Limitations of SDNs (and OpenFlow in particular)

- We have seen that scalability is a problem…
  - Generally solvable with: more resources, better distribution of tasks among resources

- A more fundamental issue: lack of expressiveness
What is OpenFlow about?

• It should be noted that OpenFlow never promised to make centralized network management easy.

• The promise of OpenFlow is to make centralized network management possible in a standardized, vendor-agnostic fashion.

• Besides providing the abstraction of a centralized control plane, however, OpenFlow does not offer much.
OpenFlow capabilities

- OpenFlow switches’ API enables creation/removal of entries in match/action tables.

- Each row in a match/action table specifies a set of conditions on packet headers, and one action to be performed on packets that match.

- Actions: forward, drop, send to domain controller.
### Match/action tables

<table>
<thead>
<tr>
<th>MAC src</th>
<th>MAC dst</th>
<th>IP src</th>
<th>IP dst</th>
<th>TCP port src</th>
<th>TCP port dst</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>192.168.1.12</td>
<td>192.168.1.2</td>
<td>*</td>
<td>80</td>
<td>DROP</td>
</tr>
</tbody>
</table>

**Conditions on packet header fields (match)** (note: many more fields can be specified; only a few shown for simplicity)
Mechanism vs Policy

• OpenFlow switches and their ability to query domain controllers provide a **mechanism** for network management

• However, they do not provide a **policy**…

• …or even an easy way to specify one!
Network programming

• The task of defining the behavior of an entire network by specifying policies is broadly referred to as **network programming**

• **Policy-based networking** is also used (the former refers to the mean to accomplish the latter :-))

• Policies can be specified…

  • …implicitly, by telling the network what to do with each type of packet (limited scalability)

  • …explicitly, by defining who can (or cannot) talk to who, which types of traffic should be inspected by IDS/proxy, etc. **(requires more complex tooling!)**
Some terminology

Configure match/action table entries

Configure network policies

(Source: Floodlight/Big Switch Networks)
Network programming, circa 2010

- NOX: one of the most popular OpenFlow controllers
- Uses Python-based API
- Its “northbound API” is a set of functions to configure match/action tables, but not much more
- In practice, a (very) thin layer of abstraction on top of native features of OpenFlow switches
Using NOX - example

Example: configure the switch in this topology:

```
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)])
```

(Received traffic on physical port 1 is rebroadcast to the controller via port 2. The controller installs a rule to forward packets from port 1 to port 2.)

(Source: Foster et al., ICFP 2011)
Limitations of NOX

• Various, but mainly:

  • Programming interface is very low-level

  • It forces programmer to reason about computation concurrently happening on controller and switches (two-tiered architecture)

  • It does not prevent or take into account race condition (mostly due to multiple packets of a flow being forwarded to the controller after a flow action has been established)

  • Flow policies are not composable
Low-level programming interface

- Mostly a set of callbacks triggered on events:
  - `packet_in, stats_in, switch_join, etc.`

- And a set of primitives to perform operations on forwarding tables:
  - `install, uninstall, etc.`

- No facilities to perform high-level operations such as logging, load-balancing, etc. - user must implement them manually or use different tools!
Two-tiered architecture

- Simple, static operations run entirely on switches
  - Example: repeater from a few slides ago
  - Controller involvement is limited to installing rules (and removing them when no longer necessary)
- What about **dynamic operations**?
  - E.g., “count packets for each new host”
  - Requires cooperation between controller and switch
Two-tiered architecture - II

Example (from Foster et al., ICFP 2011): repeater + count packets arriving at port 2 by source host:

```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)
```

If a packet from a new host is seen...

...then create a new pattern matching it...

...then repeat all packets matching the pattern on port 1...

...but also keep track of their stats

This part of the policy runs on the controller, while the actual statistics gathering/forwarding runs on the switch!
Two-tiered architecture - III

- Why is it a problem?
  - Forces programmer to have to reason about a small (or large?) distributed system:
    - Part of the computation runs on the switch(es)
    - Part of the computation runs on the controller
  - Difficult, makes programmer errors more likely
Race conditions

• Problem:

• Additional complexity: application of policy decision must be idempotent

  • An operation is idempotent if it can be applied multiple times without changing the result beyond the initial application
Non-composability

• In general, network administrator may want to apply multiple policies at the same time

  • Common example: enforce forwarding policy while gathering statistics

• NOX API does not transparently allow this

  • I.e., performing policy application $P_1$ and then $P_2$ does not yield $P_1 \circ P_2$

• Programmer must manually compose policies
Better network programming

- Simple, low-level APIs such as the one provided by NOX are limited and risk hindering OpenFlow adoption

- In order for OpenFlow to truly simplify network management, higher-level interfaces are necessary

- Ideally, administrator should specify policy and/or intent, and not functional details

- Frenetic does not quite get there, but significantly simplifies things
Abstract

Modern networks provide a variety of interrelated services including routing, traffic monitoring, load balancing, and access control. Unfortunately, the languages used to program today’s networks lack modern features—they are usually defined at the low level of abstraction supplied by the underlying hardware and they fail to provide even rudimentary support for modular programming. As a result, network programs tend to be complicated, error-prone, and difficult to maintain.

This paper presents Frenetic, a high-level language for programming distributed collections of network switches. Frenetic provides a declarative query language for classifying and aggregating network traffic as well as a functional reactive combination library for describing high-level packet-forwarding policies. Unlike prior work in this domain, these constructs are—by design—fully compositional, which facilitates modular reasoning and enables code reuse. This important property is enabled by Frenetic’s novel runtime system which manages all of the details related to installing, uninstalling, and querying low-level packet-processing rules on physical switches.

Overall, this paper makes three main contributions: (1) We analyze the state-of-the-art in languages for programming networks and identify the key limitations; (2) We present a language design that addresses these limitations, using a series of examples to motivate and validate our choices; (3) We describe an implementation of the language and evaluate its performance on several benchmarks.

Categories and Subject Descriptors: D.3.2 [Programming Languages]: Language Classifications—Specialized application languages

General Terms Languages, Design

Keywords Network programming languages, domain-specific languages, functional reactive programming, OpenFlow

1. Introduction

Today’s networks consist of hardware and software components that are closed and proprietary. The difficulty of changing these components has had a chilling effect on innovation, and forced network administrators to express policies through complicated and frustratingly brittle interfaces. As discussed in recent New York Times article [30], the rise of data centers and cloud computing have brought these problems into sharp relief and led a number of network researchers to reconsider the fundamental assumptions that underpin today’s network architectures.

In particular, significant momentum has gathered behind OpenFlow, a new platform that opens up the software that controls the network while also allowing packets to be processed using fast, commodity switching hardware [31]. OpenFlow defines a standard interface for installing flexible packet-forwarding rules on physical network switches using a programmable controller that runs separately on a stock machine. The most well-known controller platform is NOX [20], though there are several others [1, 8, 25, 39]. OpenFlow is supported by a number of commercial Ethernet switch vendors, and has been deployed in several campus and backbone networks. Using OpenFlow, researchers have already created a variety of controller applications that introduce new network functionality, like flexible access control [9, 33], Web server load balancing [21, 40], energy-efficient networking [22], and seamless virtual-machine migration [38].

Unfortunately, while OpenFlow and NOX now make it possible to implement exciting new network services, they do not make it easy. OpenFlow programmers must constantly grapple with several difficult challenges.

First, networks often perform multiple tasks, like routing, access control, and traffic monitoring. Unfortunately, decoupling these tasks from each other and implementing them independently in separate modules is effectively impossible, since packet-handling rules (un)installed by one module often interfere with overlapping rules (un)installed by other modules.

Second, the OpenFlow/NOX interface is defined at a very low level of abstraction. For example, the OpenFlow rule algebra directly reflects the capabilities of the switch hardware (e.g., bit patterns and integer priorities). Simple high-level concepts such as set difference require multiple rules and priorities to implement correctly and more powerful “wildcard” rules are a limited hardware resource that programmers must manage by hand.

Third, controller programs only receive events for packets the switches do not know how to handle. Code that installs a forwarding rule might prevent another, different event-driven call-back from being triggered. As a result, writing programs for OpenFlow/NOX quickly becomes a difficult exercise in two-level programming—programmers must simultaneously reason about the packets that will be processed on switches and those that will be processed on the controller.

Fourth, because a network of switches is a distributed system, it is susceptible to various kinds of race conditions. For example, a common NOX programming idiom is to handle the first packet of each network flow on the controller and install switch-level rules to handle the remaining packets. However, such programs can be susceptible to errors if the second, third, or fourth packets in a
Frenetic - intro

- Network programming language for OpenFlow networks
- Proposed around 2010
- Addresses many shortcomings of early OpenFlow programming interfaces
- Still does not provide a fully declarative/intent-based solution to the network programming problem
Frenetic - primer

• Broadly divided in a query language and policy component
  • Queries compute stats on network
  • Policy defines network operations (forwarding etc.)

• High-level, does not directly expose match-action tables

• Incorporates elements of functional programming, reactive programming, and relational algebra
Frenetic query language

Example (Foster et al., ICFP 2011):

```python
def host_query():
    return (Select(sizes) *
        Where(inport_fp(2)) *
        GroupBy([dstmac]) *
        Every(60))
```

**Goal:** summarize volume of traffic arriving on switch port 2, grouped by host, every 60 seconds

Operators allow to select data of interest, filter/group them by parameter and/or time interval, etc.
“See-every-packet” abstraction

- In OpenFlow, rules may interact in unexpected ways - remember that each packet can only be mapped to one action!

  - If multiple rows in a match/action table match a packet, only the rule with the highest priority is applied

- **This complicates statistics gathering** (e.g. what if I need to forward all packets w/ TCP dst port 80, but also count the ones coming from 192.168.12.2?)

- Frenetic allows programmer to issue multiple queries/forwarding policies in parallel; the Frenetic runtime takes care of composing them so that they all work

- From the point of view of the programmer, every query can “see every packet” **(no masking effect due to other rules)**
Query cost model

• Most Frenetic queries require controller to analyze one packet per flow

  • Flow: set of packets all having same header values (e.g. one direction of a TCP/IP connection)

• The “cost” of a query can therefore be approximated as the number of different flows that are relevant for a query

• Only works for per-flow statistics!

  • Per-packet statistics gathering requires controller to analyze every packet (advice: don’t do it :-))
Cheap queries vs expensive queries

Cheap: report every time a host connects to a different switch port:

```python
def learning_query():
    return (Select(packets) *
            Where(true_fp()) *
            GroupBy([srcmac]) *
            SplitWhen([inport]) *
            Limit(1))
def connection_printer():
    learning_query() >> Print()
```

Expensive: perform intrusion detection on every packet in the controller:

```python
def web_packets_query():
    return (Select(packets) *
            Where(srcport_fp(80)))
def dpi():
    web_packets_query() >> analyze_packet()
```

(Source: Foster et al., ICFP 2011)
Frenetic policy language

• Fundamentally, Frenetic policy language enables associating rules to patterns

• Compared to the basic NOX interface, Frenetic’s expressive power is considerably higher

• Patterns use a subset of the query language

• A rule associated with a switch forms a network policy
Policy language - example

- Language is based on the concept of event - an infinite stream of values of a certain type
- Forwarding programs are defined as an event source, an event sink, and event functions which transform events along the way
- Example: static rules for repeater functionality (Foster et al., ICFP 2011):

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

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

- Event source: generate a value every time a switch joins the network
- Event function: takes as input a switch ID and returns a policy associating the switch w/ the static rules above
- Event sink: takes as input a policy generated in the line above and registers it with the runtime
Policy language - considerations

• More powerful and complicated than shown here
  • See load balancer examples in the paper

• Based on \textit{reactive programming} principles
  • Network behavior defined as responses of certain types of events

• Significantly more powerful than NOX interface
  • Programmer can easily extract and aggregate data from network flow, build policies which are either static or dynamic (based on information extracted from packets), etc.
How does Frenetic help with…

• Low-level nature of NOX API?
  • Hopefully clear by now :-)
  • Higher-level interface based on functional reactive programming (for policies) and relational algebra (for queries)

• Complications due to two-tier programming?
  • No explicit switch rule management (install etc.) - administrator writes a policy and runtime takes care of generating the appropriate behavior on switches and controller
How does Frenetic help with…

• Race conditions?
  
  • Frenetic filters redundant packet if instructed (just use \textit{Limit(1)} in query, which means that at most one packet matching the query will be returned)
  
  • Complex to achieve, but managed by the runtime w/o programmer involvement

• Non-composability?
  
  • When deploying multiple policies/queries, Frenetic runtime ensures the generated low-level rules do not interfere with each other
Summarizing...

• Frenetic policies provides a somewhat higher-level abstraction than low-level NOX API...

• ...much of its benefit consists in automatically dealing with the biggest pain-points: data races, non-composability, and so on

• Still not an intent-based language: administrators must think in imperative (as opposed to declarative) terms

  • Policies define **how the network should accomplish goals** (“execute operation X in situation Y”, “drop packet with source A and destination B”), more than **what the network should do** (“load-balance input flows based on source IP”, “prevent A from talking to B”)
Other work in this space

• FML (“Practical Declarative Network Management”, Hinrichs et al., WREN 2009)

• Merlin (“Merlin: A Language for Provisioning Network Resources”, Soulé et al., CoNEXT 2014)

• …
The future?

- Northbound API (i.e., the primitives that allow applications/administrators to “tell” the controller what they want) are moving towards **intent-based networking**

- Intent-based networking is surrounded by a lot of hype... but at the core, means “let the administrator define network behavior in a high-level, declarative fashion, and let the network translates it to low-level consideration”

- Example: “reserve 10GB/s between front-end and storage servers”
  - Translated into low-level forwarding constraints by network programming tools

- Very desirable, easy to describe, difficult to achieve! :-(