Link Forwarding, Network Control Functions and Software Defined Networking

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

4/25/2016
Admin

- Project status
Recap: Look Inside a Router

Two key router functions:
- run routing algorithms/protocol (RIP, OSPF, BGP)
- *switching* datagrams from incoming to outgoing ports
Recap: Link Layer Services

- Framing
- Multiplexing/demultiplexing
- Media access control
- Forwarding/switching with a link-layer (Layer 2) domain
- Reliable delivery between adjacent nodes

![Ethernet Frame Diagram]
Recap: Random Media Access

- **Slotted Aloha protocol**
  - Time is divided into equal size slots (\(=\) pkt trans. time)
  - Node with pkt: transmit at beginning of next slot with probability \(p\)
  - Optimal value \(p = 1/n\), with success rate only \(1/e\)

![Diagram showing slotted Aloha protocol with nodes transmitting packets over time, with slots labeled as C, E, C, S, E, C, E, and S, and packets transmitted at the beginning of each slot. The diagram indicates success (S), collision (C), and empty (E) states.](image-url)
Summary of Problems of Aloha Protocols

- **Problems**
  - slotted Aloha has better efficiency than pure Aloha but clock synchronization is hard to achieve
  - Aloha protocols have low efficiency due to collision or empty slots
    - when offered load is optimal \( p = 1/N \), the goodput is only about 37%
    - when the offered load is not optimal, the goodput is even lower
  - undesirable steady state at a fixed transmission rate, when the number of backlogged stations varies

- **Ethernet design:** address the problems:
  - approximate slotted Aloha without clock synchronization
  - reduce the penalty of collision or empty slots
  - infer optimal transmission rate
The Basic MAC Mechanisms of Ethernet

geta packet from upper layer;
K := 0; n := 0; // K: control wait time; n: no. of collisions
repeat:
  wait for K * 512 bit-time;
  while (network busy) wait;   // sync clock
  wait for 96 bit-time after detecting no signal;
  transmit and detect collision; // not waste the whole
if detect collision
  stop and transmit a 48-bit jam signal; // not waste the whole
  n ++;
  m:= min(n, 10), where n is the number of collisions
  choose K randomly from {0, 1, 2, ..., 2^m-1}.
if n < 16 goto repeat
else give up
MAC Protocols

Goals

- efficient, fair, decentralized, simple

Three broad classes:

- **Non-partitioning**
  - random access
    - allow collisions
  - “taking-turns”
    - a token coordinates shared access to avoid collisions

- **channel partitioning**
  - divide channel into smaller “pieces” (time slot, frequency, code)
Example Channel Partitioning: CDMA

CDMA (Code Division Multiple Access)

- Used mostly in wireless broadcast channels (cellular, satellite, etc)
- A spread-spectrum technique


Examples: Sprint and Verizon, WCDMA
CDMA: Encoding

- All users share the same frequency, but each user \( m \) has its own unique “chipping” sequence (i.e., code) \( c_m \) to encode data, i.e., code set partitioning
  - e.g. \( c_m = 1 1 1 -1 1 -1 -1 -1 \)

- Assume original data are represented by 1 and -1

- Encoded signal = (original data) modulated by (chipping sequence)
  - Assume \( c_m = 1 1 1 -1 1 -1 -1 -1 \)

  - if data is \( d \), send \( d c_m \),
    - if data \( d \) is 1, send \( c_m \)
    - if data \( d \) is -1 send \( -c_m \)
CDMA: Encoding

user data $d(t)$

chipping sequence $c(t)$

resulting signal

$t_b$: bit period
$t_c$: chip period

$X = t_b$
CDMA: Decoding

- Inner-product (summation of bit-by-bit product) of encoded signal and chipping sequence
  - if inner-product > 0, the data is 1; else -1
CDMA Encode/Decode

Code of user $m \ c_m$:  1 1 1 -1 1 -1 -1 -1

- The number of bits of each chipping sequence is $M$
CDMA: Deal with Multiple-User Interference

- Two codes $c_i$ and $c_j$ are orthogonal, if
  - $c_j \cdot c_i = 0$, where we use “.” to denote inner product, e.g.

$$
\begin{bmatrix}
C_1: & 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 \\
C_2: & 1 & -1 & 1 & 1 & 1 & -1 & 1 & 1 \\
\end{bmatrix}
$$

$$
C_1 \cdot C_2 = 1 + (-1) + 1 + (-1) + 1 + (-1) + (-1) = 0
$$

- If codes are orthogonal, multiple users can “coexist” and transmit simultaneously with minimal interference:

$$
(\sum_j d_j c_j) \cdot c_i = d_i \|c_i\|
$$

Analogy: Speak in different languages!
CDMA: Two-Sender Interference

Code 1: 1 1 1 -1 1 -1 -1
Code 2: 1 -1 1 1 1 -1 1 1
Generating Orthogonal Codes

- The most commonly used orthogonal codes in current CDMA implementation are the Walsh Codes

\[ W_0 = (1) \]

\[ W_{2n} = \begin{pmatrix} W_n & W_n \\ W_n & \overline{W}_n \end{pmatrix} \]
Walsh Codes
Orthogonal Variable Spreading Factor (OSVF)

- Variable codes: Different users use different lengths spreading codes
- Orthogonal: diff. users' codes are orthogonal

If user 1 is given code [1,1], what orthogonal codes can we give to other users?
WCDMA Orthogonal Variable Spreading Factor (OSVF)

- **Flexible code (spreading factor) allocation**
  - **up link SF:** 4 - 256
  - **down link SF:** 4 - 512

### WCDMA downlink

<table>
<thead>
<tr>
<th>Spreading factor</th>
<th>Channel symbol rate (kbps)</th>
<th>Channel bit rate (kbps)</th>
<th>DPDCH channel bit rate range (kbps)</th>
<th>Max. user data rate with ½ rate coding (approx.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>7.5</td>
<td>15</td>
<td>3-6</td>
<td>1-3 kbps</td>
</tr>
<tr>
<td>256</td>
<td>15</td>
<td>30</td>
<td>12-24</td>
<td>6-12 kbps</td>
</tr>
<tr>
<td>128</td>
<td>30</td>
<td>60</td>
<td>42-51</td>
<td>20-24 kbps</td>
</tr>
<tr>
<td>64</td>
<td>60</td>
<td>120</td>
<td>90</td>
<td>45 kbps</td>
</tr>
<tr>
<td>32</td>
<td>120</td>
<td>240</td>
<td>210</td>
<td>105 kbps</td>
</tr>
<tr>
<td>16</td>
<td>240</td>
<td>480</td>
<td>432</td>
<td>215 kbps</td>
</tr>
<tr>
<td>8</td>
<td>480</td>
<td>960</td>
<td>912</td>
<td>456 kbps</td>
</tr>
<tr>
<td>4</td>
<td>960</td>
<td>1920</td>
<td>1872</td>
<td>936 kbps</td>
</tr>
<tr>
<td>4, with 3 parallel codes</td>
<td>2880</td>
<td>5760</td>
<td>5616</td>
<td>2.3 Mbps</td>
</tr>
</tbody>
</table>
Combined Random Access + Channel Partition: GSM Logical Channels and Request

- **Control channels**
  - Broadcast control channel (BCCH)
    - from base station, announces cell identifier, synchronization
  - Common control channels (CCCH)
    - paging channel (PCH): base transceiver station (BTS) pages a mobile host (MS)
    - random access channel (RACH): MSs for initial access, slotted Aloha
    - access grant channel (AGCH): BTS informs an MS its allocation
  - Dedicated control channels
    - standalone dedicated control channel (SDCCH): signaling and short message between MS and an MS

- **Traffic channels (TCH)**

- **Traffic channels (TCH)**
  - call setup from an MS

  ![Diagram](https://via.placeholder.com/150)
Discussions

- Advantages of channel partitioning over random access

- Advantages of random access over channel partitioning
Summary: Link Layer

- Several other important concepts
  - Flooding and self learning switch
  - VLAN
Recap: The Hourglass Architecture of the Internet
Outline

- Admin and recap
- Link layer
  - Overview
  - Random access
  - Channel partition link layer
- Software defined networking
  - Motivation: from simple forwarding to network functions
Simple Forwarding to Network Control Functions: Network Address Translation

- A local network uses just one public IP address as far as outside world is concerned
- Each device on the local network is assigned a private IP address

NAT gateway replaces source address with NAT IP address (e.g., 138.76.29.7) different source port numbers

All datagrams leaving local network cannot use private address use source address.
## Private IP

<table>
<thead>
<tr>
<th>RFC1918 name</th>
<th>IP address range</th>
<th>number of addresses</th>
<th>classful description</th>
<th>largest CIDR block (subnet mask)</th>
<th>host id size</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-bit block</td>
<td>10.0.0.0 – 10.255.255.255</td>
<td>16,777,216</td>
<td>single class A</td>
<td>10.0.0.0/8 (255.0.0.0)</td>
<td>24 bits</td>
</tr>
<tr>
<td>20-bit block</td>
<td>172.16.0.0 – 172.31.255.255</td>
<td>1,048,576</td>
<td>16 contiguous class Bs</td>
<td>172.16.0.0/12 (255.240.0.0)</td>
<td>20 bits</td>
</tr>
<tr>
<td>16-bit block</td>
<td>192.168.0.0 – 192.168.255.255</td>
<td>65,536</td>
<td>256 contiguous class Cs</td>
<td>192.168.0.0/16 (255.255.0.0)</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
NAT: Network Address Translation

Implementation: NAT router must:

- **outgoing datagrams**: replace (source IP address, port #) of every outgoing datagram to (NAT IP address, new port #)
  
  ... remote clients/servers will respond using (NAT IP address, new port #) as destination addr.

- **remember (in NAT translation table)** every (source IP address, port #) to (NAT IP address, new port #) translation pair

- **incoming datagrams**: replace (NAT IP address, new port #) in dest fields of every incoming datagram with corresponding (source IP address, port #) stored in NAT table
NAT: Network Address Translation

2: NAT router changes datagram source addr from 192.168.1.2, 3345 to 138.76.29.7, 5001, updates table

1: host 192.168.1.2 sends datagram to 128.119.40.186, 80

3: Reply arrives dest. address: 138.76.29.7, 5001

4: NAT router changes datagram dest addr from 138.76.29.7, 5001 to 192.168.1.2, 3345
Network Address Translation: Advantages

- No need to be allocated range of addresses from ISP: - just one public IP address is used for all devices
  - 16-bit port-number field allows 60,000 simultaneous connections with a single LAN-side address!
  - can change ISP without changing addresses of devices in local network
  - can change addresses of devices in local network without notifying outside world
- Devices inside local net not explicitly addressable, visible by outside world (a security plus)
Network Address Translation: Problems

- If both hosts are behind NAT, they will have difficulty establishing connection

- NAT is controversial:
  - Routers should process up to only layer 3
  - Violates end-to-end argument
    - NAT possibility must be taken into account by app designers, e.g., P2P applications
  - Address shortage should instead be solved by having more addresses --- IPv6!
Implication of Implementing NAT

NAT translation table

<table>
<thead>
<tr>
<th>WAN side addr</th>
<th>LAN side addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>138.76.29.7, 5001</td>
<td>192.168.1.2, 3345</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>

- Routing table no longer looks up by dest. IP, but also dest. Port
- Network needs to conduct packet transformation, beyond simple lookup
Simple Forwarding to Network Functions: Enterprise Networks

- Modern networks contain diverse types of equipment beyond simple routing/forwarding

```
Enterprise networks

Small: <=1k hosts; Medium: 1k-10k; Large: 10k-100k; Very Large: >= 100k
```

Source: [Sherry, et. al SIGCOMM’12]
Enterprise/Cloud Structure

- Tier 1
  - F1 (Firewall)
  - S1
  - LB1 (Load balancer)
  - IPS1 (Intrusion prevention)
  - S3

- Tier 2
  - F2
  - S2
  - LB2
  - IPS2
  - S4

- Tier 3
  - R1
  - IPS3
  - S5
  - S6

- Tier-1
  - VLAN 400
  - Tier-2
  - VLAN 300
  - Tier-3
  - VLAN 100
  - VLAN 200

- Internet

- Logger
Simple Forwarding to Network Functions

- OpenStack using OVS
Simple Forwarding to Network Control Functions: Mobile Networks

- Modern networks contain diverse types of equipment beyond simple routing/forwarding

![LTE Architecture Diagram]
Outline

- Admin and recap
- Link layer
- Software defined networking
  - Motivation: from simple forwarding to network functions
  - Design overview
Instead of distributed computing, directly setting up the state of network devices.

Discussion: key components of the architecture?
Datapath model:
- OpenFlow1.x
- OFDPA
- P4
- POF
...
**SDN Use Cases**

**SDN POWER USERS IN 2015**

**Top 4 ODL Use Cases**
- 1. NFV
- 2. Cloud Orchestration
- 3. Fulfillment
- 4. Network Monitoring and Analytics

**Top 3 Business Drivers**
- 1. Increased Interoperability and Portability
- 2. Increased Operational Efficiency
- 3. Greater Ability to Innovate and Compete

*Source: Neela Jacques, July 2015.*
Outline

- Admin and recap
- Link layer
- Software defined networking
  - Motivation: from simple forwarding to network functions
  - Design overview
  - Data path (OpenFlow)
OpenFlow Datapath

Ethernet Switch

port 1    port 2    port 3    port 4

5.6.7.8   1.2.3.4
OpenFlow Datapath

Software Layer

OpenFlow Firmware

Flow Table

<table>
<thead>
<tr>
<th>Match Fields</th>
<th>Priority</th>
<th>Counters</th>
<th>Instructions</th>
<th>Timeouts</th>
<th>Cookie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet + Byte Counters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forward to Port n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encapsulate and forward to controller</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send to normal processing pipeline</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify fields</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hardware Layer

<table>
<thead>
<tr>
<th>In Port</th>
<th>VLAN ID</th>
<th>Ethernet</th>
<th>IP</th>
<th>TCP</th>
<th>&amp; Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA</td>
<td>DA</td>
<td>Type</td>
<td>SA</td>
<td>DA</td>
<td>Proto</td>
</tr>
</tbody>
</table>

1.2.3.4

5.6.7.8

Controller

PC
Datapath Model: OpenFlow

<table>
<thead>
<tr>
<th>Match Fields</th>
<th>Priority</th>
<th>Counters</th>
<th>Instructions</th>
<th>Timeouts</th>
<th>Cookie</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Forward to Port n
Encapsulate and forward to controller
Drop
Send to normal processing pipeline
Modify fields

<table>
<thead>
<tr>
<th>VLAN ID</th>
<th>Ethernet</th>
<th>IP</th>
<th>TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SA</td>
<td>DA</td>
<td>Proto</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>SA</td>
<td>DA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Src</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Dst</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority</th>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>tcp_dst:22</td>
<td>drop</td>
</tr>
<tr>
<td>20</td>
<td>ip_dst: 101.22.0.0/16</td>
<td>port 2</td>
</tr>
<tr>
<td>1</td>
<td>in_port:1, mac_dst: 0xffffffffffffff</td>
<td>ports 2,3,4</td>
</tr>
</tbody>
</table>
OpenFlow Datapath: Example

OpenFlow Firmware

Flow Table

<table>
<thead>
<tr>
<th>MAC src</th>
<th>MAC dst</th>
<th>IP Src</th>
<th>IP Dst</th>
<th>TCP sport</th>
<th>TCP dport</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>5.6.7.8</td>
<td>*</td>
<td>port 1</td>
</tr>
</tbody>
</table>

Controller

PC

Software Layer

Hardware Layer

Flow Table:
- MAC src
- MAC dst
- IP Src
- IP Dst
- TCP sport
- TCP dport
- Action

Actions:
- port 1
- port 2
- port 3
- port 4

Computers:
- 5.6.7.8
- 5.6.7.8
- 1.2.3.4
# OpenFlow Datapath: Examples

## Switching

<table>
<thead>
<tr>
<th>Switch Port</th>
<th>MAC src</th>
<th>MAC dst</th>
<th>Eth type</th>
<th>VLAN ID</th>
<th>IP Src</th>
<th>IP Dst</th>
<th>IP Prot</th>
<th>TCP sport</th>
<th>TCP dport</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>00:1f:..</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>port6</td>
</tr>
</tbody>
</table>

## Flow Switching

<table>
<thead>
<tr>
<th>Switch Port</th>
<th>MAC src</th>
<th>MAC dst</th>
<th>Eth type</th>
<th>VLAN ID</th>
<th>IP Src</th>
<th>IP Dst</th>
<th>IP Prot</th>
<th>TCP sport</th>
<th>TCP dport</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>port3</td>
<td>00:20..</td>
<td>00:1f..</td>
<td>0800</td>
<td>vlan1</td>
<td>1.2.3.4</td>
<td>5.6.7.8</td>
<td>4</td>
<td>17264</td>
<td>80</td>
<td>port6</td>
</tr>
</tbody>
</table>

## Firewall

<table>
<thead>
<tr>
<th>Switch Port</th>
<th>MAC src</th>
<th>MAC dst</th>
<th>Eth type</th>
<th>VLAN ID</th>
<th>IP Src</th>
<th>IP Dst</th>
<th>IP Prot</th>
<th>TCP sport</th>
<th>TCP dport</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>22</td>
<td>drop</td>
</tr>
</tbody>
</table>
### OpenFlow Datapath: Examples

#### Routing

<table>
<thead>
<tr>
<th>Switch Port</th>
<th>MAC src</th>
<th>MAC dst</th>
<th>Eth type</th>
<th>VLAN ID</th>
<th>IP Src</th>
<th>IP Dst</th>
<th>IP Prot</th>
<th>TCP sport</th>
<th>TCP dport</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>5.6.7.8</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

#### VLAN Switching

<table>
<thead>
<tr>
<th>Switch Port</th>
<th>MAC src</th>
<th>MAC dst</th>
<th>Eth type</th>
<th>VLAN ID</th>
<th>IP Src</th>
<th>IP Dst</th>
<th>IP Prot</th>
<th>TCP sport</th>
<th>TCP dport</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>00:1f..</td>
<td>*</td>
<td>vlan1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
OpenFlow Message Flows: Reactive Mode (pkt miss)
OpenFlow Message Flows: Reactive Mode (flow_mod)

Controller

pktMiss
handler

OpenFlow Protocol flow_mod

pkt

OpenFlow Switch

Rule Action Statistics

Rule Action Statistics

Rule Action Statistics

OpenFlow Switch

OpenFlow Switch

OpenFlow Switch

OpenFlow Switch
OpenFlow Message Flows: Reactive Mode (forward)
Outline

- Admin and recap
- Link layer
- Software defined networking
  - Motivation: from simple forwarding to network functions
  - Design overview
  - Data path (OpenFlow)
  - Control path (programming model)
SDN Programming Model

Service/Policy

Network State

Program Logic

Datapath Model

Datapath Model
badPort = 22  // policy
hostTbl = {A:1,B:2,
           C:3,D:4}  // net view

def onPacketIn(p):
    if badPort == p.tcp_dst:
        drop
    else:
        forward([hostTbl(p.eth_dst)])

Assume only packets to A,B,C,D can appear.
hostTbl = {A:1,B:2,C:3,D:4}

def onPacketIn(p):
    if 22 == p.tcp_dst:
        drop

        installRule({'match':{'tcp_dst':22},
                    'action':[[]]})

    else:
        forward([hostTbl(p.eth_dst)])

    installRule({'match': {'eth_dst':p.eth_dst,
                         'tcp_dst':!-22},
             'action':[hostTbl(p.eth_dst)]})
hostTbl = {A:1,B:2,C:3,D:4}

def onPacketIn(p):
    if 22 == p.tcp_dst:
        drop

        installRule({'priority':1,
                     'match':{'tcp_dst':22},
                     'action':[]})

    else:
        forward([hostTbl(p.eth_dst)])

        installRule({'priority':0,
                     'match': {'eth_dst':p.eth_dst},
                     'action':[hostTbl(p.eth_dst)]})
def onPacketIn(p):
    if 22 == p.tcp_dst:
        drop
        installRule( {'priority':1,'match':{'tcp_dst':22},'action':[ ]})
    else:
        installRule( {'priority':0,'match':{'eth_dst':p.eth_dst},
                      'action':[hostTbl(p.eth_dst)]})
        forward([hostTbl(p.eth_dst)])

Does the Program Work?

Switch

{`priority`:0,'match':{'eth_dst':A},'action':[1]}

Controller

EthDst:A, TcpDst:80

A security bug!

EthDst:A, TcpDst:22
OVS Approach: Using Exact Match

hostTbl = \{A:1,B:2,C:3,D:4\}

def onPacketIn(p):
    if 22 == p.tcp_dst:
        drop

        installRule({'priority':1,
                     'match':exactMatch(p),
                     'action':[[]]})

    else:
        forward([hostTbl(p.eth_dst)])

    installRule({'priority':0,
                 'match':exactMatch(p),
                 'action':[hostTbl(p.eth_dst)]})
Problems of OVS Approach

- Each new TCP flow delayed for flow setup (10-100 ms)
- Number of flow table rules may exceed capacity (typically a few thousands)
Outline

- Admin and recap
- Link layer
- Software defined networking
  - Motivation: from simple forwarding to network functions
  - Design overview
  - Data path (OpenFlow)
  - Control path (programming model)
    - Problems
    - Algorithmic network programming
Goal of SDN Programming

Let programmers write the most obvious (=) data path independent) code:

```python
def onPacketIn(p):
    if 22 == p.tcp_dst:
        drop
    else:
        forward([hostTbl(p.eth_dst)])
```

Design a system that can automatically generate correct, highly-optimized data path.
• **Conceptually** programmer’s network control function $f$ is *invoked on every packet* entering the network.

• $f$ expressed in an *existing, general purpose language* (e.g., Java, Python), describing how a packet should be routed, **not** how data path flow tables are configured

$$f: \text{packet} \rightarrow \text{route}$$
Example Algorithmic Policy in Java

```java
Route f(Packet p) {
    if (p.tcpDstIs(22)) return null();
    else {
        Location sloc = hostTable(p.ethSrc());
        Location dloc = hostTable(p.ethDst());
        Route path = myRoutingAlg(topology(), sloc, dloc);
        return path;
    }
}

Route myRoutingAlg(Topology topo, Location sLoc, Location dloc) {
    if (isSensitive(sLoc) || isSensitive(dLoc))
        return secureRoutingAlg(topo, sloc, dloc);
    else
        return standardRoutingAlg(topo, sloc, dloc);
}
```

Does not specify anything on flow tables!
void insert(ACLItem acl, int pos, List<ACLItem> acls) {
    acls.add(pos, acl);
}

void delete(int pos, List<ACLItem> acls) {
    acls.remove(pos);
}

boolean permit(Packet p, List<ACLItem> acls) {
    for (ACLItem item : acls) {
        if (match(p, item)) {
            return item.action == PERMIT;
        }
    }
    return false;
}

List<ACLItem> acls;
Route f(Packet p) {
    if (!permit(p, acls)) {
        return null();
    }
    Location sloc = hostTable(p.ethSrc());
    Location dloc = hostTable(p.ethDst());
    Route path = myRoutingAlg(topology(), sloc, dloc);
    return path;
}
Challenge and Basic Idea

• Challenge: How to derive datapath (flow-tables) from a datapath-oblivious SDN controller function f?

• Basic idea of handling the challenge w/o a compiler:
  – There are two representations of computation
    • A sequence of instructions (whitebox)
    • Memorization tables (blackbox)
  – Although the decision function f does not specify how flow tables are configured, if for a given decision (e.g., drop), one know the dependency of the decision, one can construct the flow tables (aka, memorization tables).
Basic Idea

• Only requirement: Program f uses a simple library to access pkt attributes:

```
readPacketField :: Field -> Value
testEqual :: (Field, Value) -> Bool
ipSrcInPrefix :: IPPrefix -> Bool
ipDstInPrefix :: IPPrefix -> Bool
```

• Library provides both convenience and more importantly, decision dependency!
1. **Observes decision dependency** of $f$ on $pkt$ attributes.

2. Builds a **trace tree (TT)**, a universal (general), partial decision tree representation of any $f$.

3. **Compile** trace tree to generate flow tables (FTs).
Route f(Packet p) {
    if (p.tcpDstIs(22))
        return null();
    else {
        Location sloc = hostTable(p.ethSrc());
        Location dloc = hostTable(p.ethDst());
        Route path = myRoutingAlg(topology(), sloc, dloc);
        return path;
    }
}
Route f(Packet p) {
    if (p.tcpDstIs(22))
        return null();
    else {
        Location sloc = hostTable(p.ethSrc());
        Location dloc = hostTable(p.ethDst());
        Route path = myRoutingAlg(topology(),sloc,dloc);
        return path;
    }
}
Route f(Packet p) {
    if (p.tcpDstIs(22))
        return null();
    else {
        Location sloc = hostTable(p.ethSrc());
        Location dloc = hostTable(p.ethDst());

        Route path = myRoutingAlg(
            topology(), sloc, dloc);
        return path;
    }
}
Trace Tree Formal Spec

- A tree w/ 4 types of nodes
  - \(T\) node: assertion on packet attributes
  - \(V\) node: multi-way branching on a packet attribute
  - \(L\) node: leaf node labeled w/ action
  - \(?\) node: unknown

- **TT search**: traverse tree for a given packet to determine action or no answer

- **TT correctness**: if TT records an answer for a given packet, the answer is the same as the original algorithm
Trace Tree => Flow Table

match:{tcpDst==22}

match:{tcpDst!=22, ethDst:4, ethSrc:6}

match:{tcpDst!=22, ethDst:2}

match:{tcpDst==22}
Trace Tree => Flow Table

**Match:**
- **tcpDst == 22**
  - **True**:
    - **drop**
    - match:{tcpDst==22}
  - **False**:
    - **ethDst**
      - **2**
      - **drop**
      - match:{tcpDst!=22, ethDst:2}
    - **4**
    - **ethSrc**
      - **6**
      - **port 30**
      - match:{tcpDst!=22, ethDst:4, ethSrc:6}

**Barrier Rule:**
- match:{tcpDst==22}
- action:ToController
Simple, classical in-order tree traversal generates flow table rules!