States on a (Data) Plane

Jennifer Rexford
Traditional data planes are stateless
Software Defined Networks (SDN)

Program your network from a logically **central point**!
OpenFlow Rule Tables

<table>
<thead>
<tr>
<th>Prio</th>
<th>match</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dstip = 10.0.0.1</td>
<td>outport ← 1</td>
</tr>
<tr>
<td>2</td>
<td>dstip = 10.0.0.2</td>
<td>drop</td>
</tr>
</tbody>
</table>
Two-Tiered Programming Model

• **Stateless** data-plane rules
  – Process each packet independently
  – State updates are limited to traffic counters

• **Stateful** control-plane program
  – Store and update state in the controller application
  – Adapt by installing new rules in the switches

Forces packets to go to the controller... or greatly limits the set of applications
Emerging switches have stateful data planes
## Local State on Data Plane

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2</td>
<td>5</td>
</tr>
<tr>
<td>H1</td>
<td>99</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Local State on Data Plane

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2</td>
<td>5</td>
</tr>
<tr>
<td>H1</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Local State on Data Plane

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2</td>
<td>5</td>
</tr>
<tr>
<td>H1</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>match</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>value = 100</td>
<td>drop</td>
</tr>
</tbody>
</table>

value = 100
Local State on Data Plane

• Programmatic control over local state
  – P4, POF, OpenState, Open vSwitch

• Plus other important features
  – Programmable packet parsing
  – Simple arithmetic and boolean operations
  – Traffic statistics (delays, queue lengths, etc.)

• Simple stateful network functions can be offloaded to the data plane!
Hop-by-Hop Utilization-aware Load-balancing Architecture

Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford

HULA Multipath Load Balancing

- Load balancing *entirely* in the data plane
  - Collect real-time, path-level performance statistics
  - Group packets into “flowlets” based on time & headers
  - Direct each new flowlet over the current best path
Path Performance Statistics

Best-hop table

<table>
<thead>
<tr>
<th>Dest</th>
<th>ToR</th>
<th>Best Next-Hop</th>
<th>Path Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>S3</td>
<td>50%</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>S4</td>
<td>10%</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Using the best-hop table
  - *Update* the best next-hop upon new probes
  - *Assign* a new flowlet to the best next-hop
Flowlet Routing

Flowlet table

<table>
<thead>
<tr>
<th>h(flowid)</th>
<th>Dest ToR</th>
<th>Timestamp</th>
<th>Next-Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ToR 10</td>
<td>1</td>
<td>S2</td>
</tr>
<tr>
<td>1</td>
<td>ToR 0</td>
<td>17</td>
<td>S4</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

- Using the flowlet table
  - *Update* the next hop if enough time has elapsed
  - *Update* the timestamp to the current time
- *Forward* the packet to the chosen next hop
Putting it all Together

<table>
<thead>
<tr>
<th>Best Next-Hop</th>
<th>Path Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3</td>
<td>50%</td>
</tr>
<tr>
<td>S4</td>
<td>10%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dest ToR</th>
<th>Timestamp</th>
<th>Next-Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToR 10</td>
<td>1</td>
<td>S2</td>
</tr>
<tr>
<td>ToR 0</td>
<td>17</td>
<td>S4</td>
</tr>
</tbody>
</table>

current best next-hop S3

Update next-hop (if enough time elapsed) and time

chosen next-hop

data packet
Plenty of Other Applications

- Stateful firewall
- DNS tunnel detection
- SYN flood detection
- Elephant flow detection
- DNS amplification attack detection
- Sidejack detection
- Heavy-hitter detection
- ...
But, how to best *write* these stateful apps?
SNAP: Stateful Network-Wide Abstractions for Packet Processing

Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker

Writing Stateful Network Apps is Hard

• Low-level switch interface
  – Multiple stages of match-action processing
  – Registers/arrays for maintaining state

• Multiple switches
  – Placing the state
  – Routing traffic through the state

• Multiple applications
  – Combining forwarding, monitoring, etc.
Snap Language

- Hardware independent
- One Big Stateful Switch (OBSS)
- Composition
Stateless Packet Processing

- A function that specifies
  - How to process each packet on a one-big-switch
  - Based on its **fields**
- E.g., NetKat
Stateful Packet Processing

• A function that specifies
  – How to process each packet on a one-big-switch
  – Based on its **fields** and the **program state**
  – Where state is an **array** indexed by header fields
Example Snap App: DNS Reflection

```python
if srcip in CSNET & dstport = 53 then
    seen[srcip][dns.id] ← True
else if dstip in CSNET & srcport = 53 then
    if ~seen[dstip][dns.id] then
        unmatched[dstip]++;
        if unmatched[dstip] = threshold then
            susp[dstip] ← True
    else id
else id
```

- **Seen**: Keep track of DNS requests by client and DNS identifier
- **Unmatched**: Count DNS responses that don’t match prior requests
- **Susp**: Suspected victims receive many unmatched responses
Example Snap App: Stateless Forwarding

```java
if dstip = CSNET then outport ← CS
else if dstip = EENET then outport ← EE
else if dstip = ISP1NET then outport ← ISP1
else if dstip = ISP2NET then outport ← ISP2
else drop
```
if srcip in CSNET & dstport = 53 then
    seen[srcip][dns.id] ← True
else if dstip in CSNET & srcport = 53 then
    if ~seen[dstip][dns.id] then
        unmatched[dstip]++;
    if unmatched[dstip] = threshold then
        susp[dstip] ← True
    else id
else id

if dstip = CSNET then outport ← CS
else if dstip = EENET then outport ← EE
else if dstip = ISP1NET then outport ← ISP1
else if dstip = ISP2NET then outport ← ISP2
else drop
# Snap Applications

<table>
<thead>
<tr>
<th>Source</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chimera (USENIX Security’12)</td>
<td>Number of domains sharing the same IP address</td>
</tr>
<tr>
<td></td>
<td>Number of distinct IP addresses under the same domain</td>
</tr>
<tr>
<td></td>
<td>DNS TTL change tracking</td>
</tr>
<tr>
<td></td>
<td>DNS tunnel detection</td>
</tr>
<tr>
<td></td>
<td>Sidejack detection</td>
</tr>
<tr>
<td></td>
<td>Phishing/spam detection</td>
</tr>
<tr>
<td>FAST (HotSDN’14)</td>
<td>Stateful firewall</td>
</tr>
<tr>
<td></td>
<td>FTP monitoring</td>
</tr>
<tr>
<td></td>
<td>Heavy-hitter detection</td>
</tr>
<tr>
<td></td>
<td>Super-spreader detection</td>
</tr>
<tr>
<td></td>
<td>Sampling based on flow size</td>
</tr>
<tr>
<td></td>
<td>Selective packet dropping (MPEG frames)</td>
</tr>
<tr>
<td></td>
<td>Connection affinity</td>
</tr>
<tr>
<td>Bohatei (USENIX Security’15)</td>
<td>SYN flood detection</td>
</tr>
<tr>
<td></td>
<td>DNS reflection (and amplification) detection</td>
</tr>
<tr>
<td></td>
<td>UDP flood mitigation</td>
</tr>
<tr>
<td></td>
<td>Elephant flows detection</td>
</tr>
<tr>
<td>Others</td>
<td>Bump-on-the-wire TCP state machine</td>
</tr>
<tr>
<td></td>
<td>Snort flowbits</td>
</tr>
</tbody>
</table>
Snap Compiler

Composition of multiple apps

State placement and routing
Snap Compiler

- Identify State Dependencies
- Translate to Intermediate Representation (xFDD)
- Identify mapping from packets to state variables
- Optimally distribute the xFDD
- Generate rules per switch
Intermediate Representation: xFDDs

• Canonical representation of a program
• Composable
• Easily partitioned
• Simplify program analysis
Extended Forwarding Decision Diagrams (xFDDs)

- **Intermediate node**: test on header fields and state
- **Leaf**: set of action sequences
- **Three kinds of tests**
  - field = value
  - field$_1$ = field$_2$
  - state$_{var}$[e$_1$] = e$_2$
xFDD for DNS Reflection Detection

![Diagram of DNS Reflection Detection](image-url)
Optimally Distribute the xFDD

- Dependency Graph
- MILP
- Packet-State Mapping
- Traffic Matrix

Output
- State placement
- Routing
See SIGCOMM’16 paper for prototype, experiments, etc.

More Fun With State

• Extending Snap
  – More operations, e.g., field ← state[index]
  – Sharding and replication of state
  – Faster compilation

• Richer computational model
  – Limits on computation per packet
  – Different memory (array, hash table, key-value store)
  – Hash collisions, delays in adding new keys, etc.

• More stateful applications!
Conclusion

• Emerging switches have stateful data planes
  – Can run simple network functions
  – ... within and across switches!
• Standard interfaces
  – E.g., P4 (p4.org)
• Raises many new algorithmic challenges
  – New computational model
  – Compact data structures (e.g., sketches)
  – Working within hardware limitations