CSC 2229 – Software-Defined Networking

Handout # 6: Programming Software-Defined Networks

Professor Yashar Ganjali
Department of Computer Science
University of Toronto

yganjali@cs.toronto.edu
http://www.cs.toronto.edu/~yganjali

Some slides courtesy of J. Rexford (Princeton), N. Foster (Cornell) and N. Feamster (Georgia Tech), all used with permission.
Announcements

- Final project proposal
  - 2 pages
  - Due: Fri. Oct. 9th (5PM)

- In class presentations
  - Booked for next week
  - Email me if you want to be next
Programming SDNs

- **The Good**
  - Network-wide visibility
  - Direct control over the switches
  - Simple data-plane abstraction

- **The Bad**
  - Low-level programming interface
  - Functionality tied to hardware
  - Explicit resource control

- **The Ugly**
  - Non-modular, non-compositional
  - Challenging distributed programming
Programming OpenFlow is Not Easy

- OpenFlow and NOX now make it possible to implement exciting new network services
  - Unfortunately, they do not make it easy.

- Combining different applications is not straightforward

- OpenFlow provides a very low level abstraction

- We have a two-tier architecture

- Network of switches is susceptible to race conditions
Programming OpenFlow is Not Easy

- OpenFlow and NOX now make it possible to implement exciting new network services
  - Unfortunately, they do not make it easy.

- Combining different applications is not straightforward

- OpenFlow provides a very low level abstraction

- We have a two-tier architecture

- Network of switches is susceptible to race conditions
Problem 1: Anti-Modularity

- Combining different applications is challenging

- **Example**: monitor + route + firewall + load balancing
  - How these applications will work together?
  - How are messages from switches delivered to these applications?
  - How are messages from these apps aggregated to be sent to switches?
  - Do OpenFlow and NOX provide a way for each app to perform its job without impacting other apps?

- **Question**: How can we combine these applications?
Combining Many Networking Tasks

Monolithic application

Controller Platform

Monitor + Route + FW + LB

Hard to program, test, debug, reuse, port, ...
Modular Controller Applications

A module for each task

Monitor  Route  FW  LB

Controller Platform

Easier to program, test, and debug
Greater reusability and portability
Beyond Multi-Tenancy

Each module controls a different portion of the traffic

Relatively easy to partition rule space, link bandwidth, and network events across modules
Modules Affect the *Same Traffic*

Each module *partially* specifies the handling of the traffic.

How to combine modules into a complete application?
Anti-Modularity Example

- Consider a simple network
- Want to add two applications

- Simple repeater
  - Port 1 $\rightarrow$ Port 2
  - Port 2 $\rightarrow$ Port 1

- Web monitor
  - Packet and byte counts
  - Incoming web traffic
NOX Events and Commands

Events:

- **switch_join(switch):** triggered when switch joins the network
- **stats_in(switch, xid, pattern, packets, bytes),** triggered when switch returns the *packets* and *bytes* counters in response to a request for statistics about rules contained in *pattern*

Commands:

- **install(switch, pattern, priority, timeout, actions):** installs a rule in the flow table of *switch*
- **query_stats(switch, pattern):** issues a request for statistics from all rules contained in *pattern* on *switch*
Anti-Modularity Example

Repeater

```python
def switch_join(switch):
    repeater(switch)

def repeater(switch):
    pat1 = {in_port:1}
    pat2 = {in_port:2}
    install(switch, pat1, DEFAULT, None, [output(2)])
    install(switch, pat2, DEFAULT, None, [output(1)])
```

Web Monitor

```python
def monitor(switch):
    pat = {in_port:2, tp_src:80}
    install(switch, pat, DEFAULT, None, [])
    query_stats(switch, pat)

def stats_in(switch, xid, pattern, packets, bytes):
    print bytes
    sleep(30)
    query_stats(switch, pattern)
```
Anti-Modularity Example

**Repeater**

```python
def switch_join(switch):
    repeater(switch)

def repeater(switch):
    pat1 = {in_port:1}
    pat2 = {in_port:2}
    install(switch, pat1, DEFAULT, None, [output(2)])
    install(switch, pat2, DEFAULT, None, [output(1)])
```

**Web Monitor**

```python
def monitor(switch):
    pat = {in_port:2, tp_src:80}
    install(switch, pat, DEFAULT, None, [])
    query_stats(switch, pat)

def stats_in(switch, xid, pattern, packets, bytes):
    print bytes
    sleep(30)
    query_stats(switch, pattern)
```

**Repeater/Monitor**

```python
def switch_join(switch):
    repeater_monitor(switch)

def repeater_monitor(switch):
    pat1 = {in_port:1}
    pat2 = {in_port:2}
    pat2web = {in_port:2, tp_src:80}
    install(switch, pat1, DEFAULT, None, [output(2)])
    install(switch, pat2web, HIGH, None, [output(1)])
    install(switch, pat2, DEFAULT, None, [output(1)])
    query_stats(switch, pat2web)

def stats_in(switch, xid, pattern, packets, bytes):
    print bytes
    sleep(30)
    query_stats(switch, pattern)
```

- blue = from repeater
- red = from web monitor
- green = from neither
Programming OpenFlow is Not Easy

- OpenFlow and NOX now make it possible to implement exciting new network services
  - Unfortunately, they do not make it easy.
- Combining different applications is not straightforward
- OpenFlow provides a very low level abstraction
- We have a two-tier architecture
- Network of switches is susceptible to race conditions
Problem 2: Low-Level API

- OpenFlow is a low level programming interface
  - Derived from the features of the switch hardware
  - Rather than ease of use

- Programmer must describe *low-level details* that do not affect the *overall behavior* of the program

- Example: to implement simple set difference we require
  - Multiple rules
  - Priorities
  - All need to be managed by the programmer

- Focusing on the big picture not easy
Low-Level API Example

- Extend the repeater and monitoring
  - Monitor all incoming web traffic except traffic destined to 10.0.0.9 (on internal network)
- We need to express a logical “difference” of patterns
  - OpenFlow can only express positive constraints

```python
def repeater_monitor_noserver(switch):
    pat1 = {in_port: 1}
    pat2 = {in_port: 2}
    pat2web = {in_port: 2, tp_src: 80}
    pat2srv = {in_port: 2, nw_dst: 10.0.0.9, tp_src: 80}
    install(switch, pat1, DEFAULT, None, [output(2)])
    install(switch, pat2srv, HIGH, None, [output(1)])
    install(switch, pat2web, MEDIUM, None, [output(1)])
    install(switch, pat2, DEFAULT, None, [output(1)])
    query_stats(switch, pat2web)
```
Programming OpenFlow is Not Easy

- OpenFlow and NOX now make it possible to implement exciting new network services
  - Unfortunately, they do not make it easy.

- Combining different applications is not straightforward

- OpenFlow provides a very low level abstraction

- We have a two-tier architecture

- Network of switches is susceptible to race conditions
Problem 3: Two-Tiered System

- Control program manages networks by
  - Installing/uninstalling switch-level rules

- Programmer needs to specify communication patterns between controller and switches
  - Deal with tricky concurrency issues

- Controller does not have full visibility
  - Sees only packets that the switches do not know how to handle.
  - Previously installed rules
    - Reduce the load on the controller
    - Make it difficult to reason

- Detour: proactive vs. reactive rule installation
Two-Tiered Programming Example

- Extending the original repeater
  - Monitor the total amount of incoming traffic
  - By destination host

- Cannot install all of the rules we need in advance
  - Address of each host is unknown *a priori*

- The controller must dynamically install rules for the packets seen at run time
Two-Tiered Programming Example

```python
def repeater_monitor_hosts(switch):
    pat = {in_port:1}
    install(switch,pat,DEFAULT,None,[output(2)])

def packet_in(switch,inport,packet):
    if inport == 2:
        mac = dstmac(packet)
        pat = {in_port:2,dl_dst:mac}
        install(switch,pat,DEFAULT,None,[output(1)])
        query_stats(switch,pat)
```

- Two programs depended on each other
- Complex concurrency issues can arise
- Reading/understanding the code is difficult
- Details are sources of significant distraction
Programming OpenFlow is Not Easy

- OpenFlow and NOX now make it possible to implement exciting new network services
  - Unfortunately, they do not make it easy.

- Combining different applications is not straightforward

- OpenFlow provides a very low level abstraction

- We have a two-tier architecture

- Network of switches is susceptible to race conditions
Problem 4: Race Conditions

- Race conditions can cause complications
  - We have a distributed system
    - Of switches
    - And controllers

- Example 1: rule install delay
  - One new flow
  - Multiple packets
Race Conditions Example

- Example 2: firewall application
  - Running on multiple switches
  - Allow A initiate a flow to B
    - Two way
  - But, block flows started by B
Race Conditions Example

- Example 2: firewall application
  - Running on multiple switches
  - Allow A initiate a flow to B
    - Two way
  - But, block flows started by B
Northbound API

- Programming abstraction for applications
- Hides low-level details
- Helps orchestrate and combine applications

Example uses
- Path computation
- Loop avoidance
- Routing
- Security
Who Will Use the Northbound API?

- Service providers
- Sophisticated network operators
  - Or, enthusiastic network administrators
- Vendors
- Researchers
- Or anyone who wants to add new capabilities to their network
Benefits of the Northbound API

- Vendor independence
- Ability to quickly modify or customize control applications through simple programming

Example applications:
- Large virtual switch
- Security applications
- Resource management and control
- Middlebox integration
Frenetic Language

- **Declarative Design**
  - *What the programmer* might want
  - Rather than *how the hardware* implements it.

- **Modular Design**
  - Primitives have *limited network-wide effects* and semantics
  - Independent of the context in which they are used.

- **Single-tier Programming**
  - *See-every-packet* abstraction

- **Race-free Semantics**
  - Automatic race detection and packet suppression

- **Cost control**
  - Core query logic can be executed on network switches
Network Control Loop

- **Compute Policy**
- **OpenFlow Switches**
  - Read state
  - Write policy

- **CSC 2229 - Software-Defined Networking**
- **University of Toronto – Fall 2015**
Language-Based Abstractions

Query abstractions

Update abstractions

OpenFlow Switches

Writing/combining modules
Frenetic Language

- Abstractions for querying network state
  - An integrated query language
    - Select, filter, group, sample sets of packets or statistics
    - Designed so that computation can occur on data plane

- Abstractions for specifying a forwarding policy
  - A functional stream processing library (based on FRP)
    - Generate streams of network policies
    - Transform, split, merge, filter policies and other streams

- Implementation
  - A collection of Python libraries on top of NOX
Frenetic Queries

\[ q ::= \text{Select}(a) \ast \\
\hspace{1cm} \text{Where}(fp) \ast \\
\hspace{1cm} \text{GroupBy}([qh_1, \ldots, qh_n]) \ast \\
\hspace{1cm} \text{SplitWhen}([qh_1, \ldots, qh_n]) \ast \\
\hspace{1cm} \text{Every}(n) \ast \\
\hspace{1cm} \text{Limit}(n) \]

\[ a ::= \text{packets} \mid \text{sizes} \mid \text{counts} \]

\[ qh ::= \text{inport} \mid \text{srcmac} \mid \text{dstmac} \mid \text{ethtype} \mid \\
\hspace{1cm} \text{vlan} \mid \text{srcip} \mid \text{dstip} \mid \text{protocol} \mid \\
\hspace{1cm} \text{srcport} \mid \text{dstport} \mid \text{switch} \]

\[ fp ::= \text{true}_\text{fp}() \mid qh_\text{fp}(n) \mid \\
\hspace{1cm} \text{and}_\text{fp}([fp_1, \ldots, fp_n]) \mid \\
\hspace{1cm} \text{or}_\text{fp}([fp_1, \ldots, fp_n]) \mid \\
\hspace{1cm} \text{diff}_\text{fp}(fp_1, fp_2) \mid \text{not}_\text{fp}(fp) \]
Frenetic Queries

**Goal:** measure total web traffic on port 2, every 30 seconds

```python
def web_query():
    return (Select(sizes) *
            Where(inport_fp(2) & srcport_fp(80)) *
            Every(30))
```

**Key Property:** query semantics is independent of other program parts
Policy in OpenFlow

• Defining “policy” is complicated
  • All rules in all switches
  • Packet-in handlers
  • Polling of counters

• Programming “policy” is error-prone
  • Duplication between rules and handlers
  • Frequent changes in policy (e.g., flowmods)
  • Policy changes affect packets in flight
Frenetic Forwarding Policies

- **Rules** are created using the Rule Constructor, which takes a *pattern* and a list of *actions* as arguments.

- **Network policies** associate rules with switches:
  - Dictionaries mapping switches to list of rules.

- **Policy events** are infinite, time-indexed streams of values, just like the events generated from queries:
  - Programs control the installation of policies in a network *over time* by generating *policy events*.

- **Listeners** are event consumers:
  - Print: send to console
  - Send: transfer packet to switch and apply actions
  - Register: apply network wide policy
Power of Policy as a Function

- **Composition**
  - **Parallel:** Monitor + Route
  - **Sequential:** Firewall >> Route

- \( A >> (B + C) >> D \)
- \((A >> P) + (B >> P) \quad (A + B) >> P\)
Frenetic Forwarding Policies

Goal: implement a repeater switch

```
rules = [Rule(inport_fp(1), [forward(2)]),
         Rule(inport_fp(2), [forward(1)])]

def repeater():
    return (SwitchJoin() >> Lift(lambda switch: {switch: rules}))
```

Key Property: Policy semantics independent of other queries/policies
Parallel Composition

Monitor on source + Route on destination

Controller Platform

srcip = 5.6.7.8 → count

dstip = 1.2.3.4 → fwd(1)
dstip = 3.4.5.6 → fwd(2)
Parallel Composition

Monitor on source + Route on destination

Controller Platform

srcip = 5.6.7.8 → count

dstip = 1.2.3.4 → fwd(1)
dstip = 3.4.5.6 → fwd(2)

srcip = 5.6.7.8, dstip = 1.2.3.4 → fwd(1), count
srcip = 5.6.7.8, dstip = 3.4.5.6 → fwd(2), count
srcip = 5.6.7.8 → count
dstip = 1.2.3.4 → fwd(1)
dstip = 3.4.5.6 → fwd(2)
Sequential Composition

$\text{srcip} = 0^*, \text{dstip}=1.2.3.4 \rightarrow \text{dstip}=10.0.0.1$

$\text{srcip} = 1^*, \text{dstip}=1.2.3.4 \rightarrow \text{dstip}=10.0.0.2$

dstip = 10.0.0.1 $\rightarrow$ fwd(1)

dstip = 10.0.0.2 $\rightarrow$ fwd(2)

Controller Platform

Load Balancer $\gg$ Routing
Sequential Composition

srcip = 0*, dstip=1.2.3.4 → dstip=10.0.0.1
srcip = 1*, dstip=1.2.3.4 → dstip=10.0.0.2

dstip = 10.0.0.1 → fwd(1)
dstip = 10.0.0.2 → fwd(2)

Load Balancer >> Routing

Controller Platform

srcip = 0*, dstip = 1.2.3.4 → dstip = 10.0.0.1, fwd(1)
srcip = 1*, dstip = 1.2.3.4 → dstip = 10.0.0.2, fwd(2)
Dividing the Traffic Over Modules

- Predicates
  - Specify which traffic traverses which modules
  - Based on input port and packet-header fields

Web traffic
\( \text{dstport} = 80 \)

Non-web
\( \text{dstport} \neq 80 \)

Load Balancer

Routing

Monitor

Routing
Program Composition

Goal: implement both web monitoring and repeater

```python
def host_query():
    return (Select(counts) *
            Where(inport_fp(1)) *
            GroupBy([srcmac]) *
            Every(60))

def secure(host_policy_stream): ...

def main():
    web_query() >> Print()
    secure(Merge(host_query(), repeater())) >> Register()
```

Key Property: queries and policies compose
Frenetic Runtime System

High-level Language
- Integrated query language
- Effective support for composition and reuse

Run-time System
- Interprets queries, policies
- Installs rules
- Tracks stats
- Handles asynchronous events