Software-Defined Networking and Advanced Network Control Programming

6 SDN Programming

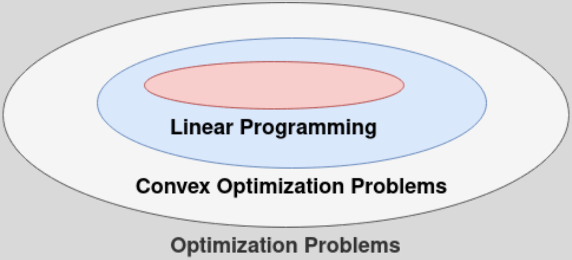
Kai Gao
kaigao@scu.edu.cn

School of Cyber Science and Engineering
Sichuan University

Recap

Types of Optimization Problems

- Convex optimization problem (COP): the objective function and constraint functions are convex
- Linear programming problem (LP): the objective function and constraint functions are linear
- The smaller the scope is, the more restricted the objective and constraint functions are, and more efficient algorithms exist



Standard Form of LP in the Matrix Representation

- decision variables:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}$$

- coefficients of the objective function:

$$\boldsymbol{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_N \end{bmatrix}$$

- coefficients of the constraint functions:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \dots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{M1} & \dots & a_{MN} \end{bmatrix}$$

The problem can be formulated as:

$$\max \boldsymbol{c}^T \boldsymbol{x}$$

$$f(\boldsymbol{x}) = \begin{bmatrix} c_1, & \dots, & c_N \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}$$

subject to:

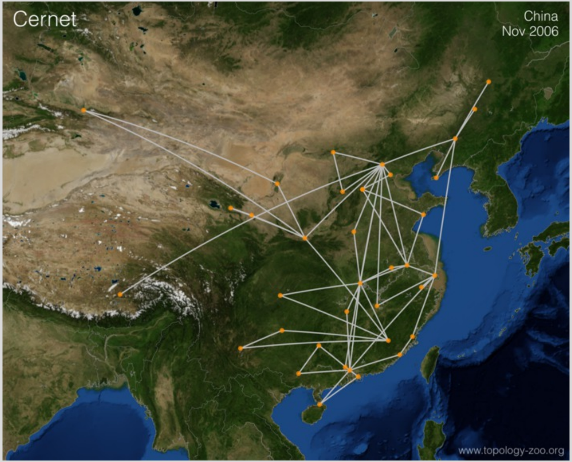
$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

$$\begin{bmatrix} a_{11} & \dots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{M1} & \dots & a_{MN} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_M \end{bmatrix}$$

Traffic Engineering (I)

Traffic engineering is a real networking problem in ISP networks. We consider a simplified problem:

- Assume there are N nodes and M uni-directional links, the i-th link has a source s_i and the destination d_i
- Link capacity $\boldsymbol{c} = (c_i)_M$ where c_i is the capacity for the i-th link
- The traffic matrix $T = (t_{ij})_{N \times N}$: t_{ij} denote the traffic from node i to node j
- Link utilization u_i: the traffic carried by a link divided by the link capacity
- Objective: to minimize the maximum link utilization



http://www.topology-zoo.org/gallery.html

Integer Linear Programming

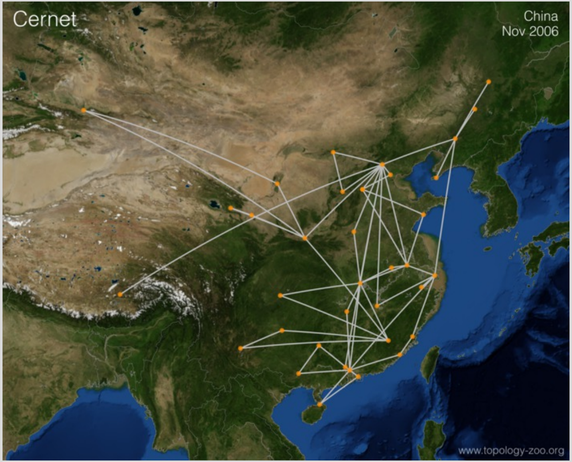
Integer linear programming (ILP) has the same format as an LP with an additional constraint that some decision variables must be integers.

- ILP is NP-hard.
- If some variables are continuous, the problem is known as mixed integer linear programming (MILP)
- If all variables are either 0 or 1, the problem is known as binary linear programming

Traffic Engineering (II)

Now consider the traffic engineering problem using tunnels

- Assume there are N nodes and M uni-directional links, the i-th link has a source s_i and the destination d_i
- Link capacity $\boldsymbol{c} = (c_i)_M$ where c_i is the capacity for the i-th link
- The traffic matrix $T = (t_{ij})_{N \times N}$: t_{ij} denote the traffic from node i to node j
- Link utilization u_i: the traffic carried by a link divided by the link capacity
- Objective: to minimize the maximum link utilization



http://www.topology-zoo.org/gallery.html

本期学习目标

- 了解 SDN 控制编程语言的主要目标、研究对象和组成部分

本期学习目标

- 了解 SDN 控制编程语言的主要目标、研究对象和组成部分
- 了解 Frenetic、Pyretic、Maple 和 Magellan 解决的问题和解决思路

本期学习目标

- 了解 SDN 控制编程语言的主要目标、研究对象和组成部分
- 了解 Frenetic、Pyretic、Maple 和 Magellan 解决的问题和解决思路
- 掌握根据踪迹树生成流表规则的方法

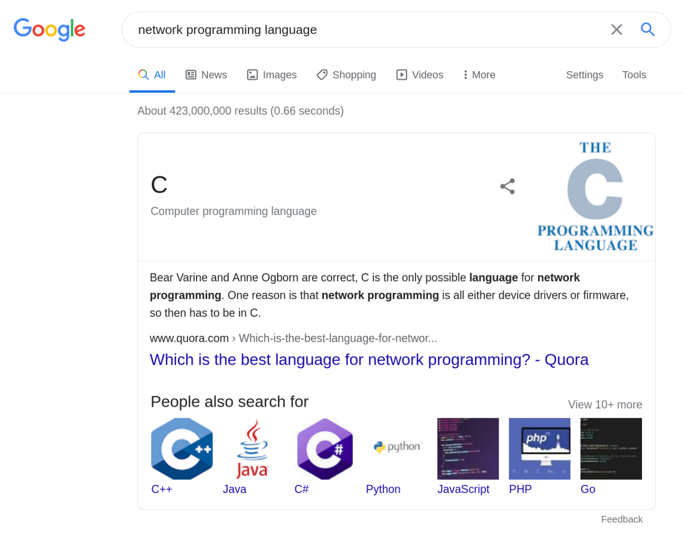
Overview

Network Programming Languages

Network programming languages (网络编程语言) can refer to general-purpose programming languages that have high-performance and/or easy-to-use network/web APIs and libraries.

Examples:

- C
- Java
- Go
- Python
- ...

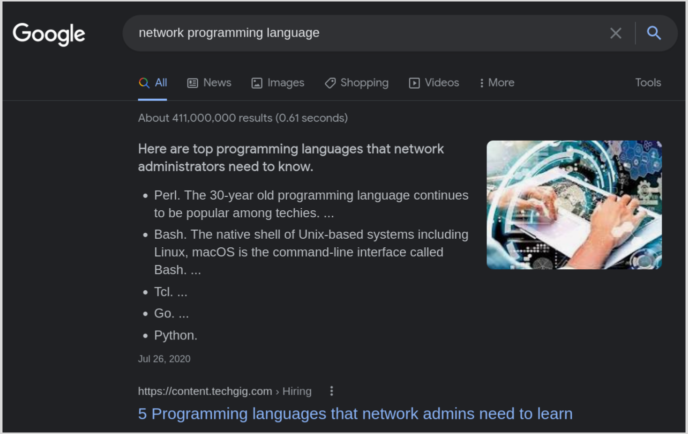


Network Programming Languages

Network programming languages can refer to specialized programming languages that simplify or automate the network operations.

Examples:

- Perl
- Bash
- Tcl
- Python
- ...



Network Programming Language

Network programming language in this course refer to programming languages in the SDN architecture

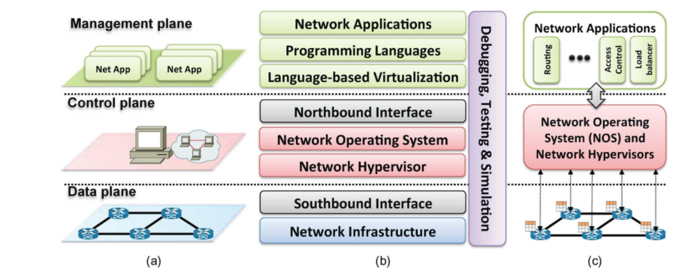


Fig. 6. *Software-Defined Networks in (a) planes, (b) layers, and (c) system design architecture.*

D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76

Why Network Programming Language?

Programming Language from the Computer Architecture's View

Low-level machine language

```
1139:   55                      push   %rbp
113a:   48 89 e5                mov    %rsp,%rbp
113d:   48 83 ec 10             sub    $0x10,%rsp
1141:   c7 45 f8 01 00 00 00    movl   $0x1,-0x8(%rbp)
1148:   c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp)
114f:   8b 55 f8                mov    -0x8(%rbp),%edx
1152:   8b 45 fc                mov    -0x4(%rbp),%eax
1155:   01 d0                   add    %edx,%eax
1157:   89 c6                   mov    %eax,%esi
1159:   48 8d 05 a4 0e 00 00    lea    0xea4(%rip),%rax
1160:   48 89 c7                mov    %rax,%rdi
1163:   b8 00 00 00 00          mov    $0x0,%eax
1168:   e8 c3 fe ff ff          call   1030 <printf@plt>
116d:   b8 00 00 00 00          mov    $0x0,%eax
1172:   c9                      leave
1173:   c3                      ret
1174:   66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
117b:   00 00 00
117e:   66 90                   xchg   %ax,%ax
```

High-level programming language

```c
#include "stdlib.h"
#include "stdio.h"

int main() {
  int a, b;
  a = 1;
  b = 2;
  printf("%d\n", a + b);
  return 0;
}
```

Why Network Programming Language?

Programming languages provide

- **portability** (可移植性) that the same high-level program can be realized on multiple targets
- **abstractions** (编程抽象) that simplify the implementation and maintenance of programs
  - variables, data structures, loops, branches, etc.
- **compilers** (编译器) that transform high-level programming languages into low-level machine languages
  - code optimization, static code analysis, linter, etc.
- **runtime** (运行时) that monitors and manages system behaviors simultaneously
  - memory protection, garbage collection, etc.

```
#include "stdlib.h"
#include "stdio.h"

int main() {
  int a, b;
  a = 1;
  b = 2;
  printf("%d\n", a + b);
  return 0;
}
```

We focus on network programming language

in SDN control plane

Why Network Programming Language

Network programming languages have similar constructs:

- **Targets**: low-level networking configurations & APIs
  - OpenFlow tables, BGP configurations, etc.
- **Abstractions**: high-level concepts and data structures to be programmed
  - network object, predicate, route objects, etc.
- **Compilation**: transforming high-level language to low-level
  - traffic optimization, state placement, etc.
- **Runtime**: monitor system state and update accordingly
  - topology change, policy change, etc.

Why Network Programming Language

Network programming languages have similar constructs:

- **Targets**: low-level networking configurations & APIs
  - OpenFlow tables, BGP configurations, etc.
- **Abstractions**: high-level concepts and data structures to be programmed
  - network object, predicate, route objects, etc.
- **Compilation**: transforming high-level language to low-level
  - traffic optimization, state placement, etc.
- **Runtime**: monitor system state and update accordingly
  - topology change, policy change, etc.

Why Network Programming Language

Network programming languages have similar constructs:

- **Targets**: low-level networking configurations & APIs
  - OpenFlow tables, BGP configurations, etc.
- **Abstractions**: high-level concepts and data structures to be programmed
  - network object, predicate, route objects, etc.
- **Compilation**: transforming high-level language to low-level
  - traffic optimization, state placement, etc.
- **Runtime**: monitor system state and update accordingly
  - topology change, policy change, etc.

Why Network Programming Language

Network programming languages have similar constructs:

- **Targets**: low-level networking configurations & APIs
  - OpenFlow tables, BGP configurations, etc.
- **Abstractions**: high-level concepts and data structures to be programmed
  - network object, predicate, route objects, etc.
- **Compilation**: transforming high-level language to low-level
  - traffic optimization, state placement, etc.
- **Runtime**: monitor system state and update accordingly
  - topology change, policy change, etc.

Why Network Programming Language

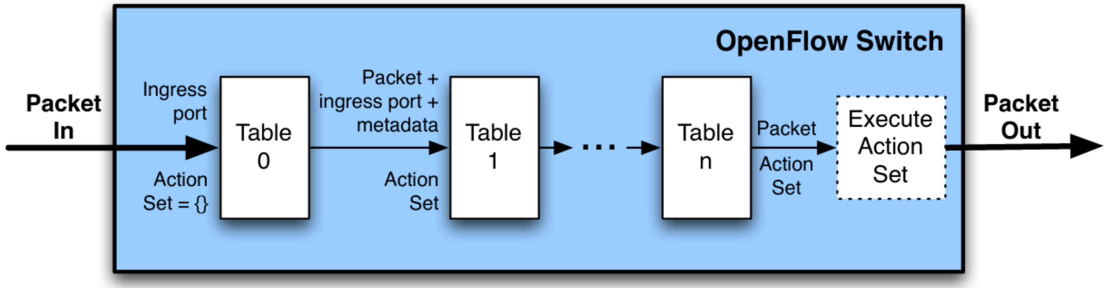
Network programming languages have similar constructs:

- **Targets**: low-level networking configurations & APIs
  - OpenFlow tables, BGP configurations, etc.
- **Abstractions**: high-level concepts and data structures to be programmed
  - network object, predicate, route objects, etc.
- **Compilation**: transforming high-level language to low-level
  - traffic optimization, state placement, etc.
- **Runtime**: monitor system state and update accordingly
  - topology change, policy change, etc.

Why Network Programming Language

Low-level Machine Language in Networking

OpenFlow Tables



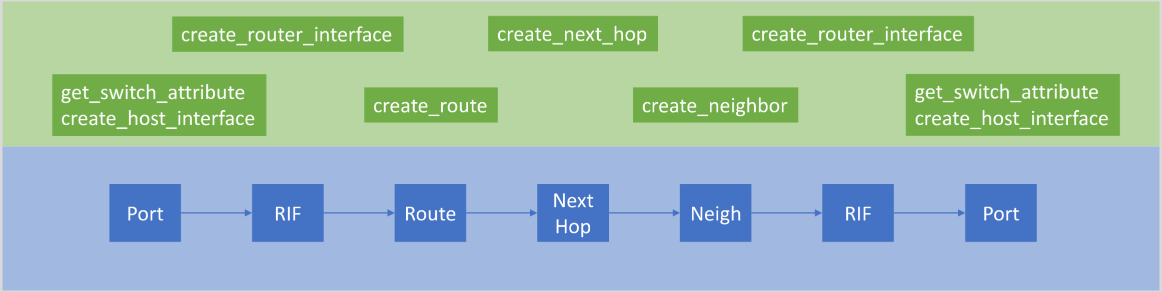
(a) Packets are matched against multiple tables in the pipeline

ONF. *OpenFlow Switch Specification (Version 1.3.5)*. Open Networking Foundation, Mar. 26, 2015, p. 177. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf (visited on 09/05/2021)

Why Network Programming Language

Low-level Machine Language in Networking

SONiC Pipeline



Lihua Yuan. “SONiC: Software for Open Networking in the Cloud”. APNet’18. 2018

Why Network Programming Language

Low-level Machine Language in Networking

BGP Configuration

```
router bgp 300
 network 1.0.0.0
 network 2.0.0.0

 neighbor 10.10.10.10 remote-as 100
 neighbor 10.10.10.10 route-map localonly out

!--- Outgoing policy route-map that filters routes to service provider A (SP-A).

 neighbor 20.20.20.20 remote-as 200
 neighbor 20.20.20.20 route-map localonly out

!--- Outgoing policy route-map that filters routes to service provider B (SP-B).

end
```

https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/23675-27.html

Representative Network Programming Language

Different network programming languages provide different abstractions, compiler, and runtime.

In this lecture, we will cover a few representative network programming languages

- Frenetic
- Pyretic
- Maple

Representative Network Programming Language

Different network programming languages provide different abstractions, compiler, and runtime.

In this lecture, we will cover a few representative network programming languages

- Frenetic
- Pyretic
- Maple

We focus on

- What are the problems?
- How do these languages solve these problems?

Frenetic

Frenetic

Frenetic: A Network Programming Language

Nate Foster
Cornell University

Rob Harrison
Princeton University

Michael J. Freedman
Princeton University

Christopher Monsanto
Princeton University

Jennifer Rexford
Princeton University

Alec Story
Cornell University

David Walker
Princeton University

Nate Foster et al. "Frenetic: A Network Programming Language". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: `http://doi.acm.org/10.1145/2034773.2034812`

Problems

Interactions between Network Programs

Consider an SDN network as below

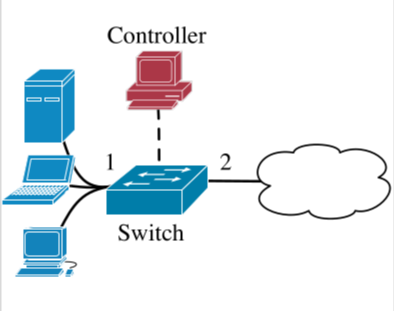


Figure 2. Simple Topology.

Program 1: Simple switching

```
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)])
```

Program 2: Traffic monitoring

```
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)
```


Nate Foster et al. "Frenetic: A Network Programming Language". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Problems

Interactions between Network Programs

What happens if the two programs are applied?

```
def repeater_monitor_wrong(switch):
  repeater(switch)
  monitor(switch)
```

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming.* ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Program 1: Simple switching

```
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)])
```

Program 2: Traffic monitoring

```
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)
```

Problems

Interactions between Network Programs

What happens if the two programs are applied?

```
def repeater_monitor_wrong(switch):
  repeater(switch)
  monitor(switch)
```

match	*priority*	*action*
in_port: 1	DEFAULT	output: 2
in_port: 2	DEFAULT	output: 1

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Program 1: Simple switching

```
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)])
```

Program 2: Traffic monitoring

```
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)
```

Problems

Interactions between Network Programs

What happens if the two programs are applied?

```
def repeater_monitor_wrong(switch):
  repeater(switch)
  monitor(switch)
```

match	*priority*	*action*
in_port: 1	DEFAULT	output: 2
in_port: 2	DEFAULT	output: 1
in_port: 2, tp_src: 80	DEFAULT	(drop)

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Program 1: Simple switching

```
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)])
```

Program 2: Traffic monitoring

```
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)
```

Problems

Interactions between Network Programs

What is the correct composition result?

Problems

Interactions between Network Programs

What is the correct composition result?

match	*priority*	*action*
in_port: 1	DEFAULT	output: 2
in_port: 2	DEFAULT	output: 1
in_port: 2, tp_src: 80	DEFAULT	(drop)

Problems

Interactions between Network Programs

What is the correct composition result?

match	*priority*	*action*
in_port: 1	DEFAULT	output: 2
in_port: 2	DEFAULT	output: 1
in_port: 2, tp_src: 80	HIGH	output:1

Problems

Interactions between Network Programs

What is the correct composition result?

match	*priority*	*action*
in_port: 1	DEFAULT	output: 2
in_port: 2	DEFAULT	output: 1
in_port: 2, tp_src: 80	HIGH	output:1

```
def repeater_monitor(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  pat2web = {in_port:2,tp_src:80}
  install(switch,pat1,[output(2)],DEFAULT)
  install(switch,pat2web,[output(1)],HIGH)
  install(switch,pat2,[output(1)],DEFAULT)
  query_stats(switch,pat2web)
```

Nate Foster et al. "Frenetic: A Network Programming Language". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Problems

Low-level Network Control API

Low-level API requires programmers to manually plan the match fields and priorities (手动设计流表项的匹配域和优先级)

```
def repeater_monitor_noserver(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  pat2web = {in_port:2,tp_src:80}
  pat2srv = {in_port:2,nw_dst:10.0.0.9,tp_src:80}
  install(switch,pat1,DEFAULT,None,[output(2)])
  install(switch,pat2srv,HIGH,None,[output(1)])
  install(switch,pat2web,MEDIUM,None,[output(1)])
  install(switch,pat2,DEFAULT,None,[output(1)])
  query_stats(switch,pat2web)
```

Nate Foster et al. "Frenetic: A Network Programming Language". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Problems

2-tier Programming with Race Conditions

Low-level API forces programmers to write 2-tier programs (both in the data plane and in the control plane)

Such a setting leads to race conditions (竞争条件). Consider two consequent packets:

in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 80
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 8080

```
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)
```

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Problems

2-tier Programming with Race Conditions

Low-level API forces programmers to write 2-tier programs (both in the data plane and in the control plane)

Such a setting leads to race conditions (竞争条件). Consider two consequent packets:

in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 80
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 8080

The flow table:

match	*priority*	*action*

```
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)
```

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Problems

2-tier Programming with Race Conditions

Low-level API forces programmers to write 2-tier programs (both in the data plane and in the control plane)

Such a setting leads to race conditions (竞争条件). Consider two consequent packets:

in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 80
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 8080

The flow table:

match	*priority*	*action*
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B	DEFAULT	output: 1

```
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)
```

Nate Foster et al. "Frenetic: A Network Programming Language". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Problems

2-tier Programming with Race Conditions

Low-level API forces programmers to write 2-tier programs (both in the data plane and in the control plane)

Such a setting leads to race conditions (竞争条件). Consider two consequent packets:

in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 80
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 8080

The flow table:

match	*priority*	*action*
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B	DEFAULT	output: 1
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B	DEFAULT	output: 1

```
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)
```

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Problems

2-tier Programming with Race Conditions

Low-level API forces programmers to write 2-tier programs (both in the data plane and in the control plane)

Such a setting leads to race conditions (竞争条件). Consider two consequent packets:

in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 80
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 8080

The flow table:

match	*priority*	*action*
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B	DEFAULT	output: 1
in_port: 2, dl_dst: C5:85:2D:D6:B6:8B	DEFAULT	output: 1

To enforce correctness, the program must store the state to ignore duplicated packets

```
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)
```

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Frenetic Solutions

Frenetic has the following components

- high-level network query language (网络查询语言)
- general-purpose functional reactive network policy (函数响应式网络策略) management library

Nate Foster et al. "Frenetic: A Network Programming Language". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Frenetic Solutions

Frenetic has the following components

- high-level network query language (网络查询语言)
- general-purpose functional reactive network policy (函数响应式网络策略) management library

Queries	q ::=	Select(a) *
		Where(fp) *
		GroupBy([$qh_1, \ldots, qh_n$]) *
		SplitWhen([$qh_1, \ldots, qh_n$]) *
		Every(n) *
		Limit(n)
Aggregates	a ::=	packets | sizes | counts
Headers	qh ::=	inport | srcmac | dstmac | ethtype |
		vlan | srcip | dstip | protocol |
		srcport | dstport | switch
Patterns	fp ::=	true_fp() | qh_fp(n) |
		and_fp([$fp_1, \ldots, fp_n$]) |
		or_fp([$fp_1, \ldots, fp_n$]) |
		diff_fp(fp_1, fp_2) | not_fp(fp)

Figure 3. Frenetic query syntax

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Frenetic Solutions

Frenetic has the following components

- high-level network query language (网络查询语言)
- general-purpose functional reactive network policy (函数响应式网络策略) management library

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Events

$$\begin{aligned}
\texttt{Seconds} &\in \mathsf{int}\ \mathsf{E}\\
\texttt{SwitchJoin} &\in \mathsf{switch}\ \mathsf{E}\\
\texttt{SwitchExit} &\in \mathsf{switch}\ \mathsf{E}\\
\texttt{PortChange} &\in (\mathsf{switch} \times \mathsf{int} \times \mathsf{bool})\ \mathsf{E}\\
\texttt{Once} &\in \alpha \to \alpha\ \mathsf{E}
\end{aligned}$$

Basic Event Functions

$$\begin{aligned}
\texttt{>>} &\in \alpha\ \mathsf{E} \to \alpha\ \beta\ \mathsf{EF} \to \beta\ \mathsf{E}\\
\mathsf{Lift} &\in (\alpha \to \beta) \to \alpha\ \beta\ \mathsf{EF}\\
\texttt{>>} &\in \alpha\ \beta\ \mathsf{EF} \to \beta\ \gamma\ \mathsf{EF} \to \alpha\ \gamma\ \mathsf{EF}\\
\texttt{ApplyFst} &\in \alpha\ \beta\ \mathsf{EF} \to (\alpha \times \gamma)\ (\beta \times \gamma)\ \mathsf{EF}\\
\texttt{ApplySnd} &\in \alpha\ \beta\ \mathsf{EF} \to (\gamma \times \alpha)\ (\gamma \times \beta)\ \mathsf{EF}\\
\texttt{Merge} &\in (\alpha\ \mathsf{E} \times \beta\ \mathsf{E}) \to (\alpha\ \mathsf{option} \times \beta\ \mathsf{option})\ \mathsf{E}\\
\texttt{BlendLeft} &\in \alpha \times \alpha\ \mathsf{E} \times \beta\ \mathsf{E} \to (\alpha \times \beta)\ \mathsf{E}\\
\texttt{BlendRight} &\in \beta \times \alpha\ \mathsf{E} \times \beta\ \mathsf{E} \to (\alpha \times \beta)\ \mathsf{E}\\
\texttt{Accum} &\in (\gamma \times (\alpha \times \gamma \to \gamma)) \to \alpha\ \gamma\ \mathsf{EF}\\
\texttt{Filter} &\in (\alpha \to \mathsf{bool}) \to \alpha\ \alpha\ \mathsf{EF}
\end{aligned}$$

Listeners

$$\begin{aligned}
\texttt{>>} &\in \alpha\ \mathsf{E} \to \alpha\ \mathsf{L} \to \mathsf{unit}\\
\texttt{Print} &\in \alpha\ \mathsf{L}\\
\texttt{Register} &\in \mathsf{policy}\ \mathsf{L}\\
\texttt{Send} &\in (\mathsf{switch} \times \mathsf{packet} \times \mathsf{action})\ \mathsf{L}
\end{aligned}$$

Rules and Policies

$$\begin{aligned}
\texttt{Rule} &\in \mathsf{pattern} \times \mathsf{action\ list} \to \mathsf{rule}\\
\texttt{MakeForwardRules} &\in (\mathsf{switch} \times \mathsf{port} \times \mathsf{packet})\ \mathsf{policy}\ \mathsf{EF}\\
\texttt{AddRules} &\in \mathsf{policy}\ \mathsf{policy}\ \mathsf{EF}
\end{aligned}$$

Figure 4. Selected Frenetic Operators.

Network Query Language

A SQL-like query language that

- considers the input as a stream of all packets or a stream of statistics (based on the selected attribute)
- applies functional operators on the input stream
- some filters are executed on the data plane

Queries	q ::=	`Select`(a) *
		`Where`(fp) *
		`GroupBy`([$qh_1, \ldots, qh_n$]) *
		`SplitWhen`([$qh_1, \ldots, qh_n$]) *
		`Every`(n) *
		`Limit`(n)
Aggregates	a ::=	`packets` | `sizes` | `counts`
Headers	qh ::=	`inport` | `srcmac` | `dstmac` | `ethtype` |
		`vlan` | `srcip` | `dstip` | `protocol` |
		`srcport` | `dstport` | `switch`
Patterns	fp ::=	`true_fp`() | qh`_fp`(n) |
		`and_fp`([$fp_1, \ldots, fp_n$]) |
		`or_fp`([$fp_1, \ldots, fp_n$]) |
		`diff_fp`(fp_1, fp_2) | `not_fp`(fp)

Figure 3. Frenetic query syntax

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming.* ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Network Query Language: Example

Traffic monitoring in NOX:

```
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)
```

Traffic monitoring in Frenetic:

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

Nate Foster et al. "Frenetic: A Network Programming Language". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Network Policy Management

A functional reactive library that

- considers network events as a stream
- transfers event streams based on operators
- defines domain-specific “sinks” for the streams

Nate Foster et al. “Frenetic: A Network Programming Language”. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812

Events

`Seconds` $\in$ int E
`SwitchJoin` $\in$ switch E
`SwitchExit` $\in$ switch E
`PortChange` $\in$ (switch $\times$ int $\times$ bool) E
`Once` $\in \alpha \rightarrow \alpha$ E

Basic Event Functions

`>>` $\in \alpha$ E $\rightarrow \alpha\ \beta$ EF $\rightarrow \beta$ E
`Lift` $\in (a \rightarrow \beta) \rightarrow \alpha\ \beta$ EF
`>>` $\in \alpha\ \beta$ EF $\rightarrow \beta\ \gamma$ EF $\rightarrow \alpha\ \gamma$ EF
`ApplyFst` $\in \alpha\ \beta$ EF $\rightarrow (\alpha \times \gamma)\ (\beta \times \gamma)$ EF
`ApplySnd` $\in \alpha\ \beta$ EF $\rightarrow (\gamma \times \alpha)\ (\gamma \times \beta)$ EF
`Merge` $\in (\alpha$ E $\times\ \beta$ E$) \rightarrow (\alpha$ option $\times\ \beta$ option$)$ E
`BlendLeft` $\in \alpha \times \alpha$ E $\times\ \beta$ E $\rightarrow (\alpha \times \beta)$ E
`BlendRight` $\in \beta \times \alpha$ E $\times\ \beta$ E $\rightarrow (\alpha \times \beta)$ E
`Accum` $\in (\gamma \times (\alpha \times \gamma \rightarrow \gamma) \rightarrow \alpha\ \gamma$ EF
`Filter` $\in (\alpha \rightarrow$ bool$) \rightarrow \alpha\ \alpha$ EF

Listeners

`>>` $\in \alpha$ E $\rightarrow \alpha$ L $\rightarrow$ unit
`Print` $\in \alpha$ L
`Register` $\in$ policy L
`Send` $\in$ (switch $\times$ packet $\times$ action) L

Rules and Policies

`Rule` $\in$ pattern $\times$ action list $\rightarrow$ rule
`MakeForwardRules` $\in$ (switch $\times$ port $\times$ packet) policy EF
`AddRules` $\in$ policy policy EF

Figure 4. Selected Frenetic Operators.

Understanding Notations

Type system:

Notation	*Meaning*
α E	a stream of type α
$\alpha\beta$ EF	a function that maps a stream of type α to a stream of type β

Examples:

`ApplyFst`: $\alpha\beta$ EF $\mapsto (\alpha \times \gamma)(\beta \times \gamma)$ EF

- Input of `ApplyFst`, $\alpha\beta$ EF, is a function that transform a stream of type α to a stream of type β
- Output of `ApplyFst`, $(\alpha \times \gamma)(\beta \times \gamma)$ EF, is a function that takes a stream of a tuple type $(\alpha \times \gamma)$ to a stream of tuple type $(\beta \times \gamma)$

Understanding Notations

Example in Java:

java.util.stream

Interface Stream<T>

Type Parameters:

T - the type of the stream elements

java.util.function

Interface Function<T,R>

Type Parameters:

T - the type of the input to the function

R - the type of the result of the function

https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html

https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

```
interface EF<Alpha, Beta>
extends Function<Stream<Alpha>, Stream<Beta>> {
}

interface ApplyFirst<Alpha, Beta, Gamma>
extends EF<Tuple<Alpha, Gamma>, Tuple<Beta, Gamma>> {

}
```

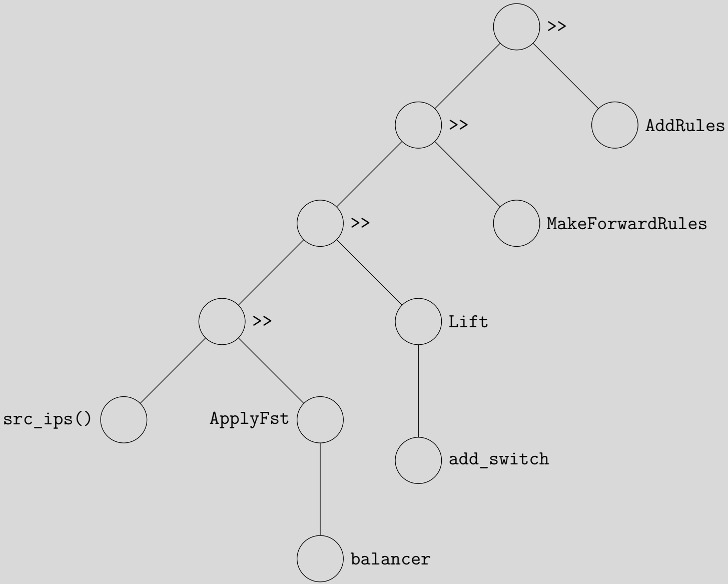
Frenetic Policy Example

We use an example to explain the functional reactive library

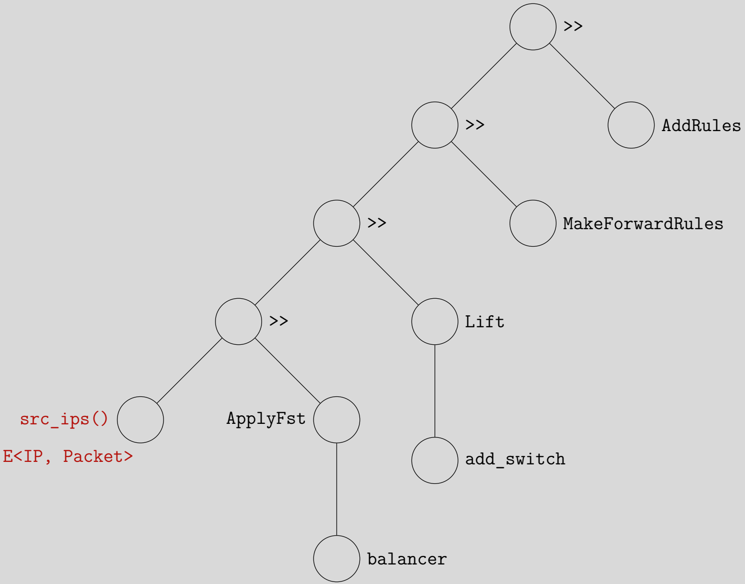
```
# query returning one packet per source IP
def src_ips() =
  return (Select(packets) *
          Where(inport_fp(1)) *
          GroupBy([srcip]) *
          Limit(1))

# helper to add switch to a port-packet pair
def add_switch(port,packet):
  return (switch(header(packet)),port,packet)

# parameterized load balancer
def balance(balancer):
  return \
    (src_ips()          >> # (IP*packet) E
     ApplyFst(balancer) >> # (port*packet) E
     Lift(add_switch)   >> # (switch*port*packet) E
     MakeForwardRules() >> # policy E
     AddRules())           # policy E
```

Figure 5. A Parameterized Load Balancer

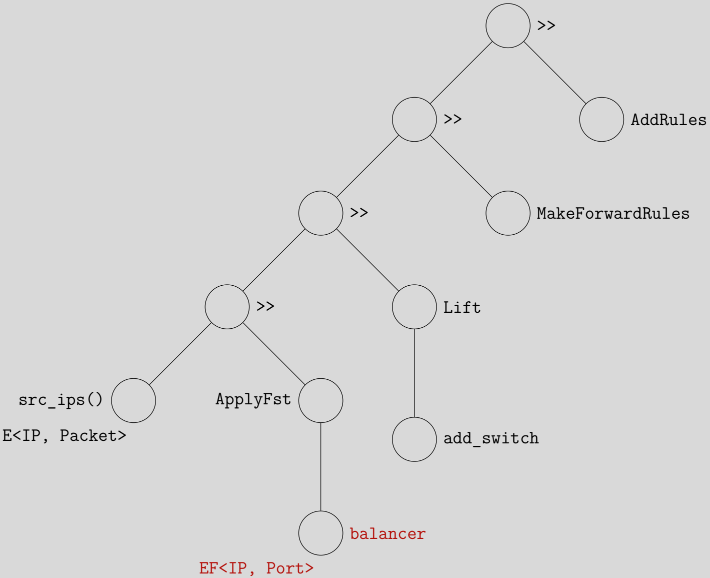
Frenetic Policy Example



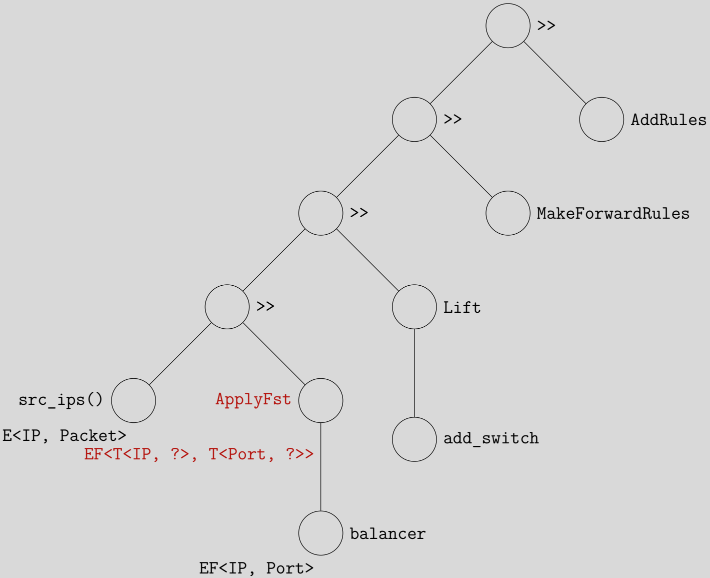
Frenetic Policy Example



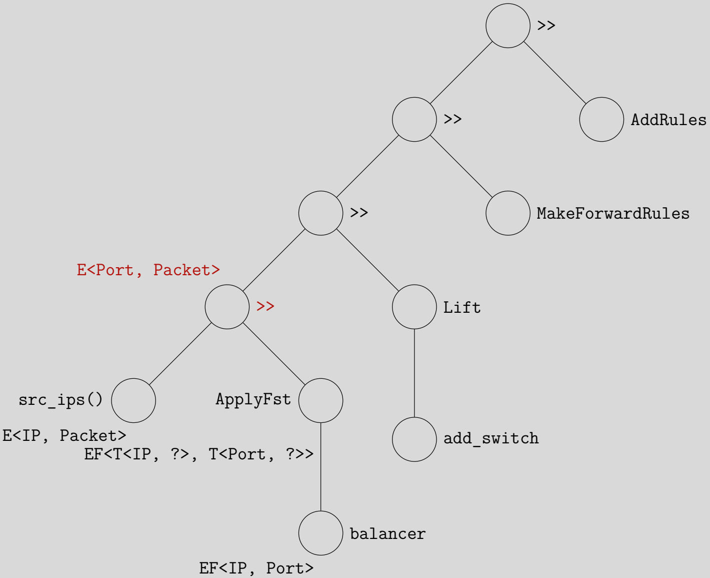
Frenetic Policy Example



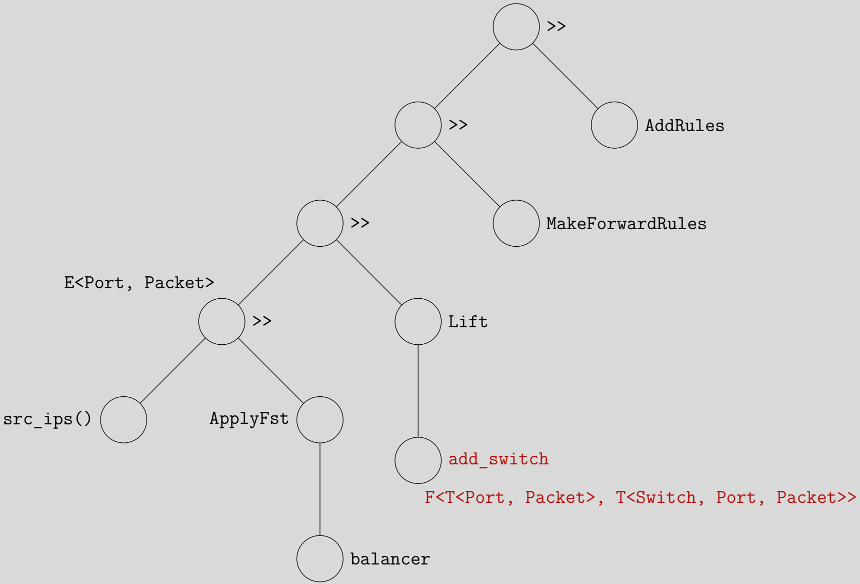
Frenetic Policy Example



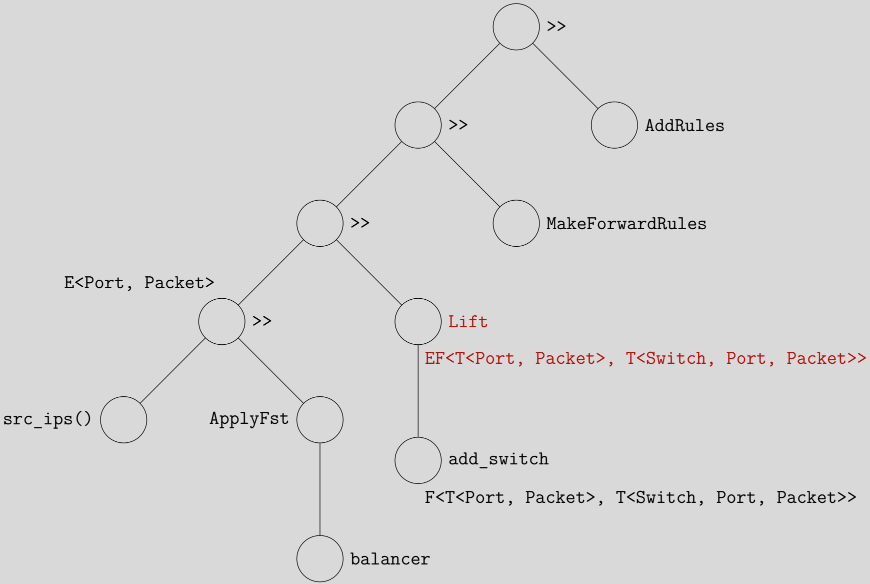
Frenetic Policy Example



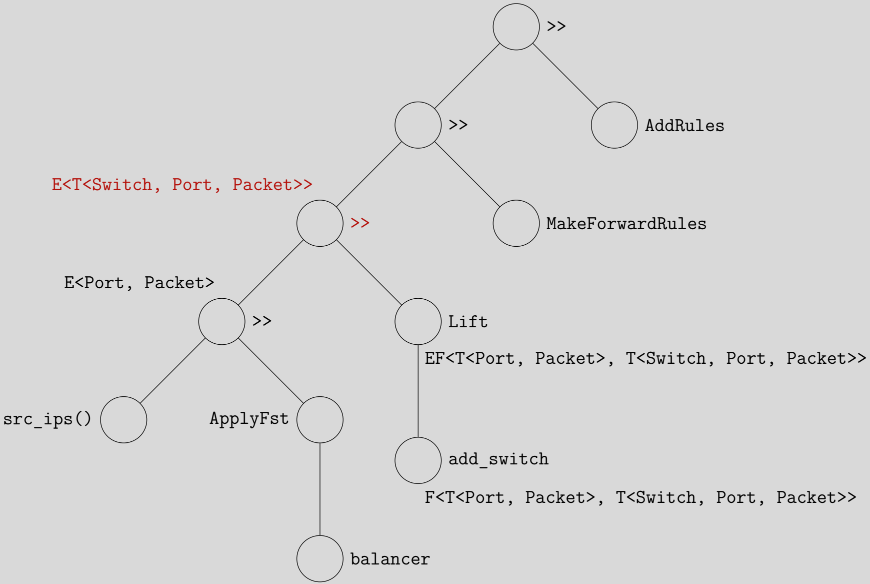
Frenetic Policy Example



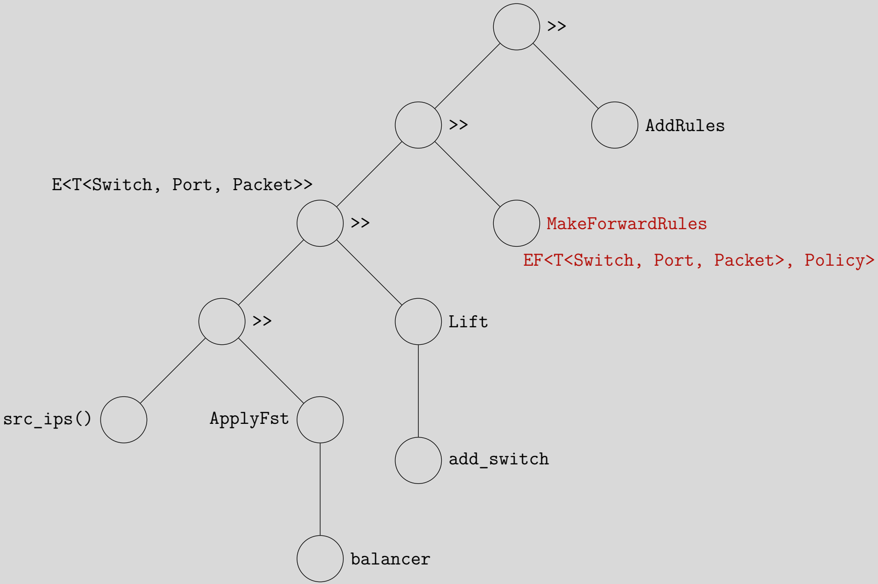
Frenetic Policy Example



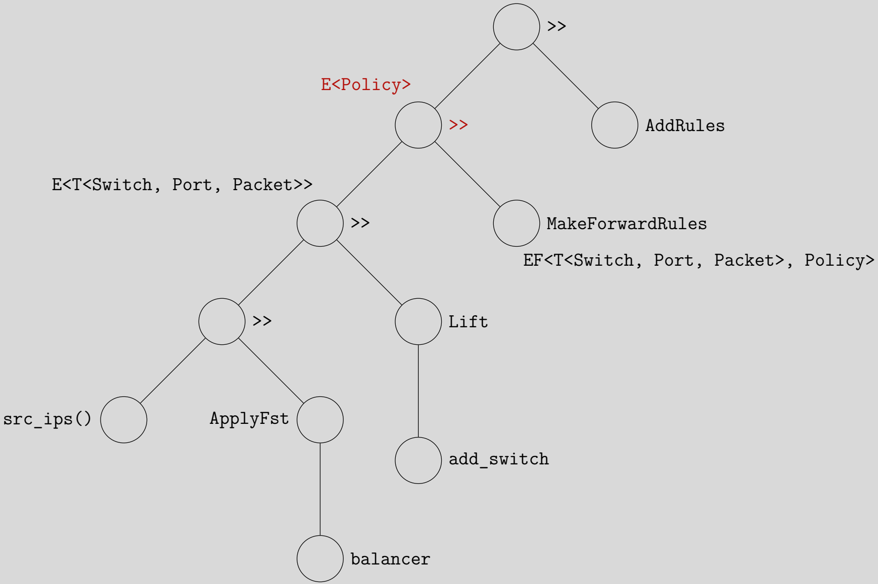
Frenetic Policy Example



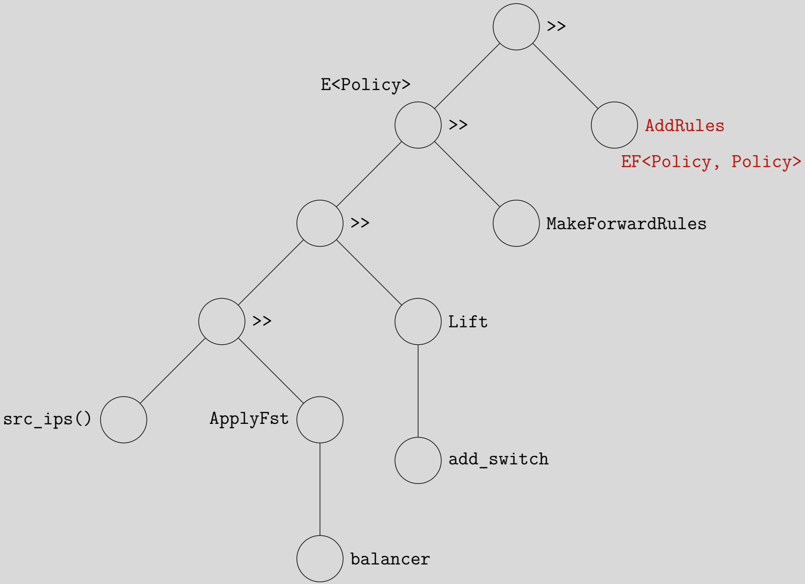
Frenetic Policy Example



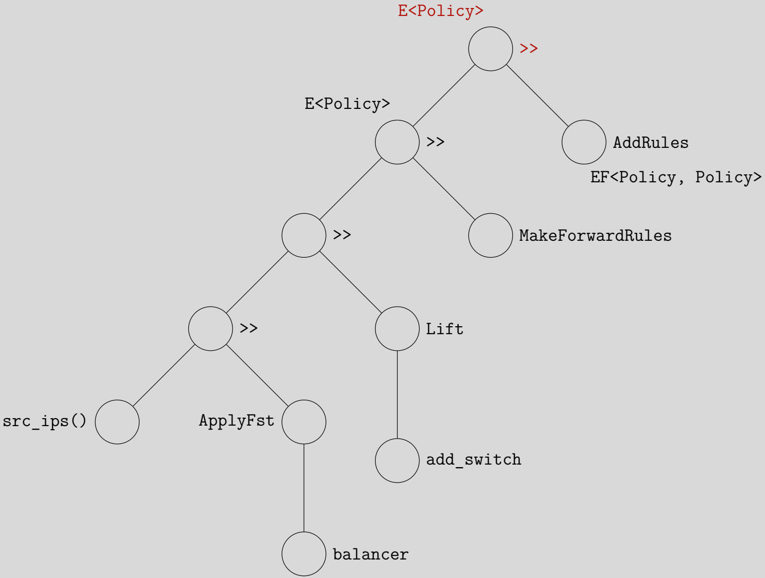
Frenetic Policy Example



Frenetic Policy Example



Frenetic Policy Example



Summary

- Programming with low-level API leads to complex development and error-prone execution
- Frenetic provides the abstraction of network query and functional reactive policy, modeling network events as streams and control workflow as transformations

Pyretic

Pyretic

Composing Software-Defined Networks

Christopher Monsanto[*], *Joshua Reich*[*], *Nate Foster*[†], *Jennifer Rexford*[*], *David Walker*[*]
[*]Princeton [†]Cornell

Christopher Monsanto et al. "Composing Software Defined Networks". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, 2013, pp. 1–13. URL: `https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto`

Goals

Pyretic aims to achieve two goals:

- Modular development of network policies
- Programming on virtualized networks

Goals

Pyretic aims to achieve two goals:

- Modular development of network policies
- Programming on virtualized networks

To achieve the goals, Pyretic uses

- Composition of network policies (网络策略组合)
- Network objects and policy transformations

Policy Composition

Pyretic supports two composition operators:

- **Sequential composition** (>>, 顺序组合): `a >> b` represents that a packet must first be processed by `a` and then be processed by `b`
  - let `p` represent a packet, `(a >> b)(p) = b(a(p))`
- **Parallel composition** (|, 并行组合): `a | b` represents that a packet is processed by `a` and `b` at the same time
  - let `p` represent a packet, `(a | b)(p) = a(p) | b(p)`

Policy Composition: Example

Monitor
`srcip=5.6.7.8` → `count`

Route
`dstip=10.0.0.1` → `fwd(1)`
`dstip=10.0.0.2` → `fwd(2)`

Load-balance
`srcip=0*,dstip=1.2.3.4` → `dstip=10.0.0.1`
`srcip=1*,dstip=1.2.3.4` → `dstip=10.0.0.2`

Compiled Prioritized Rule Set for "Monitor | Route"
`srcip=5.6.7.8,dstip=10.0.0.1` → `count,fwd(1)`
`srcip=5.6.7.8,dstip=10.0.0.2` → `count,fwd(2)`
`srcip=5.6.7.8` → `count`
`dstip=10.0.0.1` → `fwd(1)`
`dstip=10.0.0.2` → `fwd(2)`

Compiled Prioritized Rule Set for "Load-balance >> Route"
`srcip=0*,dstip=1.2.3.4` → `dstip=10.0.0.1,fwd(1)`
`srcip=1*,dstip=1.2.3.4` → `dstip=10.0.0.2,fwd(2)`

Figure 1: Parallel and Sequential Composition.

Christopher Monsanto et al. "Composing Software Defined Networks". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, 2013, pp. 1–13. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto

Network Object and Transformation

Two examples of network virtualization

Many-to-one Mapping:

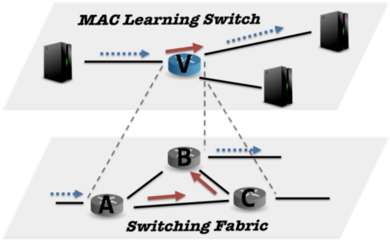


Figure 2: Many physical switches to one virtual.

One-to-many Mapping:

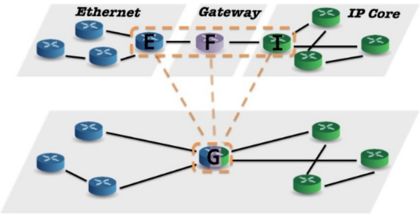


Figure 3: One physical switch to many virtual.

Christopher Monsanto et al. "Composing Software Defined Networks". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, 2013, pp. 1–13. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto

Pyretic: Language Design

Primitive Actions:

A ::= `drop` | `passthrough` | `fwd(port)` | `flood` |
`push(h=v)` | `pop(h)` | `move(h1=h2)`

Predicates:

P ::= `all_packets` | `no_packets` | `match(h=v)` |
`ingress` | `egress` | P & P | (P | P) | ~P

Query Policies:

Q ::= `packets(limit,[h])` | `counts(every,[h])`

Policies:

C ::= A | Q | $P[C]$ | (C | C) | C >> C | `if_`(P,C,C)

Figure 4: Summary of static NetCore syntax.

Christopher Monsanto et al. "Composing Software Defined Networks". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, 2013, pp. 1–13. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto

Virtual Packet Attribute

Pyretic extends a packet header with virtual packet attributes (虚拟数据包头):

- header fields contained in the packet: srcip, dstip, . . .
- location of the packet: switch, inport, vswitch, . . .

The extended packet header refers to h in the syntax

Primitive Actions:

A ::= `drop` | `passthrough` | `fwd(port)` | `flood` |
`push(h=v)` | `pop(h)` | `move(h1=h2)`

Predicates:

P ::= `all_packets` | `no_packets` | `match(h=v)` |
`ingress` | `egress` | P & P | (P | P) | ~P

Query Policies:

Q ::= `packets(limit,[h])` | `counts(every,[h])`

Policies:

C ::= A | Q | $P[C]$ | (C | C) | C >> C | `if_`(P,C,C)

Figure 4: Summary of static NetCore syntax.

Christopher Monsanto et al. "Composing Software Defined Networks". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, 2013, pp. 1–13. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto

Primitive Actions

Pyretic supports the following actions:

- `drop`: discard the packet by rewriting the location of the packet
- `passthrough`: do not change the packet
- `fwd(port)`: set the output port of a packet by rewriting the location of the packet
- `flood`: flood the packet
- `push(h=v)`: set the value of a header field
- `pop(h)`: drop a header field
- `move(h1=h2)`: rewrite a packet header field with value of another header field

Example:

Initial located packet:

`{inport: 1, sw: s1, srcip=10.0.0.1}`

After `pop(srcip)`:

`{inport: 1, sw: s1 }`

After `push(srcip=192.168.0.1)`

`{inport: 1, sw: s1, srcip=192.168.0.1 }`

Predicate

Pyretic uses predicate (声明) to specify packet selection

- `all_packets`: match all packets
- `no_packets`: match no packets
- `match(h=v)`: match packets whose header field `h` is `v`
- `ingress`: match incoming packets
- `egress`: match outgoing packets
- `P & P`, `P | P`, `~P`: predicate composition

Example:

- `match(srcip=10.0.0.1)`: all packets whose source IPv4 address is `10.0.0.1`
- `~match(srcip=10.0.0.1)`: all packets whose source IPv4 address is not `10.0.0.1`
- `ingress & match(dstip=192.168.0.2)`: all incoming packets whose destination IPv4 address is `192.168.0.2`

Network Query

Pyretic supports similar but less expressive network queries with Frenetic

- Select `limit` packets that match predicate `h`:
  - Pyretic: `packets(limit, [h])`
  - Frenetic: `Select(packet) * GroupBy([h]) * Limit(limit)`
- Report the number of packets match predicate `h` in every `n` packets:
  - Pyretic: `counts(n, [h])`
  - Frenetic: `Select(count) * GroupBy([h]) * Every(n)`

Policy Composition

Pyretic supports composition of policies:

- `A`, `Q`, `P [C]`: Basic policies
- `C >> C`: sequential composition
- `C | C`: parallel composition
- `if_(P, C1, C2)`: branching, same as `P [C1] | ~P [C2]`

Example:

```
pop(dstip) >> push(dstip='10.0.0.1') >> fwd(3)
```

- This policy has the format of `A >> A >> A`
- Rewrite destination IP address and then forward

Example

- This policy has the format of `P [Q] | A`
- Monitor traffic for packets with source `1.2.3.4`
- Use flooding to route traffic

```
def printer(pkt):
  print pkt

def dpi():
  q = packets(None,[])
  q.when(printer)
  return match(srcip='1.2.3.4')[q]

def main():
  return dpi() | flood
```

Summary

- Pyretic introduces composition operators to enable modular network policy development
- Pyretic introduces virtual packet headers to enable transformations of network topologies

Maple

Maple

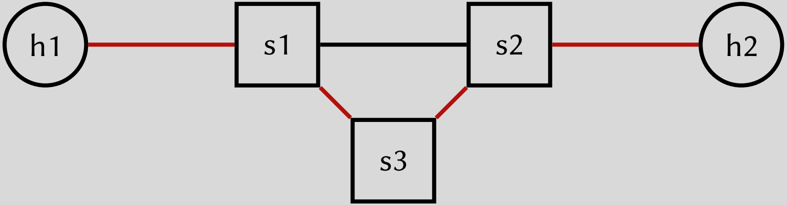
Maple: Simplifying SDN Programming Using Algorithmic Policies

Andreas Voellmy* Junchang Wang*† Y. Richard Yang* Bryan Ford* Paul Hudak*
*Yale University †University of Science and Technology of China
{andreas.voellmy, junchang.wang, yang.r.yang, bryan.ford, paul.hudak}@yale.edu

Andreas Voellmy, Junchang Wang, et al. "Maple: Simplifying SDN Programming Using Algorithmic Policies". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 87–98. URL: http://doi.acm.org/10.1145/2486001.2486030

Algorithmic Policy

Consider the example below




```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            install(match(srcip=192.168.0.0/24, srcport=22), path(s1, s3, s2))
        else:
            install(match(srcip=192.168.0.0/24), path(s1, s2))
```

Expected behavior

- Outgoing SSH traffic uses path s1-s3-s2
- Outgoing normal traffic uses path s1-s2

Problems

Logic Mismatch

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22: # <---- should match srcport=22
            install(match(srcip=192.168.0.0/24, srcport=23), path(s1, s3, s2))
            # <--- mistakenly specified as 23
        else:
            install(match(srcip=192.168.0.0/24), path(s1, s2))
```

Result: Embedded rules conflict with the algorithmic logic (实现与目标冲突)

Problems

Priority Management

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            install(match(srcip=192.168.0.0/24, srcport=22), path(s1, s3, s2))
        else:
            install(match(srcip=192.168.0.0/24), path(s1, s2))
```

If two rules are both set to DEFAULT priority:

match	*priority*	*action* (on s1)
srcip=192.168.0.0/24, srcport=22	DEFAULT	output(3)
srcip=192.168.0.0/24	DEFAULT	output(2)

Result: Nondeterministic behavior for srcip=192.168.0.1 and srcport=22

Problems

Order of Packet-In

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            install(match(srcip=192.168.0.0/24, srcport=22), path(s1, s3, s2))
        else:
            install(match(srcip=192.168.0.0/24), path(s1, s2))
```

If a packet with srcip=192.168.0.1 and srcport=80 arrives first:

match	*priority*	*action* (on s1)
srcip=192.168.0.0/24	DEFAULT	output(2)

Result: No packet-in message will be triggered for srcip=192.168.0.1 and srcport=22

Maple: High-level Idea

Maple discovers that:

- matches and priorities can be automatically derived from algorithmic logic (根据算法逻辑生成流表规则)
- barrier rules are needed to enforce correct data plane visibility (正确生成规则保证数据平面可视性)

Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:
srcip=192.168.0.1, srcport=80

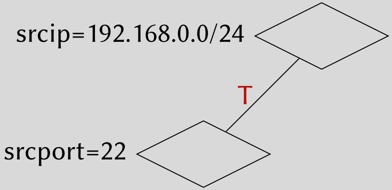
Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

```python
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:

srcip=192.168.0.1, srcport=80



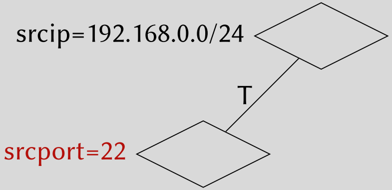
Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:

srcip=192.168.0.1, srcport=80



Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

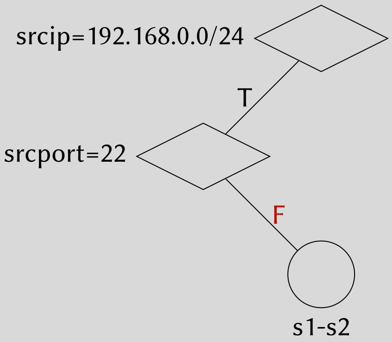
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:

srcip=192.168.0.1, srcport=80

Arrival:

srcip=192.168.0.1, srcport=22

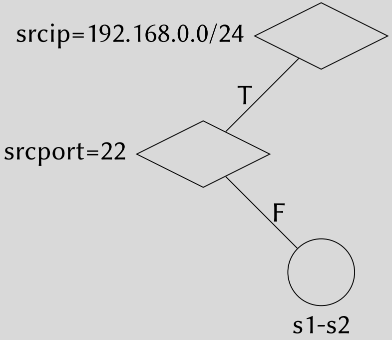


Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:
srcip=192.168.0.1, srcport=80
Arrival:
srcip=192.168.0.1, srcport=22



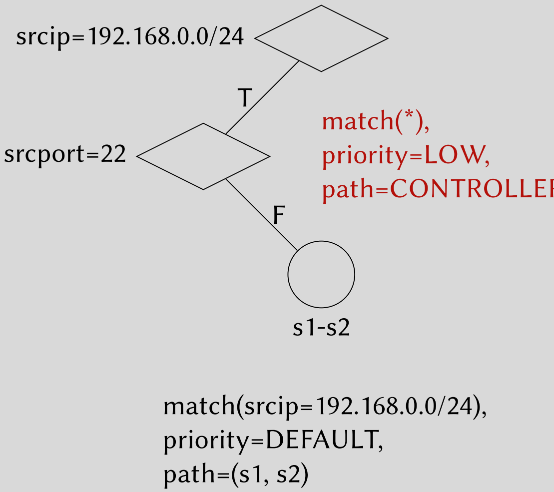
match(srcip=192.168.0.0/24),
priority=DEFAULT,
path=(s1, s2)

Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:
srcip=192.168.0.1, srcport=80
Arrival:
srcip=192.168.0.1, srcport=22

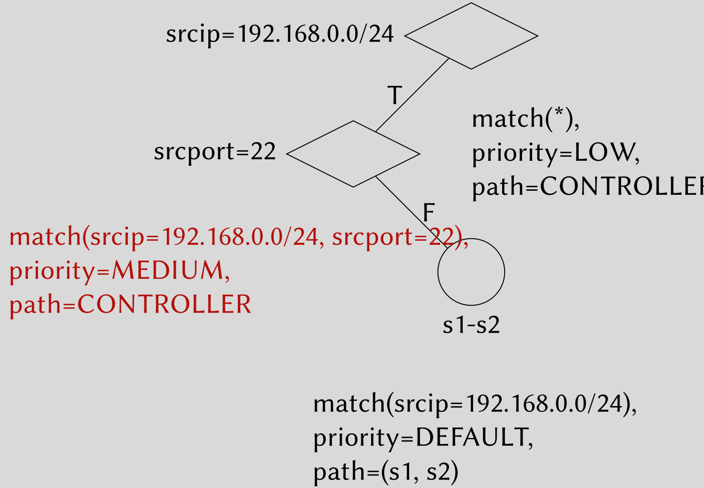


Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:
srcip=192.168.0.1, srcport=80
Arrival:
srcip=192.168.0.1, srcport=22



Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

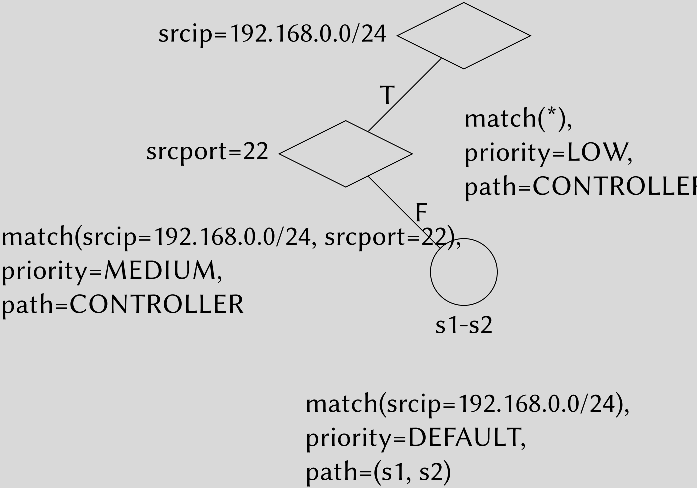
```python
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:

srcip=192.168.0.1, srcport=80

Arrival:

srcip=192.168.0.1, srcport=22

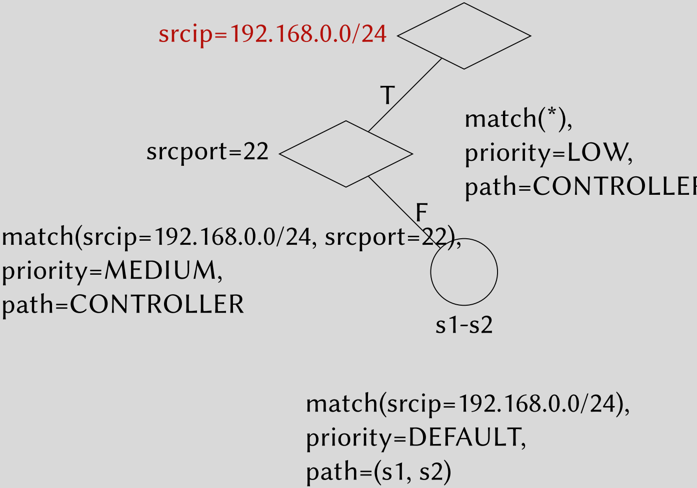


Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:
srcip=192.168.0.1, srcport=80
Arrival:
srcip=192.168.0.1, srcport=22



Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

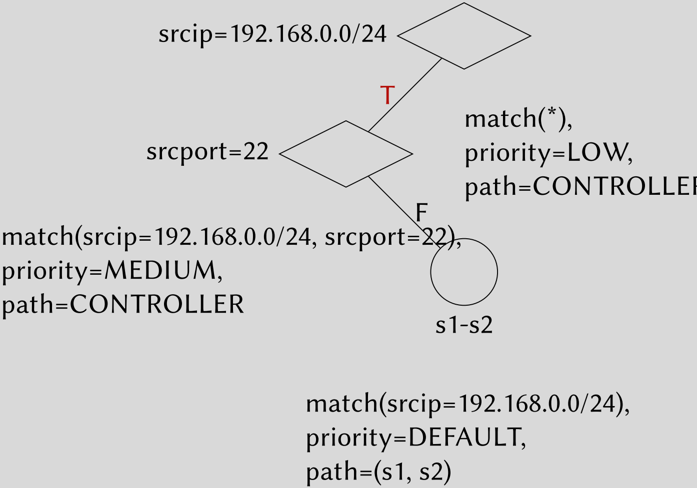
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:

srcip=192.168.0.1, srcport=80

Arrival:

srcip=192.168.0.1, srcport=22



Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

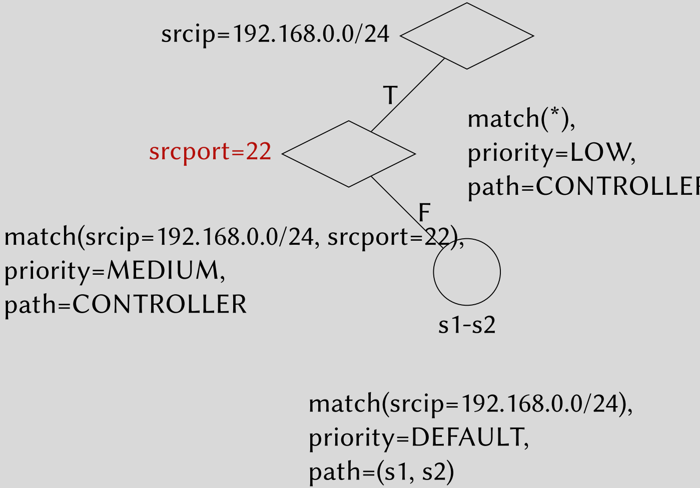
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:

srcip=192.168.0.1, srcport=80

Arrival:

srcip=192.168.0.1, srcport=22

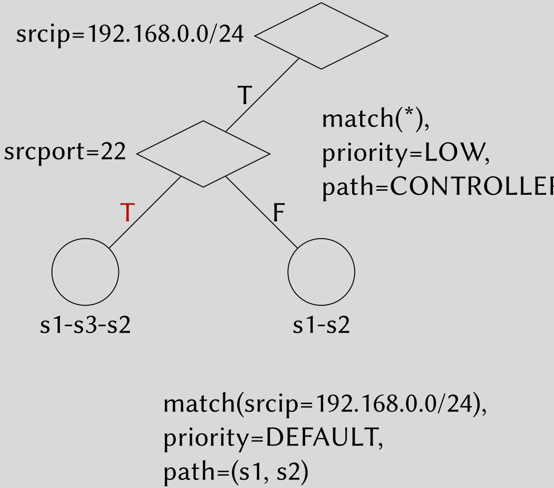


Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:
srcip=192.168.0.1, srcport=80
Arrival:
srcip=192.168.0.1, srcport=22



Trace Tree

Trace tree (踪迹树) is the key data structure in Maple to realize the goals.

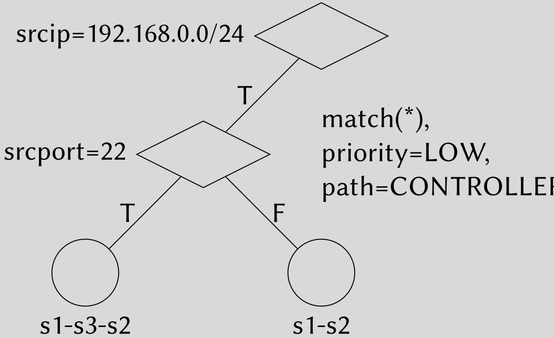
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Arrival:

srcip=192.168.0.1, srcport=80

Arrival:

srcip=192.168.0.1, srcport=22



match(srcip=192.168.0.0/24, srcport=22),
priority=MEDIUM,
path=(s1, s3, s2)

match(srcip=192.168.0.0/24),
priority=DEFAULT,
path=(s1, s2)

Rule Generation

Trace tree:

- records the packet decision process
- is used to construct flow rules
  - `True` branch: add the condition to the match
  - `False` branch: do not add the condition to the match
  - If a branch is missing, add a barrier rule
  - `True` branch has higher priority for positive test (e.g., srcip in 192.168.0.0/24, srcport=22) and lower priority for negative test (e.g., srcport != 22)

Summary

- Maple enables reactive algorithmic policies
- Flow rules (matches, priorities and actions) are automatically constructed based on the decision logic
- The decision logic is used to incrementally construct the trace tree data structure

Magellan

Magellan

Magellan: Generating Multi-Table Datapath from Datapath Oblivious Algorithmic SDN Policies

Andreas Voellmy[+] Shenshen Chen[*] Xing Wang[*] Y. Richard Yang[*+]
Yale University[+] Tongji University[*]

Andreas Voellmy, Shenshen Chen, et al. "Magellan: Generating Multi-Table Datapath from Datapath Oblivious Algorithmic SDN Policies". In: *Proceedings of the 2016 ACM SIGCOMM Conference.* SIGCOMM '16: ACM SIGCOMM 2016 Conference. Florianopolis Brazil: ACM, Aug. 22, 2016, pp. 593–594. URL: https://dl.acm.org/doi/10.1145/2934872.2959064 (visited on 10/24/2021)

Basic Idea

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

- In Maple, the trace tree is reactively constructed based on packets.
- Basic idea: proactively explore all potential execution paths and construct the trace tree

Naive Idea: Symbolic Execution

Magellen uses symbolic execution (符号计算) to proactively explore every potential branch

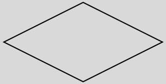
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Naive Idea: Symbolic Execution

Magellen uses symbolic execution (符号计算) to proactively explore every potential branch

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

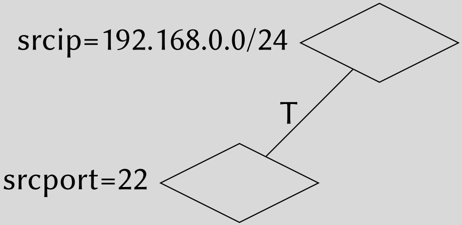
srcip=192.168.0.0/24



Naive Idea: Symbolic Execution

Magellen uses symbolic execution (符号计算) to proactively explore every potential branch

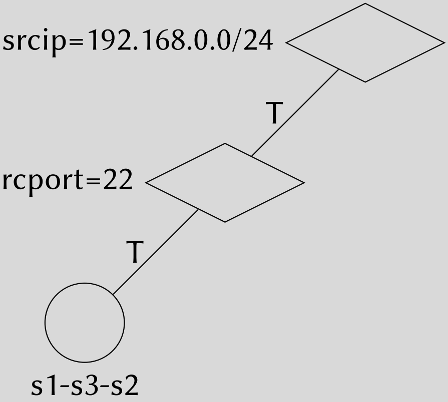
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```



Naive Idea: Symbolic Execution

Magellen uses symbolic execution (符号计算) to proactively explore every potential branch

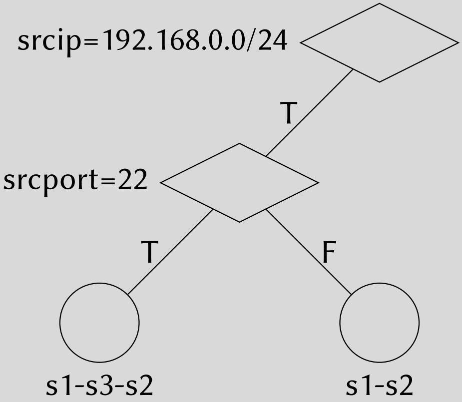
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```



Naive Idea: Symbolic Execution

Magellen uses symbolic execution (符号计算) to proactively explore every potential branch

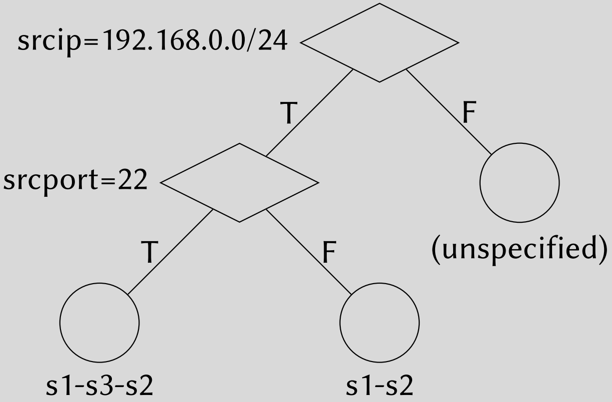
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```



Naive Idea: Symbolic Execution

Magellen uses symbolic execution (符号计算) to proactively explore every potential branch

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```



Problem of Symbolic Execution

```
def policy(pkt):
    srcvlan = mac2vlan[pkt.srcmac]
    dstvlan = mac2vlan[pkt.dstmac]
    use_path(vlan2path[srcvlan, dstvlan])
```

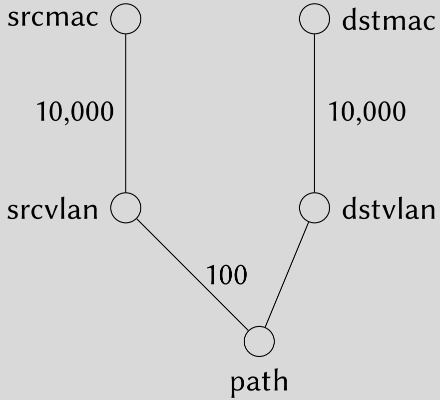
If the size of `mac2vlan` table is 10,000, the number of vlan is 100, and the size of `vlan2path` is 1000

- Exploring all execution paths results in a trace tree of 10,000 $\times$ 10,000 nodes
- Same as the "cross-product" effect of OpenFlow tables

Data Flow Graph

```
def policy(pkt):
    srcvlan = mac2vlan[pkt.srcmac]
    dstvlan = mac2vlan[pkt.dstmac]
    use_path(vlan2path[srcvlan, dstvlan])
```

With static analysis, one can build the data flow graph of the policy:

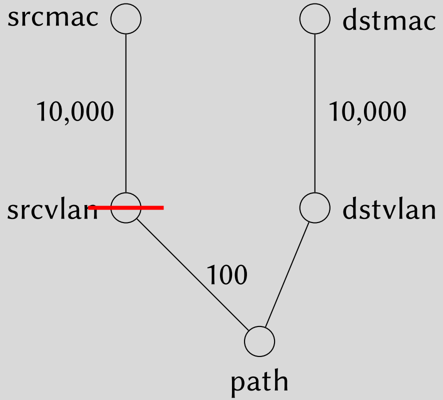


- Each "partition" is a table layout
- Different layout has different table size: number of potential inputs

Data Flow Graph

```
def policy(pkt):
    srcvlan = mac2vlan[pkt.srcmac]
    dstvlan = mac2vlan[pkt.dstmac]
    use_path(vlan2path[srcvlan, dstvlan])
```

With static analysis, one can build the data flow graph of the policy:

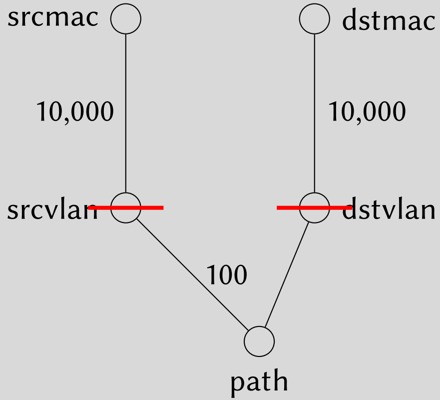


- Each "partition" is a table layout
- Different layout has different table size: number of potential inputs
  - Layout 1: 10,000 + 100 * 10,000 = 1,010,000

Data Flow Graph

```
def policy(pkt):
    srcvlan = mac2vlan[pkt.srcmac]
    dstvlan = mac2vlan[pkt.dstmac]
    use_path(vlan2path[srcvlan, dstvlan])
```

With static analysis, one can build the data flow graph of the policy:

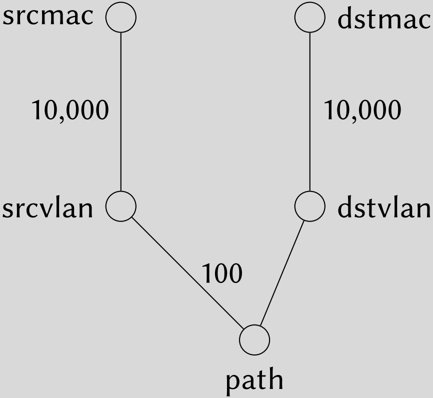


- Each "partition" is a table layout
- Different layout has different table size: number of potential inputs
  - Layout 1: 10,000 + 100 * 10,000 = 1,010,000
  - Layout 2: 10,000 + 10,000 + 1,000 = 21,000

Data Flow Graph

```
def policy(pkt):
    srcvlan = mac2vlan[pkt.srcmac]
    dstvlan = mac2vlan[pkt.dstmac]
    use_path(vlan2path[srcvlan, dstvlan])
```

With static analysis, one can build the data flow graph of the policy:



- Each “partition” is a table layout
- Different layout has different table size: number of potential inputs
  - Layout 1: 10,000 + 100 * 10,000 = 1,010,000
  - Layout 2: 10,000 + 10,000 + 1,000 = 21,000
  - Layout 3: 10,000 * 10,000 = 100,000,000

Summary

- Magellan proactively builds flow rules from algorithmic policies
- It uses static data flow analysis to avoid the cross-product problem
- By finding the optimal partition of the DFG, Magellan optimizes the flow table layout

The End

Quiz

对于下面的程序，对于下面控制器收到的数据包序列，绘制对应的踪迹树及生成的全局路由规则

```
if pkt.dstip in '192.168.0.0/24':
    use_path('gw', 's1')
elif pkt.srcip in '192.168.0.0/24':
    if pkt.dstip in '192.168.1.0/24':
        use_path('s1', 'gw')
    else:
        use_path('s1', 'billing', 'gw')
else:
    drop()
```

Packets:

- srcip=192.168.0.2, dstip=192.168.1.2
- srcip=192.168.3.2, dstip=192.168.0.2

Thanks!

kaigao@scu.edu.cn

References I

[1] Nate Foster et al. "Frenetic: A Network Programming Language". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291. URL: http://doi.acm.org/10.1145/2034773.2034812.

[2] D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76.

[3] Christopher Monsanto et al. "Composing Software Defined Networks". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, 2013, pp. 1–13. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto.

[4] ONF. *OpenFlow Switch Specification (Version 1.3.5)*. Open Networking Foundation, Mar. 26, 2015, p. 177. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf (visited on 09/05/2021).

[5] Andreas Voellmy, Shenshen Chen, et al. "Magellan: Generating Multi-Table Datapath from Datapath Oblivious Algorithmic SDN Policies". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16: ACM SIGCOMM 2016 Conference. Florianopolis Brazil: ACM, Aug. 22, 2016, pp. 593–594. URL: https://dl.acm.org/doi/10.1145/2934872.2959064 (visited on 10/24/2021).

[6] Andreas Voellmy, Junchang Wang, et al. "Maple: Simplifying SDN Programming Using Algorithmic Policies". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 87–98. URL: http://doi.acm.org/10.1145/2486001.2486030.

[7] Lihua Yuan. "SONiC: Software for Open Networking in the Cloud". APNet'18. 2018.