# Data Plane and P4

Version 1, Presented at QuTech (Delft) on 12 September 2019

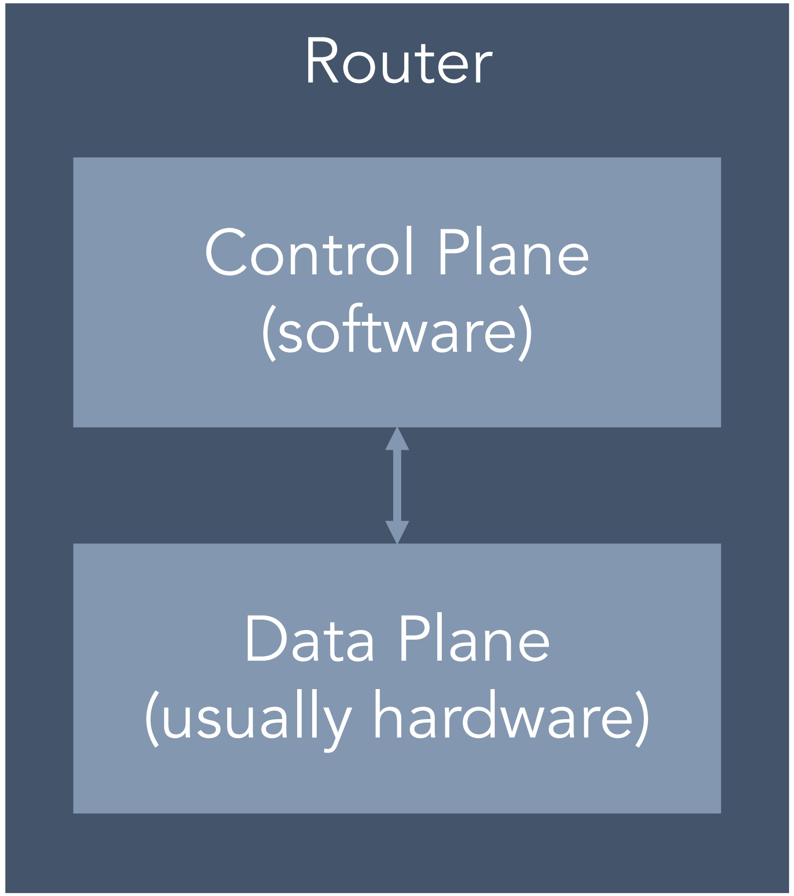
# Agenda

1. Control plane versus data plane
2. Fixed-function classical data plane
3. Programmable classical data plane
4. Control plane – data plane interface
5. Programmable quantum data plane

Section 1

# Control plane versus data plane

# Control plane versus data plane



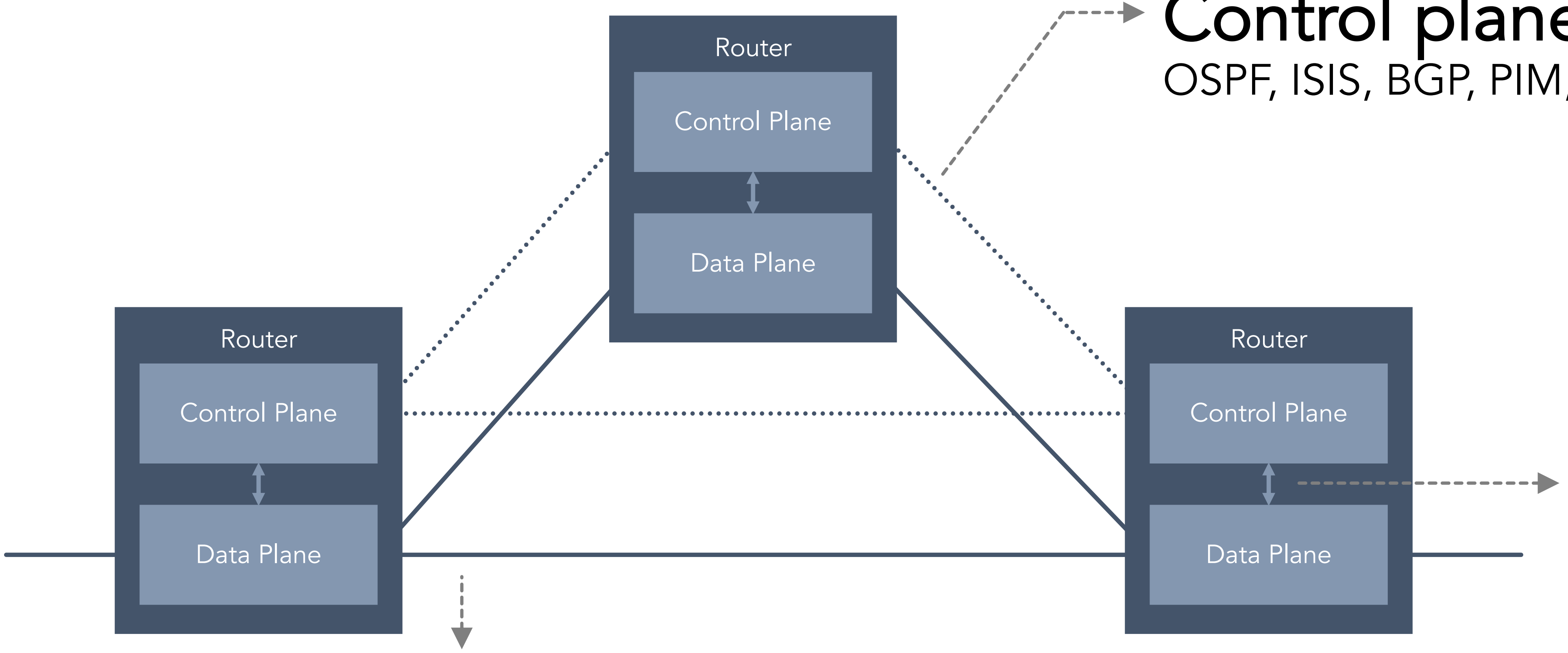

## Control plane

- Discover network topology and resources
- Program the forwarding plane (e.g. forwarding table)
- Flexibility is main concern
- Software on general purpose OS and CPU

## Data plane (aka forwarding plane)

- Forward packets
- Using the forwarding table populated by the control plane
- Speed is main concern
- Fixed-function hardware, programmable hardware, software

# "Traditional" distributed control plane

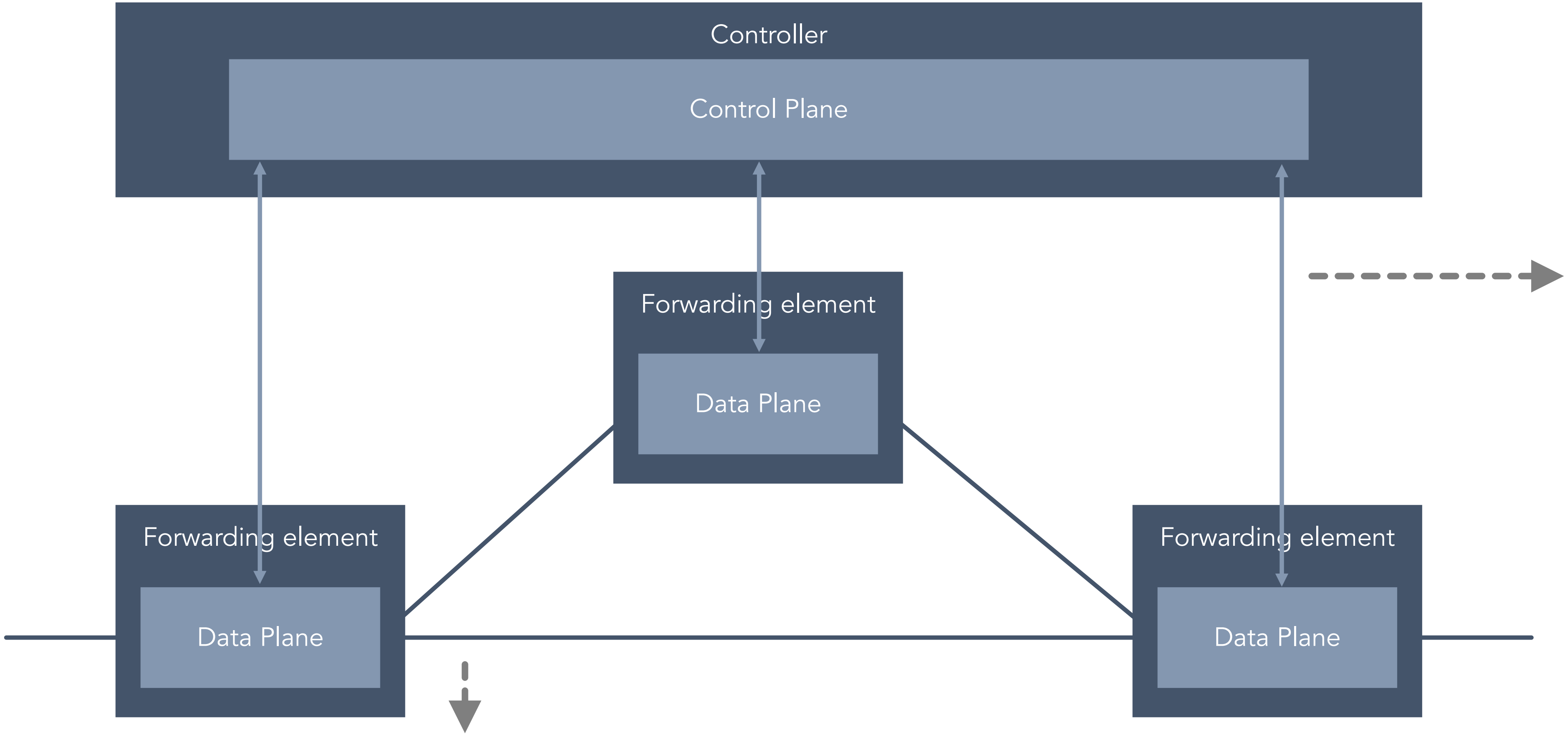



## Control plane protocols
OSPF, ISIS, BGP, PIM, RSVP, LDP, …

## API
proprietary,
ForCES,
OpenFlow,
SAI,
P4 generated…

## Data plane protocols
Ethernet, VLAN, IPv4, IPv6, MPLS, VXLAN, …

# SDN: centralized control plane (simplified)




**API**
proprietary,
ForCES,
OpenFlow,
SAI,
P4 generated…

**Data plane protocols**
Ethernet, VLAN, IPv4, IPv6, MPLS, VXLAN, …

# More planes

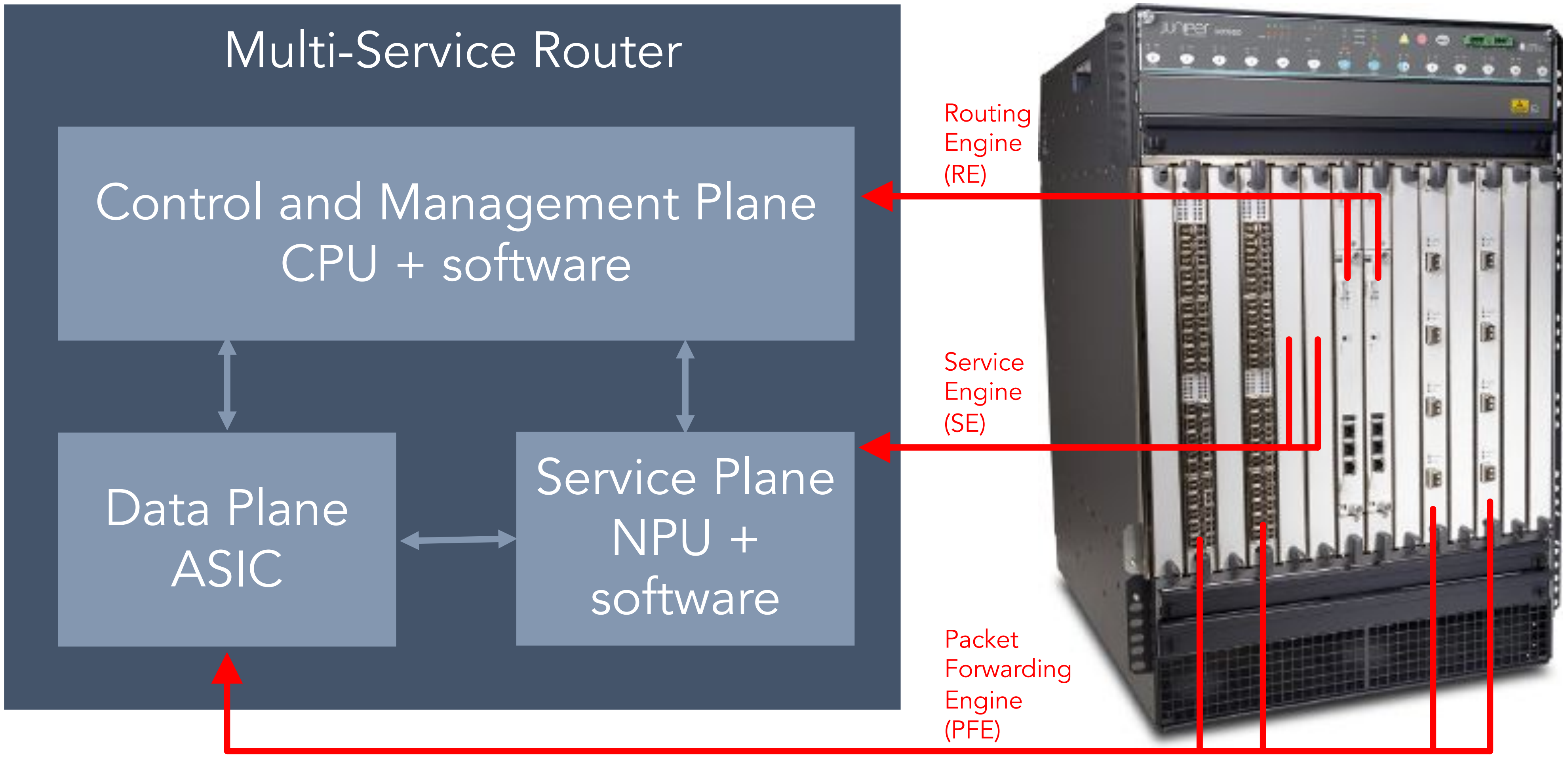
## Management plane
Configure and monitor device
General purpose CPU with general purpose OS
Examples: SSH, NETCONF, SNMP, …

## Control plane
Routing protocols, signaling protocols
General purpose CPU with general purpose OS
Examples: OSPF, ISIS, BGP, LDP, PIM, …

## Service plane
Stateful flow processing
Networking NPU with real-time OS
Example: Application layer firewall (reconstruct TCP stream)

## Data plane (aka forwarding plane)
Stateless packet-by-packet forwarding
Networking ASIC
Examples: Ethernet, IPv4, IPv6, MPLS, ..

# Physical implementation of the four planes



Section 2

# Fixed-function classical data plane

# What does the data plane do? (A lot!)

- Parse received packets, compose (“de-parse”) sent packets
- Layer 3 longest prefix match lookup of destination IP address
- Layer 2 exact match lookup of destination Ethernet address
- Exact match lookup of VLAN tag(s) and MPLS tag(s)
- Lookup of source address for Ethernet address learning and IP reverse path validation
- Handle congestion: buffer packets and implement drop policies
- Implement Quality of Service (QoS): scheduling and packet marking
- Implement (Non-) Equal Cost Multipath (N)ECMP: compute hashes, balance flows
- Apply policies, also known as Access Control Lists (ACLs)
- Track metrics (counters, thresholds, …), generate in-line telemetry
- Make copies for multicast and mirroring
- Interface with the control plane (punt control packets, program tables, …)
- … etc. etc. etc. … (*lots more*)

# Fixed pipeline data plane

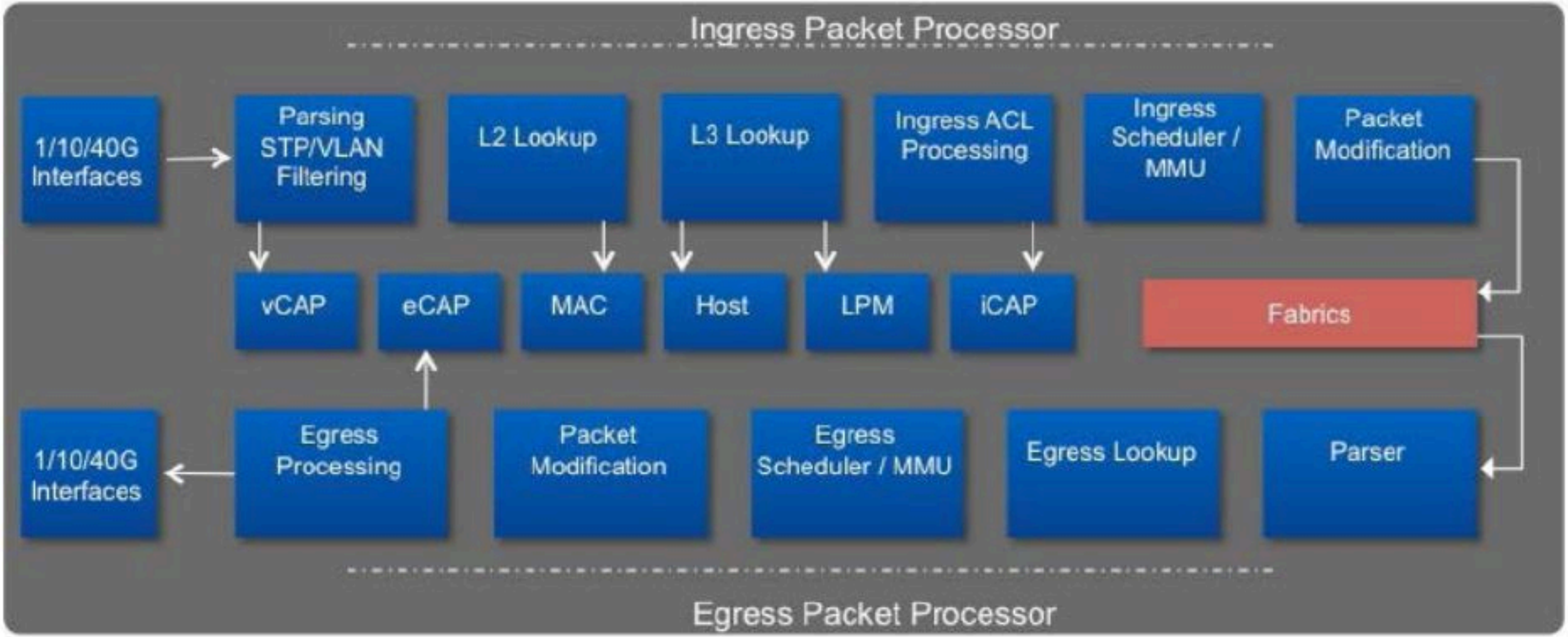
**Most things are fixed in hardware and cannot be changed:**

- The processing steps for each packet are fixed in hardware
- Fixed set of encapsulations (e.g. Ethernet, IPv4, IPv6, GRE, VXLAN)
- Fixed sequence of lookup tables (e.g. VLAN, Ethernet, IP, ACLs, …)
- Fixed set of features (e.g. multicast, ECMP, telemetry, …)

**However, <u>not everything is fixed</u>:**

- Dynamically populate contents of tables (e.g. routes, ACLs, …)
- Dynamically configure certain features (e.g. speed of interface 10/40)
- Confusingly, this is sometimes called "the control plane programs the data plane"

# Example fixed forwarding pipeline

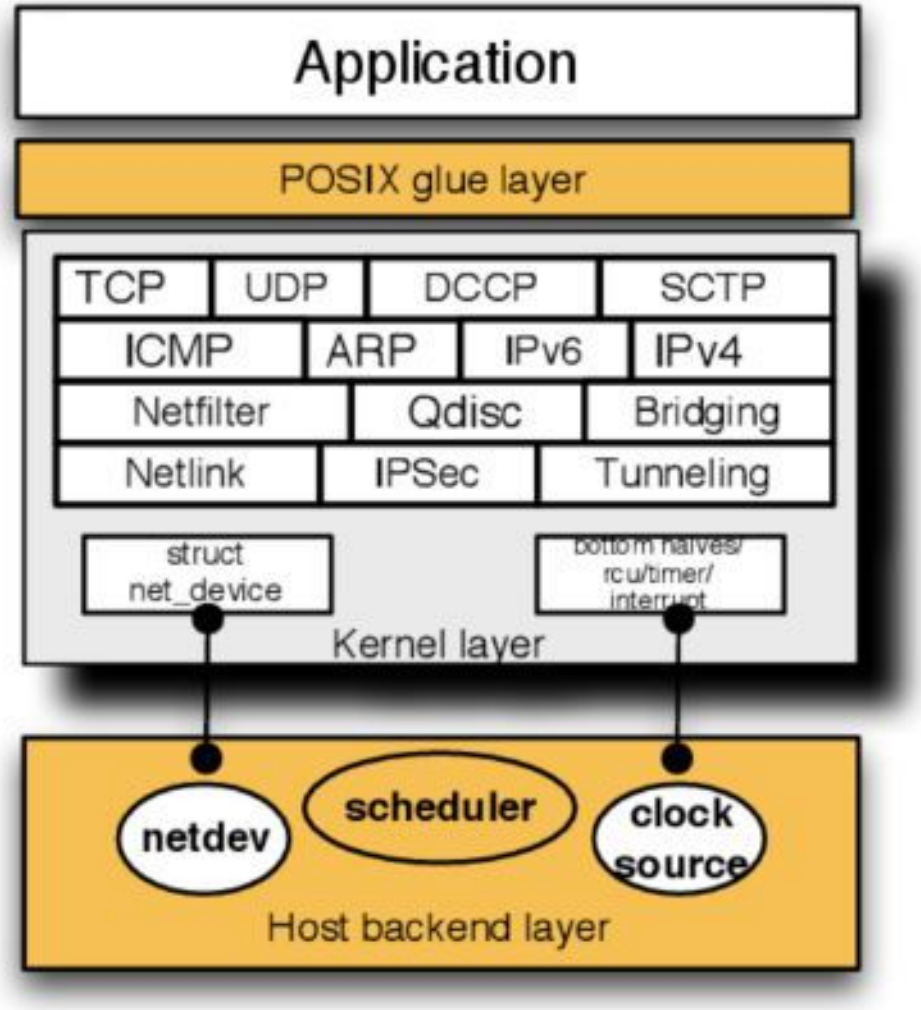


# Fixed pipeline in hardware: Tomahawk 3


## Broadcom Tomahawk 3

- ~ Dec 2017
- StrataXGS BCM56980
- 12.8 terabits per second (Tbps)
- 128x100Gbps / 64x200Gbps / 32x400Gbps
- ~ 8 billion packets per second (0.04 nsec/packet)
- 500ns or 300ns latency
- Relatively small forwarding tables
- Relatively shallow buffers
- Lean feature set
- **Data plane is not programmable**

# Fixed pipeline in software: Linux kernel stack




## Legacy Linux networking stack

- Linux can be configured to be a software switch
- Ethernet, IPv4, IPv6, VLANs, ACLs, …
- Everything implemented in software (with option for hardware acceleration)
- Socket interface to send and receive packets (inet sockets, raw sockets)
- Socket interface to populate the forwarding tables and to configure options (netlink sockets)
- Note: recent versions of Linux have a programmable network stack (discussed later)

# Data plane considerations

## Optimize for (trade-off):

- Cost
- Throughput
- Latency
- Buffer depth
- Table sizes (routes, ACLs,…)
- Features (protocols, telemetry, …)
- Flexibility (programmability)
- Power (cooling)
- Density

## Constrained by:

- Die size limits (800 $mm^2$)
- Process geometry
- Internal / external memory
- IO speed (SERDES)
- Power dissipation
- Development cost
- Device cost
- Time to market

# Single-stage versus multi-stage data plane



*Source: Facebook, OCP*

| | FB Backpack | Next Gen |
|---|---|---|
| **Capacity** | 128x100GbE | 32x400GbE / 128x100GbE |
| **Front panel** | 128x QSFP28 | 32x QSFP-DD or OSFP |
| **Size** | 8U Chassis | 1U Fixed |
| **# Switch Chips** | 12 x 3.2T | 1 x 12.8T |

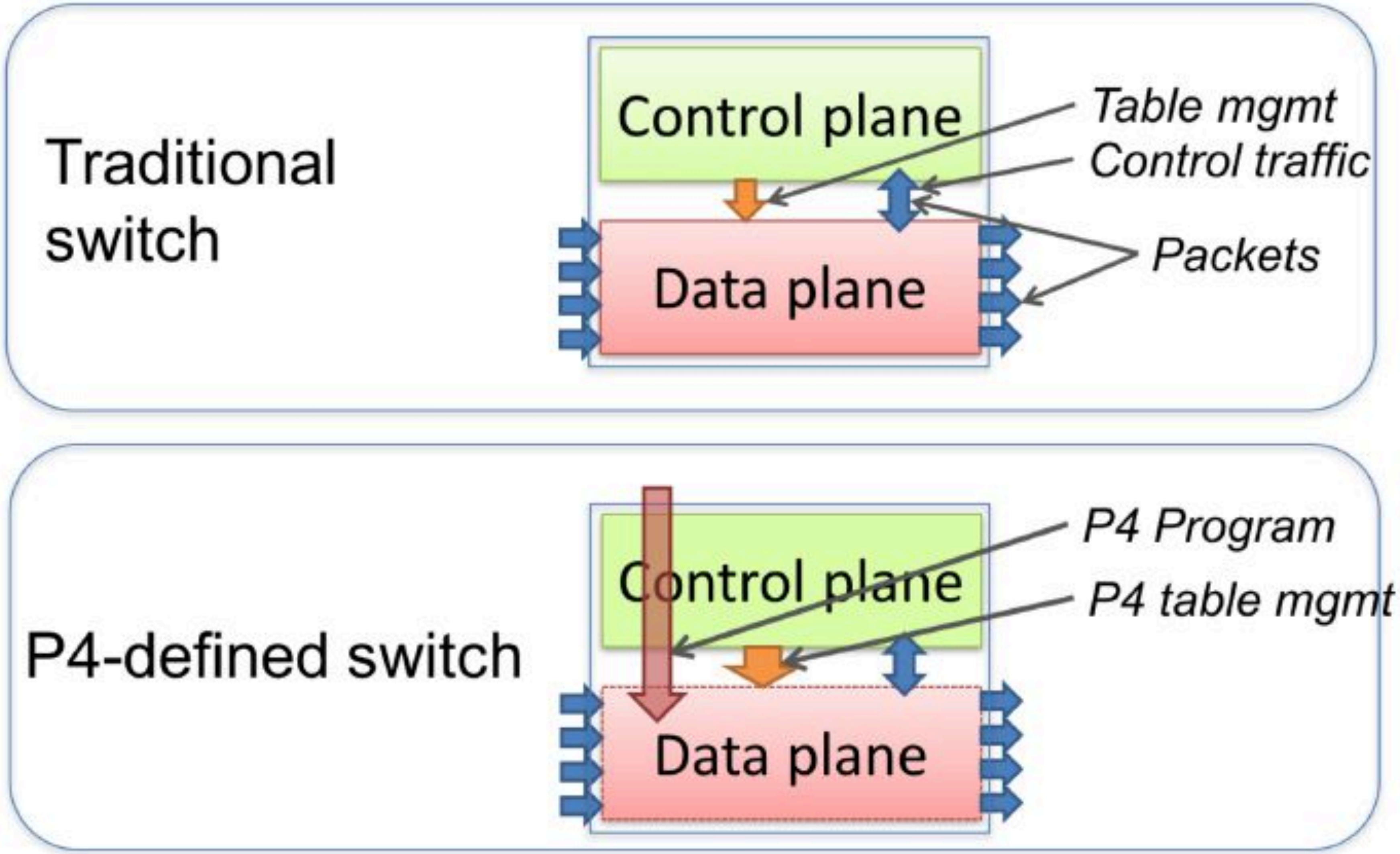
**75% Reduction in System Power, 85% reduction in System Cost ***

Section 3

# Programmable classical data plane

( Using P4 as the main example )

# Programmable pipeline vs fixed pipeline



# Controllable vs automated quantum nodes

Definitions from *"A Link Layer Protocol for Quantum Networks"*:

- **Controllable node**: offers the possibility to perform controllable quantum operations as well as storing qubits. Specifically, these nodes enable decision making
- **Automated node**: are typically only timing controlled, i.e. they perform the same preprogrammed action in each time step.

Actions fixed in hardware

Pipeline fixed in hardware,

Controlled by tables

Tables populated by software

Fixed match-action architecture + set of offload blocks

Programmable pipeline

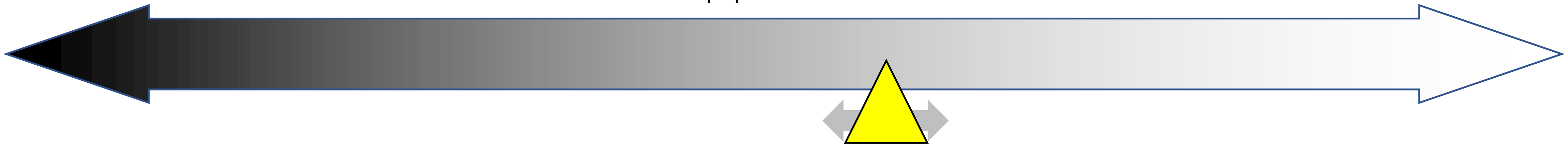
Programmable pipeline

+

Service plane

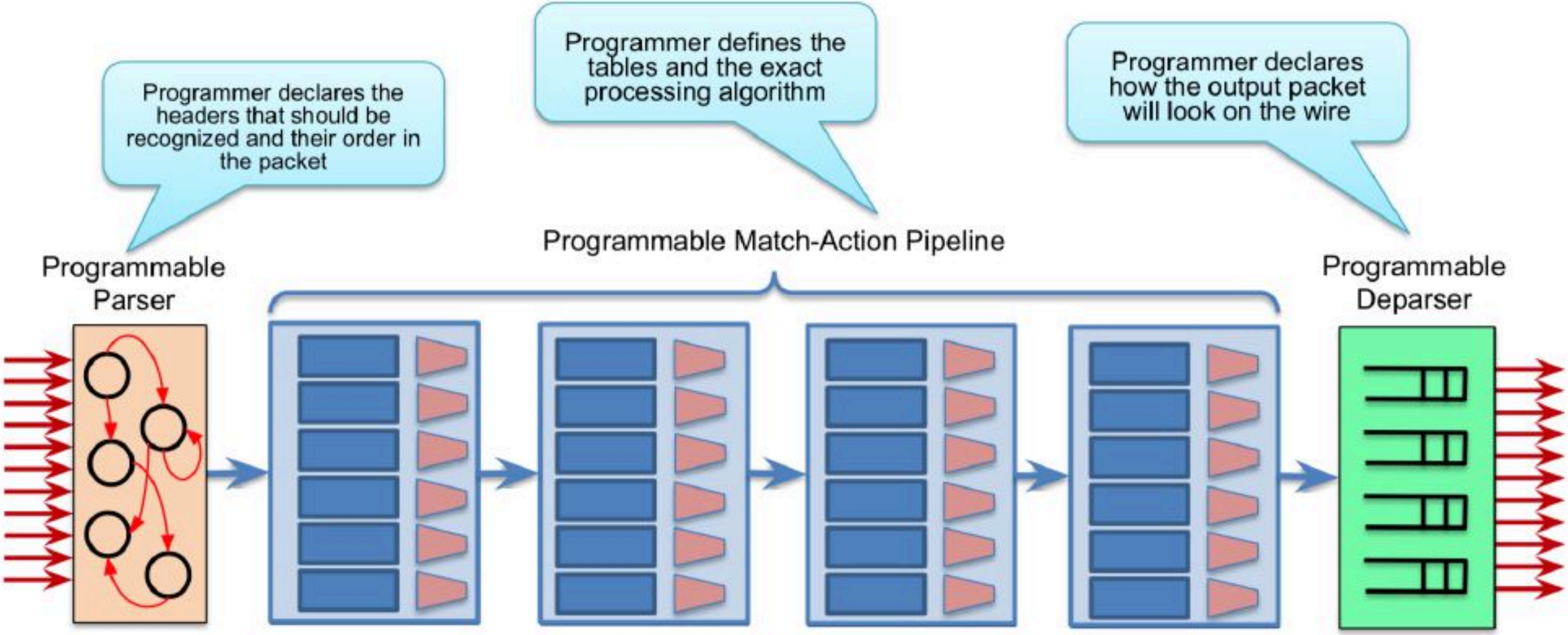
Fully programmable

Possibly using specialized instruction set



Influences the interface between control plane and data plan

# Parser, match-action tables, de-parser



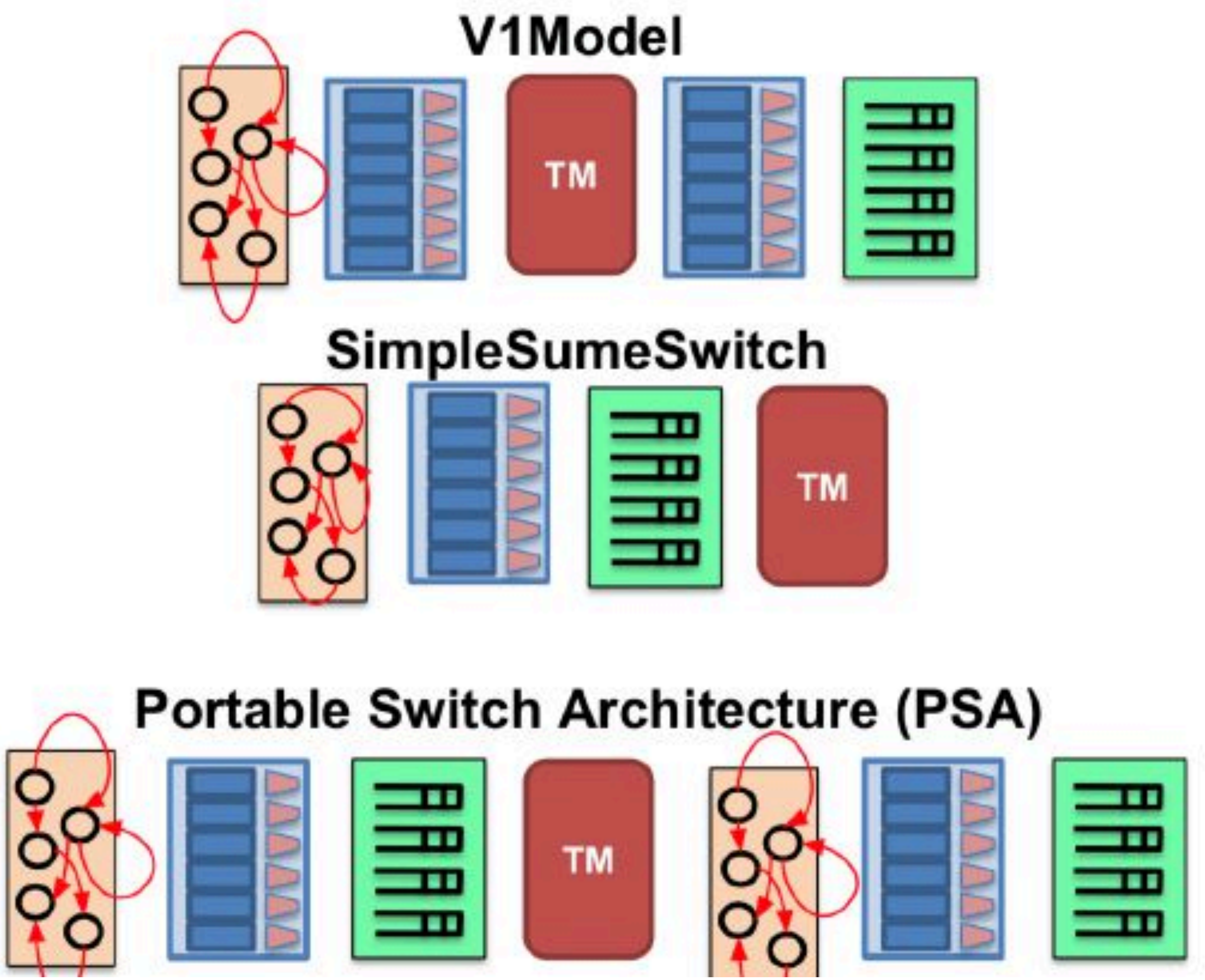
# Advantages of a programmable pipeline

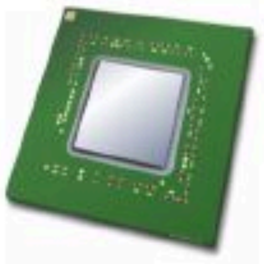
- Ability to make the data plane do what you want it to do
- Ability to define new data plane protocols
- Remove unused data plane protocols (reduced complexity)
- Flexible and efficient use of table resources
- Greater visibility (new telemetry protocols)
- Software-style of development
- Keep your own ideas

**The data plane for quantum networks is not yet well-defined!**

*So we absolutely need a programmable data plane*

# Architectural diversity of P4 targets

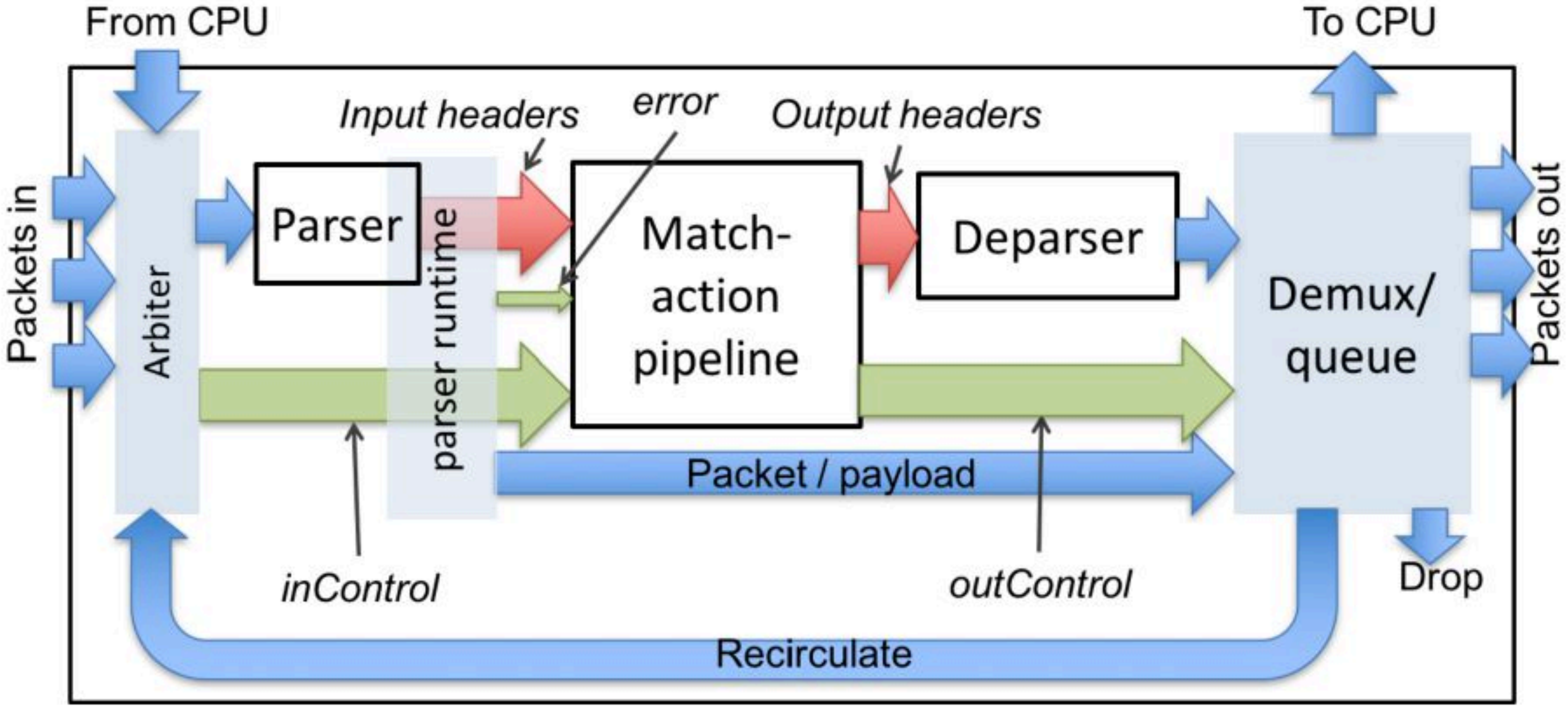


# P4 architecture definition and externs library

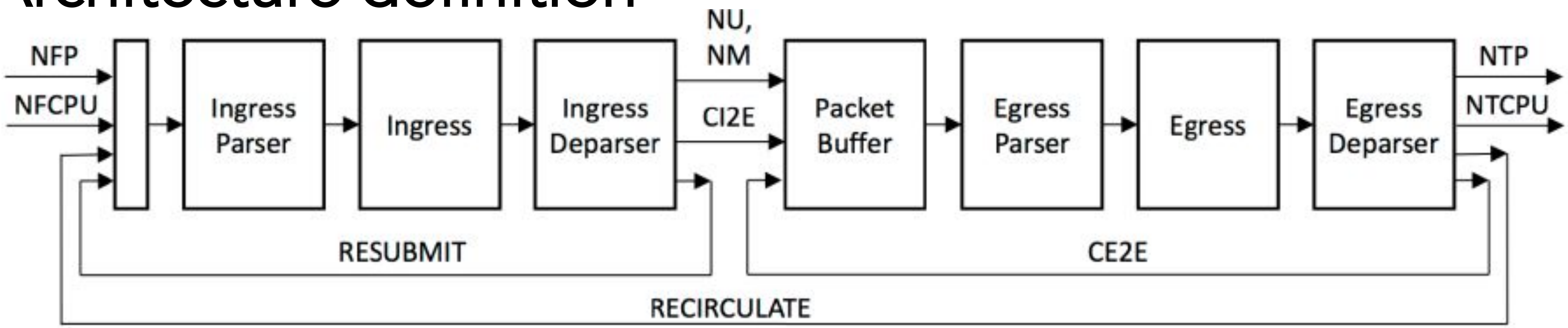
| Term | Explanation |
|---|---|
| P4 Target | An embodiment of a specific hardware implementation |
| P4 Architecture | Provides an interface to program a target via some set of P4-programmable components, externs, fixed components |

# Very simple switch architecture



# Portable Switch Architecture (PSA)

## Architecture definition

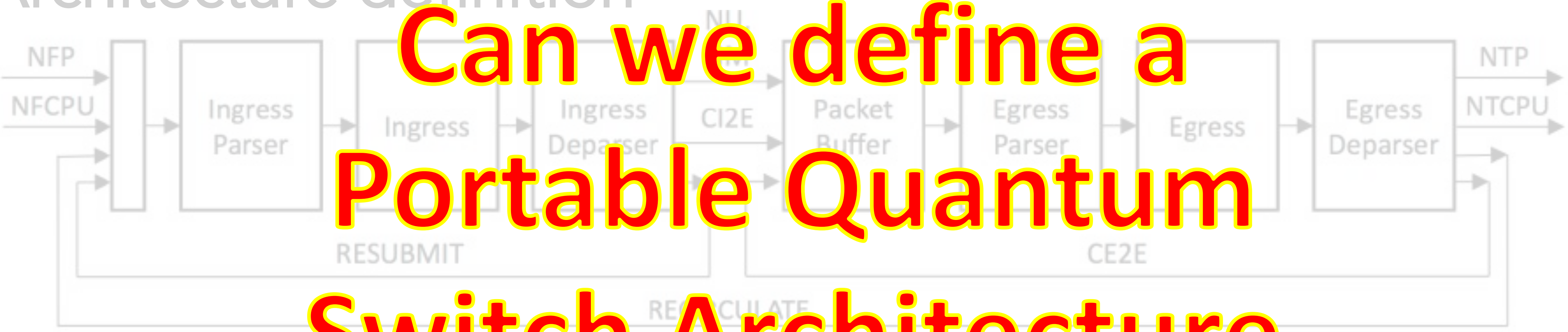



## Externs library

| | | | | | | |
|---|---|---|---|---|---|---|
| Packet Replication Engine | Buffering and Queueing Engine | Hashes | Checksums | Counters | Meters | Registers |
| | | Random | Action Profile | Action Selector | Timestamp | Packet Digest |

# Portable Switch Architecture (PSA)

## Architecture definition




## Can we define a Portable Quantum Switch Architecture (PQSA) ?

## Externs library

- Packet Replication Engine
- Buffering and Queueing Engine
- Hashes
- Checksums
- Counters
- Meters
- Registers
- Random
- Action Profile
- Action Selector
- Timestamp
- Packet Digest

# Example P4 header definitions

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

```
header IPv4_h {
    bit<4>        version;
    bit<4>        ihl;
    bit<8>        diffserv;
    bit<16>       totalLen;
    bit<16>       identification;
    bit<3>        flags;
    bit<13>       fragOffset;
    bit<8>        ttl;
    bit<8>        protocol;
    bit<16>       hdrChecksum;
    IPv4Address   srcAddr;
    IPv4Address   dstAddr;
}
```

# Example P4 meta data definitions (structs)

## Architecture defined meta-data

```
struct psa_egress_input_metadata_t {
  ClassOfService_t  class_of_service;
  PortId_t          egress_port;
  PSA_PacketPath_t  packet_path;
  EgressInstance_t  instance;
  Timestamp_t       egress_timestamp;
  ParserError_t     parser_error;
}

struct psa_egress_output_metadata_t {
  bool              clone;
  CloneSessionId_t  clone_session_id;
  bool              drop;
}
```

## Program defined meta-data

```
struct Parsed_packet {
    Ethernet_h ethernet;
    IPv4_h     ip;
}
```

# Example P4 parser (state machine)

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

    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            0x0800: parse_ipv4;
        }
    }

    state parse_ipv4 {
        b.extract(p.ip);
        verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);
        verify(p.ip.ihl == 4w5, error.IPv4OptionsNotSupported);
        transition accept;
    }
}
```

# Example P4 control block (match-action tables)

```
control TopPipe(inout Parsed_packet headers,
                in error parseError,
                in InControl inCtrl,
                out OutControl outCtrl) {
...
      action Set_nhop(IPv4Address ipv4_dest, PortId port) {
          nextHop = ipv4_dest;
          headers.ip.ttl = headers.ip.ttl - 1;
          outCtrl.outputPort = port;
      }
...
      table ipv4_match {
         key = { headers.ip.dstAddr: lpm; }  // longest-prefix match
         actions = {
              Drop_action;
              Set_nhop;
         }
         size = 1024;
         default_action = Drop_action;
     }
```

# Example P4 control block (apply section)

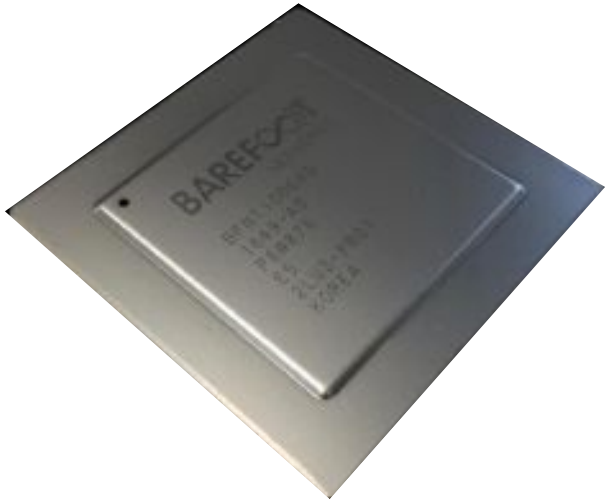
```
...
    apply {
...
          // Lookup destination IPv4 address in IPv4 route table to determine next-hop
          ipv4_match.apply();
          if (outCtrl.outputPort == DROP_PORT) return;

          // Lookup remaining TTL in TTL table to decide whether to punt to CPU
          check_ttl.apply();
          if (outCtrl.outputPort == CPU_OUT_PORT) return;

          // Lookup next-hop IP address in dmac table to determine destination MAC address
          dmac.apply();
          if (outCtrl.outputPort == DROP_PORT) return;

          // Lookup outgoing port in smac table to determine source MAC address
          smac.apply();
    }
}
```

# Programmable pipeline hardware (P4)




## **Barefoot Tofino 2** (Now Intel)

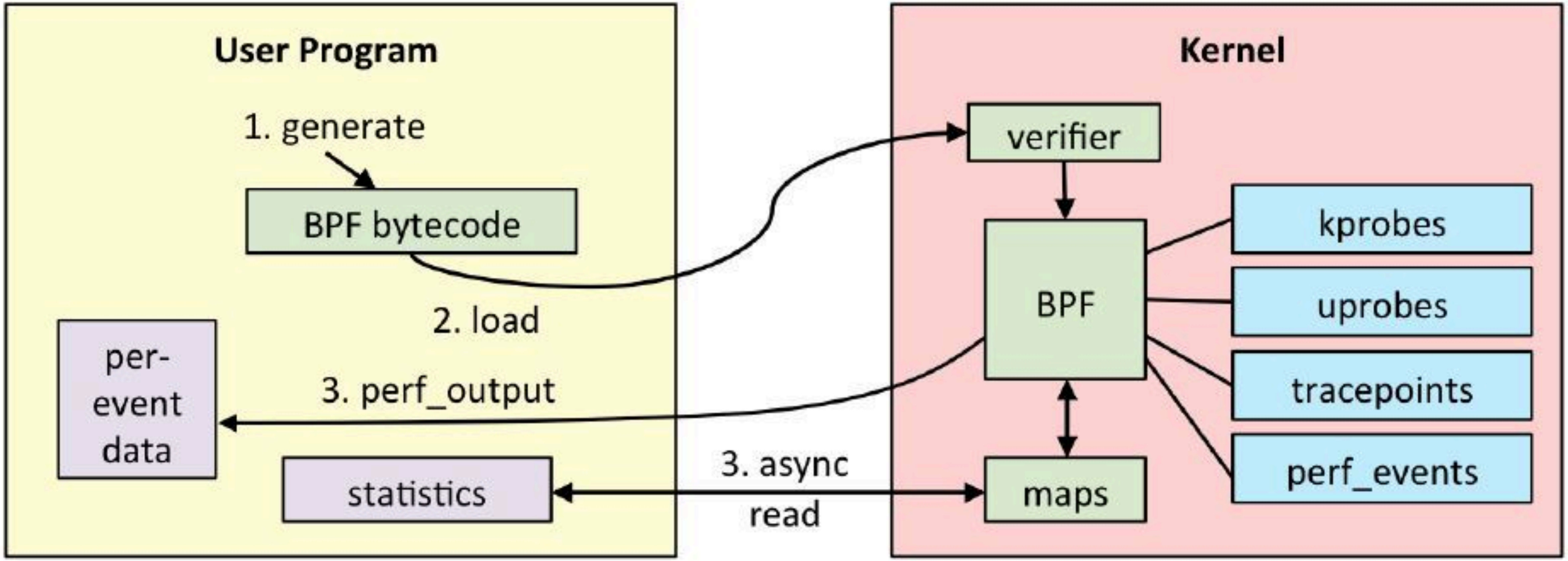
- ~ Jan 2019
- 12.8 Tbps
- Rich feature set
- Programmable using P4 high-level language
  https://p4.org

# Programmable pipeline hardware (vendor-specific)


## Broadcom Trident 4

- ~ June 2019
- StrataXGS BCM56880
- 12.8 terabits per second (Tbps)
- Rich feature set
- Programmable using NPL high-level language
  https://nplang.org/

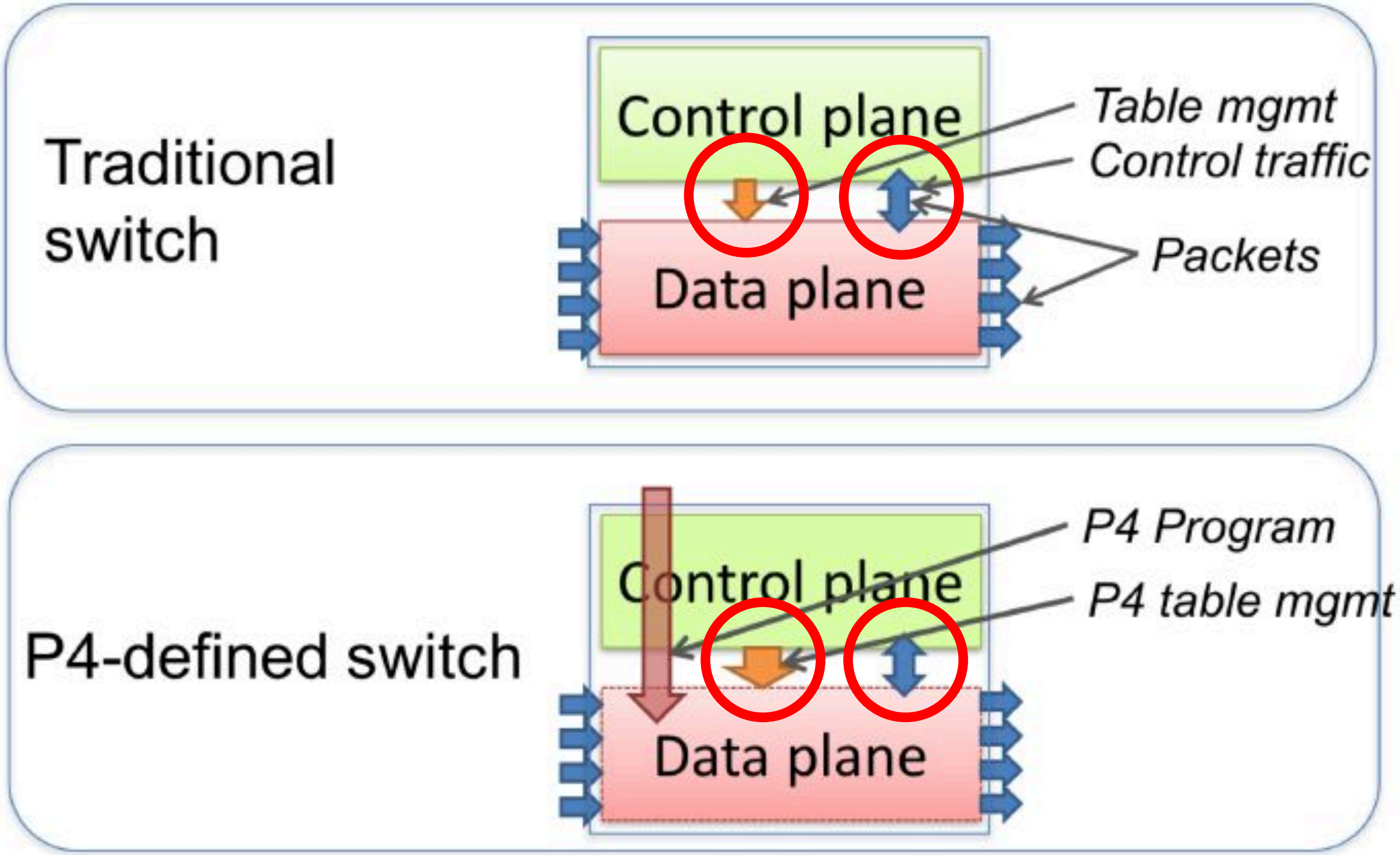
# Programmable pipeline in software: Linux eBPF



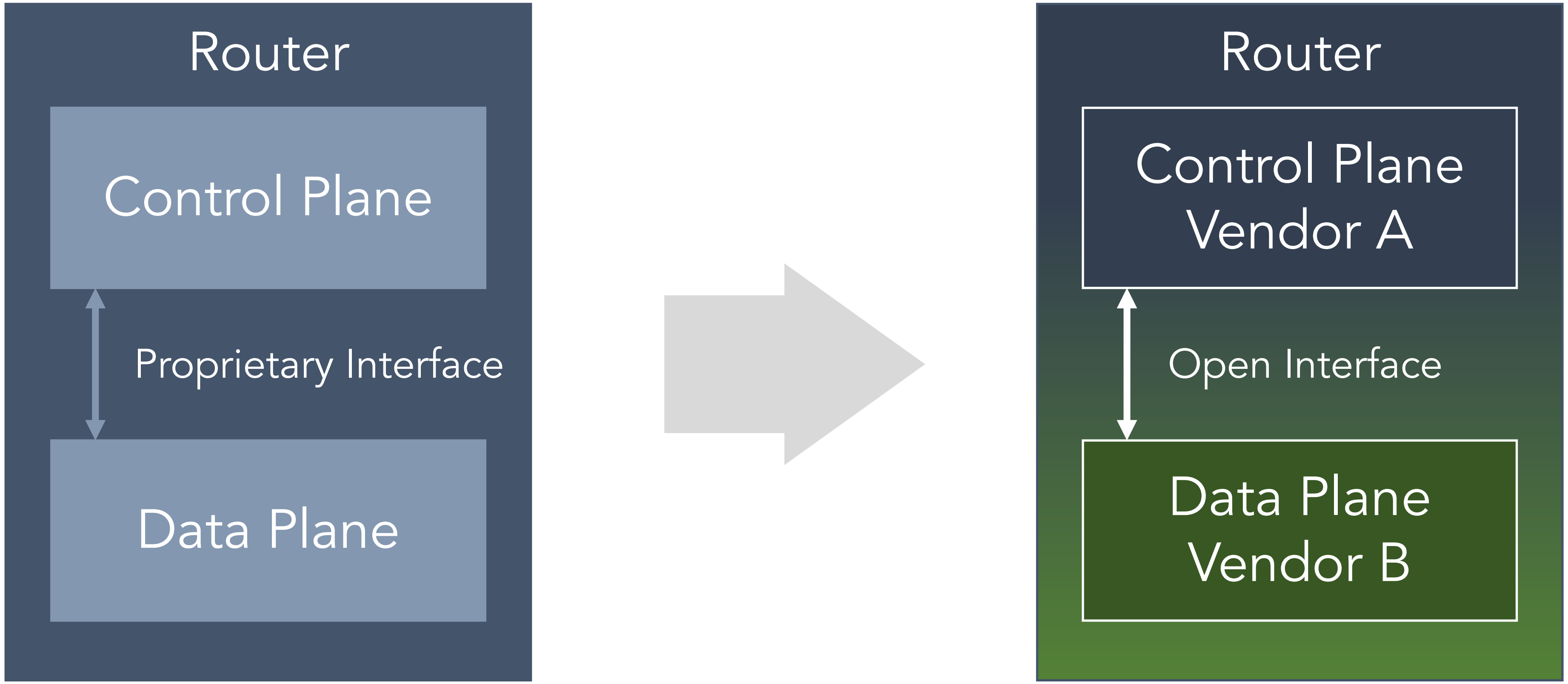
Section 4

# Control plane - data data interface

# What interface are we talking about?



# Disaggregation and open interfaces




Monolithic closed single-vendor router

Disaggregated open multi-vendor router

# “Standard” control plane - data plane interfaces

OpenFlow

CUMULUS

NETLINK

sai

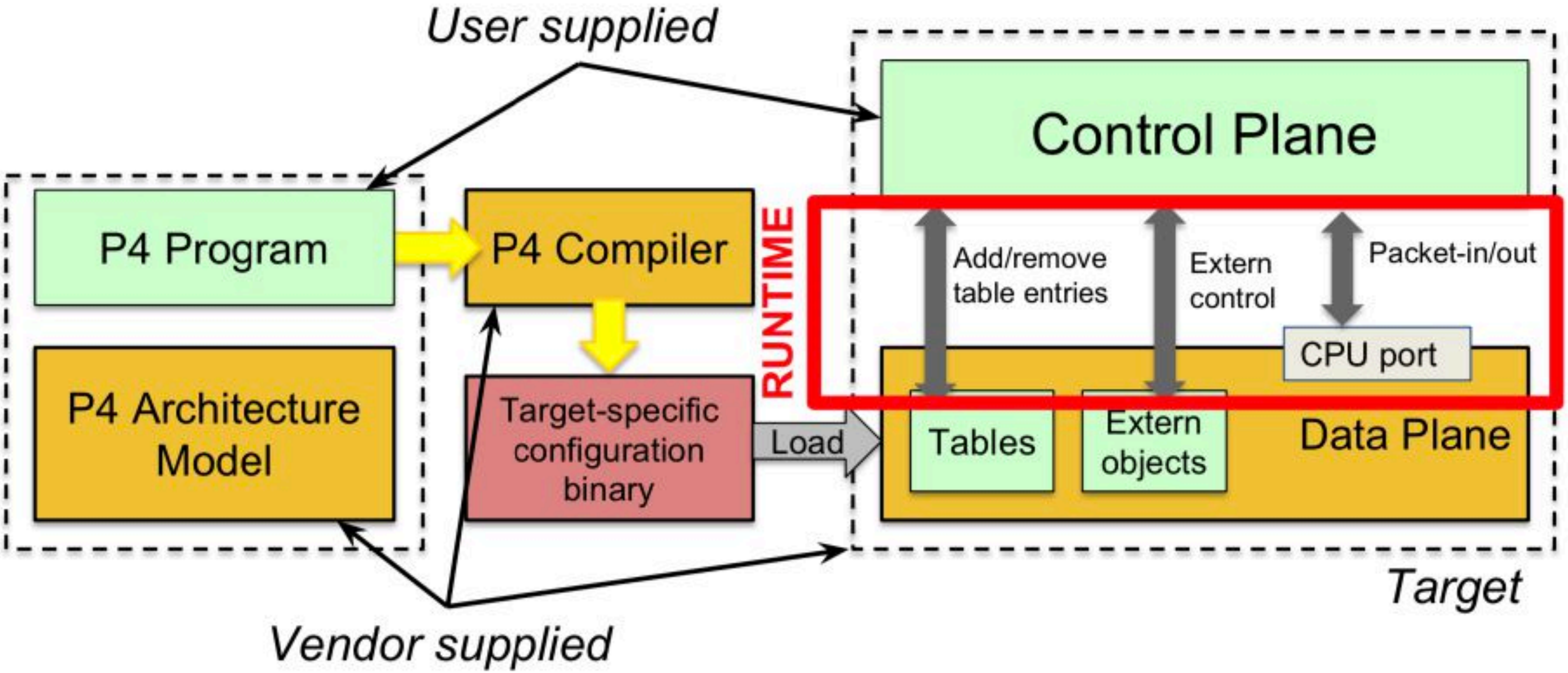
P4

IETF

ForCES

SONiC

# P4 runtime project

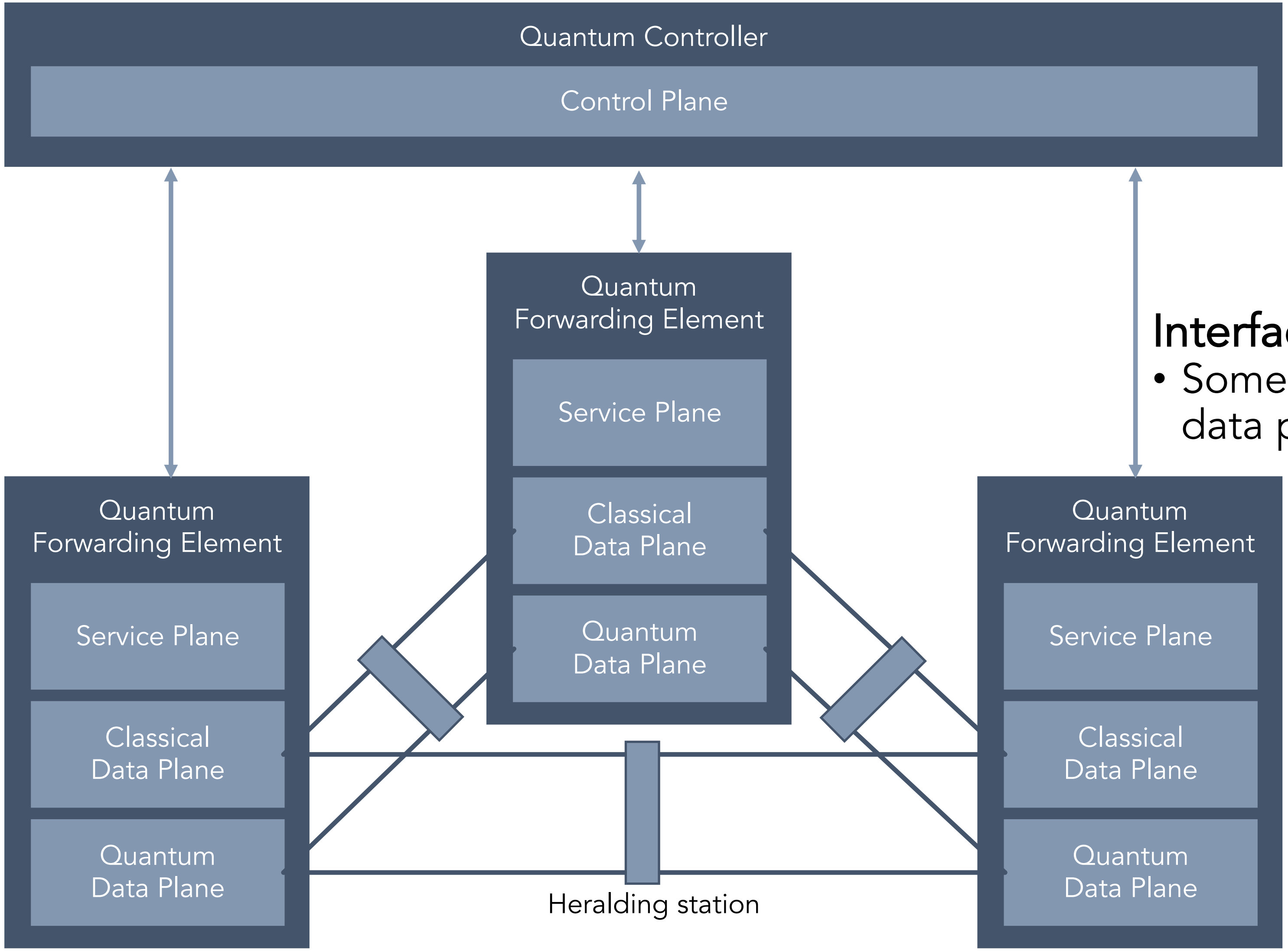


Section 5

# Programmable quantum data plane

# Quantum planes (centralized control plane)

## Helpful to think of RWA in OTN networks as an analogy




**Aware of end-to-end network service**

- Aware of full traffic demand matrix (rate)
- Aware of required fidelity
- Aware of technology model
- Computes optimal paths and schedules
- Translates to network primitives

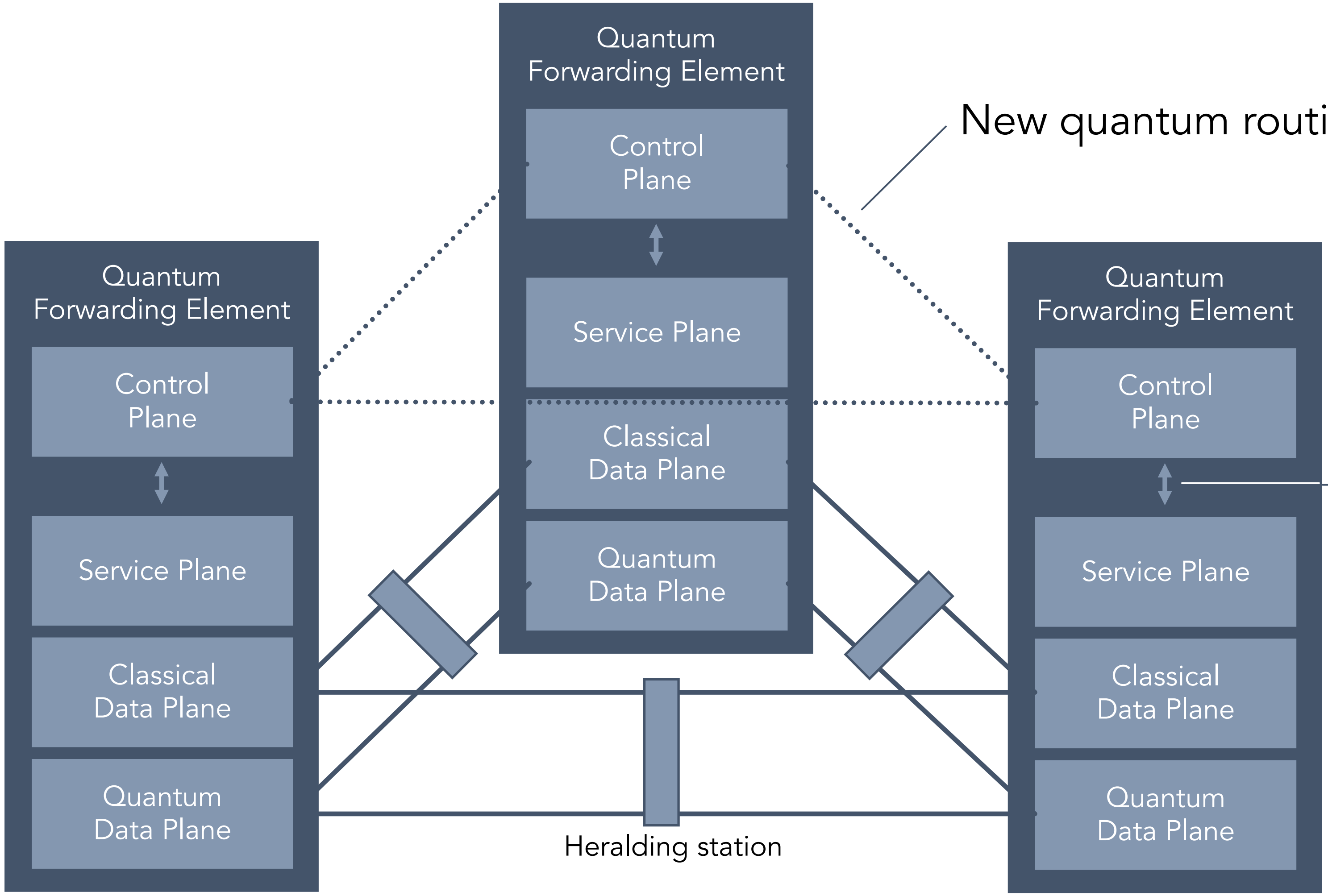
**Interface**

- Some protocol to program primitives into data plane

**Implements local behavior**

- Expressed in simple primitives
- Two data planes: classical and quantum
- Data plane will probably be modeled using state machines and scheduled timeslots to drive entanglement attempts and local operations (e.g. swap attempts)
- Something akin to "service plane" probably needed to implement stateful datalink protocols

# Quantum planes (distributed control plane)




New quantum routing and signaling protocols

Ideally, same interface that is used by the centralized control plane protocol

# Key questions for the quantum data plane

- What are the right abstractions for the quantum data plane?
  - Abstract enough to abstract away the details of the underlying technology
  - Rich enough to expose all capabilities of the underlying technology
- What service primitives does the quantum data plane offer?
  - Generate point-to-point entangled Bell Pairs, swaps, error correction, distillation, resource discovery and management (e.g. number of memory qubits and characteristics)
- What is the interface between the quantum control plane and the quantum data plane?
- How do we make the quantum data plane programmable?
- Can we re-use / generalize existing mechanisms (e.g. P4) for the quantum data plane API / programming?

# Some additional interesting observations

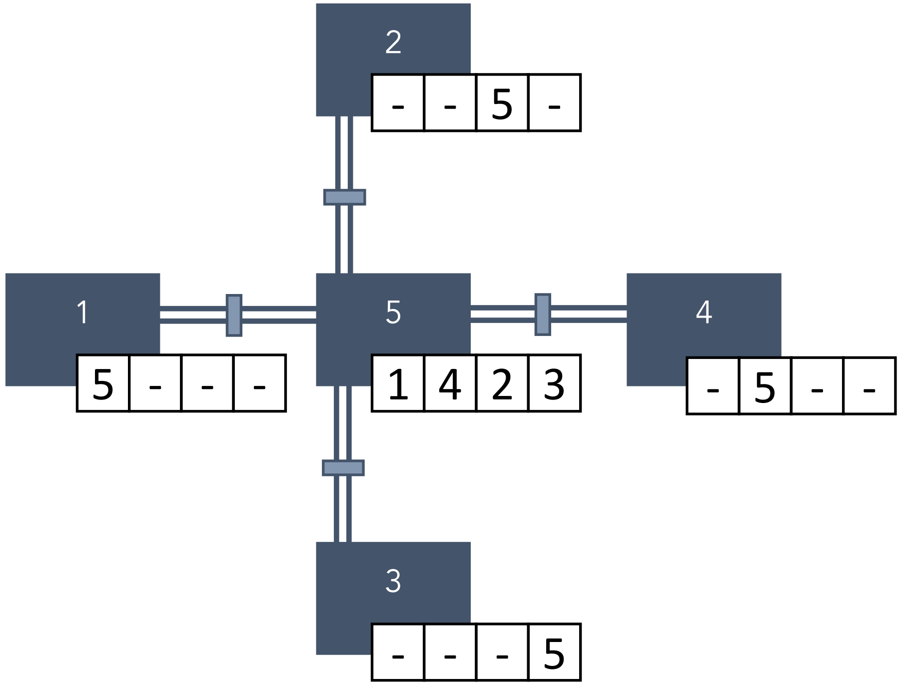
- Qubits do not have headers nor payload
  - Protocol "headers" must be in associated classical messages
  - But received qubits do have implicit meta-data: incoming port and timeslot
- Qubits can only be stored for a very short time (decoherence)
  - The network layer protocol needs to optimize for using entanglement as soon as possible
  - Any involvement of the control plane is out of the question (local and datalink operations only)
- There are many widely diverging candidate technologies for the quantum data plane
  - Need a very strong abstraction layer for the data plane
- Quantum data plane protocols are very stateful (as opposed to stateless IP/Ethernet)
  - Match-action tables and registers are probably too simple
  - Probably need an NPU akin to the service layer to process datalink protocols. "Punt to service"
- The control plane must be aware of the distributed nature of Bell Pairs and the physical model of the technology: noise models, loss models, etc.
  - But the data plane can probably get away without know any of this (analogy with OTN)
- The quantum data plane is inherently uni-directional
  - Quantum repeaters only send qubits.
  - Heralding stations only receive qubits.
  - Entanglement success is reported in a classical message.

# Primitive operations in the quantum data plane

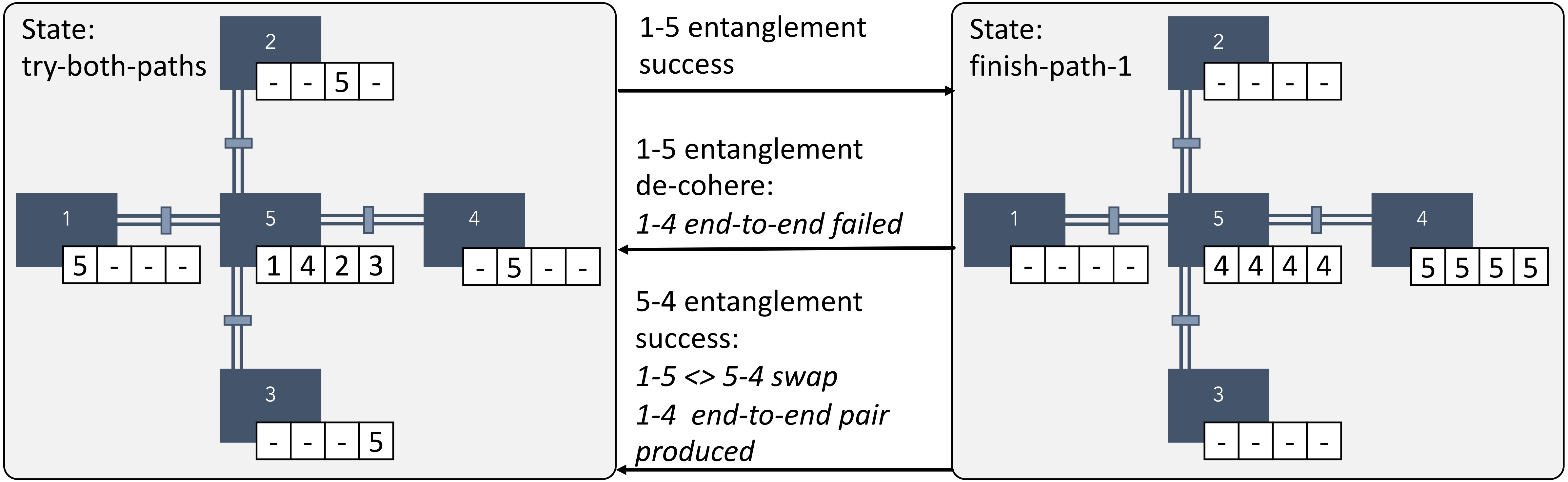
- Classical data plane
  - Re-use existing classical architecture (parser, match-action tables, de-parser)
  - Probably need an NPU to implement datalink protocols (akin to service plane)
- Quantum data plane
  - Primitives to send and receive qubits for local entanglement attempts
  - Primitives for local operations
  - What is the right level of abstraction for local operations?
    - Unitary gates?
    - Or higher level abstractions such as swap attempts, distillation attempts, etc.
- Where on the spectrum between controllable and automated are the quantum nodes?
  - Routers vs repeaters vs heralding stations?

Some initial thoughts on a possible abstraction

# Timeslot schedules (similar to GMPLS + TDM)



# A possible optimization to minimize lost opportunities due to decoherence
# Timeslot schedules and state machines



# P4 seems promising for quantum data plane

- P4 is is extremely flexible extensible
  - We probably can use P4 for the quantum data plane with no or little language syntax changes
- Use concept of ”architecture packages” to represent unique characteristics of quantum router hardware pipeline
  - “Programmable Quantum Switch Architecture” equivalent of PSA
- Concept of “externals” to represent unique quantum hardware components
  - Swap attempts, distillation attempts, move to/from storage qubit, decoherence events, possibly individual gates, …
- Represent received qubit as empty packet with only implicit meta-data (port and timeslot)

Thank you.