Advanced Topics in Communication Networks
Programming Network Data Planes

Laurent Vanbever
nsg.ee.ethz.ch

ETH Zürich
Oct 4 2018

Materials inspired from p4.org
Last week on

Advanced Topics in Communication Networks
We will start diving into the P4 ecosystem and look at our first practical usage.

- **P4 environment**
  - What is needed to program in P4?

- **P4 language**
  - Deeper-dive into the language constructs

- **P4 in practice**
  - In-network obfuscation

[USENIX Sec'18]
What is needed to program in P4?

- P4 environment
- P4 language
- P4 in practice
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
We'll rely on a simple P4\textsubscript{16} switch architecture (v1model) which is roughly equivalent to "PISA"

source: https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf
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\textsubscript{16} should be target-independent. In P4\textsubscript{14} almost 1/3 of the constructs were target-dependent.

Think of externs as Java interfaces only the signature is known, not the implementation.
≠ architectures → ≠ metadata & ≠ externs

NetFPGA-SUME


more info
Deeper dive into
the language constructs (*)

(*) full info https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html
P4_{16} 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
- `float`: not supported
- `string`: not supported
P4_{16} is a statically-typed language with base types and **operators to derive composed ones**

```
header Ethernet_h {
  bit<48> dstAddr;
  bit<48> srcAddr;
  bit<16> etherType;
}

header Mpls_h {
  bit<20> label;
  bit<3>  tc;
  bit     bos;
  bit<8>  ttl;
}

Mpls_h[10] mpls;
```

Array of up to 10 MPLS headers

```
header_union IP_h {
  IPV4_h v4;
  IPV6_h v6;
}
```

Either IPv4 or IPv6 header is present

only one alternative
P4$_{16}$ is a statically-typed language with base types and operators to derive composed ones

**Struct**
Unordered collection of named members

```p4
struct standard_metadata_t {
    bit<9> ingress_port;
    bit<9> egress_spec;
    bit<9> egress_port;
    ...
}
```

**Tuple**
Unordered collection of unnamed members

```p4
tuple<bit<32>, bool> x;
x = { 10, false };
```
P4 operations are similar to C operations and vary depending on the types (unsigned/signed ints, ...)

- arithmetic operations: +, −, *
- logical operations: ~, &, |, ^, >>, <<
- non-standard operations: [m:l] Bit-slicing
  ++ Bit concatenation

✗ no division and modulo (can be approximated)

more info: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html
Variables have local scope and their values is not maintained across subsequent invocations.

Variables **cannot** be used to maintain state between different network packets.

**important**

Instead to maintain state, you can only use two stateful constructs:

- tables
- extern objects

More on this next week.
This week on

Advanced Topics in Communication Networks
stateful programming
statefulness in practice
probabilistic data structures

How do you build stateful apps?
fast network convergence
bloom filters
part 1
How do you build stateful apps?

fast network convergence

bloom filters

part 1
Stateless and stateful objects in P4

Stateless objects
Reinitialized for each packet

Variables
Headers

Stateful objects
Keep state between packets

Tables
Registers
Counters
Meters
...
Stateless and stateful objects in P4

Stateless objects
Reinitialized for each packet

Stateful objects
Keep state between packets

- Variables
- Headers

- Tables
- Registers
- Counters
- Meters
  ...

Stateful objects in P4

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>managed by the control plane</td>
</tr>
<tr>
<td>Register</td>
<td>store arbitrary data</td>
</tr>
<tr>
<td>Counter</td>
<td>count events</td>
</tr>
<tr>
<td>Meter</td>
<td>rate-lmiting</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Stateful objects in P4

- Table: managed by the control plane
- Register: store arbitrary data
- Counter: count events
- Meter: rate-limiting
- ...
Registers are useful for storing (small amounts of) arbitrary data

```
<Type>
```

```
r.write(…)
r.read(…)
```
Registers are assigned in arrays

```
register<Type>(N) r;

r.write(0, value)

r.read(result, 0)
```
Example: Calculating inter packet gap

```c
register<bit<48>>(16384) last_seen;

action get_inter_packet_gap(out bit<48> interval, bit<32> flow_id) {
    bit<48> last_pkt_ts;

    /* Get the time the previous packet was seen */
    last_seen.read(last_pkt_ts, flow_id);

    /* Calculate the time interval */
    interval = standard_metadata.ingress_global_timestamp - last_pkt_ts;

    /* Update the register with the new timestamp */
    last_seen.write(flow_id, standard_metadata.ingress_global_timestamp);

    ...
}
```
Example: Stateful firewall

Only allow incoming packets if they belong to an established connection
Example: Stateful firewall

TCP Packet

- from internal? (yes/no)
  - yes
    - SYN flag? (yes/no)
      - yes: add flow to register
      - no: flow in register? (yes/no)
        - yes: forward
        - no: drop
  - no: drop
Example: Stateful firewall

control MyIngress(...) {
    register<bit<1>>(4096) known_flows;
    ...
    apply {
        meta.flow_id = ... // hash(5-tuple)
        if (hdr.ipv4.isValid()){
            if (hdr.tcp.isValid()){
                if (standard_metadata.ingress_port == 1){
                    if (hdr.tcp.syn == 1){
                        known_flows.write(meta.flow_id, 1);
                    }
                }
            }
            if (standard_metadata.ingress_port == 2){
                known_flows.read(meta.flow_is_known, meta.flow_id);
                if (meta.flow_is_known != 1){
                    drop(); return;
                }
            }
        }
        ipv4_lpm.apply();
    }
}

Stateful objects in P4

- **Table**: managed by the control plane
- **Register**: store arbitrary data
- **Counter**: count events
- **Meter**: rate-limiting
- **...**: ...

externs in v1model
Counters are useful for... counting

```
c.count(...)  # Only from the control plane
```

Diagram:
- Type
- `c.count(...)`
- `c.read(...)`
- "only from the control plane"
Counters can be of three different types

- `Type`
- `c.count(...)`
- `packets`
- `bytes`
- `packets_and_bytes`
Like registers, counters are assigned in arrays

counter(N,Type) c;

c.count(0)
Example: Counting packets and bytes arriving at each port

```pseudocode
control MyIngress(...) {
    counter(512, CounterType.packets_and_bytes) port_counter;
    apply {
        port_counter.count((bit<32>)standard_metadata.ingress_port);
    }
}
```

use the ingress port as counter index
Example: Reading the counter values from the control plane

```
control MyIngress(...) {
    counter(512, CounterType.packets_and_bytes) port_counter;
    apply {
        port_counter.count((bit<32>)standard_metadata.ingress_port);
    }
}
```

Control Plane

RuntimeCmd: counter_read MyIngress.port_counter 1
MyIngress.port_counter[1] = BmCounterValue(packets=13, bytes=1150)

use the ingress port as counter index
Direct counters are a special kind of counters that are attached to tables.

<table>
<thead>
<tr>
<th>Match Key</th>
<th>Action ID</th>
<th>Data</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each entry has a counter cell that counts when the entry matches.
Example: Counting packets and bytes arriving at each port *using a direct counter*

```java
control MyIngress(...) {

direct_counter(CounterType.packets_and_bytes) direct_port_counter;

table count_table {
    key = {
        standard_metadata.ingress_port: exact;
    }
    actions = {
        NoAction;
    }
    default_action = NoAction;
    counters = direct_port_counter;
    size = 512;
}

apply {
    count_table.apply();
}
}
```
Stateful objects in P4

- **Table**: managed by the control plane
- **Register**: store arbitrary data
- **Counter**: count events
- **Meter**: rate-limiting
- ...
Meters

Stream of packets → Meter → Stream of colored packets
Meters

Stream of packets

Meter

Exceeds the PIR
Does not exceed PIR but exceeds CIR
Does not exceed PIR and CIR

Parameters:
- PIR: Peak Information Rate [bytes/s] or [packets/s]
- CIR: Committed Information Rate [bytes/s] or [packets/s]

more info https://tools.ietf.org/html/rfc2698
Like registers and counters, meters are assigned in arrays.

```
m.execute(0)
meter(N, Type) m;
```
Example: Using a meter for rate-limiting

max. 1 packet/s
Example: Using a meter for rate-liming

IP Packet

Map the sender (source MAC address) to a meter index

<table>
<thead>
<tr>
<th>m_read</th>
<th>11:11:...</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22:22:...</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>33:33:...</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>44:44:...</td>
<td>3</td>
</tr>
</tbody>
</table>

Map the meter tag to a policy

m_filter
- NoAction
- drop
- drop
Example: Using a meter for rate-limiting

```c
control MyIngress(...) {
  meter(32w16384, MeterType.packets) my_meter;

  action m_action(bit<32> meter_index) {
    my_meter.execute_meter(bit<32>>(meter_index, meta.meter_tag);
  }

  table m_read {
    key = { hdr.ethernet.srcAddr: exact; }
    actions = { m_action; NoAction; }
    ...
  }
  table m_filter {
    key = { meta.meter_tag: exact; }
    actions = { drop; NoAction; }
    ...
  }

  apply {
    m_read.apply();
    m_filter.apply();
  }
}
```
Direct meters are a special kind of meters that are attached to tables.

<table>
<thead>
<tr>
<th>Match Key</th>
<th>Action ID</th>
<th>Data</th>
<th>Meter</th>
</tr>
</thead>
</table>

Each entry has a meter cell that is executed when the entry matches.
Example: Using a meter for rate-limiting

control MyIngress(...) {
  direct_meter<bit<32>>(MeterType.packets) my_meter;

  action m_action(bit<32> meter_index) {
    my_meter.read(meta.meter_tag);
  }

  table m_read {
    key = { hdr.ethernet.srcAddr: exact; }
    actions = { m_action; NoAction; }
    meters = my_meter;
    ...
  }
  table m_filter { ...

  apply {
    m_read.apply();
    m_filter.apply();
  }
}
## Summary

<table>
<thead>
<tr>
<th>Object</th>
<th>Data plane interface</th>
<th>Control plane interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>read: apply()</td>
<td>read: yes</td>
</tr>
<tr>
<td></td>
<td>modify/write: —</td>
<td>modify/write: yes</td>
</tr>
<tr>
<td>Register</td>
<td>read: read()</td>
<td>read: yes</td>
</tr>
<tr>
<td></td>
<td>write: write()</td>
<td>modify/write: yes</td>
</tr>
<tr>
<td>Counter</td>
<td>—</td>
<td>reset</td>
</tr>
<tr>
<td>Meter</td>
<td>execute()</td>
<td>configuration only</td>
</tr>
</tbody>
</table>
Stateless and stateful objects in P4

- **Stateless objects**: Reinitialized for each packet
- **Stateful objects**: Keep state between packets

Variables
Headers

Tables
Registers
Counters
Meters
...
How do you build stateful apps?

fast network convergence

bloom filters

part 1
How do you build stateful apps?

fast network convergence

bloom filters

part 1
Programming more advanced stateful data structures
Programming more advanced stateful data structures

We are provided with built-in stateful data structures such as arrays of registers, counters or meters

We need to deal with severe limitations such as a limited number of operations and memory
Programming more advanced stateful data structures

We are provided with built-in stateful data structures such as arrays of registers, counters or meters

We need to deal with severe limitations such as a limited number of operations and memory

**Today**: how can we implement a set with its usual methods i.e., add an element, membership query, delete an element, lookup, listing
There are two common strategies to implement a set

<table>
<thead>
<tr>
<th></th>
<th>strategy #1</th>
<th>strategy #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Deterministic</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>number of required operations</td>
<td>Probabilistic</td>
<td>Deterministic</td>
</tr>
</tbody>
</table>
There are two common strategies to implement a set

<table>
<thead>
<tr>
<th></th>
<th>strategy #1</th>
<th>strategy #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Deterministic</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>number of required operations</td>
<td>Probabilistic</td>
<td>Deterministic</td>
</tr>
</tbody>
</table>
Intuitive implementation of a set
Intuitive implementation of a set
Separate–chaining
Intuitive implementation of a set
Separate-chaining
Intuitive implementation of a set
Separate-chaining

N elements and M cells

<table>
<thead>
<tr>
<th>list size</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>N/M</td>
</tr>
<tr>
<td>worse-case</td>
<td>N</td>
</tr>
</tbody>
</table>
Intuitive implementation of a **set**

Separate–chaining

**Pros:** accurate and fast in the average case

**Con:** only works in hardware if there is a low number of elements (e.g. < 100)
There are two common strategies to implement a set

<table>
<thead>
<tr>
<th></th>
<th>strategy #1</th>
<th>strategy #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Deterministic</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>number of required operations</td>
<td>Probabilistic</td>
<td>Deterministic</td>
</tr>
</tbody>
</table>
A simple approach for **insertions** and **membership queries**
A simple approach for insertions and membership queries

Insert "TCP"

1-bit cells
A simple approach for **insertions** and **membership queries**

**insert "Hello"**

Hello

1-bit cells
A simple approach for **insertions** and **membership queries**

Insert "Fine"
A simple approach for **insertions** and **membership queries**

is "Hello" in the set?

1 = YES

1-bit cells
A simple approach for **insertions** and **membership queries**

Is "Bye" in the set?

Bye

0 = NO

1-bit cells
A simple approach for **insertions** and **membership queries**

**is "P4" in the set?**

Hello

1 = YES

False Positive!
A simple approach for **insertions** and **membership queries**

**N elements and M cells**

- Probability of an element to be mapped into a particular cell: \( \frac{1}{M} \)
- Probability of an element not to be mapped into a particular cell: \( 1 - \frac{1}{M} \)
- Probability of a cell to be 0: \( (1 - \frac{1}{M})^N \)
- False positive rate (FPR): \( 1 - (1 - \frac{1}{M})^N \)
- False negative rate: \( 0 \)
A simple approach for **insertions** and **membership queries**

<table>
<thead>
<tr>
<th># of elements</th>
<th># of cells</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10000</td>
<td>9.5%</td>
</tr>
<tr>
<td>1000</td>
<td>100000</td>
<td>1%</td>
</tr>
</tbody>
</table>
A simple approach for **insertions** and **membership queries**

**Pros:** simple and only one operation per insertion or query

**Con:** roughly 100x more cells are required than the number of element we want to store for a 1% false positive rate
Bloom Filters: a more memory-efficient approach for *insertions* and *membership queries*
Bloom Filters: a more memory-efficient approach for insertions and membership queries

**insert "TCP"**

<table>
<thead>
<tr>
<th>hash #1</th>
<th>hash #2</th>
<th>hash #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>TCP</td>
<td>TCP</td>
</tr>
</tbody>
</table>

1-bit cells

- 0
- 0
- 0
- 0
- 0
- 0
- 0
- 0
- 0
- 0
Bloom Filters: a more memory-efficient approach for **insertions** and **membership queries**

**insert "Hello"**

```
Hello  hash #1

Hello  hash #2

Hello  hash #3
```

```
0
1
1
0
0
0
0
0
1
1
0
0
0
0
0
```

1-bit cells
Bloom Filters: a more memory-efficient approach for insertions and membership queries

**insert "Fine"**

- Hash #1
- Hash #2
- Hash #3

1-bit cells
Bloom Filters: a more memory-efficient approach for insertions and membership queries

An element is considered in the set if all the hash values map to a cell with 1

An element is not in the set if at least one hash value maps to a cell with 0
Bloom Filters: a more memory-efficient approach for **insertions** and **membership queries**

An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0
Bloom Filters: a more memory-efficient approach for **insertions** and **membership queries**

is "Bye" in the set?

An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0
Bloom Filters: a more memory-efficient approach for insertions and membership queries

**False Positive!**

**is "Fire" in the set?**

**Yes**

An element is considered in the set if **all** the hash values map to a cell with 1.

An element is not in the set if **at least** one hash value maps to a cell with 0.

```
1
1
1
1
0
0
0
```

1-bit cells
Bloom Filters: a more memory–efficient approach for insertions and membership queries

N elements, M cells and K hash functions

- Probability of an element to be mapped into a particular cell: \[ \frac{1}{M} \]
- Probability of an element not to be mapped into a particular cell: \[ 1 - \frac{1}{M} \]
- Probability of a cell to be 0: \[ (1 - \frac{1}{M})^{KN} \]
- False positive rate: \[ (1 - (1 - \frac{1}{M})^{KN})^{K} \]
- False negative rate: \[ 0 \]
Bloom Filters: a more memory-efficient approach for **insertions** and **membership queries**

<table>
<thead>
<tr>
<th># of elements</th>
<th># of cells</th>
<th># hash functions</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10000</td>
<td>7</td>
<td>0.82%</td>
</tr>
<tr>
<td>1000</td>
<td>100000</td>
<td>7</td>
<td>≈ 0%</td>
</tr>
</tbody>
</table>
Bloom Filters: a more memory-efficient approach for insertions and membership queries

**Pro**: consumes roughly 10x less memory than the simple approach

**Con**: Requires slightly more operations than the simple approach (7 hashes instead of just 1)
Dimension your Bloom Filter

Memory

Number of operations

Tradeoff

False Positive Rate
Dimension your Bloom Filter

N elements
M cells
K hash functions
FP false positive rate
Dimension your Bloom Filter

N elements
M cells
K hash functions
FP false positive rate

\[ FP = \left(1 - \left(1 - \frac{1}{M}\right)^{KN}\right)^K \approx \left(1 - e^{-KN/M}\right)^K \]

asymptotic approx.

with calculus you can
dimension your bloom filter
Dimension your Bloom Filter

N = 1000
M = 10000
K hash functions
FP false positive rate

False Positive Rate (%)
Dimension your Bloom Filter

N = 1000
M = 10000
K hash functions
FP false positive rate

False Positive Rate (%)

increases the fraction of 0 bits more chance to find a 0 bit for an element not in the set

K (number of hash functions)
Dimension your Bloom Filter

N = 1000
M = 10000
K hash functions
FP false positive rate

K (number of hash functions)
False Positive Rate (%)

there is always a
global minimum when
\[ K = \ln 2 \times \frac{M}{N} \]
found
by taking the derivative
of
\[ \approx \left(1 - e^{-KN/M}\right)^K \]

increases the fraction of 0 bits
more chance to find a 0 bit for an element not in the set
Implementation of a Bloom Filter in P4_{16}

You will have to use hash functions

```p4
v1model

enum HashAlgorithm {
  crc32,
  crc32_custom,
  crc16,
  s,
  random,
  identity,
  csum16,
  xor16
}

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

more info
https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4
Implementation of a Bloom Filter in P4₁₆

You will have to use hash functions, as well as registers

```p4
v1model 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);
}
```

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4
Implementation of a Bloom Filter in P4\textsubscript{16} with 2 hash functions

```c
control MyIngress(...) {
    register register<bit<1>>(NB CELLS) bloom_filter;
}
```
Implementation of a Bloom Filter in P4\textsubscript{16} with 2 hash functions

```
control MyIngress(...) {
    register register<bit<1>> (NB CELLS) bloom_filter;

    apply {
        hash(meta.index1, HashAlgorithm.my_hash1, 0,
             {meta.dstPrefix, packet.ip.srcIP}, NB CELLS);
        hash(meta.index2, HashAlgorithm.my_hash2, 0,
             {meta.dstPrefix, packet.ip.srcIP}, NB CELLS);
    }
```
Implementation of a Bloom Filter in P4_{16} with 2 hash functions

```c
control MyIngress(...) {

    register register<bit<1>>(NB_CELLS) bloom_filter;

    apply {
        hash(meta.index1, HashAlgorithm.my_hash1, 0, {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
        hash(meta.index2, HashAlgorithm.my_hash2, 0, {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

        if (meta.to_insert == 1) {
            bloom_filter.write(meta.index1, 1);
            bloom_filter.write(meta.index2, 1);
        }

        if (meta.to_query == 1) {
            bloom_filter.read(meta.query1, meta.index1);
            bloom_filter.read(meta.query2, meta.index2);

            if (meta.query1 == 0 || meta.query2 == 0) {
                meta.is_stored = 0;
            } else {
                meta.is_stored = 1;
            }
        }
    }
}
```
Implementation of a Bloom Filter in P4 with 2 hash functions

```c
control MyIngress(...) {

    register register<bit<1>>(NB_CELLS) bloom_filter;
    apply {
        hash(meta.index1, HashAlgorithm.my_hash1, 0,
             {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
        hash(meta.index2, HashAlgorithm.my_hash2, 0,
             {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

        if (meta.to_insert == 1) {
            bloom_filter.write(meta.index1, 1);
            bloom_filter.write(meta.index2, 1);
        }

        if (meta.to_query == 1) {
            bloom_filter.read(meta.query1, meta.index1);
            bloom_filter.read(meta.query2, meta.index2);

            if (meta.query1 == 0 || meta.query2 == 0) {
                meta.is_stored = 0;
            } else {
                meta.is_stored = 1;
            }
        }
    }
}
```

Everything in bold red must be adapted for your program.
Depending on the hardware limitations, splitting the bloom filter might be required.

- M cells are split into $M/K$ disjoint groups.
- An element is hashed to $K$ cells, one in each group.
- One hash function per group.
- Same performance, may be easier to implement or parallelize.
Because deletions are not possible, the controller may need to regularly reset the bloom filters.

Resetting a bloom filter takes some time during which it is not usable.

Common trick: use two bloom filters and use one when the controller resets the other one.
So far we have seen how to do insertions and membership queries.

<table>
<thead>
<tr>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of required operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>strategy #1</th>
<th>strategy #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>Probabilistic</td>
<td>Deterministic</td>
</tr>
</tbody>
</table>

Bloom Filters
However Bloom Filters do not handle deletions.
However Bloom Filters do not handle deletions

If deleting an element means resetting 1s to 0s, then deleting "Fine" also deletes "TCP"
But we can easily extend them to handle deletions. This extended version is called a *Counting Bloom Filter*.
But we can easily extend them to handle deletions.
This extended version is called a **Counting Bloom Filter**

To add an element, increment the corresponding counters.
But we can easily extend them to handle deletions. This extended version is called a **Counting Bloom Filter**.

To add an element, increment the corresponding counters.
But we can easily extend them to handle deletions
This extended version is called a **Counting Bloom Filter**

To add an element, increment the corresponding counters
But we can easily extend them to handle deletions
This extended version is called a **Counting Bloom Filter**

To add an element, increment the corresponding counters

To delete an element, decrement the corresponding counters
But we can easily extend them to handle deletions. This extended version is called a **Counting Bloom Filter**.

To add an element, increment the corresponding counters.

To delete an element, decrement the corresponding counters.

All of our prior analysis for standard bloom filters applies to counting bloom filters.
Counting Bloom Filters do handle deletions at the price of using more memory.
Counting Bloom Filters do handle deletions at the price of using more memory.

Counters must be large enough to avoid overflow. If a counter eventually overflows, the filter may yield false negatives.
Counting Bloom Filters do handle deletions at the price of using more memory.

Counters must be large enough to avoid overflow. If a counter eventually overflows, the filter may yield false negatives.

Poisson approximation suggests 4 bits/counter. The average load (i.e., \( \frac{NK}{M} \)) is \( \ln 2 \) assuming \( K = \ln 2 \times \frac{M}{N} \).

With \( N=10000 \) and \( M=80000 \) the probability that some counter overflows if we use \( b \)-bit counters is at most:

\[
M \times Pr(Poisson(\ln 2) \geq 2^b) = 1.78e-11
\]
Add a new element
Implementation of a Counting Bloom Filter in P4

Delete an element

```c
control MyIngress(...) {

    register register<bit<32>>>(NB_CELLS) bloom_filter;

    apply {
        hash(meta.index1, HashAlgorithm.my_hash1, 0,
            {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
        hash(meta.index2, HashAlgorithm.my_hash2, 0,
            {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

        // Delete a element only if it is in the set
        bloom_filter.read(meta.query1, meta.index1);
        bloom_filter.read(meta.query2, meta.index2);

        if (meta.query1 > 0 && meta.query2 > 0) {
            bloom_filter.write(meta.index1, meta.query1 - 1);
            bloom_filter.write(meta.index2, meta.query2 - 1);
        }
    }
}
```
So far we have seen how to do insertions, deletions and membership queries.
Invertible Bloom Lookup Tables (IBLT) stores key–value pairs and allows for lookups and a complete listing.
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.

Each cell contains three fields:

- **count** which counts the number of entries mapped to this cell.
- **keySum** which is the sum of all the keys mapped to this cell.
- **valueSum** which is the sum of all the values mapped to this cell.
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for **lookups** and a complete **listing**

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Invertible Bloom Lookup Tables (IBLT) stores key–value pairs and allows for lookups and a complete listing.

Add a new key–value pair (assuming it is not in the set yet)

For each hash function
hash the key to find the index

Then at this index
increment the count by one
add key to keySum
add value to valueSum
Invertible Bloom Lookup Tables (IBLT) stores key–value pairs and allows for **lookups** and a complete **listing**

**Delete** a key–value pair (assuming it is in the set)

**For each** hash function

- **hash** the key to find the index

Then at this index

- **subtract** one to the count
- **subtract** key to keySum
- **subtract** value to valueSum
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for **lookups** and a complete **listing**.

<table>
<thead>
<tr>
<th></th>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>152</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>152</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>152</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Insert**

key:152 value:3

hash #1: 152

hash #2: 152

hash #3: 152
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.

**Insert**

**Key:** 7  **Value:** 98

```
hash #1
7

hash #2
7

hash #3
7
```

```
<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
<tr>
<td>1</td>
<td>152</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>159</td>
<td>101</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>152</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
</tbody>
</table>
```

...
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.

**Insert**

**key:50 value:45**

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>209</td>
<td>146</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
</tbody>
</table>
Invertible Bloom Lookup Tables (IBLT) stores key–value pairs and allows for lookups and a complete listing

Key–value pair lookup

The value of a key can be found if the key is associated to at least one cell with a count = 1
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for **lookups** and a complete **listing**

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>209</td>
<td>146</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
</tbody>
</table>
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for **lookups** and a complete **listing**.
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.

The value for the key 50 can’t be found.
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.

**Listing the IBLT**

While there is an index for which count = 1
  Find the corresponding key-value pair and return it
Delete the corresponding key-value pair
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.

**Listing the IBLT**

**While** there is an index for which count = 1

**Find** the corresponding key-value pair and return it

**Delete** the corresponding key-value pair

Unless the number of iterations is very low, loops can’t be implemented in hardware

**The listing is done by the controller**
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for **lookups** and a complete **listing**.

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Delete key: 7 value: 98
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In this example, a complete listing is not possible.
In many settings, we can use XORs in place of sums. For example, to avoid overflow issues.
For further information about Bloom Filters, Counting Bloom Filters and IBLT


