Abstractions For
Software-Defined Networks

Nate Foster
Cornell

Jen Rexford & David Walker
Princeton
Software-Defined Networking

The Good
- Logically-centralized architecture
- Direct control over the network
Software-Defined Networking

**The Good**
- Logically-centralized architecture
- Direct control over the network

**The Bad**
- Low-level programming interfaces
- Functionality derived from hardware
Software-Defined Networking

The Good
- Logically-centralized architecture
- Direct control over the network

The Bad
- Low-level programming interfaces
- Functionality derived from hardware

The Ugly
- Program pieces don't compose
- Many distributed systems challenges
Programming Abstractions
Programming abstractions are crucial for achieving the vision of software-defined networking.
Programming Abstractions

Benefits
- Modularity
- Portability
- Efficiency
- Assurance

Programming abstractions are crucial for achieving the vision of software-defined networking.
SDN Basics
• Architecture
• Programming model

Network-Wide Abstractions
• Global network view
• Network updates

Modularity
• Composing programs
• Declarative policies and queries

Vision
• Challenges
• Opportunities
SDN Basics
**Switches**

**Table:** prioritized list of rules

**Rule:** pattern, actions, and counters

**Pattern:** prefix match on headers

**Action:** forward or modify

**Counters:** total bytes and packets processed

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
<th>Bytes</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>Drop</td>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>010*</td>
<td>Forward(2)</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>011*</td>
<td>Controller</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Controllers

NOX

Network Events
- Topology changes
- Diverted packets
- Traffic statistics

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

- NOX
- Beacon
- Floodlight
- Trema
- ONIX
- POX

**NOX**

**Network Events**
- Topology changes
- Diverted packets
- Traffic statistics

**Control Messages**
- Install rule
- Uninstall rules
- Query counters
Example: Reactive Applications
Example: Reactive Applications

Network Event
Forwarding table miss
Example: Reactive Applications

Application
Calculates new rules
Example: Reactive Applications

Control Messages
(Un)install rules
Example: Reactive Applications

Subsequent Packets
Processed in fast path

Of course, purely proactive applications also possible
Network-Wide Abstractions
Network-Wide Abstractions

“Holy grail” of network management
Write one program that specifies the behavior of the whole network

- Packet forwarding
- Traffic monitoring
- Access control
Network-Wide Abstractions

**Slogan:** configuration = function(view)

**NOX**
- Global network view
- Eventual consistency
Network-Wide Abstractions

**Slogan**: configuration = function(view)

- **NOX**
  - Global network view
  - Eventual consistency

- **ONIX**
  - Network information base (NIB)
  - Controller handles replication

![Diagram showing application, NIB, and controller layers with physical network](image)
Network-Wide Abstractions

**Slogan:** configuration = function(view)

- **NOX**
  - Global network view
  - Eventual consistency

- **ONIX**
  - Network information base (NIB)
  - Controller handles replication

- **POX and others**
  - Network Object Model (NOM)
  - Can write programs that create virtual network elements
Network Updates

We said configuration = function(view)...
...what happens when the view changes?

Network Updates
- Routine maintenance
- Unexpected failures
- Traffic engineering
- Changes to ACLs
Network Updates

We said configuration = function(view)...
...what happens when the view changes?

Network Updates
- Routine maintenance
- Unexpected failures
- Traffic engineering
- Changes to ACLs

Desired Invariants
- No lost packets
- No broken connections
- No forwarding loops
- No security holes
Abstractions for Network Update

**Challenges**

- The network is a distributed system
- Can only update one element at a time
- Very easy to make mistakes
Abstractions for Network Update

Challenges

- The network is a distributed system
- Can only update one element at a time
- Very easy to make mistakes

At 12:47 AM PDT on April 21st, a network change was performed as part of our normal scaling activities...

The traffic shift was executed incorrectly and the traffic was routed onto the lower capacity redundant network. This led to a “re-mirroring storm”...

The trigger for this event was a network configuration change.
Abstractions for Network Update

Challenges

- The network is a distributed system
- Can only update one element at a time
- Very easy to make mistakes

Possible Approaches

1. Programmer specifies update protocol
2. Controller provides an abstraction
   \texttt{update(config)}
   with “reasonable” semantics

At 12:47 AM PDT on April 21st, a network change was performed as part of our normal scaling activities...

The traffic shift was executed incorrectly and the traffic was routed onto the lower capacity redundant network. This led to a “re-mirroring storm”...

The trigger for this event was a \texttt{network configuration change}.\[1\]
**Abstractions for Network Update**

**Atomic Updates**
- Seem sensible...
- ...but are costly to implement...
- ...and reasoning about effects on in-flight packets is hard!
**Atomic Updates**
- Seem sensible...
- ...but are costly to implement...
- ...and reasoning about effects on in-flight packets is hard!

**Per-Packet Consistent Updates**
Every packet processed with the old configuration or the new configuration, but not a mixture of the two
Atomic Updates

• Seem sensible...
• ...but are costly to implement...
• ...and reasoning about effects on in-flight packets is hard!

Per-Packet Consistent Updates

Every packet processed with the old configuration or the new configuration, but not a mixture of the two

Per-Flow Consistent Updates

Every packet in the same flow processed with old or new configuration, but not a mixture of the two
Consistent Updates in Action

Security Policy

<table>
<thead>
<tr>
<th>Src</th>
<th>Traffic</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>🌐 Web</td>
<td>Allow</td>
<td></td>
</tr>
<tr>
<td>🍄 Non-web</td>
<td>Drop</td>
<td></td>
</tr>
<tr>
<td>🍀 Any</td>
<td>Allow</td>
<td></td>
</tr>
</tbody>
</table>
Consistent Updates in Action

Security Policy

<table>
<thead>
<tr>
<th>Src</th>
<th>Traffic</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>🧢</td>
<td>Web</td>
<td>Allow</td>
</tr>
<tr>
<td>🧢</td>
<td>Non-web</td>
<td>Drop</td>
</tr>
<tr>
<td>🧢</td>
<td>Any</td>
<td>Allow</td>
</tr>
</tbody>
</table>

Configuration A

Process black-hat traffic on F1
Process white-hat traffic on {F2,F3}
Consistent Updates in Action

Security Policy

<table>
<thead>
<tr>
<th>Src</th>
<th>Traffic</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>Allow</td>
<td></td>
</tr>
<tr>
<td>Non-web</td>
<td>Drop</td>
<td></td>
</tr>
<tr>
<td>Any</td>
<td>Allow</td>
<td></td>
</tr>
</tbody>
</table>

Configuration A

- Process black-hat traffic on F1
- Process white-hat traffic on {F2,F3}
Consistent Updates in Action

Security Policy

<table>
<thead>
<tr>
<th>Src</th>
<th>Traffic</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>🧢</td>
<td>Web</td>
<td>Allow</td>
</tr>
<tr>
<td>🧢</td>
<td>Non-web</td>
<td>Drop</td>
</tr>
<tr>
<td>🧢</td>
<td>Any</td>
<td>Allow</td>
</tr>
</tbody>
</table>

Configuration A
- Process black-hat traffic on F1
- Process white-hat traffic on {F2,F3}

Configuration B
- Process black-hat traffic on {F1,F2}
- Process white-hat traffic on F3
# Configuration A

I = {IN_PORT:1, IN_PORT:2, IN_PORT:3, IN_PORT:4, IN_PORT:5, IN_PORT:6}

# Configuration B

I_configB = [Rule({IN_PORT:1}, [forward(5)]),
             Rule({IN_PORT:2}, [forward(6)]),
             Rule({IN_PORT:3}, [forward(7)]),
             Rule({IN_PORT:4}, [forward(7)])]

F1_configB = [Rule({TP_DST:80}, [forward(2)]),
               Rule({TP_DST:22}, [])]

F2_configB = [Rule({TP_DST:80}, [forward(2)]),
               Rule({TP_DST:22}, [])]

F3_configB = [Rule({}, [forward(2)])]

cfgB = {I: SwitchConfiguration(I_configB),
        F1: SwitchConfiguration(F1_configB),
        F2: SwitchConfiguration(F2_configB),
        F3: SwitchConfiguration(F3_configB)}

# Main Function

topo = Topo(...)
update(configA, topo)
...wait for traffic load to shift...
update(configB, topo)

<table>
<thead>
<tr>
<th>Src</th>
<th>Traffic</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>Web</td>
<td>Allow</td>
</tr>
<tr>
<td>Non-web</td>
<td>Non-web</td>
<td>Drop</td>
</tr>
<tr>
<td>Any</td>
<td>Any</td>
<td>Allow</td>
</tr>
</tbody>
</table>
One abstraction, many implementations

**Composition principles**
- Combine updates, preserve consistency

**Two-phase commit**
- Construct versioned internal and edge configurations
  - Phase 1: Install internal configuration
  - Phase 2: Install edge configuration

**Pure Extension**
- Update strictly adds paths

**Pure Retraction**
- Update strictly removes paths

**Slice Update**
- Update only affects a few switches
Network Updates, Formally

\[ \langle C, Q \rangle \xrightarrow{u} \langle C', Q' \rangle \]
Theorem

An update \( u \) from \( C_1 \) to \( C_2 \) is per-packet consistent if and only if it preserves all properties satisfied by \( C_1 \) and \( C_2 \).
Network Updates, Formally

Theorem
An update $u$ from $C_1$ to $C_2$ is per-packet consistent if and only if it preserves all properties satisfied by $C_1$ and $C_2$. 

Verified
Corollary
To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it.
Corollary
To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it.

Network Model
**Corollary**

To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it.
Corollary
To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it.
Verification

**Corollary**
To verify that a property is invariant across an update, simply check that the old and new configurations both satisfy it.

**Properties**
- Connectivity
- Loop freedom
- Blackhole freedom
- Access control
- Waypointing
- Totality
Modularity
Composing Programs

Many applications decompose naturally into components
Composing Programs

Many applications decompose naturally into components
Composing Programs

Many applications decompose naturally into components

Want to write these components once, and use them many times...
Composing Programs

Many applications decompose naturally into components

Want to write these components once, and use them many times...

...but this is difficult to achieve using current controllers

- Network events processed by each component (in some specified order)
- May either propagate or suppress each event
- Components manipulate switch state directly
- State generated by one component can be accessed by others
Modularity Problems

- Forward from port 1 to 2 and port 2 to 1
- Monitor incoming web traffic
Modularity Problems

Forward from port 1 to 2 and port 2 to 1

Monitor incoming web traffic
Modularity Problems

Problems
- Repeater rules too coarse grained
- Monitoring rules don’t forward
Example: Repeater + monitor

Repeater

def switch_join(switch):
    # Repeat Port 1 to Port 2
    p1 = {in_port:1}
    a1 = [forward(2)]
    install(switch, p1, DEFAULT, a1)

    # Repeat Port 2 to Port 1
    p2 = {in_port:2}
    a2 = [forward(1)]
    install(switch, p2, DEFAULT, a2)

When a switch joins the network, install two rules
Example: Repeater + monitor

Web Monitor

```python
def switch_join(switch):
    # Web traffic from Internet
    p = {inport:2,tp_src:80}
    install(switch, p, DEFAULT, [])
    query_stats(switch, p)

def stats_in(switch, p, bytes,...)
    print bytes
    sleep(30)
    query_stats(switch, p)
```

When a switch joins the network, install a monitoring rule
Example: Repeater + monitor

```python
def switch_join(switch):
    p1 = {inport:1}
    a1 = [forward(2)]
    install(switch, pat1, DEFAULT, None, a1)
    p2 = {inport:2}
    pat2web = {in_port:2, tp_src:80}
    a2 = [forward(1)]
    install(switch, pat2web, HIGH, None, a2)
    install(switch, pat2, DEFAULT, None, a2)
    query_stats(switch, pat2web)

def stats_in(switch, p, bytes, ...):
    print bytes
    sleep(30)
    query_stats(switch, p)
```

Must think about both tasks at the same time
Example: Repeater + monitor

Repeater + Web Monitor

def switch_join(switch):
    p1 = {inport: 1}
    a1 = [forward(2)]
    install(switch, pat1, DEFAULT, None, a1)
    p2 = {inport: 2}
    pat2web = {in_port: 2, tp_src: 80}
    a2 = [forward(1)]
    install(switch, pat2web, HIGH, None, a2)
    install(switch, pat2, DEFAULT, None, a2)
    query_stats(switch, pat2web)

def stats_in(switch, p, bytes, ...):
    print bytes
    sleep(30)
    query_stats(switch, p)

Must think about both tasks at the same time
Network Programming Language
- Streaming functional language—no events!
- Declarative semantics
- Separates reads (queries) from writes (policy)

Compiler and Run-time System
- Translates high-level programs to switches
- Automatically manages low-level resources
Frenetic By Example

### Repeater

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

def repeater():
    return \n    (SwitchJoin() >>
     Lift(lambda s:{s:policy}))
```

Policies have a declarative semantics that is independent of other program pieces.
# Frenetic By Example

## Repeater

```python
code
policy = [Rule(inport_fp(1), [forward(2)]), Rule(inport_fp(2), [forward(1)])]
def repeater():
    return \
    (SwitchJoin() >>
     Lift(lambda s:{s:policy}))
```

## Web Monitor

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

Queries have a declarative semantics that is independent of other program pieces.
Frenetic By Example

Repeater

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

def repeater():
    return \ (SwitchJoin()
            \ Lift(lambda s:{s:policy}))
```

Web Monitor

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

Repeater + Web Monitor

```
def main():
    web_query() >> Print()
    repeater() >> Register()
```

Program pieces compose
Frenetic System Overview

High-level Language
- Declarative policies
- Integrated queries
- Effective support for composition

Compiler and Run-time System
- Translates policies and queries
- Manages forwarding rules
- Tracks statistics
- Handles asynchronous events
Vision
(and Challenges)
I call it my billion-dollar mistake.

It was the invention of the null reference in 1965.

My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.

This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
Programming Language Abstractions

Many high-profile mistakes!

- Polymorphism + references
- Bounded quantification
- Pretty much every C compiler :-)

Abstract

We claim that Csmith is an effective bug-finding tool in part because it can be used to explore the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the morass of trade-offs between compilation speed, code quality, code size, and debuggability, compiler modularity, compiler retargetability, and other goals. It should be no surprise that optimizing compilers—like miscompilation tools—generate code—can encounter bugs quite frequently. Developers who stray outside the well-tested paths that represent atypical code are at risk of developing software that miscompiles valid inputs. As measured by the responses to our bug reports, the defects discovered by Csmith are important. Most of the bugs we have reported against GCC and LLVM have been fixed, but we have also found bugs in a variety of other compilers including GCC, LLVM, and commercial tools.

Figure 1.

int foo (void) {
    unsigned char y = 255;
    signed char x = 1;
    return x > y;
}

We found a bug in the version of GCC that shipped with Ubuntu Linux 8.04.1 for x86. At all optimization levels it compiles this function to return 1; the correct result is 0. The Ubuntu compiler had a bug that caused it to generate code that was incorrect for all optimization levels. We were able to use Csmith to find this bug in under 30 minutes.

1. Introduction

We have found in open-source C compilers.

Keywords: compiler testing, compiler defect, automated testing, random testing, random program generation

Categories and Subject Descriptors, Processors—compilers Languages—C

General Terms, Languages, Reliability
Many high-profile mistakes!

- Polymorphism + references
- Bounded quantification
- Pretty much every C compiler :-(

So language researchers have developed a body of techniques for modeling and reasoning precisely about language abstractions

- Operational semantics
- Denotational semantics
- Axiomatic semantics
- Bisimulations

\[
\begin{align*}
\langle \sigma, c \rangle & \models \phi \\
[e] & \quad P \sim P' \\
e \to e' & \quad e \downarrow v \\
\Gamma \vdash e : \tau
\end{align*}
\]
Many high-profile mistakes!

- Polymorphism + references
- Bounded quantification
- Pretty much every C compiler :-)

So language researchers have developed a body of techniques for modeling and reasoning precisely about language abstractions

- Operational semantics
- Denotational semantics
- Axiomatic semantics
- Bisimulations

Proving “obvious” theorems often reveals bugs

Writing down a semantics is an efficient way to communicate ideas

A lot of effort has gone into making these techniques scalable!
Opportunities and Challenges
Opportunities and Challenges

SDNs offer a unique opportunity to

• Define new abstractions for networks
• Develop their mathematical properties
• Design efficient implementations
• Deploy verification tools that provide assurance

and avoid (the analogues of) Hoare’s mistake!
Opportunities and Challenges

SDNs offer a unique opportunity to

• Define new abstractions for networks
• Develop their mathematical properties
• Design efficient implementations
• Deploy verification tools that provide assurance and avoid (the analogues of) Hoare’s mistake!

Challenge #2

• Want to program virtual networks
• Slices? Logical forwarding plane?
• Want to validate implementations, prove isolation properties
Opportunities and Challenges

SDNs offer a unique opportunity to

• Define new abstractions for networks
• Develop their mathematical properties
• Design efficient implementations
• Deploy verification tools that provide assurance
  and avoid (the analogues of) Hoare’s mistake!

Challenge #1
• Combining conflicting policies
• Constraint-based policies?
• FML [Hinrichs+ ’09] and Cologne [Liu+ ’12]

Challenge #2
• Want to program virtual networks
• Slices? Logical forwarding plane?
• Want to validate implementations, prove isolation properties
Thank You!

Collaborators
Shrutarshi Basu (Cornell)
Mike Freedman (Princeton)
Stephen Gutz (Cornell)
Rob Harrison (West Point)
Chris Monsanto (Princeton)
Joshua Reich (Princeton)
Mark Reitblatt (Cornell)
Emin Gün Sirer (Cornell)
Cole Schlesinger (Princeton)
Alec Story (Cornell)
Jen Rexford (Princeton)
David Walker (Princeton)

Funding

http://frenetic-lang.org