balancer should hand the packet to forwarding for routing, which will insert flows for the ICMP packet from the source to the destination. The ICMP request packet will make it to the destination, and the destination will respond to the request with an ICMP reply, which will be handled the exact same way - just in reverse. If you wait some time, the ARP entries will expire and the ARP process will occur again, which is probably why you see the load balancer entering that section of code again.

**PART II**

**Implementation of Random Algorithm and Statistics Display in floodlight Load Balancer**

**Chapter 5**

**Load Balancer working mechanism**

5.1 Objective

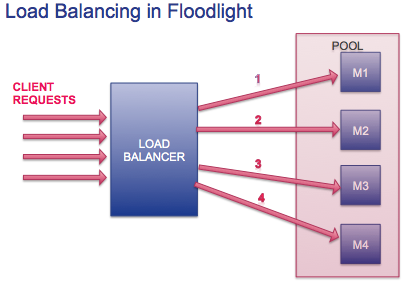
Currently there is only one policy that is Round Robin that is used to select a server from a pool of server for load balancing purposes. We have created and included another policy called Radom Detect.We have created an application that demonstrates how Load Balancing works using the policies defined in the Controller. We have also done a statistical analysis of the packets that are hitting the controller wherein we display the packet payload, the source and destination address to name a few.

5.2 Application development process

Before we go into the application, let’s understand how Load Balancing functions in Floodlight.

**5.2.1 How Load Balancing works in Floodlight**

* Load Balancer works using the principle of Virtual IP address (VIP) Pools that is already pre-defined and configured before a packet is sent.
* A Pool is created and all servers that participate in load balancing is added as members to the Pool.
* We assign a virtual IP address to the Pool which clients use as the destination address to communicate to the server.
* When the controller receives the request from the client, uses the load balancing algorithm present to select one of the available servers.



**5.2.2 What APIs are invoked in the process?**

Now, let’s look at the APIs that are called when we create the VIP Pools and its members. Floodlight uses REST as the interface mechanism to communicate to the controller. In our application, we use curl commands that send the HTTP messages to the Controller invoking REST.

**5.2.2.1 Creation of the VIP Pool address that clients use to talk to server**

The below curl command is used to create the VIP Pool where the address is specified in the address section.

*curl -X POST -d '{"id":"1","name":"vip1","protocol":"icmp","address":"10.0.0.100","port":"8"}'http://localhost:8080/quantum/v1.0/vips/*

The curl command invokes the createVip API present in VipsResource class to create the pool.

*@Put*

*@Post*

***public*** *LBVip createVip(String postData) {*

*LBVip vip=****null****;*

***try*** *{*

*vip=jsonToVip(postData);*

*}* ***catch*** *(IOException e) {*

*log.error("Could not parse JSON {}", e.getMessage());*

*}*

**5.2.2.1 Creation of the Pool**

The below curl command is used to create the Pool where the pool name is specified in the name section.

*curl -X POST -d '{"id":"1","name":"pool1","protocol":"icmp","vip\_id":"1"}' http://localhost:8080/quantum/v1.0/pools/*

The curl command invokes the createPool API present in PoolResource class to create the pool.

*@Put*

*@Post*

***public*** *LBPool createPool(String postData)*

*{*

*LBPool pool=****null****;*

***try*** *{*

*pool=jsonToPool(postData);*

*}* ***catch*** *(IOException e) {*

*log.error("Could not parse JSON {}", e.getMessage());*

*}*

**5.2.2.1 Add members to a Pool**

The below curl command is used to add members to a Pool where the members are specified in the address section.

*curl -X POST -d '{"id":"1","address":"10.0.0.3","port":"8","pool\_id":"1"}' http://localhost:8080/quantum/v1.0/members/*

*curl -X POST -d '{"id":"2","address":"10.0.0.4","port":"8","pool\_id":"1"}' http://localhost:8080/quantum/v1.0/members/*

The curl command invokes the createMember API present in MemberResource class to add the servers.

*@Put*

*@Post*

***public*** *LBMember createMember(String postData)*

*{*

*LBMember member=****null****;*

***try*** *{*

*member=jsonToMember(postData);*

*}* ***catch*** *(IOException e) {*

*log.error("Could not parse JSON {}", e.getMessage());*

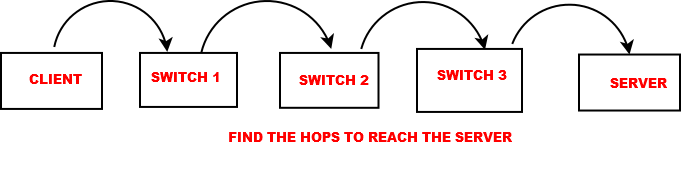
*}*

5.3 Packet Flow in case of a VIP address

Previously, we have only seen what happens when a client send a packet to an address other than the VIP address. In our application, the clients use the VIP address and let’s look into the APIs that are called when a client sends a packet with destination address as the VIP address. There are two main APIs in the controller that handles request:

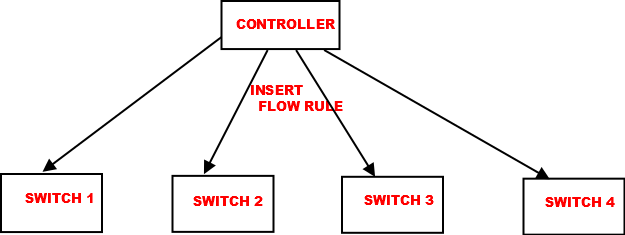
**5.3.1 pushBidirectionalVipRoutes():**

* Once the server that will handle the client’s request is identified, the controller now needs to find the path the packets should take from the client to the server.
* This API essentially finds the switches the client need to traverse to successfully reach the server.
* The controller additionally checks if both the client and server are in the same domain or island.



**5.3.2 pushStaticVipRoute()**

* Now that the hops or the path has been identified, the Controller needs to inform the intermediate switches along the path of the same.
* The controller hence inserts flow rules to each and every switch along the path so the packet is properly routed to the server.



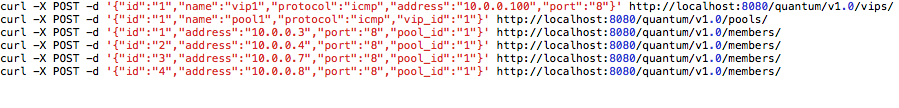
**Chapter 6**

**Project Part-II Implementation**

6.1 Implementation of Random Algorithm in Load Balancer

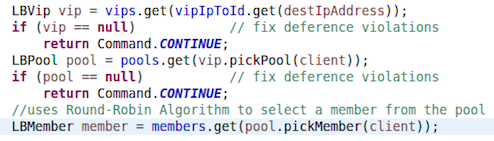
In the processPacketIn API in the LoadBalancer.java class, if the destination address is Virtual IP address, then the VIP id and its corresponding pool is checked and validated. The corresponding load balancer algorithm will then select the destination host from the pool members list according to the selected algorithm.

The routine to pick the member from the pool is written in LBPool.java class. The load balancer is triggered when you ping to the VIP address 10.0.0.100. The pool members are 10.0.0.3 , 10.0.0.4, 10.0.0.7 and 10.0.0.8 as shown below, are picked with their id’s 1, 2, 3, 4 respectively by the load balancer.

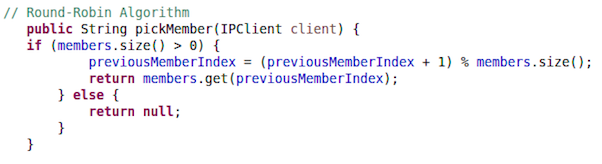


a. Round-Robin Algorithm (existing floodlight load balancing algorithm):

By default floodlight controller implements Round-Robin algorithm to select the destination host address from the pool using pool.pickMember(client) call as shown below.

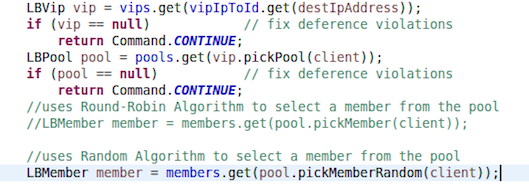
****

pickMember(client) API shown below is located in LBPool.java class, which chooses the member using member ID.It initially checkes If the pool has any member at all by ensuring member.size() is greater than 0. It adds 1 to the previousMemberIndex(which is nothing but previously selected member Id) and chooses next member Id.



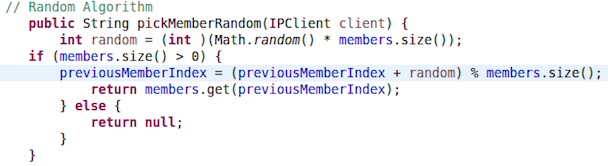
b. Random Algorithm ( newly implemented by us):

We have implemented a Random Algorithm for load balancing, the pool.pickMemberRandom(client) call will replace the pool.pickMember(client) in LoadBalancer.java class as shown below,which picks the members form pool randomly.



Round-Robin commented out

pickMemberRandom(client) API shown below is located in LBPool.java class, which chooses the member using member ID in a random fashion. Random number is generated using Math.random() and added to the previousMemberIndex(which is nothing but the member id of the previous host selected) and mod to the member.size() ( number of members in the pool). So now the pool members are picked randomly using random generator.

****

6.2 Implementation of Load Balancer Statistics Display

We have extracted the load balancer statistics and displayed it on webpage using html. We have included DisplayPktDetails() API as shown below which saves the information to be displayed in to the html file, every time the load balancer routes to particular destination host IP address.

This statistics display does not show regular ping details, it only shows the load balancer details when the VIP is pinged. The source host IP address, The VIP address pinged to, the destination host IP address selected by load balancer, the path taken by packet to reach from source host to destination host, the protocol name and the packet size are displayed as part of load balancer statistics.

Every time the controller starts the display details will be extracted in to the string builder named htmlBuilder and saved in the LoadBalancerStatistics.html file, and it continues to append every ping details that goes through load balancer.If you stop the controller and start again, the the new entry starts and the old html file is replaced with new one.



* + **ip\_pkt:** It is the Ethernet payload of incoming packet which is instance of IPv4 protocol. Source IP address can be printed by ip\_pkt.getSourceAddress() instance, the VIP address can be printed by ip\_pkt.getDestinationAddress() and the packet size can be printed using ip\_pkt.getTotalLength().
  + **member:** member argument contains the details of Load Balancer VIP pool members. It is of the type LBMember. Once load balancer picks the pool member, the destination ip addresss will be saved in the member argument. Hence the final destination host address where the packet will be routed can be printed using member.address, since we need IPv4 version of the address we use IPv4Address.of(member.address).
  + **protoName:** It is the string variable which has the details about the payload network protocol.The ip\_pkt is checked to see if it is instance of TCP,UDP or ICMP protocol, and saves the corresponding protocol name in the protoName argument.
  + **routeIn:** The routeIn information in the pushBidirectionalVipRoutes() API has the routing path details. The inbound route for all the clusters is retrieved , if source and destination clusters are same then the inbound route is copied in to routeIn using routingEngineService.getRoute() API. routeIn.getpath() has the path information of every node it traverces. The path is traversed using for loop and every node ID details are alternatively saved between “>>>” formatting string, to show the flow direction. path.get() has node ID and the correcponding port number saved one after the other,so the for loop variable i is incremented +2 to extract only node or the switch information.

6.3 Demonstration

a. Round-Robin Algorithm

First let us look how the floodlight load balancer chooses the member hosts from the pool using Round-Robin algorithm.

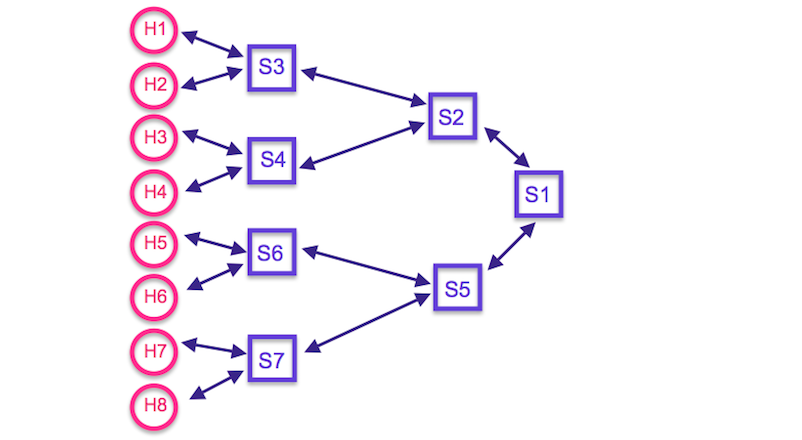
The floodlight and the mininet all installed on the same virtual machine.

The floodlight controller is started, and it listens on the port 6653.

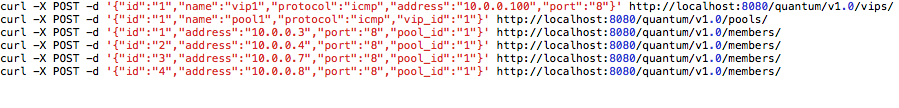
The mininet connects to the the floodlight controller using the same port number 6653 and creates the tree topology with deapth 3 by suing below command.

$ sudo mn --controller=remote, ip=127.0.0.1, port=6653 --topo=tree, 3 –switch ovsk, protocols=OpenFlow10

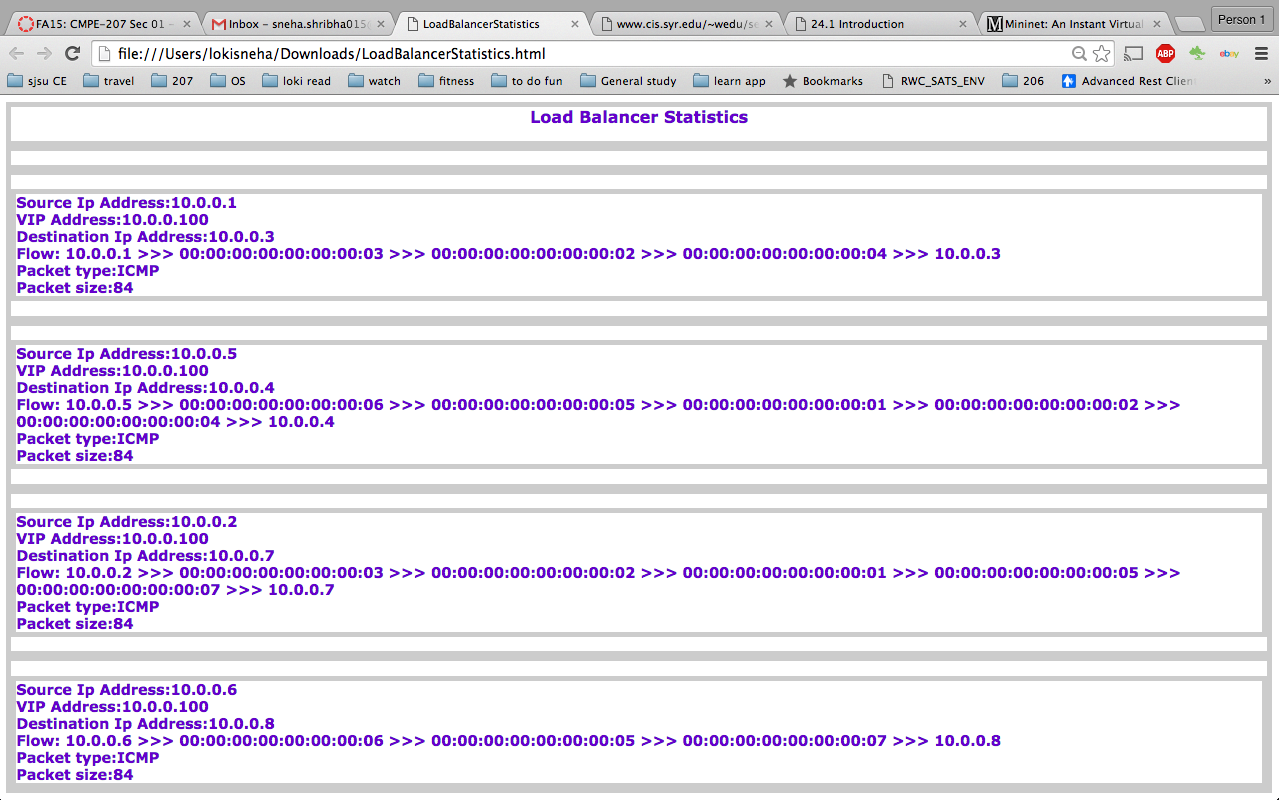
This creates the tree topology network with 8 hosts (h1,h2,h3,h4,h5,h6,h7,h8) and 7 switches(s1,s2,s3,s4,s5,s6,s7) as shown below. Pingall on mininet will establish the attachment points for all the hosts.



The test cases shown below are run, the VIP address is 10.0.0.100 and the pool members are 10.0.0.3 , 10.0.0.4, 10.0.0.7 and 10.0.0.8 with their id’s 1, 2, 3, 4 respectively.

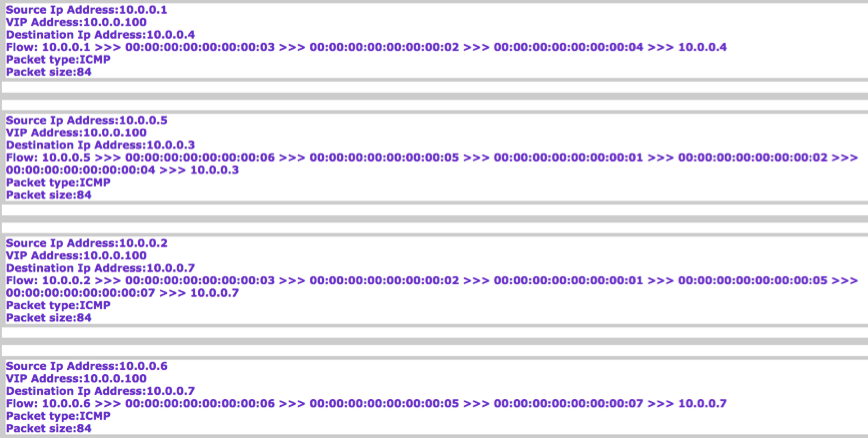


After running the test case, VIP is pinged from random hosts using mininet. We pinged to VIP address 10.0.0.100 from h1, h5, h2 and h6 one after the other. The load balancer chooses the destination hosts from the pool in the order shown in the test case in round-robin method. The load balancer statistics display shows the destination IP address and the path details as seen below.



b. Random Algorithm

The load balancer code is changed to random algorithm. Now the floodlight controller is run, mininet is connected to the controller, same topology is created, same test case is run, and the VIP address 10.0.0.100 is pinged using same set of hosts h1, h5, h2 and h6 one after the other.The load balancer statistics display shows the destination Ip address and the path details as seen below. The destination hosts are chosen randomly from pool as seen below h4,h3,h7,h7 are picked for the 4 pings.



**Chapter 7**

**Problems faced during the project and the how we overcame**

* Initally we started working on including IPv6 packets to the floodlight as the project currently supports only IPv4 packets.
* We could successfully include IPv6 handling code in the loadbalancer.java , compile and run it without any errors.
* But when tried with test cases, we had lots of error due to dependencies on all other classes in the project. To modify all the modules in the project happens to be very complex and not accomplishable in given span of time.
* Hence we shifted to implementing different load balancing algorithm and loadbalancer statistics display on an html page.
* We had difficulty in tracing the load balancer code in debugging mode, as we had timeout issues.
* We used lot of print console statements in all the methods, to check the
* status of arguments, payloads, how it is modified.

**Chapter 8**

**Conclusion**

* We understood what is SDN, how to analyze and debug a Floodlight SDN controller source code. We were also able to modify existing code, add new modules, and test successfully. All this improved our knowledge in networking concepts and increased programming skills.
* There are lots of functionalities that can be implemented in Floodlight SDN controller to extend its usability in production networks.
* Round-Robin policy among servers is based on connections, not traffic volume.
* Ability to load balance using weighted Round-Robin is something we would like to implement if we get opportunity in the future projects.

**Chapter 9**

**References**

* Check out the website:

<http://www.projectfloodlight.org/>

* Get the code:

<http://www.projectfloodlight.org/download/>

* Join the goggle group mailing list:

[https://groups.google.com/a/openflowhub.org/forum/#!forum/floodlight-dev](https://groups.google.com/a/openflowhub.org/forum/)