Accurate high-bandwidth flow measurements using P4

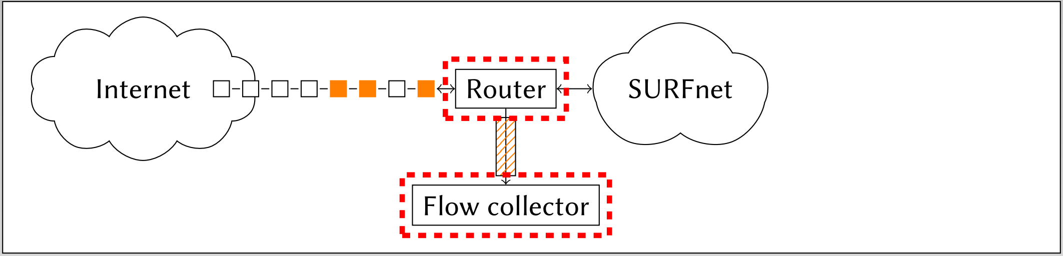
RoN++, SURFnet, the Netherlands
January 8, 2020

Luuk Hendriks
Jeroen van Ingen Schenau

DACS
Design and Analysis of Communication Systems

UNIVERSITY OF TWENTE.

Flow measurements?



Challenges in flow measurements, today

Flow measurements from high-end (high-bandwidth) devices are

- opaque: we can not look into the aggregation process;
- sampled: causing inaccuracies, and the sampling algorithms themselves are not always disclosed;
- static, inflexible: we can not configure custom flow keys.
- expensive: pricy modules / line-cards

Flow measurements tomorrow?

Can we use P4 to improve on that status quo?

- ✓ P4[1] promises flexibility at line-rate performance
- • P4 is about forwarding packets, not about measurements per se
- ? Flow measurements require state, and state is difficult when doing line-rate processing at 10, 40, 100Gbps and beyond

Two ways of keeping state in P4: **tables**, and **registers**.

[1]P4 is a new technology/paradigm to program the dataplane

```
table flows_v6 {
    reads {
        ipv6.srcAddr:   exact;
        ipv6.dstAddr:   exact;
        ipv6.flowLabel: exact;
    }
    actions {
        flow_miss_v6;
        process_packet;
    }
default_action: flow_miss_v6();
support_timeout: true;
}

action process_packet() {
    // ...
}


counter flow_stats_v6 {
    type: packets_and_bytes;
    direct: flows_v6;
}


field_list flow_key_info {
    ipv6.srcAddr;
    ipv6.dstAddr;
    ipv6.flowLabel;
    meta.frame_size;
}

action flow_miss_v6() {
    generate_digest(0, flow_key_info);
}
```

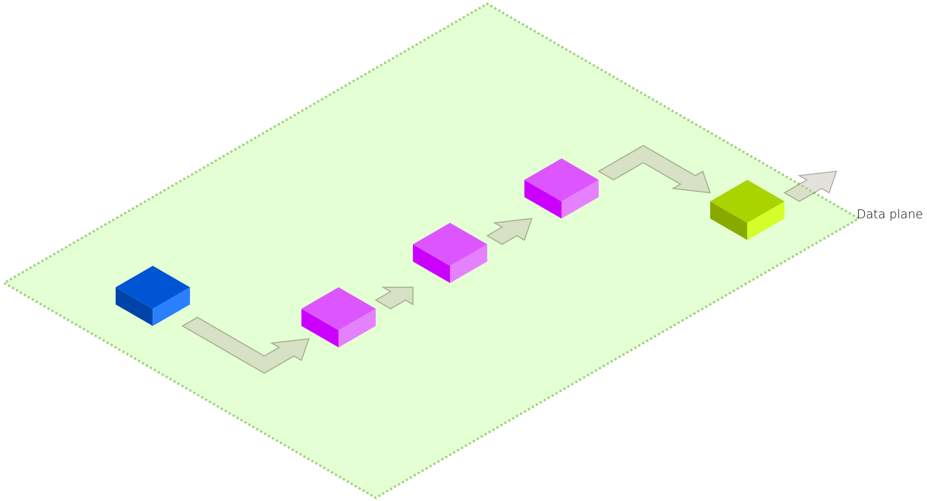
Tables

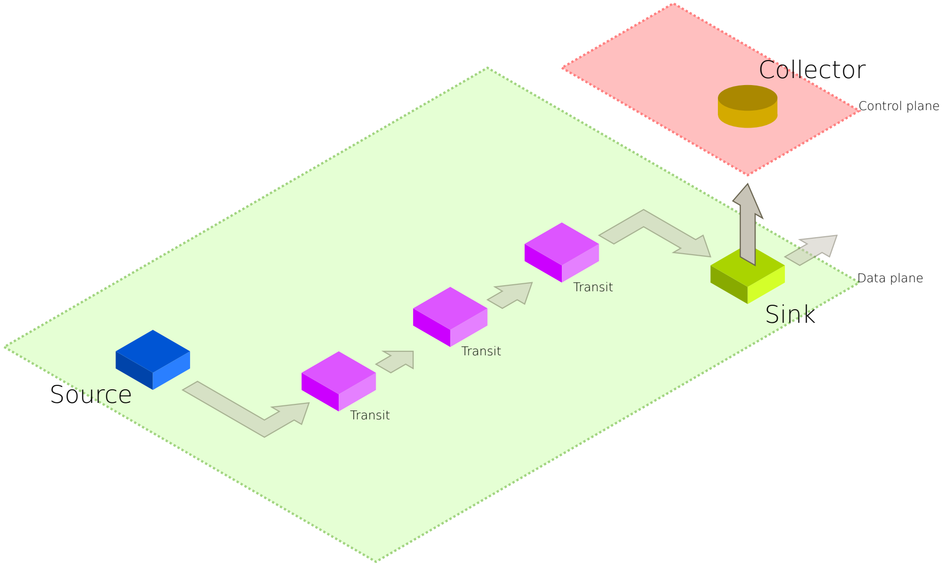
Match-Action Tables:

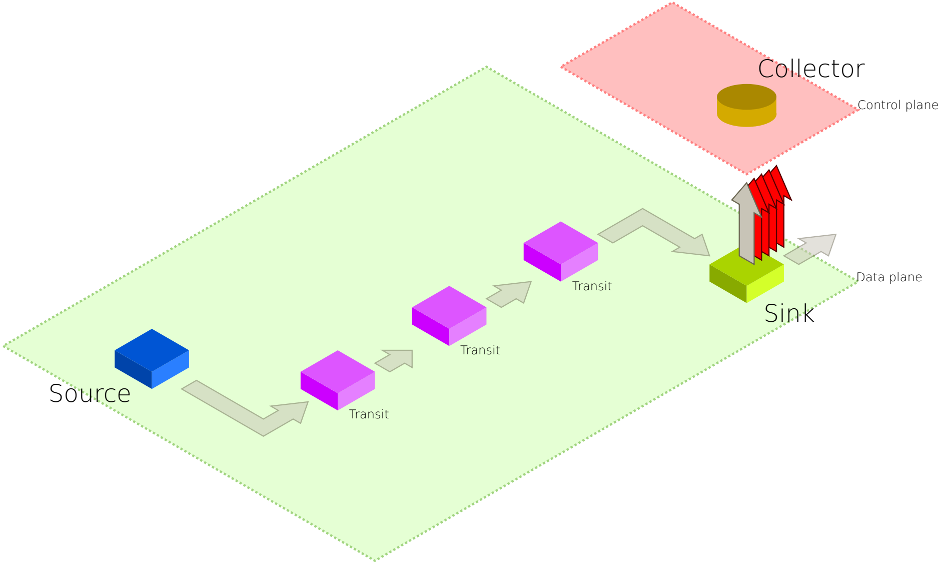
- are essential building blocks in P4 programs (e.g. a forwarding table)
- specify for each *match* which *action* should be performed
- can have counters attached to them, for packets and/or bytes

? but how do we fill these tables?

→ by *learning*, i.e. punting info to the control plane, **for every newly observed flow**


Data plane


Collector
Control plane
Data plane
Transit
Transit
Transit
Sink
Source


Collector
Control plane
Data plane
Transit
Transit
Transit
Sink
Source

While keeping track of flow statistics using Match Action tables is easy and comes, almost for free, out-of-the-box with P4, it does not scale.[2]

We just recreated the problem many devices suffer from: requiring the slow path to do measurements.

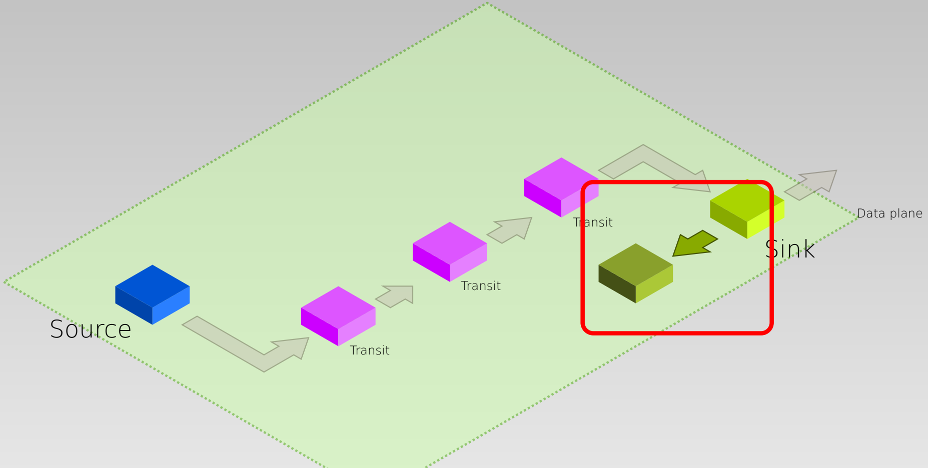
[2]there are other use cases where this approach is perfectly applicable, e.g. where the flow keys are known a priori

What about registers?

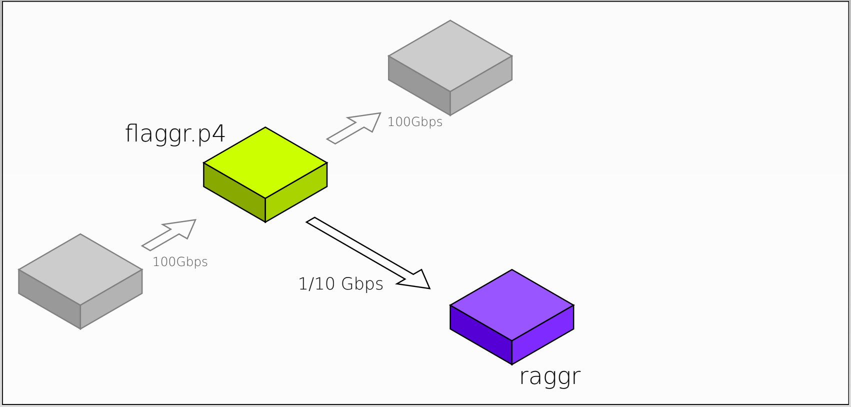
Registers allow us to keep state in the data plane.

Both reading and writing requires no interaction with the control plane.

It’s all in the data plane



Overview of topology



P4 as part of the solution

Aim:

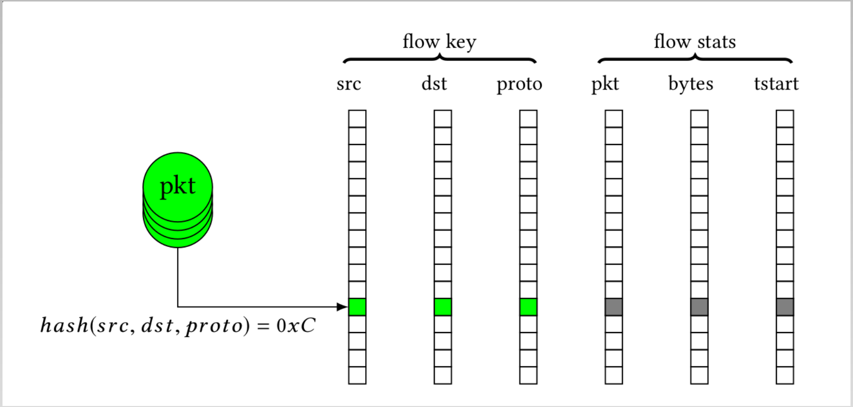
- Use P4 as a pre filtering/aggregation step. (flaggr)
- Then, let an external machine take care of the final aggregation and storage. (raggr)

Benefits:

- flexibility and power of a modern high-end x86 CPU
- because of the pre-aggregation, a smaller link between the switch and the external machine suffices.

flaggr + raggr

Collisions? Collisions!



What happens if a *different* flow hashes to `0xC` as well?

pseudo flaggr

```
// Flow key registers
reg_src_ip       = Register();
reg_dst_ip       = Register();
reg_proto        = Register();
reg_l4           = Register();

// Flow statistics registers
reg_pkt_count    = Register();
reg_byte_count   = Register();
reg_time_start   = Register();
reg_time_end     = Register();
reg_flags        = Register();

initialize_registers(hdr: PacketHeader, index:
     HashIndex, md: Metadata):
    reg_src_ip[index]         = hdr.src_ip;
    reg_dst_ip[index]         = hdr.dst_ip;
    reg_proto[index]          = hdr.proto;
    reg_l4[index]             = hdr.l4;
    reg_pkt_count[index]      = 1;
    reg_byte_count[index]     = length(hdr.ethernet
    ) + hdr.ip_len
    reg_time_start[index]     = md.timestamp;
    reg_time_end[index]       = md.timestamp;
    reg_flags[index]          = hdr.tcp_flags;
```

```
with pkt = ingress.next_packet():
    hdr = parse(pkt);
    md  = pkt.metadata;
    index = hash({hdr.src_ip, hdr.dst_ip, hdr.proto, hdr.
     l4});
    collision =
            hdr.src_ip  != reg_src_ip[index]
        ||  hdr.dst_ip  != reg_dst_ip[index]
        ||  hdr.proto   != reg_proto[index]
        ||  hdr.l4      != reg_l4[index]

    if collision:
        // Export info and keep track of new flow
        flow_record = { reg_src_ip[index],
                        reg_dst_ip[index],
                        reg_proto[index],
                        reg_l4[index],
                        reg_pkt_count[index],
                        reg_byte_count[index],
                        reg_time_start[index],
                        reg_time_end[index],
                        reg_flags[index] }
        emit({hdr.ethernet, flow_record});
        initialize_registers(hdr, index, md);
    else:
        // Update statistics of current flow
        reg_pkt_count[index]     +=  1;
        reg_byte_count[index]    +=  length(hdr.ethernet)
  + hdr.ip_len
        reg_time_end[index]      =   md.timestamp;
        reg_flags[index]         ||= hdr.tcp_flags;
```

That can’t be hard in P4!

Every piece of information stored requires a register. A register can only be accessed (read and/or written to) **ONCE** per packet.

In order to determine how we should update the statistics registers (packet counter etc.), we first need to find out whether a collision occured in the key registers (src ip etc.).

In other words, the order accessing the registers is crucial.

Registers in flaggr

The actual code, as opposed to the pseudo code, is comprised of many different controls. Each control manages at least one register:

Statistics controls:

- PacketCount
- ByteCount
- FlowTimes (2 registers!)
- TcpFlags

control PacketCount

```
control PacketCount(
    inout metadata_t md
    ){

    Register<bit<32>, bit<HASH_WIDTH>>(1 << HASH_WIDTH)
      flow_cache_packets;


    RegisterAction<bit<32>, bit<HASH_WIDTH>, bit<32>>(
      flow_cache_packets) fc_packets_reset = {
        void apply(inout bit<32> current, out bit<32>
      read_packets) {
                  read_packets = current;
                  current = 32w1;
        }
    };


    RegisterAction<bit<32>, bit<HASH_WIDTH>, bit<32>>(
      flow_cache_packets) fc_packets_increase = {
        void apply(inout bit<32> current, out bit<32>
      read_packets) {
                  read_packets = current;
                  current = current + 32w1;
        }
    };


    apply {
        if (md.clash == 1) {
            md.fc_pkts = fc_packets_reset.execute(md.
      hash_idx);
        } else {
            md.fc_pkts = fc_packets_increase.execute(md.
      hash_idx);
        }
    }
```

- operates based on a hash of the flow key
- has two `RegisterActions`
  - one to reset the counter to 1 (new flow)
  - one to increase the counter with 1

→ has an `apply` to pick one of these actions, based on whether a collision has been observed

? why no `if` in one single `RegisterAction` ?

raggr

- receives partial flow records
- aggregates the partials
- writes the full flow information to disk or UNIX pipe (currently, .csv)
- tells us about the reduction rate in terms of packets, bytes, number of partials, etc.

Evaluation: completeness

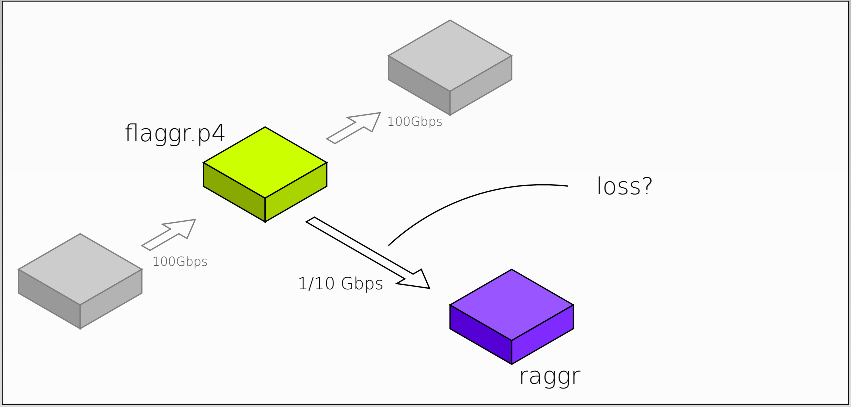
We now have a working P4-based exporter, exporting (partial) flow records based on hash collisions, and a collector performing the final aggregation.

Evaluation time!

Method:

1. Generate 100k flows (`flowgenpp`), our ground truth
2. `tcpreplay` it through the switch
3. Compare resulting .csv to ground truth:
    - ensure ALL flows from ground truth are in the measured flows
    - ensure NO other flows are ‘observed’

Problem: Did we get all flo-



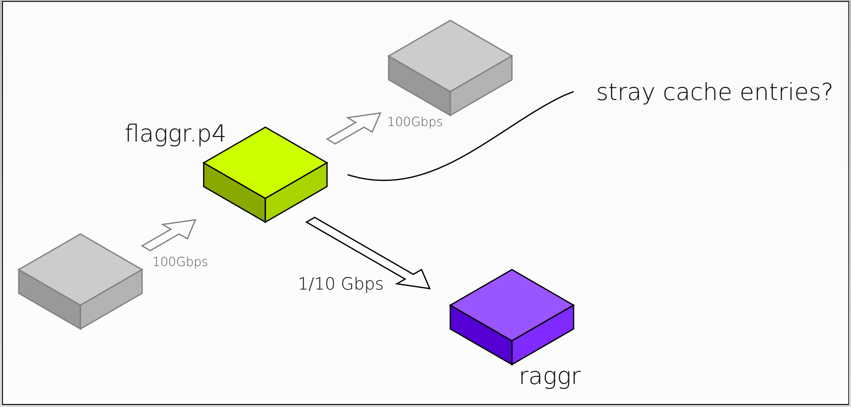
```
control Serial(
    inout metadata_t md
    ){
        Register<bit<64>, bit<1>>(1) flow_serial;
        RegisterAction<bit<64>, bit<1>, bit<64>>(
      flow_serial) fc_serial_update = {
            void apply(inout bit<64> current, out
       bit<64> read_serial) {
                    read_serial = current;
                    current = current + 1;
            }
        };

    apply {
        md.serial = fc_serial_update.execute(
      SERIAL_REG_INDEX);
    }
}
```

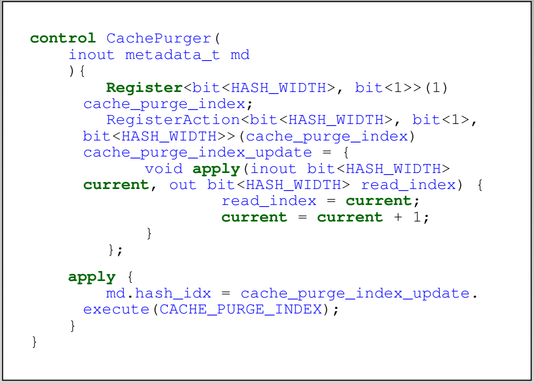
Solution: Serial numbers!

- Well-known concept in existing flow setups
- Attach a serial number to each flow record
- Collector can signal losses

→ In flaggr, we use a 64bit serial, incremented on every export

Problem: No collision?





```
control CachePurger(
    inout metadata_t md
    ){
        Register<bit<HASH_WIDTH>, bit<1>>(1)
     cache_purge_index;
        RegisterAction<bit<HASH_WIDTH>, bit<1>,
     bit<HASH_WIDTH>>(cache_purge_index)
     cache_purge_index_update = {
            void apply(inout bit<HASH_WIDTH>
     current, out bit<HASH_WIDTH> read_index) {
                    read_index = current;
                    current = current + 1;
            }
        };

    apply {
        md.hash_idx = cache_purge_index_update.
     execute(CACHE_PURGE_INDEX);
    }
}
```

Solution: send magic packets!

- The switch can only act upon an incoming packet
- Force an export by sending a magic packet
- Cache is purged, one by one, sequentially

→ *raggr* sends out these probes (EtherType `0xBEEF`)

! Note that, by configuring the interval of these probes
this can function as a poor man's *active timeout*

Evaluation: completeness

We see all the flows and nothing but the flows from the ground truth!

Statistics controls:

- PacketCount
- ByteCount
- FlowTimes (2 registers!)
- TcpFlags

Helper controls:

- Serial
- CachePurger

Evaluation: correctness

Are all statistics (packet/byte count, TCP flags) correct?
Some byte counters off by 131072, some by 262144, some by 196608 ...

```
header flow_info_h {
    bit<64> serial;
    flow_key_t  flow_key;
    bit<16> bytes; // TODO is 16 bits enough????
    (...)
}
```

Easy fix, right?

Lesson learned: control logic (such as `if`) in `RegisterActions` is expensive, and hard on the compiler!

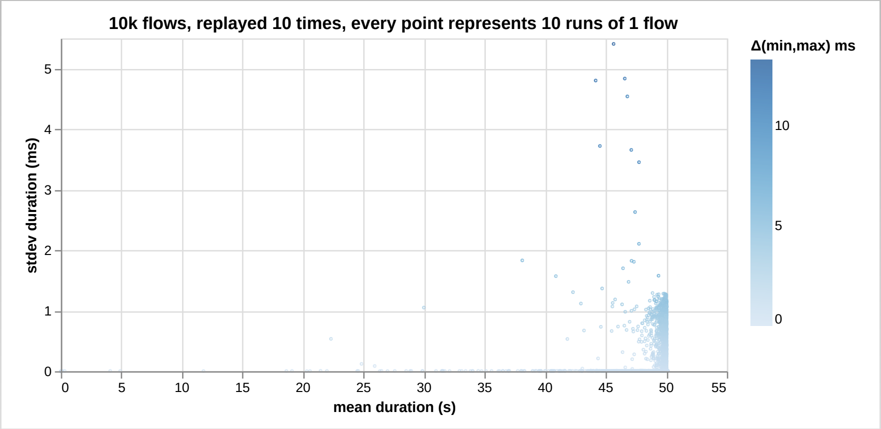
Evaluation: Flow duration precision

Method:

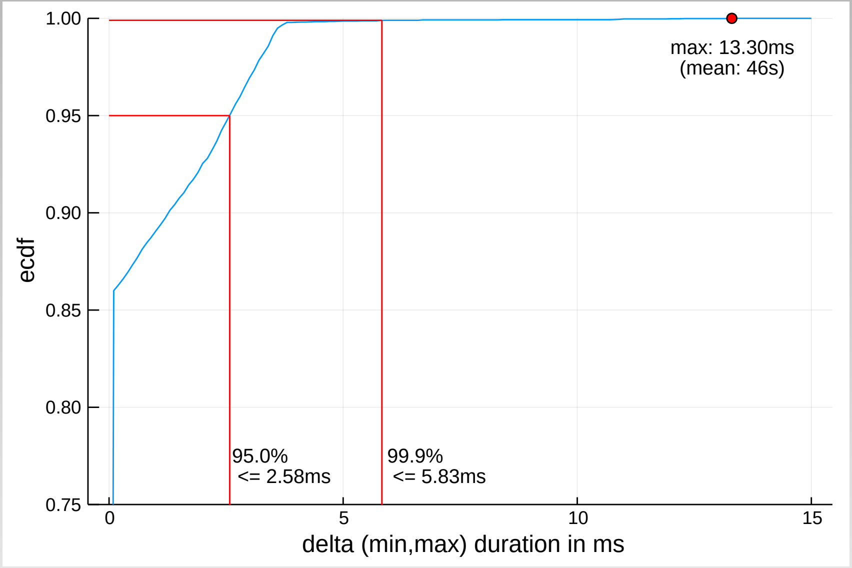
- Generate 10k flows
- Send through the switch 10 times using `tcpreplay`
- Analyze the $t_{end} - t_{start} = t_{duration}$ per flow, for all runs

! Replay times of all 10 runs were within 10∼20ms of eachother (as reported by `tcpreplay`)

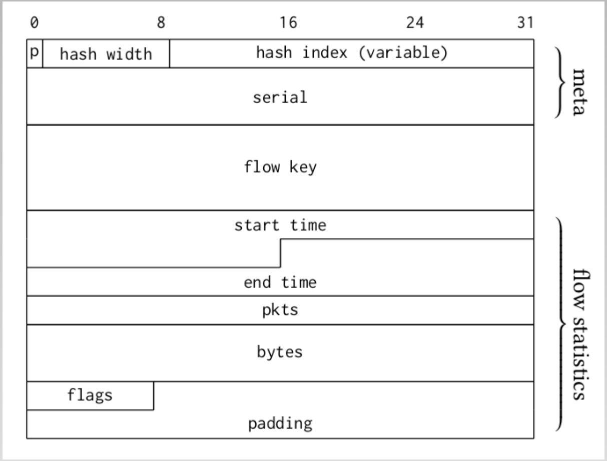
Flow duration precision



Flow duration precision



Wireformat flaggr → raggr



p = purge bit, padding is to get at least 64byte ethernet frames

Meta information can be used to analyze the nature of traffic on your network, and fine-tune your flow measurement setup.

Concluding

We can leverage P4 to realise more open, transparent flow measurements that are unsampled and accurate, on high speed links.
Much more to discover:

- At which speeds does `raggr` start to choke ...
- ... and can we leverage e.g. eBPF (offloading) to support `raggr`?
- Can we do these measurements for IPv6 (Tofino2) ?
- How can we do absolute timestamps instead of relative ones?

Next up:

- Get this setup published
- Release `flaggr`, `raggr`, and `flowgenpp` code
- Analyze nature of campus traffic (another paper)

Accurate high-bandwidth flow measurements using P4

RoN++, SURFnet, the Netherlands
January 8, 2020

Luuk Hendriks, luuk.hendriks@utwente.nl