Networking is on the verge of a paradigm shift towards deep programmability.

Why? It's really a story in 3 stages:

Stage 1: The network management crisis

"Human factors are responsible for 50% to 80% of network outages."

Ironically, this means that data networks work better during week-ends...

![Route leaks by day](chart.png)

Source: Job Snijders (NTT)
What is SDN and how does it help?

• SDN is a new approach to networking
  – Not about “architecture”: IP, TCP, etc.
  – But about design of network control (routing, TE,...)

• SDN is predicated around two simple concepts
  – Separates the control-plane from the data-plane
  – Provides open API to directly access the data-plane

• While SDN doesn’t do much, it enables a lot

OpenFlow is an API to a switch flow table

• Simple packet-handling rules
  – Pattern: match packet header bits, i.e. flowspace
  – Actions: drop, forward, modify, send to controller
  – Priority: disambiguate overlapping patterns
  – Counters: #bytes and #packets

```
10. src=1.2.*.*, dest=3.4.5.* → drop
05. src = *.*.*.*, dest=3.4.*.* → forward(2)
01. src=10.1.2.3, dest=*.*.*.* → send to controller
```

OpenFlow is not all roses

The protocol is too complex
(12 fields in OF 1.0 to 41 in 1.5)
switches must support complicated parsers and pipelines

The specification itself keeps getting more complex
extra features make the software agent more complicated

Consequences
Switches vendor end up implementing parts of the spec.
which breaks the abstraction of one API to rule-them-all
Enters… Protocol Independent Switch Architecture and P4

P4 is a high-level language for programming protocol-independent packet processors

P4 is protocol-independent
the programmer defines packet headers & processing logic

P4 is target-independent
data plane semantic and behavior can be adapted

P4 is protocol-independent
P4 specifies packet forwarding behaviors
enables to redefine packet parsing and processing

We will start diving into the P4 ecosystem and look at our first practical usage

Quick historical recap

This week on
Advanced Topics in Communication Networks

IP forwarding in P4?

LAN 1
LAN 2

1.2.3.4 1.2.3.5 1.2.3.254
1.2.3.0/24
5.6.7.0/24

5.6.7.1 5.6.7.2
5.6.7.200

1.2.3.0/24
5.6.7.0/24

forwarding table

Quick historical recap

P4 environment
P4 language
P4 in practice

2014 2015 2016 2017 2018
July
Initial paper
P4 14 specification
September
P4 14 v1.0.1
v1.0.2
v1.0.3
v1.0.4

P4 16 specification (draft)
December
P4 16 specification
May

Next week: Stateful data plane programming
Probabilistic data structures (beginning)
P4 introduces the concept of an architecture

P4 Target
a model of a specific hardware implementation

P4 Architecture
an API to program a target

Programming a P4 target involves a few key elements

P4 Program
Compiler
Control Plane
Data Plane
Tables
Externs
CPU port
User supplied
Vendor supplied
Code
Target
Architecture Model

We'll rely on a simple P4 switch architecture (v1model)
which is roughly equivalent to “PISA”

Each architecture defines the metadata it supports, including both standard and intrinsic ones

Each architecture also defines a list of “externs”, i.e. blackbox functions whose interface is known

Most targets contain specialized components
which cannot be expressed in P4 (e.g. complex computations)

At the same time, P4 should be target-independent
In P4, almost 1/3 of the constructs were target-dependent

Think of externs as Java interfaces
only the signature is known, not the implementation

vmodel

struct standard_metadata_t {
    bit<9> ingress_port;
    bit<9> egress_spec;
    bit<9> egress_port;
    bit<32> clone_spec;
    bit<32> instance_type;
    bit<1> drop;
    bit<16> recirculate_port;
    bit<32> packet_length;
    bit<32> enq_timestamp;
    bit<19> enq_qdepth;
    bit<32> deq_timedelta;
    bit<19> deq_qdepth;
    error parser_error;
    bit<48> ingress_global_timestamp;
    bit<48> egress_global_timestamp;
    bit<32> lf_field_list;
    bit<16> mcast_grp;
    bit<32> resubmit_flag;
    bit<16> egress_rid;
    bit<1> checksum_error;
    bit<32> recirculate_flag;
};

+ many others (see below)

vmodel

extern register<T> {
    register(bit<32> size);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
};

vmodel

extern void random<T>(out T result, in T lo, in T hi);

extern void hash<O, T, D, M>(out O result, in HashAlgorithm algo, in T base, in D data, in M max);

extern void update_checksum<T, O>(in bool condition, in T data, inout O checksum, HashAlgorithm algo);

+ many others (see below)
But first, the basics: *data types, operations, and statements*

P4 is a statically-typed language with *base types* and operators to derive composed ones.

- `bool` - Boolean value
- `bit<W>` - Bit-string of width W
- `int<W>` - Signed integer of width W
- `varbit<W>` - Bit-string of dynamic length ≤ W
- `match_kind` - describes ways to match table keys
- `error` - used to signal errors
- `void` - no values, used in few restricted circumstances
- `_F` - not supported
- `_X` - not supported
P4 is a statically-typed language with base types and operators to derive composed ones

Think of a header as a struct in C containing the different fields plus a hidden 'validity' field

Parsing a packet using extract() fills in the fields of the header from a network packet
A successful extract() sets to true the validity bit of the extracted header

P4 operations are similar to C operations and vary depending on the types (unsigned/signed ints, …)

Constants, variable declarations and instantiations are pretty much the same as in C too

Variables have local scope and their values is not maintained across subsequent invocations

Important variables cannot be used to maintain state between different network packets

Instead to maintain state you can only use two stateful constructs

Tables modified by control plane
Extern objects modified by control plane & data plane

More on this next week
P4 statements are pretty classical too
Restrictions apply depending on the statement location

```
return  terminates the execution of the
         action or control containing it

exit   terminates the execution of all
         the blocks currently executing
```

Conditions
```
if (x==123) {…} else {…}
```
terminates the execution of the
action or control containing it

```
switch (t.apply().action_run) {
  action1: { ...}
  action2: { ...}
}
```
Switch
Terminates the execution of all
the blocks currently executing

Parser
Match-Action Pipeline
Deparser

The parser uses a state machine to map
packets into headers and metadata

```
Packet
  a:b:c:d   →  1:2:3:4
  1.2.3.4   →  5.6.7.8
  1234      →  56789
```
Parser
Match-Action Pipeline
Deparser

The last statement in a state is an (optional) transition,
which transfers control to another state (inc. accept/reject)

```
state start {
  transition parse_ethernet;
}
state parse_ethernet {
  packet.extract(hdr.ethernet);
  transition select(hdr.ethernet.etherType) {
    0x800: parse_ipv4;
    default: accept;
  }
}
state parse_ipv4 {
  packet.extract(hdr.ipv4);
  transition select(hdr.ipv4.protocol) {
    6: parse_tcp;
    17: parse_udp;
    default: accept;
  }
}
```
Go directly to

```
state parse_myTunnel {
  packet.extract(hdr.myTunnel);
  transition select(hdr.myTunnel.proto_id) {
    TYPE_IPV4: parse_ipv4;
    default: accept;
  }
}
```
A simple example for tunneling

```
header myTunnel_t {
  bit<16> proto_id;
  bit<16> dst_id;
}
struct headers {
  ethernet_t   ethernet;
  myTunnel_t   myTunnel;
  ipv4_t       ipv4;
}
parser MyParser(…) {
  state start {…}
  state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
      0x1212: parse_myTunnel;
      0x800: parse_ipv4;
      default: accept;
    }
  }
  state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition select(hdr.ipv4.protocol) {
      6: parse_tcp;
      17: parse_udp;
      default: accept;
    }
  }
  state parse_myTunnel {
    packet.extract(hdr.myTunnel);
    transition accept;
  }
  ...
}
```
Defining (and parsing) custom headers allow you
to implement your own protocols

```
header myTunnel_t {
  bit<16> proto_id;
  bit<16> dst_id;
}
parser MyParser(…) {
  state start {
    transition parse_ethernet;
  }
  state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
      0x1212: parse_myTunnel;
      0x800: parse_ipv4;
      default: accept;
    }
  }
  state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition select(hdr.ipv4.protocol) {
      6: parse_tcp;
      17: parse_udp;
      default: accept;
    }
  }
  state parse_myTunnel {
    packet.extract(hdr.myTunnel);
    transition accept;
  }
  ...
}
```
P4 parser supports both fixed and variable-width header extraction

P4 parser supports both fixed and variable-width header extraction

Parsing a header stack requires the parser to loop

the only “loops” that are possible in P4

Header stacks

for source routing

The parser contains more advanced concepts

check them out!

Parser Match-Action Pipeline Deparser

Control

Tables match a key and return an action

Actions similar to functions in C

Control flow similar to C but without loops
Match types are specified in the P4 core library and in the architectures:

- **exact**: exact comparison
  - `0x01020304`

- **ternary**: compare with mask
  - `0x01020304 & 0x0F0F0F0F`

- **type**: longest prefix match
  - `0x01020304/24`

- **range**: check if in range
  - `0x01020304 — 0x010203FF`

Table entries are added through the control plane:

- Actions are blocks of statements that possibly modify the packets
Actions usually take directional parameters indicating how the corresponding value is treated within the block.

Directions can be of three types:

- **in**: read only inside the action; like parameters to a function
- **out**: uninitialized, write inside the action; like return values
- **inout**: combination of in and out; like “call by reference”

Let’s reconsider a known example:

```cpp
action reflect_packet(inout bit<48> src, inout bit<48> dst, in bit<9> inPort, out bit<9> outPort) {
    bit<48> tmp = src;
    src = dst;
    dst = tmp;
    outPort = inPort;
}
```

Actions parameters resulting from a table lookup do not take a direction as they come from the control plane:

```cpp
action set_egress_port(bit<9> port) {
    standard_metadata.egress_spec = port;
}
```

Interacting with tables from the control flow:

- **Applying a table**
  
  ```cpp
  ipv4_lpm.apply();
  ```

- **Checking if there was a hit**
  
  ```cpp
  if (ipv4_lpm.apply().hit) {...} else {...}
  ```

- **Check which action was executed**
  
  ```cpp
  switch (ipv4_lpm.apply().action_run) {
    ipv4_forward: { ... }
  }
  ```

Example: L3 forwarding with multiple tables:

- **IP Packet**

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1.0/24</td>
<td>1</td>
</tr>
<tr>
<td>2.2.2.0/24</td>
<td>1</td>
</tr>
<tr>
<td>3.3.3.0/24</td>
<td>2</td>
</tr>
<tr>
<td>4.4.4.0/24</td>
<td>3</td>
</tr>
</tbody>
</table>

  - Map a prefix to a next hop index
  - Map a next hop index to an egress port

- **Forwarding**

  Map a packet to the right egress port.
Example: L3 forwarding with multiple tables

```c
table ipv4_lpm {
  key = {
    hdr.ipv4.dstAddr: lpm;
  }
  actions = {
    set_nhop_index;
    drop;
    NoAction;
  }
  size = 1024;
  default_action = NoAction();
}
table forward {
  key = {
    meta.nhop_index: exact;
  }
  actions = {
    _forward;
    NoAction;
  }
  size = 64;
  default_action = NoAction();
}
```

Applying multiple tables in sequence and checking whether there was a hit

```c
control myingress(...) {
  action drop() {...}
  action set_nhop_index(...}
  action _forward(...
  table ipv4_lpm {...
    apply {
      if (hdr.ipv4.isValid(){
        if (ipv4_lpm.apply().hit) {
          forward.apply();
        }
      }
    }
  }
}
```

Validating and computing checksums

```c
extern void verify_checksum<T, O>( in bool condition,
                             in T data,
                             inout O checksum,
                             HashAlgorithm algo
                            );
```

```c
extern void update_checksum<T, O>( in bool condition,
                             in T data,
                             inout O checksum,
                             HashAlgorithm algo
                           );
```

```c
MyComputeChecksum(...) {
  apply {
    update_checksum(
      hdr.ipv4.isValid(),
      { hdr.ipv4.version,
        hdr.ipv4.ihl,
        hdr.ipv4.diffserv,
        hdr.ipv4.totalLen,
        hdr.ipv4.identification,
        hdr.ipv4.flags,
        hdr.ipv4.fragOffset,
        hdr.ipv4.ttl,
        hdr.ipv4.protocol,
        hdr.ipv4.srcAddr,
        hdr.ipv4.dstAddr },
      hdr.ipv4.hdrChecksum,
    HashAlgorithm.csum16);
  }
}
```

Control flows contain more advanced concepts

- cloning packets
- sending packets to control plane
- recirculating

```
Parser Match-Action Pipeline Deparser
```

```
Packet
```

```
Headers
```

```
Deparser
```

```
Parser
```

```
MyIngress(...) {
  action drop() {...}
  action set_nhop_index(...
  action _forward(...
  table ipv4_lpm {...
    apply {
      if (hdr.ipv4.isValid(){
        if (ipv4_lpm.apply().hit) {
          forward.apply();
        }
      }
    }
  }
}
```

More info: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

"Full circle"