Flog : Logic Programming for Software Defined Networks

Naga Praveen Katta, Jennifer Rexford, David Walker
Princeton University
Traditional networks

- Traditional network elements - special purpose devices running distributed algorithms.

Operator:
- Monitors traffic
- Identifies threats
- Indirectly configures policy

Control Plane – Complex Distributed algorithms
Data Plane – Simple packet forwarding
Traditional networks

- Managing a network is hard
  - Routers with millions of lines of code
  - Running complex distributed protocols
  - Connected to a diverse set of middleboxes

- Operating a network is expensive
  - More than half the cost of a network
  - Manual operator errors cause most outages

- Traditionally hard to innovate
  - Closed equipment with vendor specific interfaces
  - Ossified evolution
  - Few people can make changes (say, CISCO certified)
What is a Software-Defined Network?

**Controller Machine**
Arbitrary program implements control plane functionality:
- Tracks network topology
- Monitors traffic
- Installs rules to block or forward traffic.
Openflow Switches

- Switch packet-handling rules: \(<\text{pattern, action, priority}>\)
  - **Pattern**: match packet header bits
  - **Action**: drop, forward, modify, send to controller
  - **Priority**: disambiguate overlapping patterns
  - Counters: #bytes and #packets

### 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>

**priority**
Industry Thrust

• Everyone has signed on
  – Google, Facebook, Microsoft, Yahoo, Verizon, Deutsche Telekom

• New applications
  – Host mobility
  – Server load balancing
  – Network virtualization
  – Dynamic access control
  – Energy-efficiency

• Real deployments
  – Google’s usage in a Wide Area Network
  – Nicira, acquired by VMware
Software-Defined Networks

The Good
• Simple data plane abstraction
• Logically-centralized controller
• Direct control over switch policies

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

The Ugly
• Non-modular, non-compositional
• Programmer faced with challenging distributed programming problem
Programming the controller

Programming abstractions
Programming the controller

Application modules in software
Stateful Firewall

Let packets from external hosts in, only when spoken to

Internal network

Controller

External world

Let packets from external hosts in, only when spoken to
## Stateful Firewall

### Table: Filter Rules

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inport(2)</td>
<td>Fwd(1)</td>
<td>0</td>
</tr>
</tbody>
</table>
Stateful Firewall

Internal network

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inport(2)</td>
<td>Fwd(1)</td>
<td>0</td>
</tr>
</tbody>
</table>
**Stateful Firewall**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inport(2)</td>
<td>Fwd(1)</td>
<td>0</td>
</tr>
</tbody>
</table>
Stateful Firewall

**Internal network**

- **Pattern**
  - `inport(2)`: Fwd(1), Priority: 0
  - `Srcip(ip1) ^ inport(1)`: Fwd(2), Priority: 0

**Allow**
- `ip1`
Stateful Firewall

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inport(2)</td>
<td>Fwd(1)</td>
<td>0</td>
</tr>
<tr>
<td>Srcip(ip1)^ inport(1)</td>
<td>Fwd(2)</td>
<td>0</td>
</tr>
</tbody>
</table>
1. Flow Identification

# Network Events
flow(dstip=IP), inport=2 --> seen(IP).

- Events: packet-ins, switches and ports go online/offline.
- Flow identification rule

  flow(h1=x1,h2=x2,...), constraints --> rel(x1,x2,...)

- Example:

  flow(srcip=IP, vlan=V), V > 0 --> myvlans(IP,V)
2. Update Controller State

### Information Processing

- A logic program to process the monitored network-events (base facts)
- Has multiple inference rules for deriving new facts
- Two kinds of inference rules

```
# Information Processing
seen(IP) --> allow(IP).
allow(IP) --> allow(IP).
```

```
fact1, fact2, ... --> factn
<factn generated and added to current database>
```

```
fact1, fact2, ... --> factn
<factn added to a database which is used in the next epoch>
```
3. Specifying Policy

```prolog
# Policy Generation
inport(2) |> fwd(1), level(0).

allow(IP) -->
  srcip(IP), inport(1) |> fwd(2), level(0).
```

- Generate a forwarding policy for the switches
  ```prolog
  fact(V1, V2 ...) --> pattern(V1, V2...) |> action, level(i)
  ```

- Gives the pattern, action and the priority for the switch rules
Stateful Firewall

```plaintext
# Network Events
flow(dstip=IP), inport=2 --> seen(IP).

# Information Processing
seen(IP) +--> allow(IP).
allow(IP) +--> allow(IP).

# Policy Generation
inport(2) |> fwd(1), level(0).

allow(IP) -->
srcip(IP), inport(1) |> fwd(2), level(0).
```
What is Flog?

• An event-driven, forward chaining logic programming language
• Has three effects
  – Executed every time a specific network event occurs (epoch)
  – Updates the state (tables) at the controller.
  – Generates a forwarding policy based on the controller state.
• Why logic programming?
  – Good for table-driven collection and processing of network statistics
  – Inspired by success of NDlog, Overlog, Dedalus, Bloom
  – Good for incremental updates to state.
• Specialized Logic Programming in the context of SDNs
Simple Learning Switch
Simple Learning Switch

Controller

* | > flood

ip1

ip2

ip3
Simple Learning Switch

Learn

(ip1, 1)
Simple Learning Switch

Learn

- (Ip1, 1)
- (Ip3, 3)
Simple Learning Switch

# Network Events
flow(scrip=IP, inport=P) --> seen(IP, P)

# Information Processing
seen(IP, P) --> learn(IP, P).
learn(IP, P) --> learn(IP, P).

# Policy Generation
--> flood, level(0).
learn(IP, P) --> dstip(IP) --> fwd(P), level(1)
# Network Events
flow(srcip=IP, inport=P), split(inport) --> seen(IP, P).

# Information Processing
seen(IP, P) --> learn(IP, P).
seen(IP, P), learn(IP’, P’), IP!=IP’ --> learn(IP’,P’).

# Policy Generation
* |> flood, level(0).

seen(IP, P) -->
  dst(IP) |> fwd(P), level(1).

seen(IP, P), learn(IP’, P’), IP!=IP’ -->
  dst(IP’) |> fwd(P’), level(1).
Related Work

- NOX, Beacon: low-level, imperative, event driven
  - install, uninstall forwarding rules directly on the switch
- FML: high-level language for SDN based on Datalog
  - Can mention the kinds of flows to be allowed/denied.
  - not flexible, need to use other languages for stateful computation
- Frenetic provides a combination of
  - (1) a declarative query language with an SQL-like syntax for monitoring packets
  - (2) a functional packet stream-processing language, and
  - (3) a specification language for describing packet forwarding
- Flog - Best of both worlds from FML and Frenetic
Conclusion

- Programming abstractions for Software-Defined Networking
- FLOG - Logic Programming based language for programming SDN controllers
- A Flog program has three important components
  - Network events
  - Information processing
  - Policy generation

Future Work
- Full fledged compiler/run time
- Support for incremental policy updates