High-Throughput Publish/Subscribe in the Forwarding Plane

Theo Jepsen, Masoud Moushref, Antonio Carzaniga, Nate Foster, Xiaozhou Li, Milad Sharif, Robert Soulé
Università della Svizzera italiana (USI) and Barefoot Networks
Publish/Subscribe Is Critical to Distributed Applications
Publish/Subscribe Is Critical to Distributed Applications
Publish/Subscribe Is Critical to Distributed Applications
Publish/Subscribe Is Critical to Distributed Applications
Publish/Subscribe Is Critical to Distributed Applications
Publish/Subscribe Is Critical to Distributed Applications
Motivating Example: ITCH Market Feed

1. NASDAQ publishes feed:

- Ether/IP/UDP
- MOLD
- GOOGL
- MSFT
- ORCL

2. End-Points filter based on trading strategy:
   - stock == MSFT
Motivating Example: ITCH Market Feed

1. NASDAQ publishes feed:

- Ether/IP/UDP
- MOLD
- GOOGL
- MSFT
- ORCL

2. End-Points filter based on trading strategy:
   stock == MSFT

High volume, High tail latency
In-software Processing: Multicast + Kernel Bypass

- Unnecessary congestion in the network
- Burden of filtering on hosts leads to queuing
- Highlights need for “in network” solution

Nasdaq 9/30/17, 0.5% GOOGL
Challenges

- Different applications have different message formats
- Filter content based on expressive conditions
- Deep packets and multiple messages per packet
Camus: Dataplane Pub/Sub

Controller

Camus Compiler

P4 Compiler

P4 header spec

P4 parser spec

P4 program

Control plane rules

Filters (subscriptions)
A publisher simply composes and sends packets

Camus generates application-specific parsing logic

Parsing logic is static, installed once with Camus
Filters are boolean formulas of atomic predicates and an action:

\[ \text{stock} == \text{GOOGL} : \text{fwd}(1) \]

A forwarding action may be unicast or multicast:

\[ \text{stock} == \text{GOOGL} : \text{fwd}(1,2,3) \]

Rules may be stateful or compute a function:

\[ \text{stock} == \text{GOOGL} \land \text{avg(price)} > 50 : \text{fwd}(1) \]
header_type itch_add_order_t {
    fields {
        stock_locate: 16;
        /* ... */
        shares: 32;
        stock: 64;
        price: 32;
    }
}

header itch_add_order_t add_order;
@pragma query_field(add_order.shares)
@pragma query_field(add_order.price)
@pragma query_field_exact(add_order.stock)
@pragma query_counter(my_counter, 100, 1024)
header_type itch_add_order_t {
    fields {
        stock_locate: 16;
        /* ... */
        shares: 32;
        stock: 64;
        price: 32;
    }
}

header itch_add_order_t add_order;
@pragma query_field(add_order.shares)
@pragma query_field(add_order.price)
@pragma query_field_exact(add_order.stock)
@pragma query_counter(my_counter, 100, 1024)
Compiling Static Pipeline

```c
header_type itch_add_order_t {
    fields {
        stock_locate: 16;
        /* ... */
        shares: 32;
        stock: 64;
        price: 32;
    }
}
header itch_add_order_t add_order;
@pragma query_field(add_order.shares)
@pragma query_field(add_order.price)
@pragma query_field_exact(add_order.stock)
@pragma query_counter(my_counter, 100, 1024)
```
Compiling Dynamic Filters: Representing Rules with BDDs

\[
\begin{align*}
\text{shares} &= 2 : \text{fwd}(1) \\
\text{price} > 1 \land \text{shares} &= 2 : \text{fwd}(2)
\end{align*}
\]
Compiling Dynamic Filters: Representing Rules with BDDs

shares == 2 : fwd(1)
price > 1 \land shares == 2 : fwd(2)
Compiling Dynamic Filters: BDD Reductions

(i) Remove isomorphic (Standard)

(ii) Remove redundant (Standard)

(iii) Remove implicit (Domain-specific)
Compiling Dynamic Filters: BDDs to Forwarding Table (1/4)

- price > 1
  - shares = 5
    - [ ]
    - [2]
  - [3]
- price = 3
  - shares = 5
    - [1,3]
    - [1,2,3]
Compiling Dynamic Filters: BDDs to Forwarding Table (1/4)

Partition into sub-graphs by field

- **price** fields
  - price > 1
  - price = 3

- **shares** fields
  - shares = 5

- **leaves**
  - []
  - [2]
  - [3]
  - [1,3]
  - [1,2,3]
Compiling Dynamic Filters: BDDs to Forwarding Table (2/4)

price fields

price > 1

Assign an ID

price = 3

shares fields

shares = 5

Identify entry and exit node sets

leaves

[ ]

[2]

[3]

[1,3]

[1,2,3]
Compiling Dynamic Filters: BDDs to Forwarding Table (3/4)

For each path, the tuple (entry ID, match, exit ID) corresponds to an entry in its field’s table

<table>
<thead>
<tr>
<th>State</th>
<th>Match</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>=3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>&gt;1</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Match</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>=5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>=5</td>
<td>8</td>
</tr>
</tbody>
</table>
Compiling Dynamic Filters: BDDs to Forwarding Table (4/4)

Encode BDD as finite state machine in the forwarding tables
Multiple Messages Per Packet

- Parsing deep: recirculate packet and advance index
- Routing multiple messages: prune unwanted messages at egress
Evaluation
Compiler Efficiency

- Used synthetic workload generator to create queries of the form:
  \[ stock = S \land price > P : \text{fwd}(H) \]

- Can fit \( O(100K) \) queries in switch memory!

- Compiling 100K subscriptions required
  21,401 table entries and 198 multicast groups
Experiment: In-Network ITCH Filtering

1. Publisher sends feed (add order)
2. Switch filters for: stock = “GOOGL”
3. Client calculates latency

- Machine has 2 x 25 GB/s NICs
- Forward: switch forwards packets; queries evaluated in software
- Filter: switch evaluates queries
## Workloads

<table>
<thead>
<tr>
<th>Workload</th>
<th>Messages per packet</th>
<th>% GOOGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic</td>
<td>1-12 (Zipf dist.)</td>
<td>1%</td>
</tr>
<tr>
<td>Synthetic (worst case)</td>
<td>Exactly 12</td>
<td>100%</td>
</tr>
<tr>
<td>Nasdaq sample 08302017</td>
<td>Exactly 1</td>
<td>0.1%</td>
</tr>
</tbody>
</table>
Synthetic Workload

CDF of Latency

1% GOOGL, 1-12 messages / packet

With Camus, 99% finish under 20us
Without Camus, 99% finish under 500us
Worst-Case Workload

CDF of Latency

100% GOOGL, 12 messages / packet

- Baseline
- Switch Filtering
NASDAQ Workload (8/30/17)

CDF of Latency

0.1% GOOGL, 1 messages / packet

With Camus, 100% finish under 100us
Without Camus, 84% finish under 100us
Conclusion

- Camus is a pub/sub service implemented on programmable network ASICs
- Uses a novel BDD-based algorithm to translate predicates into P4 tables that can support $O(100K)$ expressions
- Increases system flexibility and reduces latency for clients

http://inf.usi.ch/phd/jepsen/