Composing Software-Defined Networks

Chris Monsanto*, Joshua Reich*
Nate Foster^, Jen Rexford*, David Walker*

www.frenetic-lang.org/pyretic

Princeton*    Cornell^
Software Defined Networks (SDN)

Enable network innovation by decoupling

**control plane**: determines rules
**data plane**: applies rules to packets
Software Defined Networks (SDN)

Achieved using open standard API
Software Defined Networks (SDN)

Enabling a shift from protocols to applications
Running Example Network

SDN Switch
w/ labeled ports

Internet

Servers

A

B
Programming in OpenFlow

Route: IP/fwd

Counters for each rule
- #bytes, #packets

Priority

<table>
<thead>
<tr>
<th>Rule</th>
<th>Pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>match(dstip=A)</td>
<td>fwd(2)</td>
</tr>
<tr>
<td>1</td>
<td>match(*)</td>
<td>fwd(1)</td>
</tr>
<tr>
<td>2</td>
<td>match(dstip=B)</td>
<td>fwd(3)</td>
</tr>
</tbody>
</table>

1: dstip!=A 
stip!=B

2: dstip=A

3: dstip=B
One API, Many Uses

Priority Ordered

Switch: MAC/fwd

match(dstmac=A)[fwd(2)]
match(dstmac=B)[fwd(3)]
match(*)[fwd(1)]
One API, Many Uses

loadBalancer: IP/mod

match(srcip=0*, dstip=P)[mod(dstip=A)]
match(srcip=1*, dstip=P)[mod(dstip=B)]
But Only Half of the Story

Controller Application

Controller Platform

Hardware Interface

OpenFlow (assembly)

Hardware Flow-Table Rules
(Machine Language)
But Only Half of the Story

**Modular & Intuitive**

![Diagram showing modular & intuitive components](image)

- Monitor
- Route
- FW
- LB

Programmer Interface

Hardware Interface

**Hardware Flow-Table Rules (Machine Language)**

OpenFlow (assembly)

?
OpenFlow Isn’t Modular

Balance then Route

Combined Rules?
(only one match)

Balances w/o Forwarding!
OpenFlow Isn’t Modular

Balance then Route

Combined Rules?
(only one match)

Forwards w/o Balancing!
<table>
<thead>
<tr>
<th>Abstracts</th>
<th>Providing</th>
<th>Supporting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Policy</strong></td>
<td>Compositional Operators</td>
<td>Functional Composition</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>Layered Abstract Topologies</td>
<td>Topological Decomposition</td>
</tr>
<tr>
<td><strong>Packet</strong></td>
<td>Extensible Headers</td>
<td>Policy &amp; Network Abstractions</td>
</tr>
</tbody>
</table>
Compositional Operators: A Monitoring Load Balancer

- Traffic to P re-addressed and forwarded to either A or B, based on source
- Counted if from source X
Module 1:
Balance
Rewrite $\text{dstip} = P$ to $A$, if $\text{srcip} = 0^*$
$B$, if $\text{srcip} = 1^*$

Module 2:
Route
Based on $\text{dstip}$

Module 3:
Monitor
count if $\text{srcip} = X$

Compositional Operators:
A Monitoring Load Balancer
Gateway acts like:

- Legacy router
- Legacy switch

- ARP responder
- Hybrid MAC-rewriter, legacy router/switch
Topology Abstraction:
A Legacy Gateway Replacement

Legacy Ethernet

Module 1: MAC-learn

Module 2: Forward (ARP or MAC Rewrite)

Module 3: IP-Route

Legacy IP Core
Pyretic’s Design

• Monitoring Load Balancer
  – Encode policies as functions
  – Compositional operators
  – Queries as forwarding policies

• MAC-Learning Module
  – Dynamic Policies

• “One Big Switch” Topology Abstraction
  – Extensible packet model
Pyretic Drop Policy

Goal: Drop packets (i.e., OpenFlow drop)
Write: drop
Means: eval(drop,p) = {}

evaluate given policy on packet results in
Pyretic Forward Policy

Goal: Forward packets out port a
Write: fwd(a)
Means: eval(fwd(a), p) = {p[outport:=a]}

located packet w/ fields for
• switch
• inport
• outport
One Pyretic Policy
For Each OpenFlow Action

- drop: 0 packets
- fwd(port): 1 packets
- flood: 0, 1, or more packets
- mod(h=v): 1 packets
Pyretic Policy

A function mapping a located packet to a set of located packets

\[
\text{eval}(\text{policy}, \text{packet}) = \{ \text{packet} \}
\]

Puts focus on *meaning* instead of *mechanics*
Enabling Compositional Operators

Parallel ‘|’:  *Do both C1 and C2 simultaneously*
\[
\text{eval}(C1 \mid C2,p) = \text{eval}(C1,p) \cup \text{eval}(C2,p)
\]

Sequential ‘>>’:  *First do C1 and then do C2*
\[
\text{eval}(C1 >> C2,p) = \bigcup \{\text{eval}(C2,p') \mid p' \in \text{eval}(C1,p)\}
\]

match(dstip=A)[fwd(2)] | match(dstip=B)[fwd(3)] | ~(match(dstip=A) | match(dstip=b))[fwd(1)]

No priorities needed!
Querying as Forwarding

\[
\text{bucket(\text{limit},[h]) \quad \text{count\_bucket(every},[h])}
\]

Abstract location corresponding to a data-structure that store packet-data and callback processing routines

\[
b = \text{count\_bucket(every}=1) \\
b.\text{register\_callback(print)} \\
\text{match(srcip=X)[fwd(b)]}
\]
Monitoring Load Balancer

balance =
  match(srcip=0*, dstip=P)[mod(dstip=A)] |
  match(srcip=1*, dstip=P)[mod(dstip=B)] |
  ~match(dstip=P)[id]

route =
  match(dstip=A)[fwd(2)] |
  match(dstip=B)[fwd(3)] |
  ~(match(dstip=A) | match(dstip=B))[fwd(1)]

b = counts(every=1)
b.register_callback(print)
monitor = match(srcip=X)[fwd(b)]

mlb = (balance >> route) | monitor
Compared to

install_flowmod(5, srcip=X & dstip=P, [mod(dstip=A), fwd(2)])
install_flowmod(4, srcip=0* & dstip=P, [mod(dstip=A), fwd(2)])
install_flowmod(4, srcip=1* & dstip=P, [mod(dstip=B), fwd(3)])
install_flowmod(4, srcip=X & dstip=A, [fwd(2)])
install_flowmod(4, srcip=X & dstip=B, [fwd(3)])
install_flowmod(3, dstip=A, [fwd(2)])
install_flowmod(3, dstip=B, [fwd(3)])
install_flowmod(2, srcip=X, [fwd(1)])
install_flowmod(1, *, [fwd(3)])
Pyretic’s Design

• Monitoring Load Balancer
  – Encode Policies as Functions
  – Compositional Operators
  – Queries as Forwarding Policies

• MAC-Learning Module
  – Dynamic Policies

• “One Big Switch” Topology Abstraction
  – Extensible packet model
How Do We Change Policies?

*Dynamic policy*

a time-series of policies

\[ P_{t=0} \quad P_{t=1} \quad P_{t=3} \]
MAC-Learning Module

class learn():
  def init(self):
    b = bucket(limit=1,[‘srcmac’,’switch’])
    b.register_callback(update)
    self.P = flood | fwd(b)

  def update(self,pkt):
    self.P = if_(match(dstmac=pkt[‘srcmac’],
                  switch=pkt[‘switch’]),
                   fwd(pkt[‘inport’]), self.P)

if_(P,C1,C2) = P[C1] | ~P[C2]

Update current val to flood

If newly learned MAC

Forward directly to learned port

Otherwise, policy unchanged
class learn():
    def init(self):
        b = bucket(limit=1,['srcmac','switch'])
        b.register_callback(update)
        self.P = flood | fwd(b)

    def update(self, pkt):
        self.P = if_(match(dstmac=pkt['srcmac'],
                          switch=pkt['switch'],
                          fwd(pkt['inport']), self.P)
                          P, self.P)
Pyretic’s Design

• Monitoring Load Balancer
  – Encode Policies as Functions
  – Compositional Operators
  – Queries as Forwarding Policies

• MAC-Learning Module
  – Dynamic Policies

• “One Big Switch” Topology Abstraction
  – Extensible packet model
## Extensible Pyretic Packet Model

<table>
<thead>
<tr>
<th>Field</th>
<th>Val[0]</th>
<th>Val[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcmac</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dstmac</td>
<td></td>
<td></td>
</tr>
<tr>
<td>proto</td>
<td></td>
<td></td>
</tr>
<tr>
<td>srcip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inport</td>
<td></td>
<td></td>
</tr>
<tr>
<td>outport</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- All OpenFlow fields
- Location fields
- Virtual fields
- Stacks of values
  - push(h=v)
  - pop(h)
  - Actions and matches use (currently) top value

Implemented on OpenFlow by mapping extended field values to VLAN tags/MPLS labels
“One Big Switch” Topology Abstraction

- Simplest of topology abstraction examples
- Build a distributed middlebox by running centralized middlebox app on V!
def abstract(ingress, fabric, egress, derived):

Returns a new policy for the underlying network (i.e., on nodes S and T) that “does” the derived policy on the abstract topology (i.e., on node V) using 3 partial transformation policies to:

- handle packets entering abstract switch
- move packets through abstract switch
- handle packets exiting abstract switch
Implementing abstract()

def abstract(ingress, fabric, egress, derived):
    return ingress >> derived >> lower_packet >> fabric >> egress
    # defines part of transform
    # app run on abstract topo
    # built-in
    # defines part of transform
    # defines part of transform
    # defines part of transform

<table>
<thead>
<tr>
<th>Field</th>
<th>$V_0$</th>
<th>$V_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch</td>
<td>⌚</td>
<td>V</td>
</tr>
<tr>
<td>inport</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>vinport</td>
<td>⌚</td>
<td></td>
</tr>
<tr>
<td>voutport</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Summary: Pyretic Policy Syntax
(You may already be a Pyretic programmer!)

8 Actions
\[ A ::= \text{drop} \mid \text{fwd(port)} \mid \text{flood} \mid \text{mod(h=v)} \mid \text{id} \mid \text{push(h=v)} \mid \text{pop(h)} \mid \text{move(h1=h2)} \]

6 Predicates
\[ P ::= \text{all_packets} \mid \text{no_packets} \mid \text{match(h=v)} \mid \text{P} \& \text{P} \mid (\text{P} \mid \text{P}) \mid \neg \text{P} \]

2 Query Buckets
\[ B ::= \text{bucket(limit,[h])} \mid \text{count_bucket(every,[h])} \]

5 Policies
\[ C ::= A \mid \text{fwd(B)} \mid P[C] \mid (C \mid C) \mid C \gg C \]
## Summary: Abstractions

<table>
<thead>
<tr>
<th>Policy</th>
<th>Pyretic</th>
<th>Current APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rich Composition</td>
<td>Little Composition</td>
</tr>
<tr>
<td>Network</td>
<td>Layered Abstract</td>
<td>Concrete Network</td>
</tr>
<tr>
<td></td>
<td>Topologies</td>
<td></td>
</tr>
<tr>
<td>Packet</td>
<td>Extensible Headers</td>
<td>Fixed OpenFlow Headers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Related Work:

[Frenetic, Maestro, FRESCO] / [Click]

<table>
<thead>
<tr>
<th>Policy</th>
<th>Pyretic</th>
<th>Current APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rich Composition</td>
<td>Some / Full Composition</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
<th>Layered Abstract Topologies</th>
<th>Concrete Network</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Packet</th>
<th>Extensible Headers</th>
<th>Fixed OpenFlow Headers</th>
</tr>
</thead>
</table>

But only for a single software switch not multiple hardware switches
Related Work:
[FlowVisor] / [Nicira NVP, OpenStack Quantum]

<table>
<thead>
<tr>
<th>Policy</th>
<th>Pyretic</th>
<th>Current APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rich Composition</td>
<td>Little Composition</td>
</tr>
<tr>
<td>Network</td>
<td>Layered Abstract Topologies</td>
<td><strong>Disjoint Slices / Topology Hiding</strong></td>
</tr>
<tr>
<td>Packet</td>
<td>Extensible Headers</td>
<td>Fixed OpenFlow Headers</td>
</tr>
</tbody>
</table>

Both approaches support multi-tenancy, but not topological decomposition (of functional composition)
Pyretic Interpreter and Suite of Apps
Available at [www.frenetic-lang.org/pyretic](http://www.frenetic-lang.org/pyretic)

- Monitoring & DPI
- Load Balancers
- Hub
- ARP
- Firewalls
- MAC learner

• Abstractions
  - Big switch (one-to-many)
  - Spanning tree (many-to-many)
  - Gateway (many-to-one)

And bigger applications built by combining these.
And More!
Available at www.frenetic-lang.org/pyretic

• Features Request
• Bug reporting
• Link to github
• Discuss list
• Join the project

• Dev Roadmap
  – Reactive (microflow) runtime
  – Proactive (compilation) runtime
  – Optimizations
  – Caching
Thanks for Listening!