Programming
The Network Data Plane

Changhoon Kim
This is roughly how I process packets …

- Prone to bugs
- Very long and unpredictable lead time

in English

Switch OS

Run-time API

Driver

Fixed-function ASIC
Tons of beautiful ideas to enhance your network …

- For better and easier management
  - Allocate h/w resources adaptively
  - Ensure visibility for debugging and diagnostics
  - Verify network behavior
  - Improve OAM capabilities
Tons of beautiful ideas to enhance your network …

- To enable critical new features
  - Introduce custom encapsulations and/or headers
  - Better network-fabric load-balancing
  - Enhanced congestion control
  - Improve robustness
  - Embed some middlebox functions inline
  - any many more …
Extremely limited way of turning ideas into reality

- No DIY – must work with vendors at “feature” level
- Extremely complicated and involved process to build consensus and pressure for “feature”
- Painfully long and unpredictable lead time
- To use new features, you must get new switches
- What you finally get != what you asked for
Programmable network devices come to rescue

- Some devices are or will be more programmable than fixed-function ASICs
- CPUs: 10s of Gb/s
- FPGAs, NPUs: 100s of Gb/s
- Protocol-Independent Switch Architecture (PISA) chips: a few Tb/s
  - RMT [SIGCOMM’13] and a few emerging solutions, including one from Barefoot
  - Merchant silicon with fully programmable parser and generic match-action logic
  - In next few years this kind of silicon will dominate
Turning the tables

“<i>This is precisely how you must process packets</i>”

in P4

Switch OS

Run-time API

Driver

PISA device

(Protocol-Independent Switch Architecture)
What does this mean?

- **To network device vendors**
  - S/W programming practices and tools used in every phase
  - Extremely fast iteration and feature release
  - Differentiation in capabilities and performance
  - Can fix even data-plane bugs in the field

- **To large on-line service providers and carriers**
  - No more “black boxes” in the “white boxes”
  - Your devs can program, test, and debug your network devices all the way down
  - You keep your own ideas

You wear both hats 😊
The rest of the talk:

- How PISA works
- Why we call it protocol-independent forwarding
- What kind of cool things you can do with PISA and P4
- What kinds of research problems arise
PISA: An architecture for high-speed programmable packet forwarding
PISA: Protocol Independent Switch Architecture
PISA: Protocol Independent Switch Architecture

Programmable Parser → Ingress → Buffer → Egress
PISA: Protocol Independent Switch Architecture

Mix of SRAM and TCAM for: lookup tables, counters, meters, Bloom filters

ALUs for: Standard boolean and arithmetic operations & add/delete fields, hashes

Programmable Packet Generator

Programmable Parser

Recirculation
Why we call it protocol-independent packet processing
Device does not understand any protocols until it gets programmed

Logical Data-plane View
(your P4 program)

Switch Pipeline

Queues

CLK
Mapping logical data-plane design to physical resources

Logical Data-plane View
(your P4 program)

Switch Pipeline

Programmable Parser

L2 Table
L2 Action Macro

IPv4 Table
IPv4 Action Macro

IPv6 Table
IPv6 Action Macro

ACL Table
ACL Action Macro

Queues

CLK
Re-program in the field

Logical Data-plane View (your P4 program)

Switch Pipeline
What exactly does the compiler do?

- Programmable Parser
- Cross Bar
- Hash Gen
- Match Table (SRAM or TCAM)
- Action & Instr Mem
- PHV (Pkt Hdr Vector)
- ALUs
- Queues
- CLK
What does a P4 program look like?

```
header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
    }

    parser parse_ethernet {
        extract(ethernet);
    }

    return select(latest.etherType) {
        0x8100 : parse_vlan;
        0x800  : parse_ipv4;
        0x86DD : parse_ipv6;
    }
}

header_type my_encap_t {
    fields {
        foo : 12;
        bar : 8;
        baz : 4;
        qux : 4;
        next_protocol : 4;
    }
}
```
What does a P4 program look like?

```
control ingress {
    apply(l2);
    apply(my_encap);
    if (valid(ipv4) {
        apply(ipv4_lpm);
    } else {
        apply(ipv6_lpm);
    }
    apply(acl);
}
action set_next_hop(nhop_ipv4_addr, port) {
    modify_field(metadata.nhop_ipv4_addr, nhop_ipv4_addr);
    modify_field(standard_metadata.egress_port, port);
    add_to_field(ipv4.ttl, -1);
}
```

**Table ipv4_lpm**
```
{ }
reads {
    ipv4.dstAddr;
}
actions {
    set_next_hop;
}
```
What does a P4 program look like?

```c
/* Example: A typical IPv4 routing table */
table ipv4_lpm {
    reads {
        ingress_metadata.vrf : exact;
        ipv4.dstAddr : lpm;
    }
    actions {
        nop;
        l3_l2_switch;
        l3_multicast;
        l3_nexthop;
        l3_ecmp;
        l3_drop;
    }
    size : 65536;
}
```

These are the only possible actions. Each particular entry in the table is associated with ONE of these:

<table>
<thead>
<tr>
<th>vrf</th>
<th>ipv4.dstAddr / prefix</th>
<th>action</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>192.168.1.0 / 24</td>
<td>l3_l2_switch</td>
<td>port_id=64</td>
</tr>
<tr>
<td>10</td>
<td>10.0.16.0 / 22</td>
<td>l3_ecmp</td>
<td>ecmp_index=12</td>
</tr>
<tr>
<td>1</td>
<td>192.168.0.0 / 16</td>
<td>l3_nexthop</td>
<td>nexthop_index=451</td>
</tr>
<tr>
<td>1</td>
<td>0.0.0.0 / 0</td>
<td>l3_nexthop</td>
<td>nexthop_index=1</td>
</tr>
</tbody>
</table>
What kind of cool things can you do by programming data planes?
Let’s start with a demo!
Demo setup

/* Reference p4 program
switch.p4 */

#include "includes/headers.p4"
#include "includes/parser.p4"

/********************************/
/* Device Personality Selection */
/********************************/

// #define P_MINIMAL
#define P_BASE
//#define P_MSDC_SPINE
//#define P_MSDC_TOR
//#define P_MSDC_BL
//#define P_INTERNET_ROUTER ...

// Flexible Tables
#define LPM_TABLE_SIZE 16384
#define IPV6_LPM_TABLE_SIZE 4096
#define HOST_TABLE_SIZE 131072
#define IPV6_HOST_TABLE_SIZE 32768 ...

header_type routing_metadata_t {
  fields {
    bd : BD_BIT_WIDTH; /* bridge domain */
    vrf : VRF_BIT_WIDTH; /* routing domain */
    v6_vrf : VRF_BIT_WIDTH; /* routing domain */
    ...
  }
  table bridge_domain {
    reads {
      routing_metadata.bd : exact;
    }
    actions {
      nop; // Not used
      bd_set;
    ...
  }...
Scenario: Debugging long latency tail

HTTP download (from H1 to H3)

H1  5Mbps
SW1 (dc.p4)  5Mbps
H2  20Mbps
SW2 (dc.p4)  20Mbps
SW3 (dc.p4)  5Mbps
Monitor  5Mbps
H3  5Mbps
Scenario: Debugging Long Latency Tail

**SW1 (dc.p4)**
- 5Mbps
- Payload
- TCP Hdr
- L2 & L3 Hdr

**SW2 (dc.p4)**
- 20Mbps
- Payload
- TCP Hdr
- L2 & L3 Hdr

**SW3 (dc.p4)**
- 5Mbps
- SW1 ID & Latency
- Payload
- TCP Hdr
- L2 & L3 Hdr

**H1**
- 5Mbps

**H2**
- 20Mbps

**H3**

**Monitor**
- 5Mbps
- SW3 ID & Latency
- Payload
- TCP Hdr
- L2 & L3 Hdr
In-band Network Telemetry in P4

```p4
table int_table {
  reads {
    ip.protocol;
  }
  actions {
    export_queue_latency;
  }
}
```

```p4
action export_queue_latency (sw_id) {
  add_header(int_header);
  modify_field(int_header.kind, TCP_OPTION_INT);
  modify_field(int_header.len, TCP_OPTION_INT_LEN);
  modify_field(int_header.sw_id, sw_id);
  modify_field(int_header.q_latency, intrinsic_metadata.deq_timedelta);
  add_to_field(tcp.dataOffset, 2);
  add_to_field(ipv4.totalLen, 8);
  subtract_from_field(ingress_metadata.tcpLength, 12);
}
```

Add TCP Options & copy switch ID and queue latency Into the options
INT open-source code and spec


---

Improving Network Monitoring and Management with Programmable Data Planes

By Mukesh Hira & LJ Webber

Quicklinks: The INT specification - INT GitHub repository - INT demo video

Compute virtualization and the widespread deployment of virtual machines have led to an extension of the network into the hypervisor – a domain where multiple isolated software instances need to communicate. To address this requirement, we propose a network services - logical switches, virtual router interfaces, virtual router interfaces, virtual switch interfaces, virtual switch interfaces, virtual switch interfaces. In this regard, P4 is a logical network so as to enable deployment of virtual services over any physical network infrastructure, the only requirement from the physical network being IP connectivity between the hypervisors.

While the decoupling of physical and virtual topologies has advantages, it is important to have some interaction between the physical and virtual switches to allow for end-to-end monitoring of the entire physical + virtual network from a "single pane of glass" and to help in troubleshooting and fault isolation in complex physical + virtual topologies.

We propose methods for various network elements to collect and report their state in real-time, allowing for improved cooperation between the virtual and physical layers without requiring intermediate layers such as CPU driven control planes. The general term we have applied to these methods is INT: Inband Network Telemetry.

Some examples of useful network state to be reported by network elements are:

1. <Switch-ID, Input port ID, output port ID> -- This allows for determination and monitoring of the different paths between a pair of end-points. Current mechanisms for determining multiple paths between a pair of end-points are based on IP traceroute and can only discover equal-cost layer-3 paths (ECMP routes), but cannot discover or report the existence of
And a lot more exciting apps …
We’ve just started to scratch the surface
Always-on path & latency tracking

- **How does it work?**
  - Collect physical path and hop latency of **every** packet via INT
  - Last hop creates one record upon each new connection or path/latency change of existing connections

- **Why is this useful?**
  - Confirm the effect of routing config or policy changes real time
  - Verify and audit network behavior thoroughly
  - Quantify RIB-FIB inconsistency
  - Identify connections affected by maintenance, failure, or recovery events
  - Detect hop-latency increases and identify the victims
Add L4 load balancing to every switch

- **How does it work?**
  - Ensure **per-connection consistency**: Must forward every packet in a connection to the same DIP
  - Each switch should maintain a large amount of connection state

- **Why is this useful?**
  - Cost saving; employ zero or fewer servers to offer L4 load-balancing service
  - Predictably high throughput and low latency for LB traffic even under DoS
  - Service differentiation and protection via h/w-based throttling

- P4 prototype available and demo’ed at the P4 workshop
Custom traffic monitoring and filtering

- General-purpose stateful memory & Custom hashing
  → Explosion of probabilistic traffic monitoring and filtering schemes

- Bloom-filter-based whitelist
  - Can remember $O(10^{7-8})$ items with very low false positives

- Heavy-hitter detection via Count-min sketch
  - Can track the frequency of $O(10^6 \sim 7)$ items

- A better NetFlow (FlowRadar [NSDI’16])
  - Switches encode flow-sets using Invertible Bloom Filter and export the encodings frequently to monitoring servers -- once every few msec
  - Monitors decode the encodings network wide and produce NetFlow-like records
Dynamic source routing

- Forward packets/flowlets/flows based on the current path conditions
  - Path condition: Link utilization, queue depth, hop latency, end-to-end latency, etc.
- Hop-by-hop decision making is also feasible and arguably more appealing
  - Prototype and results (namely “HULA”) to be published at SOSR’16

1. Keep track of path status
2. Detect flowlets and manage their state
3. Assign best paths to flowets
4. Export hop-level link utilization
5. Generate path-utilization feedback

Sender
Host/ToR

Intermediate
SW

Intermediate
SW

Receiver
Host/ToR

pkt

path1 = 50%

path2 = 30%

path3 = 60%

path4 = 40%

best = path2

feedback

path1

path2

path3

path4

Export hop-level link utilization

1. Keep track of path status
2. Detect flowlets and manage their state
3. Assign best paths to flowets

Receive
Host/ToR

Sender
Host/ToR

Intermediate
SW

Intermediate
SW

4. Export hop-level link utilization
Various modes of congestion control

- **Explicit congestion-control protocols running in switches**
  - RCP, XCP,TeXCP, etc.

- **Hybrid congestion control – or Timely++**
  - Each switch exposes its id and queuing latency
  - End hosts keep deriving the best rate for each connection

- **Host-to-dst-ToR admission control (network-level VoQ)**
  - Last-hop ToR enforces the “hose-model” traffic via admission control
  - High throughput, low latency, and (nearly) lossless without pausing
  - Can enrich the design if hosts expose more info to network, such as traffic type, message size, deadline, etc.
Research problems

- How should P4 evolve?
  - Ensure composability, portability, and language-architecture separation
  - Could we further improve the abstractions offered by P4?

- How do we test equivalence of two P4 programs?
- Could we automatically derive test cases for a P4 program?
- How can we prove correctness of a P4 program?
- What can we verify using a P4 program?
- Can we auto-generate P4 programs from higher-level policies?
Research problems – cont’d

- **What novel use cases can we enable?**
  - Advanced network monitoring, analysis, and diagnostics
  - Assisting distributed applications via in-network processing
    - E.g., MoM [NSDI’15], NetPaxos [SOSR’15]
  - Jointly designing the network dataplane and the apps running on it
    - E.g., SwitchKV [NSDI’16]

- **Can we build programmable schedulers?**
  - PIFO [HotNets’15] and UPS [NSDI’16]
  - What’s the right language-level abstraction for these?

- **And many more …**
P4 development environment

- Open-source P4 development tools
  - P4 compiler, reference P4 programs, P4-programmable S/W switch, test framework, etc.
  - Apache 2.0 license
  - Available at http://github.com/p4lang

- There will be more programmable forwarding targets
  - Both hardware and software devices (OVS, eBPF, etc.)
  - Switches, NICs, middle-boxes, etc.

This trend is going to be irreversible.
Recapping: Why is data-plane programmability a big deal?
Key benefits of programmable forwarding

1. **New features**: Add new protocols
2. **Reduce complexity**: Remove unused protocols
3. **Efficient use of resources**: Flexible use of tables
4. **Greater visibility**: New diagnostics, telemetry, OAM etc.
5. **Modularity**: Compose forwarding behavior from libraries
6. **Portability**: Specify forwarding behavior once; compile to many devices
7. **Own your own IP**: No need to tell the chip vendor your features
Closing remark

*Networks will become a programmable target, repeating the same evolution pattern that already took place in computing and storage industries.*

Join the 3rd P4 Workshop, May/23 – 24 at Stanford
(more info at http://p4.org)