Recent Advances in Network Functions

Aditya Akella

*UW-Madison*

HotMiddlebox 2016
Network functions

Routing and switching

Functionality added to routers
- Filtering
- Load balancing
- Rate limiting

Implemented as custom packet processing boxes

“Middleboxes”
- Firewalls, proxies,
- intrusion detection systems,
- scrubbing, load balancing,
- Security, WAN optimization
Network functions virtualization
– VMs
– Containers
– Micro-services (lambas?)

Platforms for implementing, deploying NFs
– CoMB
– NetBricks

Point impl. of specific functions
– SDN-based NATs/filtering
– Software load balancers
This talk

• Some fundamental issues, and systems we built
  – State management
  – Software implementation
  – Composition

• Some interesting open problems
State Management
Dynamic reallocation in distributed processing

Load balancing  Elastic scaling  High availability
Network migration  Remote invocation  Always updated NFs
Stateful operation

Dynamically updated per packet

NF’s action for packet depends on state
Multiple instances of an NF should collectively produce the same output as a single instance.

Difficult to achieve:
- **Output** depends on state
- Desire for ↑ **performance** and ↓ **resource usage**
Packet loss

SLO: < 1%

Perform resource usage output equiv.

Reroute new flows

Reroute existing flows

Wait for flows to die

Packet loss icon

SLA: <1%

SLO: < 1%

Diagram with arrows and icons
The state management problem

Performance + resource use + output equiv.

Quickly move or copy NF state alongside updates to network forwarding state

Safety guarantees on updates (none lost; no reordering)
OpenNF

Control Application

move(http, \textit{NF}_1, \textit{NF}_2)

g\textit{et}(http)

\textit{NF}_1

\textit{NF}_2

state chunks

put(state)

\textit{forward}(http, \textit{NF}_2)
Events for loss-free move

1. enableEvents(red) on Bro₁
2. get/delete on Bro₁
3. Buffer events at controller
4. put on Bro₂
5. Flush packets in events to Bro₂
6. Update forwarding

Loss free move

Order-preserving move

Eventual, strict, strong consistency for state sharing
Safety guarantees

PRADS asset detector processing 5K pkts/sec
Move per-flow state for 500 flows

Operations are efficient, but guarantees come at a cost!
Open issues: state

• Better approaches to move/copy state?
  – E.g., FTMB’s copy operation is faster, offers stronger consistency; but no support for move

• Other safety guarantees
  – Timing of packets, cross-session ordering?

• What if state is externalized?
  – Copy may be simplified
  – Move is still nuanced
    • Coordinating processing reallocation
    • Local caching of state may complicate matters
Automating Modification Using StateAlyzr

A system that relies on data and control-flow analysis to automatically identify state objects that need explicit handling.

- **Soundness** means that the system must not miss any critical types, storage locations, allocations, or uses of state.

- **Precision** means that the system identifies the minimal set of state that requires special handling.

Leverage middlebox code structure to improve precision without compromising soundness.
StateAlyzr offers useful *improvements* in *precision*

Theoretically *proved* the *soundness* of our algorithms
Software Implementation
Host-based functions
Goal: CPU, memory efficiency; speed
- Careful allocation of cores
- Cache allocation
- VM-to-VM copy
...

Network-wide functions
Also: meet global goals
- Relatively unexplored
- E.g., network load balancing
Network congestion: flows of all types suffer

Example

- Elephant throughput is cut by half
- TCP RTT is increased by 100X per hop (Rasley, SIGCOMM’14)
# Traffic Load Balancing Schemes

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Hardware changes</th>
<th>Transport changes</th>
<th>Granularity</th>
<th>Pro-/reactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECMP</td>
<td>No</td>
<td>No</td>
<td>Coarse-grained</td>
<td>Proactive</td>
</tr>
<tr>
<td>Centralized</td>
<td>No</td>
<td>No</td>
<td>Coarse-grained</td>
<td>Reactive (control loop)</td>
</tr>
<tr>
<td>CONGA/ Juniper VCF</td>
<td>Yes</td>
<td>No</td>
<td>Fine-grained</td>
<td>Proactive</td>
</tr>
<tr>
<td>MPTCP</td>
<td>No</td>
<td>Yes</td>
<td>Fine-grained</td>
<td>Reactive</td>
</tr>
<tr>
<td>Presto</td>
<td>No</td>
<td>No</td>
<td>Fine-grained</td>
<td>Proactive</td>
</tr>
</tbody>
</table>

Can we do this purely in software?
Presto at a High Level

- Spine
  - vSwitch
  - NIC
  - TCP/IP

- Leaf
  - vSwitch
  - NIC
  - TCP/IP

**Near uniform-sized data units**

- Proactively distributed evenly over symmetric network by vSwitch sender
- Receiver masks packet reordering due to multipathing below transport layer
Presto LB Granularity

- Presto: load-balance on *flowcells*
- What is flowcell?
  - *A set of TCP segments with bounded byte count*
  - Bound is maximal TCP Segmentation Offload (TSO) size
    - Maximize the benefit of TSO for high speed
    - *64KB* in implementation

- What’s TSO?

![Diagram showing TCP/IP, NIC, Segmentation & Checksum Offload, and MTU-sized Ethernet Frames]
Presto LB Granularity

• Presto: load-balance on *flowcells*

• What is flowcell?
  
  – *A set of TCP segments with bounded byte count*
  
  – Bound is maximal TCP Segmentation Offload (TSO) size
    
    • Maximize the benefit of TSO for high speed
    
    • *64KB* in implementation

• Examples

Start

25KB

30KB

30KB

TCP segments

Flowcell: 55KB
Presto LB Granularity

• Presto: load-balance on *flowcells*

• What is flowcell?
  – *A set of TCP segments with bounded byte count*
  – Bound is maximal TCP Segmentation Offload (TSO) size
    • Maximize the benefit of TSO for high speed
    • *64KB* in implementation

• Examples

```
Flowcell: 7KB (the whole flow is 1 flowcell)
```
Host A

**TCP/IP**

50KB

**id, label**

Host B

**TCP/IP**

**vSwitch**

**NIC**

Spine

Leaf

Presto Sender

**vSwitch** receives TCP segment #1

**flowcell #1:** vSwitch encodes *flowcell ID*, rewrites label

**NIC** uses TSO and chunks segment #1 into MTU-sized packets
Benefits

• Many flows smaller than 64KB [Benson, IMC’11]
  – the majority of mice are not exposed to reordering

• Most bytes from elephants [Alizadeh, SIGCOMM’10]
  – traffic routed on uniform sizes

• Fine-grained and deterministic scheduling over disjoint paths
  – near optimal load balancing
• Packet reordering for large flows due to multipath
  – Interferes with GRO – rendering GRO useless
  – Flowcell-aware GRO

• Distinguish loss from reordering
  – Smart heuristic based on flowcell routing properties
Evaluation

Presto’s throughput is within $1 - 4\%$ of Optimal, even when the network utilization is near $100\%$; In non-shuffle workloads, Presto improves upon ECMP by 38-72% and improves upon MPTCP by 17-28%.

Optimal: all the hosts are attached to one single non-blocking switch.
Open issues: implementation

• Leveraging server features and network to accelerate network functions
  – Hashing, checksumming, multiple queues, RDMA

• Data plane programmability, e.g., PIFO, DOMINO, P4
  – Selectively offload to switches for performance or visibility

• Programmatic control over such network functions

• Moving other network-wide functions to the edges
  – Failure recovery, cross-flow caching/analysis, QoS
Chain Composition
Compose chains to realize joint intent?
Do NFs compose? NF ordering?
Deconstruct P1 and P2 into ACLs, chaining rules

Understand flow space relations

Understand joint intent of P1 and P2

P1 supersedes P2

Insert a drop rule

Understand NF operation to order NFs

if_(match(srcip=Mktg, tcp, dstport=8080, dstip=CRM),
   LB>>BC>>route,
   if_(match(dstip=CRM),
      drop,
      if_(match(srcip=Empl, tcp, 
        dstport=80|8080, 
        dstip=Servers), 
        BC>>route, 
        drop)))
Composition challenges

Constructs not rich enough to **encode policy intent**
- In a policy: e.g., an ACL *must* apply
- Across policies: joint intent

Need to know about **NFs actions** to **compose** them

---

**PGA (policy graph abstraction):**
A first attempt aimed at **common** cases

Raises the level of abstraction

Composition support by **construction**
Constructs (1): Policy Graph basics

**EPG** – end-points that satisfy a membership predicate, defined over **labels**

**Edge attributes** – whitelist and service chain → allowed communication

- **Marketing** → **CRM** (dstport = 8080)
- **Employees** → **Servers** (dstport = 80, 8080)

**Label space**

- Employees
  - Marketing
  - Non-Marketing
- CRM
- Non-CRM
- Servers
- Locations
  - Campus
  - Data Center

Contains

Disjoint

**Constructs (2):**

\[(\text{CRM} \lor \text{Non-CRM}) \land \text{Data Center}\]
Constructs (2): Composing NFs

**Constraints**: allowed policy changes when a policy graph is composed with others

No need to specify joint intent, e.g., action precedence

---

**Policy 1**

<table>
<thead>
<tr>
<th>Classifier</th>
<th>NF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>Add</td>
</tr>
<tr>
<td>srcIP=Mktg, dstIP=CRM, dstPt=8080</td>
<td>Y</td>
</tr>
<tr>
<td>dstIP=CRM</td>
<td>N</td>
</tr>
</tbody>
</table>

---

OK to **further constrain** the packet space, but cannot completely disallow

**No other** communication is allowed

Constraints on **NF actions**
Normalization: separate input EPG into an equivalent set of disjoint EPGs by rewriting predicate in **positive disjunctive normal form**

Graph union: merging edges using constraints

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Match</th>
<th>Add</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcIP=Mktg, dstIP=CRM, dstPt=8080</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>dstIP=CRM</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Constructs (3): Grayboxing NFs

Grayboxing: specifies the high level packet processing behavior of NFs

Byte Counter

match: (dstip=Servers.RIPs) >> count_bytes: {group_by: [dstip]}

Load Balancer

match(dstip=CRM.virtIP) >> modify(dstip=CRM.RIPs)

Full specification for some NFs

Input/output packet spaces

Not rich enough: necessary but not sufficient
NF ordering: dependency analysis

**LB**

*Policy:*
match: (dstip=CRM.virtIP) >>
modify: (dstip=CRM.RIPs)

**Byte Counter**

*Policy:*
match: (dstip=Servers.RIPs) >>
count_bytes: {group_by: [dstip]}

Topological sort over a dependency graph
PGA Implementation

- **PGA runtime (Pyretic)**
  - srcIP, dstIP
  - 1st pkt
  - Flow rules

- **Graph composer**
  - Input graphs
  - report
  - Composed graph
  - query
  - policy

- **Policy graph repo.**
  - Build predicates
  - Look up norm. EPGs
  - Look up policy

- **Controller (POX)**
  - OpenFlow
  - End-point join/leave/update
**Evaluation**

**Small synthetic** input:
11 EPGs, 4 NFs, 7 EPG-EPG edges

Composed to: **8 EPGs, 20 edges**

**Large real** input:
137 departments, 4340 subnets, 20K ACLs

Composed to: **3.7K EPGs, 2.1M edges**

---

**Small Graph**

<table>
<thead>
<tr>
<th>Composition time(s)</th>
<th>Graph Union</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0016</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0004</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Large Graph**

<table>
<thead>
<tr>
<th>Composition time(s)</th>
<th>Graph Union</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Time (ms)**

- Small
  - Graph Union: 0.02 ms
  - Normalization: 0.04 ms

- Large
  - Graph Union: 400 ms
  - Normalization: 300 ms
Open issues: composition

• Support for richer policies
  – QoS/performance metrics
  – Stateful or event-driven/dynamic policies

• Accommodating other general NFs

• Limitations of graph-based abstractions
  – Network-wide objectives (TE, load balance)
Network functions: rich space!

• Some initial contributions in state management, software implementation and chain composition

• Plenty of ground still to cover