frenetic

A Network Programming Language

Nate Foster, Mike Freedman, Rob Harrison, Chris Monsanto, Jen Rexford, Alec Story, and Dave Walker
Traditional Networks

Data Plane (hardware):
- Forwards, filters, buffers, tags, rate-limits; collects stats

Control Plane (software):
- Tracks topology; computes routes; modifies data plane state

Operator:
- Monitors traffic, Configures policy

Data Plane (hardware):
Forwards, filters, buffers, tags, rate-limits; collects stats
Software-Defined Networks

**Idea:** move control off of switches and onto a separate, general-purpose computer.

Controller Machine
Arbitrary program implements control plane functionality:
- Monitors traffic,
- Tracks topology,
- Selects routes,
- Installs forwarding rules.
Everyone has signed on Microsoft, Google, Cisco, Yahoo, Facebook, Deutch Telekom,…

New Applications

- Host mobility
- Virtualization
- Dynamic access control
- Energy-efficiency
- Load balancing
New Challenges

OpenFlow makes it *possible* to program the network, but it does not make it *easy!*

- Provides a thin veneer over switch hardware
- Like programming in assembly

Our goal

- Develop new abstractions for programming networks
  - More convenient
  - More modular
  - More reliable
  - More secure
This Talk

OpenFlow in more depth
- Existing programming model and problems

Frenetic Language
- New abstractions for network programming

Frenetic Run-time System
- Implementation strategy and experience
## OpenFlow Switches

### Flow Table

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
<th>Bytes</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010</td>
<td>Drop</td>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>010*</td>
<td>Forward(n)</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>011*</td>
<td>Controller</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
OpenFlow Controllers (NOX)

- Network Events
  - Packets
  - Stats
  - Topology changes

- Control Messages
  - Install rules
  - Uninstall rules
  - Query counters

Controller

Switches

NOX Program
Typical OpenFlow Application

- Network Events:
  - Forwarding table miss

- Controller

- Control Messages:
  - (Un)install rules

- Switches
Problem I: Anti-Modular

Controller Application

Repeater Module

P: Forward 1 → 2 and 2 → 1

Monitoring Module

Q: Query web traffic

P installed

Doesn’t work because repeater rules too coarse-grained; monitoring rules don’t forward
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)])

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)

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
Problem II: Two-tiered Model

Tricky problem:
- Controller activity is driven by packets
- For efficiency, applications install rules to forward packets in hardware

Constant questions:
- “Will that packet come to the controller and trigger my computation?”
- “Or is it already being handled invisibly on the switch?”
Problem III: Network Race Conditions

A challenging sequence of events:

- **Switch**
  - sends packet to controller

- **Controller**
  - analyzes packet
  - updates its state
  - initiates installation of new packet-processing rules

- **Switch**
  - hasn’t received new rules
  - sends new packets to controller

- **Controller**
  - confused
  - packets in the same flow handled inconsistently
Three problems with a common cause

Three problems

- Anti-modular
- Two-tiered model
- Network race conditions

One cause

No effective *abstractions* for *reading* network state
Separate network programming into two parts:

- Abstractions for querying network state
  - Reads have *no effect* on forwarding policy
  - Reads able to *see every packet*

- Abstractions for specifying a forwarding policy
  - Forwarding *policy* must be separated from *implementation mechanism*

A natural decomposition that mirrors two fundamental tasks: monitoring and forwarding
This Talk

OpenFlow & Nox in more depth
- Existing programming model and problems

Frenetic Language
- New abstractions for network programming

Frenetic Run-time System
- Implementation strategy and experience
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

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
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
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
This Talk

OpenFlow & Nox in more depth
- Existing programming model and problems

Frenetic Language
- New abstractions for network programming

Frenetic Run-time System
- Implementation strategy and experience
Frenetic System Overview

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
Run-time Activities

Frenetic Program

Query
Register

Query Interpreter
Policy Interpreter

Update Stats
Monitoring Loop

Check Subscribers
Check Rules
Do Actions

Packet In
Send Packet
Install Flow
Stats Request
Stats In

NOX

Frenetic Runtime System

Packet In

NOX

Legend:
- Frenetic Program
- NOX
- Runtime Module
- Runtime Data Structure
Preliminary Evaluation

Core Network Applications
- Learning Switch
- Spanning Tree
- Shortest path routing
- DHCP server
- Centralized ARP server
- Generic load balancer

Additional Applications
- Memcached query router
- Network scan detector
- DDOS defensive switch

Micro Benchmarks
- Coded in Frenetic and NOX
MicroBench: Lines of Code

Lines of Code

Forwarding Policy:
- HUB: Floods out other ports
- LSW: Learning Switch

Monitoring Policy

NOX
Frenetic

No monitoring
Web Statistics
Heavy Hitters
Forwarding Policy:
HUB: Floods out other ports
LSW: Learning Switch

Monitoring Policy
No monitoring
Web Statistics
Heavy Hitters

NOX
Frenetic
Ongoing and Future Work

Performance evaluation & optimization
- Measure controller response time and network throughput
- Wildcard rules and proactive rule installation
- Support for parallelism

Program Analysis
- Establish key invariants

Hosts and Services
- Extend queries & controls to end hosts

More abstractions
- Virtual network topologies
- Network updates with improved semantics
## Conclusion: An Analogy

<table>
<thead>
<tr>
<th>Concern</th>
<th>Assembly Languages</th>
<th>Programming Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x86</td>
<td>NOX</td>
</tr>
<tr>
<td><strong>Resource Allocation</strong></td>
<td>Move values to/from registers</td>
<td>Install/uninstall rules on switches</td>
</tr>
<tr>
<td><strong>Resource Tracking</strong></td>
<td>Have I spilled that value?</td>
<td>Will that packet arrive at the controller?</td>
</tr>
<tr>
<td><strong>Coordination</strong></td>
<td>Unregulated calling conventions</td>
<td>Unregulated rule installation</td>
</tr>
<tr>
<td><strong>Portability</strong></td>
<td>Hardware Dependent</td>
<td>Hardware Dependent</td>
</tr>
<tr>
<td></td>
<td>Haskell/ML</td>
<td>Frenetic++</td>
</tr>
<tr>
<td></td>
<td>Declare/use program variables</td>
<td>Construct/ register policy</td>
</tr>
<tr>
<td></td>
<td>Program variables always accessible</td>
<td>Queries can read every packet</td>
</tr>
<tr>
<td></td>
<td>Function calls managed automatically</td>
<td>Policies managed automatically</td>
</tr>
<tr>
<td></td>
<td>Hardware Independent</td>
<td>Hardware Independent</td>
</tr>
</tbody>
</table>
The Team

Nate Foster
Mike Freedman
Rob Harrison

Chris Monsanto
Jen Rexford
Alec Story
Dave Walker

frenetic

http://frenetic-lang.org
### Implementation Options

#### Rule Granularity
- **microflow** (exact header match)
  - simpler; more rules generated
- **wildcard** (multiple header match in single rule)
  - more complex; fewer rules (may be) generated

#### Rule Installation
- **reactive** (lazy)
  - first packet of each new flow goes to controller
- **proactive** (eager)
  - new rules pushed to switches