Lumbergh: An Algorithmic Middleware Manager for the Maple Controller

Henry Davidge Advisor: Richard Yang

Abstract

Software-Defined Networking (SDN) and Network Function Virtualization (NFV) represent two solutions to the challenges presented by evolving network traffic patterns. By interacting with the network through a centralized controller, SDN makes it possible for users to manipulate the entire state of the network rather than reason about the behavior of individual switches. At the same time, NFV allows administrators to rapidly configure and deploy virtual services in lieu of expensive dedicated hardware. However, these technologies are not a panacea for growing network complexity. To effectively leverage these exciting new paradigms, we need tools that provide the right level of abstraction. Building upon the Maple controller, we present Lumbergh, a middleware manager that allows users to express service chains algorithmically in the style of a Maple general purpose program. Lumbergh decouples the abstract notion of a service chain from the concrete process of spinning up virtual machines on a network, simplifying the process of managing NFs while still providing the necessary level of flexibility and control. In a regular Java class, the user can define both named services that are shared between flows and anonymous services that are isolated to a single flow. These services can then be incrementally composed into service chains over the course of the program. Lumbergh translates the service chains into a sequence of routes that efficiently perform the requested functions and direct the flow to its final destination. While Lumbergh requires additional functionality in order to be a practical tool for a network administrator, we believe that its programmatic API and emphasis on abstraction reflect the future of SDN and NFV.

1 Background

1.1 Software-Defined Networking

Software-Defined Networking (SDN) is a new development in network architecture that responds to the inability of existing techniques to meet the needs of modern use patterns. Traditional techniques in networking focused on static architecture well suited to handling many simultaneous one-client-to-one-server connections. Several recent trends have stressed the need for more dynamic and adaptive networks. The bandwidth demands consequent to increasing access to high speed internet and mobile data necessitate more distributed applications – it’s not uncommon for web requests today to hit several machines before returning information to the user. Additionally, the boom of infrastructure as a service (IaaS) and cloud computing has led to the creation of huge datacenters where virtual machines
are constantly communicating with each and rapidly being created and destroyed. The dynamic nature of these networks necessitates tools to easily update network configuration and routing behavior on the fly.

In response to these challenges, SDN provides users with a new configuration paradigm. Rather than communicate with the switches directly, the user interacts with a central controller that then installs rules at each switch using the OpenFlow protocol[5]. This centralized model and the abstractions it enables create a logical separation between the control plane, where administrators manage routing policies and network state, and the data plane, which deals with the actual movement of packets. Therefore, users avoid much of the complexity of creating distributed forwarding rules for a large network and gain the power to quickly change the configuration of their network.

1.2 Middleboxes and Network Function Virtualization

Administrators often desire certain functions to be performed on traffic before it reaches its final destination. These services cover a wide swath of categories including access control, intrusion detection, load balancing, and compression. Traditionally these functions are provided by dedicated hardware known as middleboxes, but, paralleling the rise of server virtualization and cloud infrastructure, are increasingly being performed by virtual machines running on general purpose hardware.

1.3 Maple

Maple[6] is a system that empowers users to define routing logic by providing functions (called algorithmic policies), in the form of standard programs, that map packets to routes. The Maple runtime and optimizer examine the execution of these algorithmic policies and translate their data access patterns into OpenFlow rules. This compilation process means that users can write traditional sequential programs without worrying about how they can be expressed as operations that can be performed at the switches.

2 Statement of Purpose and Related Work

NFV and SDN have the potential for a powerful symbiotic relationship. The power and flexibility of these two technologies pave the way for a future of network design and administration where the user interacts with concise, powerful APIs that empower her to manipulate the entire network rather than lose sight of the forest for the trees by programming individual switches and hosts. However, SDN and NFV do not guarantee these lofty goals in and of themselves. Effective network virtualization requires software frameworks that provide a convenient interface between the control plane and the data plane.

Several open source projects are exploring the question of how best to provide both the control and convenience necessary for efficient configuration of NFV. Neutron[3], one component of the comprehensive OpenStack project, provides a flexible API that provides users with great control over the creation, configuration, and placement of NFs on a network. At a slightly higher level, the Open Daylight[4] project builds on Neutron to provide a reference framework
centered around NFV management. While details on their website are fairly sparse at the moment, the project seems focused on creating an intuitive and pleasant interface for the manual manipulation of NFs.

In this report we approach the problem from an alternative perspective. Rather than ask how we can provide effective manual management tools, Lumbergh attempts to empower the user to think in terms of service chains and flow processing rather than virtual machine management while still providing adequate control and flexibility. This ideal runs parallel to the central tenet of SDN – the separation of the data and control planes. In an effort to accomplish the goal, we build upon Maple to provide a system for the assignment of service chains to flows through general purpose programs.

3 API and Examples

A Lumbergh user specifies an algorithmic policy creating a Java class that extends the abstract class `NFVFunction`. This class must implement a method `nfvOnPacket` that maps flows to an `NFVRoute` comprised of a source, a destination, and a chain of services. The services contained in the service chain can either be member variables of the user class or local variables defined inside of the `nfvOnPacket` method. By defining a service as a member variable, the user can specify that multiple flows (here identified by `(srcIP, dstIP)` tuple) should pass through the same instance of a service, for example to maintain global network statistics or to ensure that an intrusion detection system with cross-activating rules has adequate knowledge across flows. In contrast, local service variables service only one flow.

At the beginning of `nfvOnPacket`, the user specifies hosts on the network that can be used for service virtualization. Each host is identified by a `(MACAddress, IPAddress, Name)` tuple. Additionally, the user provides a capacity for each host. Lumbergh uses an abstract notion of capacity to facilitate service placement – the quantity can correspond to real world values such as processing power, memory, or disk space. During service placement, Lumbergh identifies potential hosts by comparing their remaining capacity to the capacity required by a service.

Services are defined by extending the abstract class `NFVService`. Each service must implement the function `bwToCapacity(int capacity)` that maps a bandwidth to capacity. Like capacity, bandwidth is an abstract term and can be used as a relative quantity within the network. This function acknowledges the varying complexity of different NFs and ensures that the initial service configuration will meet the anticipated needs of the network.

Now, we will present and annotate an example Lumbergh program, written by our old friend and network programmer Alice.

```java
public class Example extends NFVFunction {
    NFVService s1;
    NFVService s2;
    NFVService s3;
    Set<String> bad_hosts;

    public Example() {
```
super();
s = new TestService(1);
s2 = new TestService(2);
s3 = new TestService(1);
}

In the class declaration and constructor, Alice declares and initializes her instance services that will be used at later points in the program and shared between flows just as she would initialize any other member object. In addition, she defines a set of badhosts that she does not want to permit to communicate over the network.

```java
public NFVRoutenfvOnPacket(Packet p) {
  addHost(new NFVHost(17, 2, "10.0.0.17", "h17"));
  addHost(new NFVHost(4, 2, "10.0.0.4", "h4"));
  addHost(new NFVHost(5, 1, "10.0.0.5", "h5"));
  List<NFVService> services = new ArrayList<NFVService>();

  if (bad_hosts.contains(fromIPv4Address(p.ipSrc()))) {
    return null;
  }
  services.add(s1); // Every flow should go through s1

  if (p.ipSrc() != ipAddress("10.0.0.9") && p.ipDst() != ipAddress("10.0.0.9")) {
    services.add(s2);
  }

  if (p.ipSrc() == ipAddress("10.0.0.8") && p.ipDst() == ipAddress("10.0.0.7")) {
    services = new ArrayList<NFVService>();
    services.add(new TestService(1));
  }

  return NFVRoute.unicast(hostLocation(p.ethSrc()), hostLocation(p.ethDst()), services);
}
```

At the beginning of `nfvOnPacket` Alice specifies the hosts that can be used to run the services. Note that this specification must be performed inside of the function body rather than in the constructor since the constructor is called immediately upon startup before the controller discovers the network topology via OpenFlow. She also initializes the list of services that she will manipulate over the course of the function to specify the service chain.

In the core of the program, Alice adds services to the chain depending on the attributes of the packet. The bad hosts bit serves to show that despite the focus on NFV mangament, Lumbergh can still leverage Maple to provide other basic network utilities. Returning null in lieu of an `NFVRoute` instructs the controller to drop the packet.
By testing this program on a simulated network, we can see the effects of this policy. The topology below is, in the language of Mininet, a tree topology with depth=2 and fanout=4. The three service hosts have an orange fill. In this demo, the code snippet `host1 ping host2` means that we send a ping message from host1 to host1.
First we demonstrate Lumbergh's most basic service routing capabilities by running `h9 ping h15`. This command results in `s1` being spun up at `h17`, the host that results in the shortest path between `h9` and `h15`. Note that the remaining capacity of `h17` is now 1.

Figure 1: `h9 ping h15` (Shortest path service placement)
Next we execute the command `h13 ping h14`. Although the optimal service placement for `s2` would be at `h17`, the placement of `s1` depleted too much of `h16`’s capacity. Therefore the chain must instead be routed through `h4`. This command illustrates Lumbergh’s network context awareness.

Figure 2: h13 ping h13 (Capacity awareness)
In a final example we demonstrate Lumbergh’s flexible API and its ability to route through existing services. The service chain from $h_7$ to $h_8$ is specified by passing an anonymous service. This mechanism can be used to quickly instantiate services that are isolated to a particular flow. In addition, the return chain shows the ability to route through existing services – we know that these are the existing service instances because otherwise there would not be enough capacity for instantiation.

Figure 3: $h_8$ ping $h_7$ (Anonymous services and shared services)
4 Architecture

4.1 Layered Graph Algorithm

Lumbergh uses the layered graph algorithm proposed by Choi et. al 2003[1] to decide the optimal placement for services on the network. This algorithm transforms the network graph into a new graph and then assigns service location by computing the shortest path. The transformed graph consists of several layers, numbered 1 to \( k \), each of which is identical to the original topology. Layer \( k \) is connected to Layer \( k+1 \) at each potential host site that possesses the capacity to perform Service \( k \) on the flow, so the transformed graph contains |Services| + 1 layers. If the shortest path includes the edge from Host \( k \) to Host \( k+1 \), where the subscript identifies the host and the superscript the layer, then service \( k \) should be assigned to Host \( i \).

However, since the structure of the layered graph remains constant over the execution of the algorithm it is possible that Service \( k \) will be assigned Host \( i \) despite a Service \( l < k \) exhausting the capacity of Host \( i \). In general it is not possible to efficiently determine the optimal service placement in the presence of such capacity constraints since the problem can be reduced to finding a Hamiltonian path in the original graph (which is known to be NP-hard) if every node except the source and sink is a potential host with the capacity to perform exactly one service and |Hosts| = |Services|. Instead, we use a heuristic: before adding an edge from Host \( k \) to Host \( k+1 \) to the shortest path tree, we backtrack through the tree to ensure that Host \( i \) still has adequate capacity. This technique ensures that the shortest path will correspond to a possible service placement, but may fail to return an assignment in some cases where a valid placement is possible.

4.2 Extension for Shared Services

In some cases the user may want multiple flows routed to the same service instance. We handle these cases by solving multiple service placement problems and concatenating the results. For example, if Service \( i \) has already been placed at Host \( i \), we first compute the service path that places the services Service \( 1 \) ... Service \( k-1 \) between the source and Host \( i \) and then append the service path that places Service \( k \) ... Service \( n \) on the path from Host \( i \) to the sink.

4.3 Implementation Details

After calculating the shortest service path, Lumbergh first breaks the path into routes at each service point, so a path with \( k \) services will be broken into \( k + 1 \) routes. To ensure that the flow goes through the entire service chain, its position in the chain must be unambiguously determinable using only OpenFlow criteria. Furthermore, in order to maximize flexibility, the service path should be allowed to contain cycles (in the original graph, not the layered graph). Therefore, packets must be modified as they progress through the service chain so that they can be uniquely identified each time the leave a given host. Lumbergh solves this problem by maintaining a hashtable that maps \((srcIP, dstIP, inPort, VLAN) \rightarrow Route\). Here a route consists of a destination switch port, an ordered set of links, and, crucially,
zero or more packet modifications. The first time Lumbergh process a flow, it records its VLAN tag if it has one and writes the value as 0. Each time a packet passes through a service its VLAN tag is then incremented by one, except for the last time when it is reset to its original value. Thus, the hashtable presented can be guaranteed to correctly identify a packet’s position in the service chain. Since each criterion in the intermediate identification can be checked at the switches, Maple is able to efficiently compile these service chains into OpenFlow rules, so a flow only has to be routed to the controller the first time it appears in the network.

5 Performance Characteristics

Unfortunately the time and resource constraints we faced did not allow us to test Lumbergh on an actual network. However, we were able to simulate a network using Mininet in order to glean some cursory scalability information. Since we were not emulating VM startup time, the benchmarks below only consider the running time of the layered graph service placement algorithm. The charts shown below were both performed on a 2013 MacBook Air with a dual core Intel i5 processor. Each datapoint represents the mean of 10 trials.
The first plot shows runtime as a function of the number of nodes (switches or hosts) in the original graph. In each of these tests three services were being placed. Each of the topologies tested is a tree topology with a depth of two and fanouts ranging from 2 to 5. Due to the limitations of Mininet we were not able to scale above this network size. However, these limited data suggest that at small network sizes the algorithm is limited more by the overhead of calling a Java function than the algorithmic complexity.

Figure 4: Runtime as a function of the number of nodes in the network
Next we see the runtime as a function of the number of services being placed. Each of these service chains were computed for a tree topology with a depth of 2 and a fanout of 4. Here we see the algorithm scaling close to linearly. Since the work of the layered graph algorithm is performed by Dijkstra’s shortest path algorithm with complexity $O(|E| + |V|\log(|V|))$ on a graph that is $k + 1$ times as large as the original graph, we expect the layered graph algorithm to take time $\sim (k + 1)|E| + (k + 1)|V|\log((k + 1)|V|)$, which is approximately what we see in this chart. This plot also reassures us that the capacity tracking module of the algorithm is not too costly, at least for networks with small diameter like the one tested.

![Number of Services vs Runtime](image)

Figure 5: Runtime as a function of the number of services
6 Conclusions and Future Work

While we believe that Lumbergh shows promise as a possible blueprint for modern NFV management, there remains much to be done. Most practically, we need to implement a way for Lumbergh to communicate with and spin up real NFs in a cloud environment. Several projects promise to be helpful in this regard. In particular, ClickOS[2] provides a lightweight Xen-managed host for NFs based on the Click modular router. The short startup time and small image size of ClickOS services would allow dynamic service placement engines like Lumbergh to avoid halting traffic during the manipulation of NFs.

Once Lumbergh is capable of running on actual hardware, it can focus on fault tolerance and NF monitoring. While automatic service placement is nice, the system could become truly useful to administrators if it can monitor NF health and load and dynamically adapt to changes in network state. Although there are sure to be implementation difficulties, this functionality can be conceptually achieved by observing packet queues and CPU load at the hosts and then re-executing the layered graph algorithm with unhealthy nodes removed. The system could also be improved by observing traffic patterns over time and moving services that are no longer in the optimal location.

Looking at the bigger picture, we expect that in the coming years abstraction and programmatic administration will become more and more important in the networking community, and are excited to see the evolution of the next generation of network configuration utilities like Lumbergh.

7 Acknowledgments

I’d like to thank the Yale Computer Science community for a rewarding four years of enthusiasm, exhaustion, and enlightenment. Thanks to all of you, both my knowledge and critical thinking faculties are light years ahead of their state at my matriculation.

In particular I want to acknowledge Prof. Yang for advising my project and making time to help me understand the important principles of SDN and system design, and Andi Voellmy for his invaluable insight and support over the course of the semester.
8 References


