Cloud Server with OpenFlow: Load Balancing

Surya Prateek Surampalli
Information Technology Department
Southern Polytechnic State University
Marietta, GA, United States
ssurampa@spsu.edu

Ying Qian
Department of Computer Science & Technology
East China Normal University
Shanghai, China
yqian@cs.ecnu.edu.cn

***Abstract*—in high-traffic Internet today, it is often desirable to have multiple servers that represent a single logical destination server to share the load. A typical configuration comprises multiple servers behind a load balancer that would determine which server would serve the request of a client. Such equipment is expensive, has a rigid set of rules, and is a single point of failure. In this paper, I propose an idea and design for an alternative load-balancing architecture with the help of an OpenFlow switch connected to a NOX controller that gains political flexibility, less expensive, and has the potential to be more robust to failure with future generations of switches**

I. INTRODUCTION

In today's increasingly internet-based cloud services, a client sends a request to URL or a logical server and receives a response from a potentially multiple servers acts as a logical address server. Google server is said to be the best example, the request is sent to server farm as soon as the client resolves the IP address from the URL [1].

Load balancers are expensive that acts as a reverse proxy and distributes network or application traffic across a number of servers. Load balancers are used to increase capacity (concurrent users) and reliability of applications. They improve the overall performance of applications by decreasing the burden on servers associated with managing and maintaining application and network sessions, as well as by performing application-specific tasks [1]. Since load balancers are not basic equipment and run custom software, policies are rigid in their choices. Specific administrators are required and also the arbitrary policies are not possible to implement. Since running policy and the switch are connected it is reduced to a single point of failure [2].

The order of magnitude will cost less than a commercial load-balancer if architecture with an OpenFlow switch is implemented which is controlled by the commodity server and also provides flexibility for writing patterns which allow the controller to be applied arbitrary political [1].

If the next generation of OpenFlow switches has the capability of connecting to multiple controllers, there is a chance of making the system more robust to abortion by keeping the any server behind the switch which that acts as the controller [1].

II. BACKGROUND

A. Load Balancing

Load balancing helps make networks more efficient. It distributes the processing and traffic evenly across a network, making sure no single device is overwhelmed [1]. Web servers, as in the example above, often use load balancing to evenly split the traffic load among several different servers. This allows them to use the available bandwidth more effectively, and therefore provides faster access to the websites they host [3].

Whether load balancing is done on a local network or a large Web server, it requires hardware or software that divides incoming traffic among the available servers. Networks that receive high amounts of traffic may even have one or more servers dedicated to balancing the load among the other servers and devices in the network. These servers are often called (not surprisingly) load balancers [1].

Load balancing can be performed using dedicated hardware devices such as load balancers or having intelligent DNS servers. A DNS server can redirect traffic data centre with a heavy load or redirect requests made by customers for a data centre that is less network stretches from clients. Many data centres use of expensive hardware load balancing equipment that makes in distributing the network traffic across multiple machines to avoid congestion on a server.

A DNS server resolves a hostname to a single IP address where the client sends the request. To the outside world there is a logical address that resolves a host name [3]. This IP address is not associated with a single machine, but is the type of service that a client requests. DNS can resolve a host name to a load balancer within a data centre. But this could be avoided for safety reasons and to avoid attacks on the device. When a client request comes to the load balancer, the request is redirected according to the policy.

B. OpenFlow Switch

An OpenFlow switch is a software program or hardware device that forwards packets in a software-defined networking (SDN) environment. OpenFlow switches are either based on the OpenFlow protocol or compatible with it [1].

In a conventional switch, packet forwarding (the data plane) and high-level routing (the control plane) occur on the same device. In software-defined networking, the data plane is decoupled from the control plane. The data plane is still implemented in the switch itself but the control plane is implemented in software and a separate SDN controller makes high-level routing decisions. The switch and controller communicate by means of the OpenFlow protocol. The OpenFlow switch on the other hand uses an external controller called NOX to add rules into its flow table.

C. NOX Controller

NOX is a network control platform, which provides a high-level programmatic interface upon which network management and control applications can be built. In brevity, NOX is an OpenFlow controller [3]. Therefore, NOX applications mainly assert flow-level control of the network meaning that they determine how each flow is routed or not routed in the network.

The OpenFlow switch is connected to the NOX controller and communicates over a secure channel using the OpenFlow protocol. The current design of OpenFlow only allows one NOX controller per switch. The NOX controller decides how packets of a new flow should be handled by the switch. When new flows arrive at the switch, the packet gets redirected to the NOX controller which then decides whether the switch should drop the packet or forward it to a machine connected to the switch. The NOX controller can also delete or modify existing flow entries in the switch.

The NOX controller can execute modules that describe how a new flow should be handled. This provides us an interface to write C++ modules that dynamically add or delete routing rules into the switch and can use different policies for handling flows.

D. Flow Table

A flow table entry of an OpenFlow switch consists of a header fields, counters and actions. Each flow table entry stores Ethernet, IP and TCP/UDP header information. This information includes destination/source MAC and IP address and source/destination TCP/UDP port numbers. Each flow table entry also maintains a counter of number of packets, and bytes arrived per flow. A flow table entry can also have one or more action fields that describe how the switch will handle packets that match the flow entry. Some of the actions include sending the packet on all output ports, forwarding the packet on an output port of a particular machine and modifying packet headers (Ethernet, IP and TCP/UDP header). If a flow entry does not have any actions, then the switch drops all packets for the particular flow.

Each Flow entry also has an expiration time after which the flow entry is deleted from the flow table. This expiration time is based on the number of seconds a flow was idle and the total amount the time (in seconds) the flow entry has been in the flow table. The NOX controller can chose a flow entry to exist permanently in the flow table or can set timers which delete the flow entry when the timer expires.

III. Load-Balancer Design

Load balancing architecture comprises an OpenFlow switch with a control device of NOX and server machines connected to output ports of the switch server. The OpenFlow switch uses an interface to connect to the Internet. Each server has a static IP address and NOX controller maintains a list of currently connected to the OpenFlow switch servers. Each server is running web server emulation on a well known port.

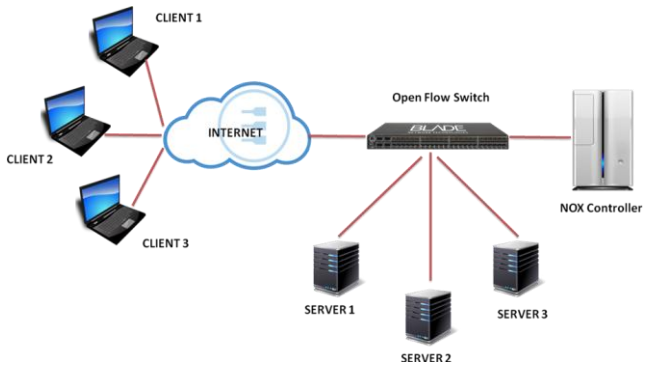

Figure1. Load-balancer architecture using OpenFlow switch and NOX controller

The hostname of server to IP address is resolved by each client and a request is sent to that IP address on the known port number. If you consider the above diagram, when a packet is reached to the switch from the client, the header information of the packet is compared with the entries of the flow table. If the header information of the packet corresponds to an inlet of the flow, the counter for the number of packets, the byte count is incremented, and the actions associated with the input of the flow are performed on the packet. If no match is found, the switch forwards the packet to NOX.

NOX decides how the packet for this flow should be handled by the switch. NOX and then inserts a new article in the cash flow of the switch using the OpenFlow protocol. To achieve load-balancing features, the modules should be written in C++ that is executed by NOX controller. NOX should perform the function of handle () when a new flow

arrives at the switch. This function sets the load balancing policy and adds new rules in the flow table of the switch.

All client requests should be destined for the same IP address, then whatever the module is executed by NOX, should add rules for each flow which can modify the destination MAC and IP address of the packet with a server's MAC and IP address. The switch will forward the packet to the server output port after modifying the packet header.

When servers return a packet to the client, the module adds an entry flow that changes the source IP address with the IP address of the host that the client sends its request. So the client should always receive packets from the same IP address. If the client connection / server connection is closed or remains idle for 10 seconds, then the inactivity timer expires causing the input stream to be deleted from the cash flow of the switch. This allows input stream recycling.

Servers wait for a NOX to register and then report their current load on some schedule similar to the Listener Pattern. NOX in a separate thread listening on a UDP socket for heartbeats with reported by server loads and maintains a table with the current loads of all servers. When applying for a new stream is received, it chooses the server with the lowest and the load current increases to the low current server. This prevents flow of all flows routed to the same server as the server reports a new load. It also breaks ties by turning it into a round robin until the servers report their actual load heartbeat.

Flow Algorithm

Require: Flow, path

```
1: sourceHost = LocateSource(flow);
2: destinationHost = LocateDestination(flow);
3: layer = setToplayer();
4: currentSwitch = LocateCurrentSwitch();
5: direction = 1; //upward
6: path = null; //list of switches
7: return search ();
```

This algorithm works as follows. When the OpenFlow controller receives a packet from a switch, it switches the control to the load balancer. Line 1 to 6 introduces the initialization for necessary variables. The load balancer firstly analyses the packet's match information including the input port on the switch that receives the packet as well as the packet's source address and destination address. Then it looks up those addresses using its knowledge about the network topology. Once the source and destination hosts are located, the load balancer calculates the top layer that the flow needs to access. We use the search direction flag. The flag has two values: 1 for upward and 0 for downward. It is initialized to 1. A path is created for saving a route grouped by a list of switches later. Line 7 calls search () that performs the search for paths recursively.

In the method search (), It firstly adds current switch into path. It returns the path if current search reaches the bottom layer. It reverses the search direction if current search reaches the top layer. Then it calls a method

```
1: search () {
2: path.add(curSwitch);
4: if isBottomLayer(curSwitch) then
5: return path;
6: end if
7: if curSwitch.getLayer ( ) == layer then
8:     direction = 0; //reverse
9: end if
10: links = findLinks(curSwitch, direction);
11: link = findWorstFitLink(links);
12: curSwitch = findNextSwitch(link);
13: return search ();
14:   }
```

that returns all links on current switch that are towards current search direction. Only one link is chosen by picking up the worst-fit link with maximum available bandwidth. And then the current switch object is updated. The method search () is called recursively layer by layer from the source to destination. At last the path will be return to the load balancer. The path information will be used for updating flow tables of those switches in the path.

Flow Scheduling

The Flow scheduling functionality works as follows. Each OpenFlow switch maintains its own flow table. Whenever any packet comes in, the switch checks the packet's match information with the entries in its flow table. The packet's match information includes ingressPort, etherType, srcMac, dstMac, vlanID, srcIP, dstIP, IP protocol, T CP/UDP srcPort, TCP/UDP dstPort. If it finds a match, it will send out the packet to the corresponding port. Otherwise it will encapsulate the packet in a PACKET IN message and send the message to the controller. As a module of the OpenFlow controller, the load balancer will handle the PACKET IN message. It finds a proper path by executing a search with the DLB algorithm described in Algorithm 1. The path is a list of switches from source to destination of the packet. Then the load balancer creates one FLOW MOD message for each switch in the path and sends it to the switch. This message will have the packet's match information as well as an output port number on that switch. The output port number is directly calculated by the path and network topology. If one switch receives a FLOW MOD message, it will use it to update its flow table accordingly. Those packets buffered on ports of that switch may find their matches in the updated flow table and be sent out. Otherwise the switch will repeat this process.

IV. Future Work

The OpenFlow specification includes an optional feature that would allow multiple NOXs to make active connections to the switch. In the case when the NOX is failing, another machine could resume the role of the NOX and continue routing traffic. Naturally the system would need to detect the failure, have a mechanism to remember any state associated with the current policy, and all servers would have to agree on who the new NOX was. These requirements naturally lend themselves to the Paxos consensus algorithm in which policy and leader elections can be held and preserved with provable progress [3]. We have implemented Paxos in another research project and could add it to our server implementation at the controller/signaller layer. As long as at least half of the nodes in the cluster stay up, state will be preserved and traffic should continue to flow.

V. Conclusion

It is possible to achieve similar functionality to a commercial load balancer switches using only physical commodities. The OpenFlow switch provides the flexibility to implement the arbitrary policy in software and politics separate the switch itself. Since the policy is decoupled from the switch, we can avoid the machine implementation of the policy of a single point of failure and provide a more robust system.

References

[1] OpenFlow Switch Specification. Version 0.8.9 (Wire Protocol 0x97). Current maintainer: Brandon Heller (brandonh@stanford.edu). December 2, 2008.

[2] Web caching and Zipf-like distributions: evidence and implications. Breslau, L. Pei Cao Li Fan Phillips, G. Shenker, S. Xerox Palo Alto Res. Center, CA. INFOCOM 1999.

[3] Paxos Made Simple. Leslie Lamport

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. ACM SIGCOMM, 2008.

[5] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. IEEE Transactions on Computers, 1985.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Datacenter Traffic Characteristics. SIGCOMM WREN workshop, 2009.

[7] HOPPS, C. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF, 2000.

[8] W. J. Dally and B. Towles. Principles and Practices of Interconnection Networks. Morgan Kaufmann Publisher, 2004.

[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. ACM IMC 2009.

[10] 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 CCR, 2008.

[11] R. N. Mysore, A. Pamporis, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric. ACM SIGCOMM, 2009.

[12] Beacon OpenFlow Controller https://OpenFlow.stanford.edu/display/Beacon/Home.

[13] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Definded Networks. ACM SIGCOMM, 2010.

[14] Y. Zhang, H. Kameda, S. L. Hung. Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems. Computers and Digital Techniques, IEE, 1997.

[15] OpenFlow Switch Specification, Version 1.0.0. http://www.OpenFlow.org/documents/OpenFlow-spec-v1.0.0.pdf.

[16] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve: Load-balancing web traffic using OpenFlow. ACM SIGCOMM Demo, 2009.

[17] R. Wang, D. Butnariu, J. Rexford. OpenFlow-Based Server Load Balancing Gone Wild. Hot ICE, 2011.

[18] M. Koerner, O. Kao. Multiple service load-balancing with OpenFlow. IEEE HPSR, 2012.

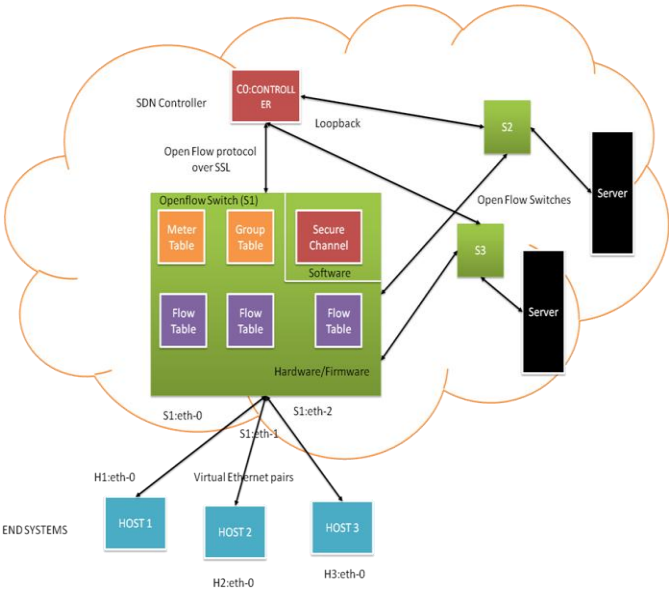

Figure 2: Load-Balancer block diagram architecture using OpenFlow switch and NOX controller.