Frenetic Tutorial

Team Frenetic

June 1, 2013

1 Introduction

The goal of this tutorial is to teach readers how to program a Software-Defined Network (SDN) running OpenFlow using the Frenetic programming language. This involves explaining the syntax and semantics of Frenetic and illustrating its use on a number of simple examples. Along the way, there are a number of exercises for the reader. Solutions appear online.

2 Motivation

[Plagiarized in part from IEEE overview paper. –DPW]

Traditional networks are built out of special-purpose devices running distributed protocols that provide functionality such as routing, trafc monitoring, load balancing, NATing and access control. These devices have a tightly-integrated control and data plane, and network operators must separately congure every protocol on each individual device. This configuration task is a challenging one as network operators must struggle with a host of different baroque, low-level, vendor-specific configuration languages. Moreover, the pace of innovation is slow as device internals and APIs are often private and proprietary, making it difficult to develop new protocols or functionality to suit client needs.

Recent years, however, have seen growing interest in software-dened networks (SDNs), in which a logically-centralized controller manages the packet-processing functionality of a distributed collection of switches. SDNs make it possible for programmers to control the behavior of the network directly, by conguring the packet-forwarding rules installed on each switch. Moreover, the Open Networking Foundation is committed to developing a standard, open, vendor-neutral protocol for controlling collections of switches. This protocol is OpenFlow.

SDNs can both simplify existing applications and also serve as a platform for developing new ones. For example, to implement shortest-path routing, the controller can calculate the forwarding rules for each switch by running Dijkstras algorithm on the graph of the network topology instead of using a more complicated distributed protocol. To conserve energy, the controller can selectively shut down links or even whole switches after directing trafc along

other paths. To enforce ne-grained access control policies, the controller can consult an external authentication server and install custom firewall rules.

But although SDNs makes it possible to program the network, they do not make it easy. Protocols such as OpenFlow expose an interface that closely matches the features of the underlying switch hardware. Roughly speaking, OpenFlow allows programmers to manually install and uninstall individual packet-processing rules. First-generation controller systems such as NOX, Beacon, and Floodlight support the same low-level interface, which forces applications to be implemented using programs that manipulate the fine-grained state of individual devices. Unfortunately, it is extremely difficult to develop independent program components, such as a router, firewall and network monitor, that collaborate to control the flow of traffic through a network since the application must ultimately install a *single* set of low-level rules on the underlying switches. This single set of rules must simultaneously implement the desired high-level semantics for each independent high-level component.

In addition, a network is a distributed system, and all of the usual complications arisein particular, control messages sent to switches are processed asynchronously. Programming asynchronous, distributed systems is notoriously difficult and error prone. Network programmers require require support to get this right.

The goal of the Frenetic language is to raise the level of abstraction for programming SDNs. To replace the low-level imperative interfaces available today, Frenetic offers a suite of declarative abstractions for querying network state, dening forwarding policies, and updating policies in a consistent way. These constructs are designed to be *modular* so that individual policies can be written in isolation, by different developers and later composed with other components to create sophisticated policies. This is made possible in part by the design of the constructs themselves, and in part by the underlying run-time system, which implements them by compiling them down to low-level OpenFlow forwarding rules. Our emphasis on modularity and compositionthe foundational principles behind effective design of any complex software systemis the key feature that distinguishes Frenetic from other SDN controllers.

3 Programming Static Frenetic Policies

Describe the basic semantics and concepts.

- located packets - policies are functions - basic functions

Static NetCore Programming Examples ———————————

0. Review the topology in tutorial-topo. Define some user-friendly switch names for our topology too:

1. Write a program to route all ip traffic as follows: - packets with destination ip 10.0.0.10 arriving at switch C go to host 10. - packets with destination ip 10.0.0.20 arriving at switch C go to host 20. - packets with destination ip 10.0.0.30 arriving at switch D go to host 30. - packets with destination ip 10.0.0.40 arriving at switch D go to host 40. - flood all arp packets arriving at

any switch Try out your program by pinging 10 from 20 and 20 from 10. What happens? What happens if you ping 10 from 30?

Start your work with this handy wrapper to handle flooding of arp packets. Replace "drop" below with some other policy that solves the problem.

"' let arpify (P:policy) = if frameType = arp then all else P

let my_sol1 = arpify(drop) "'

2. Extend the program written in (a) to route all traffic between 10, 20, 30, 40.

3. Now consider the program that routes all traffic between all hosts, which we will provide.

4. Use the NetCore query facility to discover the set of all TCP ports being used by either machines 10 or 20. Here is a [list of common port numbers](http://packetlife.net/media/library/23 ports.pdf) Note: some protocols are sending a lot of traffic (eg: HTTP, on port 80). As you investigate, narrow your searches to find the "needle in the haystack." There is one host sending a small amount of traffic to a non-standard port in a nonstandard format. Which host is it? Print the packets using that port. They contain a secret message.

5. Construct a firewall for the network that enforces the following policy. Compose it with the routing policy defined in part 3. - Machines 10, 20, 30, 40 are trusted machines. - Machine 50 is an untrusted machine. - Each of these machines is serving files using HTTP (TCP port 80). - Machines 10 and 20 have private files that may not be read by untrusted machines using HTTP. - Machines 30, 40, 50 have public files that may be read by any machine. - All traffic not explicitly prohibited must be allowed to pass through the network.

Acknowledgements. This work is supported in part by the NSF under grant CNS-1111698, the ONR under award N00014-12-1-0757, and by a Google Research Award.

References