P4 & Named Data Networks

A networking paradigm for the future

SPEAKER: Rui Miguel
Navigators, Departamento de Informática
Faculdade de Ciências (FCUL), ULisboa

Under guidance of:
• Professor Fernando Ramos, DI-FCUL
• Professor Arjuna Sathiaseelan, Cambridge University
PRESENTATION OUTLINE (committed)

§1 • The P4 programming language for networks

§2 • Named data networks (NDN)  
§3 • (tentative) Expressing NDN using P4

• ...And a round of NavTalk attendee participation!
CHAPTER I

The P4 Programming Language
Different reprogrammable technologies (switches, NICs, FPGAs) each have their own capabilities and internal language.

The OpenFlow interface started simple, but has been frequently extended to allow switches to expose their capabilities, becoming overly complex.

It would be desirable that future switches support flexible parsing and header matching, through a common, open API (a new “OpenFlow 2.0”).
P4 is a generic, high-level language to describe target-independent processing on the forwarding plane.

1. It becomes possible to express a packet processing flow that is generic, flexible, and easily changeable.
2. Untied to specific protocols.
3. It is like programming in C with no concern regarding the underlying architecture!

P4 currently has two versions: P4-14 and P4-16.
   - We’ll discuss differences between them later.
While its motivation stems from SDN and OpenFlow, P4 does not assume:
- There is an SDN controller overseeing the network.
- The network is TCP/IP, Ethernet or otherwise.
- Devices are controlled using OpenFlow.

Assumptions P4 does make:
- If a device supports P4, then its vendors provide a back-end compiler to translate it to the device’s internal language.
- If two match-action stages are independent, then they can be performed in parallel, even if specified imperatively.
• We should take a look at another assumption made for the earlier version, P4-14 (*not true for P4-16*).

  ▫ A P4 compatible device will abide to the following model/architecture:
PROGRAMMING A P4-COMPATIBLE DEVICE

P4-16 PROGRAMMER

P4 CONSORTIUM

P4 Back-End Compiler

DEVICE FABRICANT

P4 Front-End Compiler

headers.p4

parser.p4

actions.p4

main.p4
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

```p4
typedef bit<48> EthernetAddress;

header ethernet {
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
    bit<16> etherType;
}

struct Parsed_packet {
    ethernet eth;
}
```

headers.p4
P4 – Learn by example
Defining a whee, simple L2-switch for IP packets

```
parser start {
    return ethernet_st;
}

header ethernet_t ethernet;

parser ethernet_st {
1    extract(ethernet);
2    return select(latest.ethType) {
3        0x800: ingress;
4        default: parse_error e;
5    }
6 }
```

```
parser_exception e {
    parser_drop;
}
```
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

```p4
header ethernet_t ethernet;

parser ethernet_st {
1 extract(ethernet);
2 return
   select(latest.ethType) {
3 0x800: ingress;
4 default: parse_error e;
5 }
6 }

parser_exception e {
   parser_drop;
}
```
CHAPTER 1 — P4 Programming Language

P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

```
header ethernet_t ethernet;

parser ethernet_st {  
1  extract(ethernet);
2  return
3   select(latest.ethType) {  
4     0x800: ingress;
5     default: parse_error e;
6   }
7 }

parser_exception e {  
  parser_drop;
}
```
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

header ethernet_t ethernet;

parser ethernet_st {
  1 extract(ethernet);
  2 return
    select(latest.ethType) {
      3 0x800: ingress;
      4 default: parse_error e;
    }
  }
}

parser_exception e {
  parser_drop;
}

PARSED PACKET
(device representation)

Space explicitly allocated for an Ethernet header

extract()
P4 — Learn by example

Defining a whee, simple L2-switch for IP packets

```p4
header ethernet_t ethernet;

parser ethernet_st {
1  extract(ethernet);
2  return
3  select(latest.ethType) {
4    0x800: ingress;
5  }
6 }

parser_exception e {
  parser_drop;
}
```

PARSED PACKET
(device representation)

- 8:56:27:6f:2b:9c
- 8:0:20:4:5:6
- 0x800

---

May 3, 2017
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

```
header ethernet_t ethernet;

parser ethernet_st {
  1  extract(ethernet);
  2  return
      select(latest.ethType) {
        3  0x800: ingress;
        4  default: parse_error e;
      }
  6 }

parser_exception e {
  parser_drop;
}
```

PARSED PACKET
(device representation)

```
8:56:27:6f:2b:9c
8:0:20:4:5:6
0x800
```
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

```p4
typedef bit<48>
    EthernetAddress;

header ethernet {
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
    bit<16> etherType;
}

struct Parsed_packet {
    ethernet eth;
}

parser TopParser(packet_in b, 
    out Parsed_packet p) {

    state start {
        1 b.extract(p.eth);
        2 transition
            select(p.eth.etherType) {
                3 0x800: accept;
                4 _ : reject;
            }
    }
}
```

P4-16
P4 — Learn by example

Defining a whee, simple L2-switch for IP packets

```p4
typedef bit<48>
    EthernetAddress;

header ethernet {
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
    bit<16> etherType;
}

struct Parsed_packet {
    ethernet eth;
}

parser TopParser(packet_in b, out Parsed_packet p) {

    state start {
        b.extract(p.eth);
        transition select(p.eth.etherType) {
            0x800: accept;
            _ : reject;
        }
    }
}
```
CHAPTER 1 — P4 Programming Language

P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

**typedef** bit<48>
EthernetAddress;

**header** ethernet {
EthnetAddress dstAddr;
EthernetAddress srcAddr;
bit<16> etherType;
}

**struct** Parsed_packet {
ethernet eth;
}

**parser** TopParser(packet_in b, 
out Parsed_packet p) {

**state** start {
1 b.extract(p.eth);
2 transition
select(p.eth.etherType) {
3 0x800: accept;
4 _ : reject;
5 }
}

}
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

**typedef** bit<48> EthernetAddress;

**header** ethernet {
  EthernetAddress dstAddr;
  EthernetAddress srcAddr;
  bit<16> etherType;
}

**struct** Parsed_packet {
  ethernet eth;
}

**parser** TopParser(packet_in b, out Parsed_packet p) {

  **state** start {
    1 b.extract(p.eth);
    2 **transition**
    3 select(p.eth.etherType) {
      3 0x800: accept;
      4 _ : reject;
    }
  }

  }

}
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

header ethernet_t ethernet;

parser ethernet {
  extract (ethernet);
  return
  select(latest.etherType) {
    0x800: ingress;
    default: parse_error e;
  }
}

parser_exception e {
  parser_drop;
}

parser TopParser(packet_in b, Parsed_packet p) {

  state start {
    b.extract(p.eth);
    transition
    select(p.eth.etherType) {
      0x800: accept;
      _ : reject;
    }
  }
}
• Our exercise is giving us a glimpse on how to implement our own protocols in P4.
  ▫ By defining our own headers and writing our own parser, we can program the forwarding device to expect non-standard protocols... or the standard ones we already know and love!
  ▫ This also demonstrates P4 is quite flexible, as it aspired to be.
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

```p4
action set_output_port (port) {
    modify_field(standard_metadata.egress_port, port);
}

table forward {
    reads {
        ethernet.dstAddr: exact;
    }
    actions {
        set_output_port;
        drop;
    }
}

control ingress {
    apply(forward);
}
```

P4 is forbidden from WRITING (adding entries) to tables, but it can READ from them.
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

Main.p4

```p4
action set_output_port (port) {
  modify_field(standard_metadata .egress_port, port);
}

table forward {
  reads {
    ethernet.dstAddr: exact;
  }
  actions {
    set_output_port;
    drop;
  }
}

control ingress {
  apply(forward);
}
```

There’s something awkward here. We are expecting a parameter when `set_output_port()` is invoked, but we are passing none in the table declaration. So how does it know, in execution time, what parameter is being passed?

The official answer is: the control plane decides. For simpler reasoning, we can imagine it is part of the table.
**P4 — Learn by example**

**Defining a whee, simple L2-switch for IP packets**

```p4
action set_output_port (port) {
    modify_field(standard_metadata.egress_port, port);
}

table forward {
    reads {
        ethernet.dstAddr: exact;
    }
    actions {
        set_output_port;
        drop;
    }
}

control ingress {
    apply(forward);
}
```

<table>
<thead>
<tr>
<th>DESTINATION MAC</th>
<th>ACTION</th>
<th>PORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:56:27:6f:2b:9c</td>
<td>set_output_port</td>
<td>3</td>
</tr>
<tr>
<td>8:0:20:4:5:6</td>
<td>set_output_port</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>drop</td>
<td>-</td>
</tr>
</tbody>
</table>
P4 — Learn by example
Defining a wheee, simple L2-switch for IP packets

Main.p4

```
action set_output_port (port) {
  modify_field(standard_metadata.egress_spec, port);
}

table forward {
  reads {
    ethernet.dstAddr: exact;
  }
  actions {
    set_output_port;
    drop;
  }
}

control ingress {
  apply(forward);
}
```

```
control TopMainIngr(
  inout standard_metadata_t m,
  inout Parsed_packet p) {

  action set_output_port(port) {
    m.egress_spec = port;
  }

  table forward {
    key = { p.eth.dstAddr: exact; }
    actions = {
      set_output_port;
      Drop;
    }
  }
}
```
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

```p4
action set_output_port (port) {
    modify_field(standard_metadata.egress_port, port);
}

table forward {
    reads {
        ethernet.dstAddr: exact;
    }
    actions {
        set_output_port;
        drop;
    }
}

control ingress {
    apply(forward);
}

control TopMainIngr(...) {
    action set_output_port(port) {
        m.egress_spec = port;
    }
    table forward {
        ... 
    }
    apply {
        forward.apply();
    }
}
```
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

Main.p4

```p4
control egress {

}
```

```p4
control TopMainEgr(...) {

    apply {

    }

}
```
action rewrite_mac(smac) {
    modify_field(ethernet.srcAddr, smac);
}

table set_egress {
    reads {
        standard_metadata.
            egress_port : exact;
    }
    actions {
        rewrite_macs;
        drop;
    }
}

control egress {
    apply(set_egress);
}

control TopMainEgr(...) {
    action set_srcMAC(address) {
        p.eth.srcAddr = address;
    }

    table set_egress {
        key = { m.egress_port: exact;}
        actions = {
            set_srcMAC;
            Drop;
        }
    }

    apply { set_egress.apply(); }
}
P4 — Learn by example
HYPOTHETICAL L3 ROUTER REWRITING SMAC

```p4
action rewrite_mac(smac) {
    modify_field(ethernet.srcAddr, smac);
}

table set_egress {
    reads {
        standard_metadata.
        egress_port : exact;
    }
    actions {
        rewrite_macs;
        drop;
    }
}

control egress {
    apply(set_egress);
}
```

The takeaway of that slide: you can also match on metadata!
P4 — Learn by example
Defining a whee, simple L2-switch for IP packets

```p4
control TopDeparser {
    packet_out b,
    in Parsed_packet p) {

    apply {
        b.emit(p.eth);
    }
}
```

Any P4-14 device follows a common model.
- PARSER → INGRESS → EGRESS
  - But is that model fitting for all purposes (e.g. firewalls?)

P4-14 defines primitives that compatible devices will implement… most of the time.
  - Will they?
  - They may implement extra features but be unable to expose them?

Very stable core language, updated rarely.

Not bound to the assumed P4-14 model/architecture.
- Vendors specify, in P4 syntax, an architecture definition file.
  - Software switch users that define new architectures must build their own back-end compiler?

Enhanced parameterization features.
- You may have noticed earlier.

Selective deparser.

Extern blocks and functions.
ABOUT P4-16
Our P4-14 workflow

P4-16 PROGRAMMER

DEVICE FABRICANT

P4 CONSORTIUM

P4 Front-End Compiler

P4 Back-End Compiler

headers.p4
parser.p4
actions.p4
main.p4
ABOUT P4-16

Our P4-16 workflow

Our P4-16 workflow is as follows:

1. **P4-16 PROGRAMMER**
2. **P4 CONSORTIUM**
   - **Front-End Compiler**
   - **Back-End Compiler**
3. **DEVICE FABRICANT**

### ARCHITECTURE DEFINITION FILE

```cpp
/** *
 * Programmable parser.
 * @param \texttt{\textless}H\texttt{\textgreater} Type of headers.
 * @param \texttt{b} Input packet.
 * @param \texttt{p} Representation built by the parser.
 */

parser Parser<

```
The figure is a possible architecture in P4-16, but not the default, mandatory and only architecture, as in in P4-14.
The figure is a possible architecture in P4-16, but not the default, mandatory and only architecture, as in in P4-14.

- P4-16-compatible devices must provide an architecture definition file on top of the back-end compiler.
ABOUT P4-16
See how a simple drop can differ across architectures

action Drop()
{ ctrl.outPort = DROP_PORT; }
ABOUT P4-16
See how a simple drop can differ across architectures

**Action** Drop()
{ ctrl.outPort = DROP_PORT; }

**Architectural Considerations**

**Architecture A**
DROP_PORT is a special port, a `#define` in the architecture definition file.

**Architecture B**
Drop is performed by setting a metadata bit.
ABOUT P4-16
See how a simple drop can differ across architectures

**ARCHITECTURE A**

```c
action Drop()
{ ctrl.outPort = DROP_PORT; }
```

DROP_PORT is a special port, a `#define` in the architecture definition file.

**ARCHITECTURE B**

```c
action Drop()
{ metadata.dropBit = 1; }
```

Drop is performed by setting a metadata bit.

**ARCHITECTURE C**

```c
action Drop()
{ mark_for_drop(); }
```

mark_for_drop() is an `extern` function declared in the architecture definition file.
ABOUT P4-16
A sample architecture definition

```c
/** *
 * Programmable parser.
 * @param <H> Type of headers.
 * @param b Input packet.
 * @param p Representation built by the parser.
 */
parser Parser<H>(packet_in b, out H p);

struct ctrl_t {
    bit<5> inputPort;
    bit<5> outputPort;
}

/** *
 * Main match-action pipeline.
 * @param <T> Type of headers.
 * @param headers Received from the parser and sent to the deparser.
 * @param ctrl Input and output ports. Input set by control plane.
 */
control Pipe<T>(inout T headers, inout ctrl_t ctrl);
```
A sample architecture definition

```c
/** *
 * Programmable parser.
 * @param <D> Type of headers.
 * @param b Output packet.
 * @param p Representation built by the parser.
 */
control Deparser<D>(inout D outputHeaders, packet_out b);

/** *
 * Top-level package declaration - must be instantiated by user.
 * @param <S> User-defined type of the headers processed.
 */
package VerySimpleSwitch<S>(
    Parser<S> p,
    Pipe<S> map,
    Deparser<S> d);
```
ABOUT P4-16
A sample architecture definition

/** * Programmable parser.
  * @param <D> Type of headers.
  * @param b Output packet.
  * @param p Representation built by the parser.
  */
control Deparser<D>(inout D outputHeaders, packet_out b);

/** * Top-level package declaration - must be instantiated by user.
  * @param <S> User-defined type of the headers processed.
  */
package VerySimpleSwitch<S>(
  Parser<S> p,
  Pipe<S> map,
  Deparser<S> d);

The programmer instantiates this device in **Main.p4**, by writing:

VerySimpleSwitch(TopParser(), TopPipe(), TopDeparser()) main;

(TopParser is the name we gave to our parser in **Parser.p4**, and so on)
ABOUT P4-16
Extern blocks and functions

- **Extern** blocks/functions are programmed by the device manufacturer and specified in P4 syntax on the architecture definition file.
  - They are all-powerful as the manufacturer can program them to do pretty much anything!

```p4
extern register<T> {  
    register(bit<32> size); // Constructor
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}

extern void recirculate<T>(in T data);
```
ABOUT P4-16
Enhanced parameterization

//Extracts a myheader_h header into h.
parser Subparser(packet_in b, out myheader_h h)(bool specialField)
{
    state start { ... }
}

parser TopParser(packet_in b, Parsed_packet p) {
    Subparser(true) s; //Instantiates a Subparser.
    state start {
        s.apply(b, p.someHeader);
        ...
    }
}
CONCLUSIONS

• P4 is a high-level language for programmable network devices.
  ▫ It is motivated by OpenFlow’s rigidity on supporting new protocols.
  ▫ It is a paradigm similar to C programming for an arbitrary underlying processor architecture.
  ▫ Devices must provide a back-end compiler to translate it to its own internal language. Such a device is P4-compatible.
  ▫ You run whatever protocols you want... and only those you want!

• On the language.
  ▫ P4-14 is loosely typed and more verbose for some use cases.
  ▫ P4-16 is strongly typed and has more parameterization features.
On the trade-off between portability and functionality.

- P4-14 assumes a common model (parser -> ingress -> egress), which makes programs more portable. However, device manufacturers are constrained on the functionality their switches can provide.
- P4-16 allows manufacturers to expose more of their device through diverse control blocks, `extern` functions and instantiable blocks, at the cost of portability.

- Use P4-16, as P4-14 is likely to be deprecated ;)

CONCLUSIONS
CHAPTER II

Named Data Networks (NDNs)