Network Telemetry: Sketch, Marple

Qiao Xiang
Computer Science Department
Yale University
204 Watson
Email: qiao.xiang@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/
Network Verification vs. Network Telemetry

- **Verification**: how may you verify whether the behavior of a network (system) is acceptable?

- **Telemetry**: what is happening now in the network?
  - E.g., heavy hitter detection, queueing delay, etc.
Outline

- Network telemetry
  - Roadmap: from low-level telemetry data structure, to high-level telemetry programming
Outline

- Network telemetry
  - Roadmap: from low-level telemetry data structure, to high-level telemetry programming
  - Low-level data structure: sketch
Example

- Heavy hitter: a flow that are sending lots of packets within a short term.

- Q: How can you find which flow is a heavy hitter?

- Strawmans:
  1. count every single flow,
  2. sampling,
  3. ...
Sketch: Introduction

- A class of data structures that summarize the information of a large set of streaming events (packets) in a fast and compact way
- E.g., count-min sketch, bitmap, ...
Count-Min Sketch for Heavy Hitter Detection

- d independent hash functions \((h_1, h_2, \ldots)\), each of which has a range of \(w\)
- When a packet arrives, hash its 5-tuple using each hash function in parallel, and add 1 to the corresponding locations.
- Compute the number of packets of a flow \(f\)

\[
\text{Count}(f) = \min_j (\text{get}(h_j(f)))
\]
Example

- 3 independent hash functions with range 4
- p1 of flow 1 arrives, h_1=1, h_2=2, h_3=1

<table>
<thead>
<tr>
<th>Hash 1</th>
<th>Hash 2</th>
<th>Hash 3</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>
</tbody>
</table>
Example

- 3 hash functions with range 4
- p1 of flow 1 arrives, \( h_1=1, h_2=2, h_3=1 \)

<table>
<thead>
<tr>
<th>Hash 1</th>
<th>Hash 2</th>
<th>Hash 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</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>
Example

- 3 hash functions with range 4
- p1 of flow 1 arrives, $h_1=1$, $h_2=2$, $h_3=1$
- p2 of flow 2 arrives, $h_1=1$, $h_2=1$, $h_3=4$

<table>
<thead>
<tr>
<th>Hash 1</th>
<th>Hash 2</th>
<th>Hash 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</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>
Example

- 3 hash functions with range 4
- p1 of flow 1 arrives, $h_1=1$, $h_2=2$, $h_3=1$
- p2 of flow 2 arrives, $h_1=1$, $h_2=1$, $h_3=4$

<table>
<thead>
<tr>
<th>Hash 1</th>
<th>Hash 2</th>
<th>Hash 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

- 3 hash functions with range 4
- p1 of flow 1 arrives, $h_1=1$, $h_2=2$, $h_3=1$
- p2 of flow 2 arrives, $h_1=1$, $h_2=1$, $h_3=4$
- p3 of flow 1 arrives, $h_1=1$, $h_2=2$, $h_3=1$

<table>
<thead>
<tr>
<th>Hash 1</th>
<th>Hash 2</th>
<th>Hash 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

- 3 hash functions with range 4
- p1 of flow 1 arrives, $h_1=1$, $h_2=2$, $h_3=1$
- p2 of flow 2 arrives, $h_1=1$, $h_2=1$, $h_3=4$
- p3 of flow 1 arrives, $h_1=1$, $h_2=2$, $h_3=1$

<table>
<thead>
<tr>
<th>Hash 1</th>
<th>Hash 2</th>
<th>Hash 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

- 3 hash functions with range 4
- p1 of flow 1 arrives, h_1=1, h_2=2, h_3=1
- p2 of flow 2 arrives, h_1=1, h_2=1, h_3=4
- p3 of flow 1 arrives, h_1=1, h_2=2, h_3=1

<table>
<thead>
<tr>
<th>Hash 1</th>
<th>Hash 2</th>
<th>Hash 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

count(flow 1) = min(get(h_1(flow 1)), get(h_2(flow 1)), get(h_3(flow 1)))
= min (3, 2, 2) = 2
Accuracy of Count-Min Sketch

- When $d = \lceil \ln 1 / \delta \rceil$, $w = \lceil e / \epsilon \rceil$, the estimated count of flow $i$ using count-min sketch $\hat{a}_i$ satisfies (1) $\hat{a}_i \geq a_i$, and (2) with probability at least $1 - \delta$, $\hat{a}_i \leq a_i + \epsilon \|a\|_1$.

- Proof
Offline Reading

- **Measurement library of different sketches**

- **Universal sketch for multiple applications**
Discussion

- What do you like about sketch?

- What do you not like about sketch?
  - Limited applications, e.g., cannot measure queueing latency
  - Low-level programming
Outline

- Network telemetry
  - Sketch
  - Marple: High-level network telemetry programming
    - Programing model
Marple: Abstractions

- Packet stream: for each packet in the queue of switch

\[ S := (\text{switch}, \text{qid}, \text{hdrs}, \text{uid}, \text{tin}, \text{tout}, \text{qsize}) \]

- Location
- Packet identification
- Queue entry and exit timestamps
- Queue depth seen by packet

Question: what are the operators needed?
Operators on Packet Stream

\[ S := (\text{switch}, \text{qid}, \text{hdrs}, \text{uid}, \text{tin}, \text{tout}, \text{qsize}) \]

Familiar functional operators

- filter
- map
- zip
- groupby
Filter: Restrict Stream of Interest

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pktstream</td>
<td>Stream of packet performance metadata.</td>
</tr>
<tr>
<td>filter(R, pred)</td>
<td>Output tuples in R satisfying predicate pred.</td>
</tr>
</tbody>
</table>

- **Example: high queue latency packets**

  ```
  result = filter(pktstream, qid == Q and switch == S and tout - tin > 1ms)
  ```
Map: Stateless Computation over Packets

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pktstream</td>
<td>Stream of packet performance metadata.</td>
</tr>
<tr>
<td>filter(R, pred)</td>
<td>Output tuples in R satisfying predicate pred.</td>
</tr>
<tr>
<td>map(R, [exprs], [fields])</td>
<td>Evaluate expressions, [exprs], over fields of R, emitting tuples with new fields, [fields].</td>
</tr>
</tbody>
</table>

- Example: rounding packet timestamps to an epoch

```python
result = map(pktstream, [tin/epoch_size], [epoch]);
```
Groupby: Aggregating Statefully over Packets

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pktstream</td>
<td>Stream of packet performance metadata.</td>
</tr>
<tr>
<td>filter(R, pred)</td>
<td>Output tuples in R satisfying predicate pred.</td>
</tr>
<tr>
<td>map(R, [exprs],</td>
<td>Evaluate expressions, [exprs], over fields of R,</td>
</tr>
<tr>
<td>[fields])</td>
<td>emitting tuples with new fields, [fields].</td>
</tr>
<tr>
<td>groupby(R,</td>
<td>Evaluate function fun over the input stream R</td>
</tr>
<tr>
<td>[fields], fun)</td>
<td>partitioned by fields, producing tuples on emit().</td>
</tr>
</tbody>
</table>

Example: per-flow average latency

R1 = filter(S, proto == TCP)
R2 = groupby(R1, 5tuple, ewma)

```python
def ewma([avg],[tin, tout]):
    avg = (1-\alpha)*avg + \alpha*(tout-tin)
```
Groupby: Aggregating Statefully over Packets

Marple only allows three classes of aggregation operations

- Operation independent on each switch, e.g., querying latency on a switch
- Operation independent on each packet, e.g., average link utilization seen by packet along its path
- Associative and commutative operation, e.g., count
Zip: Joining Results across Queries.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pktstream</td>
<td>Stream of packet performance metadata.</td>
</tr>
<tr>
<td>filter(R, pred)</td>
<td>Output tuples in R satisfying predicate pred.</td>
</tr>
<tr>
<td>map(R, [exprs], [fields])</td>
<td>Evaluate expressions, [exprs], over fields of R, emitting tuples with new fields, [fields].</td>
</tr>
<tr>
<td>groupby(R, [fields], fun)</td>
<td>Evaluate function fun over the input stream R partitioned by fields, producing tuples on emit().</td>
</tr>
<tr>
<td>zip(R, S)</td>
<td>Merge fields in incoming R and S tuples.</td>
</tr>
</tbody>
</table>

- **Example: detect TCP incast**

R1 = map(pktstream, [tin/epoch_size], [epoch]);
R2 = groupby(R1, [5tuple, epoch], new_flow);
R3 = groupby(R2, [epoch], count);
R4 = zip(R3, pktstream);
result = filter(R4, qsize > 100 and count > 25);
Outline

- Network telemetry
  - Sketch
  - Marple: High-level network telemetry programming
    - Programming model
    - Data plane realization
Question: Can all these operators be implemented on programmable switch?

- Stateless operation: filter, map, zip
- Stateful operation: groupby
Why Stateful Operation on Switch is Hard?

```python
ewma_query = groupby(S, 5tuple, ewma)
def ewma([avg], [tin, tout]):
    avg = (1-\alpha)\times avg + \alpha\times(tout-tin)
```

- Compute and update values at switch line rate (1 pkt / ns)
  - SRAM is fast but expensive
- Scale to millions of aggregation keys (e.g., 5-tuples)
  - DRAM is cheap but slow
How to Do Things Fast and Scale?

- **Cache:** the illusion of fast and large memory

![Diagram showing on-chip cache (SRAM) and off-chip backing store (DRAM)]
Cache: the Illusion of Fast and Large Memory

```python
def ewma([avg], [tin, tout]):
    avg = (1-\alpha)*avg + \alpha*(tout-tin)
```

*Read value for 5-tuple key K*

*Modify value using ewma*

*Write back updated value*
Cache: the Illusion of Fast and Large Memory

ewma_query = groupby(S, 5tuple, ewma)

def ewma([avg], [tin, tout]):
    avg = (1-\alpha)*avg + \alpha*(tout-tin)
Cache: the Illusion of Fast and Large Memory

def ewma([avg], [tin, tout]):
    avg = (1-\alpha) * avg + \alpha * (tout-tin)
Problem Solved?

- Cache misses require accessing off-chip DRAM ==> non-deterministic latency

- Question: what do we do?
Cache w/o Reading Back

- Once evicted, never come back
- Treat cache misses as packets from new flows
Cache w/o Reading Back

Read value for key K

On-chip cache (SRAM)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>$V_0$</td>
</tr>
</tbody>
</table>

Off-chip backing store (DRAM)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
</table>


Cache w/o Reading Back

Read value for key K

On-chip cache (SRAM)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>( V_0 )</td>
</tr>
</tbody>
</table>

Evict \( K', V'_\text{cache} \)

Off-chip backing store (DRAM)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( K' )</td>
<td>( V'_\text{back} )</td>
</tr>
</tbody>
</table>
Cache w/o Reading Back

On-chip cache (SRAM)

Key | Value
---|---
K  | V₀

Evict K', V'.cache

Off-chip backing store (DRAM)

Key | Value
---|---
K'  | V'._back

Read value for key K
Cache w/o Reading Back

Read value for key K

On-chip cache (SRAM)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>$V_0$</td>
</tr>
</tbody>
</table>

Evict $K', V'_{cache}$

Nothing to wait for.

Off-chip backing store (DRAM)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K'$</td>
<td>$V'_{back}$</td>
</tr>
</tbody>
</table>
What about Accuracy?

```
ewma_query = groupby(S, 5tuple, ewma)
def ewma([avg], [tin, tout]):
    avg = (1-α)*avg + α*(tout-tin)
```

- Initial state $s_0$, backing store state $s_{backing}$, caching state $s_{cache}$ for $N$ packets after eviction
- What is the correct state?

\[
s_{correct} - (1 - \alpha)^N s_{backing} = s_{cache} - (1 - \alpha)^N s_0
\]

\[
s_{correct} = s_{cache} + (1 - \alpha)^N (s_{backing} - s_0)
\]
Can Any Aggregation Function Be Merged?

Theorem 3.1. Every aggregation function has a corresponding merge function that uses $O(n2^n)$ auxiliary bits.

- Unfortunately, memory is limited
Mergable Aggregation Function

- Two classes of aggregations
  1. Associate aggregation: e.g., addition, max, min, ...
  2. Linear-in-state aggregation:
     - $p$: last $k$ packets of a flow
     - $A(p), B(p)$: functions of $p$
     - $S = A(p) \times S + B(p)$
     - E.g., EWMA, counting, ...

**Theorem 3.2.** If an aggregation function is either linear-in-state or associative, it has a merge function that uses $O(n)$ bits of auxiliary state.
Offline Exercise

- Is there any query in previous slides non-mergable? If so, which ones?
  - If you find the answer,
  - I got Chinese cookies as prize
Outline

- Network telemetry
  - Sketch
  - Marple: High-level network telemetry programming
    - Programming model
    - Data plane realization
    - From Marple query to data plane realization
Marple: Query Compilation

Example: count out-of-sequence TCP packets over time epoch at two switches

def oos_count(([count, lastseq], [tcpseq, payload_len])):
    if lastseq != tcpseq:
        count = count + 1
        emit()
    lastseq = tcpseq + payload_len

tcps    = filter(pktstream, proto == TCP
                 and (switch == S1 or switch == S2));
tslots  = map(pktstream, [tin/epoch_size], [epoch]);
joined  = zip(tcps, tslots);
oos     = groupby(joined,
                  [5tuple, switch, epoch],
                  oos_count);
Step 1: Network-wide query to switch-local queries

- Determine packet stream location for final output stream
  - filter: decided in predicate
  - map and groupby: unchanged
  - zip: intersection of input stream locations

```python
def oos_count(count, lastseq, tcpseq, payload_len):
    if lastseq != tcpseq:
        count = count + 1
    emit()
    lastseq = tcpseq + payload_len

tcps = filter(pktstream, proto == TCP
              and (switch == S1 or switch == S2));
tslots = map(pktstream, [tin/epoch_size], [epoch]);
joined = zip(tcpseq, tslots);
join = groupby(joined,
               [Stuple, switch, epoch],
               oos_count);
```
Step 1: Network-wide query to switch-local queries

- Determine packet stream location for final output stream
- Check if operations are independent on each switch
  - filter and zip: true if output are on single switch
  - groupby: true if aggregate by switch
  - map: unchanged
Step 2: Query AST to Pipeline Configuration

- Postorder traversal of AST to generate operator sequence

```
def oos_count(([count, lastseq], [tcpseq, payload_len])):
    if lastseq != tcpseq:
        count = count + 1
        emit()
        lastseq = tcpseq + payload_len

tcps = filter(pktstream, proto == TCP
              and (switch == S1 or switch == S2));
tsslots = map(pktstream, [tin/epoch_size], [epoch]);
joined = zip(tcp, tslots);
oos = groupby(joined,
              [Stuple, switch, epoch],
oos_count);
```

\[ \text{tcp}(\text{filter}) \rightarrow \text{tslots (map)} \rightarrow \text{joined (zip)} \rightarrow \text{oos (groupby)} \]
Step 2: Query AST to Pipeline Configuration

- Postorder traversal of AST to generate operator sequence
- From operator sequence to P4 code
  - filter and zip: check predicate and set a “valid” bit on metadata
  - map: assign metadata to computed expression
  - groupby: assign and update registers for aggregation fields
Step 3: Detect Linear-in-State Aggregation

- Compute “history” of each variable in aggregation function
  - history: how many previous packets to look at to determine the variable’s value accurately
  - E.g., counter: history = ∞, last TCP SEQ: history = 2
Step 3: Detect Linear-in-State Aggregation

- Compute “history” of each variable in aggregation function

- History of state variable:
  - Finite history: linear-in-state
  - Infinite history: syntactic pattern matching “S=A*S+B”
    - Incomplete check: $S = (S^2-1) / (S-1)$
  - All state variables are linear-in-state -> aggregation is linear-in-state
Step 3: Detect Linear-in-State Aggregation

- Compute “history” of each variable in aggregation function
- History of state variable:
- Linear-in-state aggregation: compute auxiliary state

Diagram:

1. **Aggregation function code (in Marple)**
2. **Step 1:** Compute history for each state variable
3. **Step 2:** Are all state variable updates linear-in-state? (Finite history variables are trivially linear in state)
   - **YES** → Query is scalable
     - **Step 3:** Compute auxiliary state required for merge
   - **NO** → Query is not scalable
Marple: Putting Things Together

Collection servers to handle
(1) Marple query results
(2) Evictions to backing store

Network operator

Marple Queries

Marple Compiler

Switch Programs

Programmable switches with programmable key-value store

End hosts
Discussion

- What do you like about Marple?
- What are the limitations?
Offline Reading

- Telemetry using stream processor

- Partition telemetry into programmable switch and stream processor

- End-host telemetry