CS434/534: Topics in Network Systems

Topic 3: Scalable, Programmable (Software-Defined) Networking: Software-Defined Networking Programming (Maple, Trident); P4

Nov. 16, 2021

Y. Richard Yang
Computer Science Department, Yale University
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/
Admin

Final Project planning
- Nov. 18: Iteration of topics
- Dec. 3: Checkpoint (2-page update)
- Dec. 22: Due (report)
Recap: 5GC Data Path

10.10.101.1
Recap: 5GC Control Plane (CP) Architecture

- Separate into data plane (DP) and control plane (CP)
- Service Base Architecture (SBA): each network function (NF) in CP exposes HTTP/2 ReSTful API
  - CP further divided into a data layer and a control layer
Recap: NLB/NAT Flow

- Increasingly, networking layer does more than forwarding packets, e.g.,
  - NAT addresses the problem of scarce IPv4 addresses, improving security
  - LB directs traffic to multiple servers
- One trend: move application logic into the network
Recap: Multiple Networking Plane Composition

Example: Traditional L2 + L3 Networking

- Setting: Host A L3NP receives a packet above.
- Action:
  - Host A looks up destination in FIB
  - Best match has next router C (223.1.1.4)
    - Hand datagram to L2NP to send inside a L2NP frame

<table>
<thead>
<tr>
<th>Dest. Net.</th>
<th>next router</th>
<th>Nhops</th>
</tr>
</thead>
<tbody>
<tr>
<td>223.1.1/24</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>223.1.2/24</td>
<td>223.1.1.4</td>
<td>2</td>
</tr>
<tr>
<td>223.1.3/24</td>
<td>223.1.1.4</td>
<td>2</td>
</tr>
<tr>
<td>0.0.0.0/0</td>
<td>223.1.1.4</td>
<td>-</td>
</tr>
</tbody>
</table>

- Frame: src, dst addr
- Datagram source, dest address
- A’s MAC addr
- C’s MAC addr
- A’s IP addr
- E’s IP addr
- IP payload
- To Internet
Recap: SDN Main Ideas

- Separation of Data and Control Path
- Common data path abstraction
Recap: OpenFlow 1.x Datapath: Basic Abstraction

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

**Packet + byte counters**

1. Forward packet to zero or more ports
2. Encapsulate and forward to controller
3. Send to normal processing pipeline
4. Modify Fields
5. Any extensions you add!

+ mask what fields to match

Source: Scott Shenker, UC Berkeley
OpenFlow in Production

- Google Network Control Plane Orion

Recap: Higher-Level Logic with Datapath Generation

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)])

Benefit: a “thread” like view of packet control

Issue: Extracting correct flow rules can be tricky
Outline

- Admin and recap
- Scalable, programmable (software-defined) networking
  - Overview and roadmap
  - Scaling networking from a switch to the whole Internet
  - Software-defined networking
    - Overview
    - OpenFlow SDN
      - OpenFlow datapath
      - High-level OpenFlow datapath programming
        » User programming
        » Automatic, algorithmic SDN programming
Programmers write only the logic (=> data path independent):
- specify how a packet should be treated by the networking system
- has little constraints on programming (e.g., can use any control structures such as loops in a generic programming language)
  - do not invent a new language
System automatically generates correct, optimized data path
Programming Model: Programmer’s View

- 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 High-level Program

```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); // loop inside
    else
        return standardRoutingAlg(topo, sloc, dloc); // loop inside
}
```
Basic Ideas of Datapath Generation

- **Insight**
  - Although the decision function \( f \) does not specify how flow tables are configured, if for a given decision (e.g., drop), one can know the dependency on the input (pkt attributes) of the decision, one can construct the flow table.

- **Requirement**
  - Program \( f \) uses a library to access pkt attributes
  - Library provides both convenience and more importantly, decision dependency

- **Basic data structure**
  - Trace tree

```plaintext
readPacketField :: Field -> Value
testEqual :: (Field, Value) -> Bool
ipSrcInPrefix :: IPPrefix -> Bool
ipDstInPrefix :: IPPrefix -> Bool
```
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;
    }
}
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 Formalism

- 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

If there is a match on the trace tree, the tree and the program give the same result, true or false?

- Read: EthSrc
- Read: EthDst
- Assert: TcpDst==22
- True
- False

path1
Is the trace tree for a single device or it can do multiple devices?
Trace Tree => Datapath

```
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}
```
Trace Tree => Datapath

```
match: {tcpDst==22}
```

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

```
barrier rule:
match: {tcpDst==22, ethDst:2}
```

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

```
action: ToController
```

Priority
Offline Exercise: Write a FW Interpreter and Construct TT

- Assume Linux iptables format
  - Match
    - -p protocol
    - -s address/mask
    - -d address/mask
    - --sport port
    - --dport port
  - Action
    - -j ACCEPT | REJECT

- Example: allow only http (tcp port 80) traffic, block all others
  - -p tcp --dports 80 -j ACCEPT
  - -j REJECT
Outline

- Admin and recap
- Scalable, programmable (software-defined) networking
  - Overview and roadmap
  - Scaling networking from a switch to the whole Internet
  - Software-defined networking
    - Overview
    - OpenFlow SDN
      - OpenFlow datapath
      - High-level datapath programming
        » User programming
        » Automatic, algorithmic SDN programming
          » Basic idea: automatic dependency tracing
Problem: Optimizing Traces

f1: access

f2: routing
Example: Trace Reduction

f1: access

pkt.tcpDstEquals(82)

pkt.tcpDst()

pkt.tcpDst()

pkt.tcpDstEquals(80)

f2: routing

pkt.tcpDstEquals(81)

pkt.tcpDst()

Set of packets matched on trace: true
Example: Trace Reduction

\[ f_1: \text{access} \]

- pkt.tcpDstEquals(82)
- pkt.tcpDst()
- pkt.tcpDst()
- pkt.tcpDstEquals(80)

\[ f_2: \text{routing} \]

- tcpDst==82: false
- tcpDst!==82
- Read(tcpDst)
- tcpDst==81
- Read(tcpDst)
- tcpDst==81
- Read(tcpDst)
- tcpDst==81
- Read(tcpDst)
- tcpDst==80
- Read(tcpDst)

Set of packets matched on trace: true
Example: Trace Reduction

\[ f1: \text{access} \]

\[ \begin{align*}
    & \text{pkt.tcpDstEquals(82)} \\
    & \text{pkt.tcpDst()} \\
    & \text{pkt.tcpDst()} \\
    & \text{pkt.tcpDstEquals(80)} \\
    & \text{pkt.tcpDst()} \\
\end{align*} \]

\[ \text{tcpDst==82} \]
\[ \text{false} \]

\[ \text{Read(tcpDst)} \]
\[ 81 \]

\[ \text{tcpDst!=82 \land \ true} \]

\[ f2: \text{routing} \]

\[ \begin{align*}
    & \text{pkt.tcpDst()} \\
    & \text{pkt.tcpDst()} \\
\end{align*} \]

\[ \text{tcpDst==81} \]
\[ \text{true} \]

\[ \text{Read(tcpDst)} \]
\[ 81 \]
Example: Trace Reduction

f1: access

pkt.tcpDstEquals(82)

pkt.tcpDst()

pkt.tcpDst()

pkt.tcpDstEquals(80)

pkt.tcpDstEquals(81)

pkt.tcpDst()

f2: routing

Set of packets matched on trace: true

tcpDst==82 \ true = tcpDst!=82

tcpDst==82

false

Read(tcpDst)

81

Read(tcpDst)

81

tcpDst==80

false

Read(tcpDst)

81

tcpDst==81

true

Read(tcpDst)

81
Example: Trace Reduction

\[
\begin{align*}
f1: & \text{access} \\
pkt.\text{tcpDstEquals}(82) & \text{tcpDst==82} \text{ true} \\
pkt.\text{tcpDst()} & \text{tcpDst!=82} \text{ true} \\
pkt.\text{tcpDst()} & \\
pkt.\text{tcpDstEquals}(80) & \\
pkt.\text{tcpDstEquals}(81) & \\
pkt.\text{tcpDst()} & \\
\end{align*}
\]

\[
\begin{align*}
f2: & \text{routing} \\
\text{Set of packets matched on trace: true} \\
\text{tcpDst==82} & \\
\text{tcpDst=81} & \text{tcpDst!=82} \\
\end{align*}
\]
Example: Trace Reduction

f1: access

pkt.tcpDstEquals(82)

pkt.tcpDst()  

pkt.tcpDst()  

pkt.tcpDstEquals(80)

pkt.tcpDstEquals(81)

pkt.tcpDst()  

Set of packets matched on trace: true

tcpDst!=82 \ true = tcpDst=82

tcpDst=81 \ tcpDst!=82 = tcpDst=81

tcpDst=81 \ tcpDst=81 = tcpDst=81

tcpDst!=80 \ tcpDst=80 = tcpDst=81

tcpDst=81 \ tcpDst=81 = tcpDst=81

f2: routing

Read(tcpDst)

Read(tcpDst)

Read(tcpDst)

Read(tcpDst)

Read(tcpDst)
Example: Rule Update Reduction

disjoint, could use same priority.
(Offline) Priority Level Reduction: Increasing at T Nodes Only
(Offline) Priority Level Reduction: Increasing at T Nodes Only

Priority | Match                      | Action
----------|----------------------------|--------
1         | tcp_dst:22                 | drop   
0         | eth_dst:2                  | drop   
0         | eth_dst:4, eth_src:6       | port 30

Priority | Match                      | Action
----------|----------------------------|--------
1         | tcp_dst:22                 | drop   
0         | eth_dst:2                  | drop   
0         | eth_dst:4, eth_src:6       | port 30
0         | eth_dst:5, eth_src:6       | port 40
Summary: Basic Ideas of Automatic, General-Language Datapath Generation

- **Insight**
  - Although the decision function $f$ does **not** specify how flow tables are configured, if for a given decision (e.g., drop), one can know the dependency on the input (pkt attributes) of the decision, one can construct the flow table.

- **Requirement**
  - Program $f$ uses a library to access pkt attributes and environment state variables, to track dependency access

- **Basic data structures and algorithms**
  - Trace tree: tracking dependency on pkt attributes
Exercise

What if the OpenFlow device has multiple tables forming a pipeline?
Exercise

- What if reactive installation is not acceptable, but the set of representative packets is known?
Exercise: What if the Environment Changes?

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)])
Potential Approaches on Handling Dynamic Environment State

- **Timeout**
  - Wait for timeout of existing rules (e.g., Floodlight)
  - Problem: long delay; may not even be correct

- **Rerun**
  - Completely rerun program and then compare the differences (e.g., Pyretic)
  - Problem: low efficiency
Map<MAC, Location> hostTable;
List<ACLItem> acls;

Route f(Packet p) {
    hostTable.put(p.ethSrc(), p.ingressPort());
    if ( !permit(p, acls) ) return drop;
    Location src = p.ingressPort();
    Location dst = hostTable.get(p.ethDst());
    Route path = myRoutingAlg(topology(), src, dst);
    return path;
}
Unified Dependency Tracking

- Control program f uses simple library wrappers to access pkt attributes and state variables

- Reason: Provides both convenient data structures and importantly, again, decision dependency!
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;
    }
}
```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 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;
  }
}
Example Dependency Table

**Assert:**
\[ TcpDsd == 22 \]
\[ false \]
\[ true \]

**Read:**
\[ EthDst \]
\[ EthDst \]

**Component**

<table>
<thead>
<tr>
<th>Component</th>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>(hostTable, 1)</td>
<td>[false, 1]</td>
</tr>
<tr>
<td>(hostTable, 2)</td>
<td>[false, 1,2]</td>
</tr>
<tr>
<td>(hostTable, 3)</td>
<td>[false, 3]</td>
</tr>
<tr>
<td>(hostTable, 4)</td>
<td>[false, 3,4]</td>
</tr>
<tr>
<td>topology</td>
<td>[false,1,2],[false,3 ,4]</td>
</tr>
</tbody>
</table>
Tradeoffs Between Tracking Overhead and Precision

assert: TcpDsd==22

true

false

read: EthSrc

1

3

read: EthDst

2

4

(hostTable, 1)

(hostTable, 2)

(links, {1,2,3})

Dijkstra will visit only 3 links. There is no dependency on link[4].

Assume Dijkstra.
Using Dependency Table

**Assert:** TcpDsd==22

- `true` branch
- `false` branch

**Read:** EthSrc

- `1` (hostTable, 1)
- `3` (hostTable, 3)

**Read:** EthDst

- `2` (hostTable, 2)
- `4` (hostTable, 4)

**Dependency Table (aka inverted index):**

<table>
<thead>
<tr>
<th>Component</th>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>(hostTable,1)</td>
<td>[false, 1]</td>
</tr>
<tr>
<td>(hostTable,2)</td>
<td>[false, 1,2]</td>
</tr>
<tr>
<td>(hostTable,3)</td>
<td>[false, 3]</td>
</tr>
<tr>
<td>(hostTable,4)</td>
<td>[false, 3,4]</td>
</tr>
<tr>
<td>topology</td>
<td>[false,1,2],[false,3,4]</td>
</tr>
</tbody>
</table>

host 4 changes location =>

- Uses dependency table to locate dependent entries
Using Dependency Table

**Dependency table** (aka inverted index)

<table>
<thead>
<tr>
<th>Component</th>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>(hostTable,1)</td>
<td>[false, 1]</td>
</tr>
<tr>
<td>(hostTable,2)</td>
<td>[false, 1,2]</td>
</tr>
<tr>
<td>(hostTable,3)</td>
<td>[false, 3]</td>
</tr>
<tr>
<td>(hostTable,4)</td>
<td>[false, 3,4]</td>
</tr>
<tr>
<td>topology</td>
<td>[false,1,2],[false,3,4]</td>
</tr>
</tbody>
</table>

- host 4 changes location =>
  - Uses dependency table to locate dependent entries

Discussion: What can the system do?
Exercise

- Algorithmic SDN controls a network using only L2-L4 information. What if you want to control network using application L7 information (e.g., HTTP URI), i.e., application-defined networking?
Application-Defined Networking: Challenge

- Application-defined (L7) networking information is **NOT** contained in each/one packet header field
  - Comparison: L2-L4 SDN programming is natural as all L2-L4 information is contained in every single packet.

- Application information need to be extracted by network functions such as Deep Packet Inspection or joint network-application programming

Examples from Kinetic (Kim et al.) and Resonance (Kim et al.)
Basic Idea: Packet Attribute => Stream Attribute

*stream attribute*, to represent application-layer (L7) state as if it is a header field so that programmer can select packets based on the state

**Requirement:** L7 state is bound to a stream: a sequence of packets with certain common header fields.

**Example**

Consider a packet with 5-tuple: \(<10.0.0.2, 10.0.0.2, 1234, 80, tcp>\)

- HTTP URI: The HTTP URI information extracted for TCP connection with the 5-tuple: \(<10.0.0.2, 10.0.1.2, 1234, 80, tcp>\)
- Heavy hitter (source): Whether the source IP address (10.0.0.2) is classified as a heavy hitter
Stream Attribute: Detail

Define a stream attribute:

```scala
define http_uri = StreamAttribute[String]("http_uri", TCP5TUPLE)
```

- The type information, e.g., String, Int
- A descriptive name, e.g., HTTP_URI, authenticated
- The stream type (bit masks on packet header fields), e.g., TCP5TUPLE, SRC_IPADDR, DST_IPADDR
Use a stream attribute just like a packet header
pkt.http_uri, pkt.authenticated, pkt.heavy_hitter, ...

// 3-way branch
if ((pkt.authenticated)
    && (pkt.http_uri === "www.xyz.com")) {
    // true branch
} else {
    // else branch
} unknown {
    // unknown branch; if no unknown->else
}

Under the hood, a stream attribute is a virtual variable that is computed from another execution thread

Issue: Stream attribute is extracted by finite state machine, and MAY have an unknown value

Solution: Kleene’s 3-value logic

<table>
<thead>
<tr>
<th>∧</th>
<th>T</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>F</td>
<td>U</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>∨</th>
<th>T</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>

Truth tables for ∧ and ∨ in Kleene’s 3-valued logic
(T - True, F - False, U - Unknown)
Discussion: Algorithmic SDN Programming using Trace Tree

- **Pros**
  - Simple, language independent

- **Limitations**
  - Algorithmic policy instead of declarative policy
  - Reaction latency: A reactive approach that waits for punted packets (or warmup packets) to unfold the trace tree and generate rules; warm-up can help, but not a complete solution
Outline

- Admin and recap
- Scalable, programmable (software-defined) networking
  - Overview and roadmap
  - Scaling networking from a switch to the whole Internet
  - Software-defined networking
    - Overview
    - OpenFlow SDN
      - OpenFlow datapath
      - High-level datapath programming
        - User programming
        - Automatic, algorithmic SDN programming
          - Basic idea: automatic dependency tracing
          - Extension: L2-4 matching => L2-7 matching
          - Extension: Algorithmic programming => algorithmic + declarative Domain Specific Language
Overview

- Benefits of Domain Specific Language (DSL)
  - Programmer: more compact grammar
  - System: understanding of semantics, instead of blackbox operations/programs

- Issues of switching to DSL
  - Typically hard to match with the flexibility and support of a general purpose language

- Maple+Trident design point
  - Embed DSL in a general purpose language
  - Multiple DSL functions, we use routing DSL (Route Algebra) as an example
Example: Route Construction Complexity: Consistent, Correlated Routes

Requirement: The return path must be the inverse of the forward path, i.e., symmetry.

If the forward and reverse paths are computed independently using the shortest path algorithm, the requirement may not be satisfied.
**Declarative SDN DSL: Route Algebra**

### Union (∪)/Intersection (∩)/Difference (∖)
Given two route set $\Delta_1$ and $\Delta_2$, return the union/intersection/difference of $\Delta_1$ and $\Delta_2$:
- $\Delta_1 \cup \Delta_2 = \{ r \mid r \in \Delta_1 \lor r \in \Delta_2 \}$
- $\Delta_1 \cap \Delta_2 = \{ r \mid r \in \Delta_1 \land r \in \Delta_2 \}$
- $\Delta_1 \setminus \Delta_2 = \{ r \mid r \in \Delta_1 \land r \notin \Delta_2 \}$

### Concatenation (+)
Given two route sets $\Delta_1$ and $\Delta_2$, return a new route set by concatenating all route pairs $(r_1, r_2)$ in $\Delta_1 \times \Delta_2$ and removing the invalid ones:
- $\Delta_1 + \Delta_2 = \{ r_1 + r_2 \mid r_1 \in \Delta_1, r_2 \in \Delta_2, \text{dst}_{r_1} = \text{src}_{r_2} \}$

### Inversion (∼)
Given a route set $\Delta$, return the inverse of $r \in \Delta$:
- $\times \Delta = \{ \times r \mid r \in \Delta \}$

### Preference (≻)
Given two route sets $\Delta_1$ and $\Delta_2$, return the preferred route. (If there is an equivalent route in $\Delta_1$, do not use the ones in $\Delta_2$):
- $\Delta_1 \succ \Delta_2 = \{ r \mid r \in \Delta_1 \lor (r \in \Delta_2 \land \nexists r' \in \Delta_1, r \sim r') \}$

### Selection (σ)
Given a route set $\Delta$ and an evaluation function $f : R^+ \to \{0, 1\}$, return all routes in $\Delta$ that are evaluated as 1:
- $\sigma_f(\Delta) = \{ r \in \Delta \mid f(r) = 1 \}$

### Optimal selection (⌃)
Given one route set $\Delta$ and a routing cost function $d : R^+ \to R$, return any route with the minimum value:
- $\overset{\overset{\sim}{\sim}}{\sigma_d}(\Delta) = \underset{r \in \Delta}{\arg \min} d(r)$

### Arbitrary selection (∗)
Given one route set $\Delta$, return a route set containing exactly one route $r$ in $\Delta$:
- $\ast \Delta = \overset{\overset{\sim}{\sim}}{\sigma_1}(\Delta)$

- The basic unit of route algebra is route set.
Route Algebra: Exercise

- Select a route from host H to gateway GW using only link with 100G capacity. If no such path, pick one with links with 10G capacity.

  \[ (\sigma_{cap=100Gbps} (H : \rightarrow : GW) \triangleright \sigma_{cap=10Gbps} (H : \rightarrow : GW)) \]
Putting Together: L7+DSL Dependency Tracking

Example Policy: Block traffic for infected hosts and construct routes using concatenation

Events:

- When any network component on $r_1$ or $r_2$ changes, the new route $(r_1 + r_2)$ is automatically recomputed.
- When the host status changes to infected, all packets are dropped.
- When the host status is cleared (e.g., through an admin interface or timeout), a route $r_1 + r_2$ is automatically recomputed.

```
iff (pkt.is_endhost_infected) {
    drop(pkt)
} else {
    bind(pkt, r_1 + r_2)
}
```
DSL Benefit: Efficient Computation

Example:

val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)

- Blackbox programming: system waits until r finishes computation
- DSL: system returns after ps is computed and not null; rest of computation continue in the background
**DSL Benefit: Efficient Computation**

**Example:**

```scala
val p = ShortestPath(G, s, t)
val ps = snapshot(p)
val b = ffr(G, ps)
val r = any(ps >> b)
```

At time 0: no routes is ready

At time 1: the primary route p0 is ready. With efficient update, Trident selects p0 as r.

At time 2: the backup route b0 is also ready. The standard update will selects p0 as r.

At time 3: a data change e happens and invalidates the primary route.

At time 3: now p is not ready, Trident selects b0 as r.

At time 4: p has a new value p1. With efficient update, Trident selects p1 as r.

With this simple code, Trident achieves lifecycle management for backup routes.
(Offline) Efficient Computation

The general rule: For all route algebra operators, if the partial result has no unknown subsets, the output guarantees glitch-free consistency and we can apply efficient update.

<table>
<thead>
<tr>
<th>Expr</th>
<th>Known Subset</th>
<th>Unknown Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta_1 \cup \Delta_2$</td>
<td>$\mathcal{K}_1 \cup \mathcal{K}_2$</td>
<td>$\mathcal{U}_1 \cup \mathcal{U}_2$</td>
</tr>
<tr>
<td>$\Delta_1 \cap \Delta_2$</td>
<td>$\mathcal{K}_1 \cap \mathcal{K}_2$</td>
<td>$(\mathcal{K}_1 \cap \mathcal{U}_2) \cup (\mathcal{U}_1 \cap \mathcal{K}_2) \cup (\mathcal{U}_1 \cap \mathcal{U}_2)$</td>
</tr>
<tr>
<td>$\Delta_1 \setminus \Delta_2$</td>
<td>$\mathcal{T}_{\mathcal{U}_2 = \emptyset}(\mathcal{K}_1 - \mathcal{K}_2)$</td>
<td>$(\mathcal{T}_{\mathcal{U}_2 = \emptyset}(\mathcal{K}_1) \cup \mathcal{U}_1) - (\mathcal{K}_2 \cup \mathcal{U}_2)$</td>
</tr>
<tr>
<td>$\Delta_1 + \Delta_2$</td>
<td>$\mathcal{K}_1 + \mathcal{K}_2$</td>
<td>$(\mathcal{K}_1 + \mathcal{U}_2) \cup (\mathcal{U}_1 + \mathcal{K}_2) \cup (\mathcal{U}_1 + \mathcal{U}_2)$</td>
</tr>
<tr>
<td>$\preceq \Delta$</td>
<td>$\preceq \mathcal{K}$</td>
<td>$\succeq \mathcal{U}$</td>
</tr>
<tr>
<td>$\sigma_f(\Delta)$</td>
<td>$\sigma_f(\mathcal{K})$</td>
<td>$\sigma_f(\mathcal{U})$</td>
</tr>
<tr>
<td>$\diamond_d (\Delta)$</td>
<td>$\mathcal{T}_{\mathcal{U} = \emptyset}(\diamond_d (\mathcal{K}))$</td>
<td>$\diamond_d (\mathcal{T}_{\mathcal{U} = \emptyset}(\diamond_d (\mathcal{K})) \cup \diamond_d (\mathcal{U}))$</td>
</tr>
<tr>
<td>$\Delta_1 \vartriangleright \Delta_2$</td>
<td>$\mathcal{K}<em>1 \cup \mathcal{T}</em>{\mathcal{U}_1 = \emptyset}(\mathcal{K}_2 - \mathcal{K}_1)$</td>
<td>$\mathcal{U}<em>1 \cup ((\mathcal{T}</em>{\mathcal{U}_1 = \emptyset}(\mathcal{K}_2) \cup \mathcal{U}_2) \setminus (\mathcal{K}_1 \cup \mathcal{U}_1))$</td>
</tr>
<tr>
<td>$*$</td>
<td>$\ast \mathcal{K}$</td>
<td>$\ast \mathcal{U}$</td>
</tr>
</tbody>
</table>

$T_{\varepsilon}(S)$ - the value is $S \cup \{\varepsilon\}$ if $\varepsilon = \text{true}$, and $\{\varepsilon\}$ otherwise.
Summary

- Designing network-wide, high level SDN programming model remains a challenge
  - Datapath model continues to evolve (OpenFlow -> P4)
  - Scalability and robustness of a single compiler/interpreter
  - We will revisit control current large-scale SDN controllers in topic 5
Discussion: Limitations of OpenFlow Based Datapath Model

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

- Match Fields
- Action
- Stats

Packet + byte counters

1. Forward packet to zero or more ports
2. Encapsulate and forward to controller
3. Send to normal processing pipeline
4. Modify Fields
5. Any extensions you add!

Switching

<table>
<thead>
<tr>
<th>Switch Port</th>
<th>VLAN ID</th>
<th>VLAN pcp</th>
<th>MAC src</th>
<th>MAC dst</th>
<th>Eth type</th>
<th>IP Src</th>
<th>IP Dst</th>
<th>IP ToS</th>
<th>IP Prot</th>
<th>L4 sport</th>
<th>L4 dport</th>
</tr>
</thead>
</table>

+ mask what fields to match

Source: Scott Shenker, UC Berkeley
Programming Protocol-independent Packet Processors (P4): Goals

- Motivated by limitations of OpenFlow 1.x, aka OpenFlow 2.0
- Reconfigurability
  - controller can redefine the packet parsing and processing in the field
- Protocol independence
  - Not tied to specific packet formats: controller can specify (i) a packet parser for extracting header fields with particular names and types and (ii) a collection of typed match+action tables that process headers
- Target independence
  - as a C programmer does not need to know the specifics of CPU, the controller programmer need not know the details of the underlying switch
Traditional vs P4 Switches

Traditional switch

Control plane

Data plane

Table mgmt
Control traffic

Packets

P4-defined switch

Control plane

Data plane

P4 Program
P4 table mgmt
P4 Abstract Forwarding Model

https://p4.org/p4-spec/docs/P4-16-v1.2.2.html
Example: The Very Simple Switch (VSS) Architecture
VSS Architecture Specification

```c
// File "very_simple_switch_model.p4"
// Very Simple Switch P4 declaration
// core library needed for packet_in and packet_out definitions
// include <core.p4>
/* Various constants and structure declarations */
/* ports are represented using 4-bit values */
typedef bit<4> PortId;
/* only 8 ports are "real" */
const PortId REAL_PORT_COUNT = 4w8; // 4w8 is the number 8 in 4 bits
/* metadata accompanying an input packet */
struct InControl {
    PortId inputPort;
}
/* special input port values */
const PortId RECIRCULATE_IN_PORT = 0xD;
const PortId CPU_IN_PORT = 0xE;
/* metadata that must be computed for outgoing packets */
struct OutControl {
    PortId outputPort;
}
/* special output port values for outgoing packet */
const PortId DROP_PORT = 0xF;
const PortId CPU_OUT_PORT = 0xF;
const PortId RECIRCULATE_OUT_PORT = 0xD;
/* Prototypes for all programmable blocks */
/**
 * Programmable parser.
 * @param <H> type of headers; defined by user
 * @param b input packet
 * @param parsedHeaders headers constructed by parser
 */
parser Parser<H>{packet_in b,
    out H parsedHeaders};

/**
 * Match-action pipeline
 * @param <H> type of input and output headers
 * @param headers headers received from the parser and sent to the deparser
 * @param parseError error that may have surfaced during parsing
 * @param inCtrl information from architecture, accompanying input packet
 * @param outCtrl information for architecture, accompanying output packet
 */
control Pipe<H>{
inout H headers,
    in error parseError, // parse error
    in InControl inCtrl, // input port
    out OutControl outCtrl}; // output port
/**
 * VSS deparser.
 * @param <H> type of headers; defined by user
 * @param b output packet
 * @param outputHeaders headers for output packet
 */
control Deparser<H>{
inout H outputHeaders,
    packet_out b};
/**
 * Top-level package declaration - must be instantiated by user.
 * The arguments to the package indicate blocks that
 * must be instantiated by the user.
 * @param <H> user-defined type of the headers processed.
 */
package VSS<H>(Parser<H> p,
    Pipe<H> map,
    Deparser<H> d);
// Architecture-specific objects that can be instantiated
extern Checksum16 {
    Checksum16(); // constructor
    void clear(); // prepare unit for computation
    void update<T>{in T data}; // add data to checksum
    void remove<T>{in T data}; // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```
Example: Realizing L2+L3 Composition using VSS

Setting: Packet above arrives at Router C’s L3NP
Action:
- Router C looks up destination in FIB
- Hand packet to L2NP

<table>
<thead>
<tr>
<th>Dest. Net</th>
<th>router</th>
<th>Nhops</th>
<th>interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>223.1.1/24</td>
<td>-</td>
<td>1</td>
<td>223.1.1.4</td>
</tr>
<tr>
<td>223.1.2/24</td>
<td>-</td>
<td>1</td>
<td>223.1.2.9</td>
</tr>
<tr>
<td>223.1.3/24</td>
<td>-</td>
<td>1</td>
<td>223.1.3.27</td>
</tr>
<tr>
<td>0.0.0.0/0</td>
<td>-</td>
<td>-</td>
<td>223.1.4.1</td>
</tr>
</tbody>
</table>
Example: VSS Instantiation
Parser

#include <core.p4>

#include "very_simple_switch_model.p4"

This program processes packets comprising an Ethernet and an IPv4
// header, and it forwards packets using the destination IP address

typedef bit<48> EthernetAddress;
typedef bit<32> IPv4Address;

// Standard Ethernet header
header Ethernet_h {
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
    bit<16> etherType;
}

// IPv4 header (without options)
header IPv4_h {
    bit<4> version;
    bit<4> ihl;
    bit<8> diffserv;
    bit<16> totalLen;
    bit<16> identification;
    bit<3> flags;
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    IPv4Address srcAddr;
    IPv4Address dstAddr;
}

// Structure of parsed headers
struct Parsed_packet {
    Ethernet_h ethernet;
    IPv4_h ip;
}

// Parser section

// User-defined errors that may be signaled during parsing
error {
    IPv4OptionsNotSupported,
    IPv4IncorrectVersion,
    IPv4ChecksumError
}

parser TopParser(packet_in b, out Parsed_packet p) {
    Checksum16() ck; // instantiate checksum unit

    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            0x0800: parse_ipv4;
            // no default rule: all other packets rejected
        }
    }

    state parse_ipv4 {
        b.extract(p.ip);
        verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);
        verify(p.ip.ihl == 4w5, error.IPv4OptionsNotSupported);
        ck.clear();
        ck.update(p.ip);
        // Verify that packet checksum is zero
        verify(ck.get() == 16w0, error.IPv4ChecksumError);
        transition accept;
    }
control TopPipe\[inout\] Parsed_packet headers,
in error parseError, // parser error
in InControl inCtrl, // input port
out OutControl outCtrl) {
  IPv4Address nextHop; // local variable

  /**
   * Indicates that a packet is dropped by setting the
   * output port to the DROP_PORT
   */
  action Drop_action() {
    outCtrl.outputPort = DROP_PORT;
  }

  /**
   * Set the next hop and the output port.
   * Decrement ipv4_ttl field.
   * @param ipv4_dest IPv4 address of next hop
   * @param port output port
   */
  action Set_nhop(IPv4Address ipv4_dest, PortId port) {
    nextHop = ipv4_dest;
    headers.ip.ttl = headers.ip.ttl - 1;
    outCtrl.outputPort = port;
  }

  /**
   * Computes address of next IPv4 hop and output port
   * based on the IPv4 destination of the current packet.
   * Decrement packet IPv4 TTL.
   * @param nextHop IPv4 address of next hop
   */
  table ipv4_match {
    key = { headers.ip.dstAddr: lpm; } // longest-prefix match
    actions = {
      Drop_action;
      Set_nhop;
    }
    size = 1024;
    default_action = Drop_action;
  }

  /**
   * Send the packet to the CPU port
   */
  action Send_to_cpu() {
    outCtrl.outputPort = CPU_OUT_PORT;
  }

  /**
   * Check packet TTL and send to CPU if expired.
   */
  table check_ttl {
    key = { headers.ip.ttl: exact; }
    actions = { Send_to_cpu; NoAction; }
    const default_action = NoAction; // defined in core.p4
  }

  /**
   * Set the destination MAC address of the packet
   * @param dmac destination MAC address.
   */
  action Set_dmac(EthernetAddress dmac) {
    headers.ethernet.dstAddr = dmac;
  }

  /**
   * Set the destination Ethernet address of the packet
   * based on the next hop IP address.
   * @param nextHop IPv4 address of next hop.
   */
  table dmac {
    key = { nextHop: exact; }
    actions = {
      Drop_action;
      Set_dmac;
    }
    size = 1024;
    default_action = Drop_action;
  }
**
* Set the source MAC address.
* @param smac: source MAC address to use
*/

action Set_smac(EthernetAddress smac) {
    headers.ethernet.srcAddr = smac;
}

/**
* Set the source mac address based on the output port.
*/
table smac {
    key = { outCtrl.outputPort: exact; }
    actions = {
        Drop_action;
        Set_smac;
    }
    size = 16;
    default_action = Drop_action;
}

apply {
    if (parseError != error.NoError) {
        Drop_action(); // invoke drop directly
        return;
    }

    ipv4_match.apply(); // Match result will go into nextHop
    if (outCtrl.outputPort == DROP_PORT) return;

    check_ttl.apply();
    if (outCtrl.outputPort == CPU_OUT_PORT) return;

    dmac.apply();
    if (outCtrl.outputPort == DROP_PORT) return;

    smac.apply();
}
// deparser section
control TopDeparser(inout Parsed_packet p, packet_out b) {
    Checksum16() ck;
    apply {
        b.emit(p.ethernet);
        if (p.ip.isValid()) {
            ck.clear(); // prepare checksum unit
            p.ip.hdrChecksum = 16w0; // clear checksum
            ck.update(p.ip); // compute new checksum.
            p.ip.hdrChecksum = ck.get();
        }
        b.emit(p.ip);
    }
}
// Instantiate the top-level VSS package
VSS(TopParser(),
    TopPipe(),
    TopDeparser()) main;
Thinking Exercise

- What is a high-level programming model for P4 devices?