Traffic monitoring in software dataplane: a generic accuracy-aware adaptive solution

Daphné Tuncer

Department of Computing
Imperial College London (UK)
dtuncer@ic.ac.uk
Collaborative work with

Gioacchino Tangari (leading efforts – Macquarie University)

Marinos Charalambides (UCL)

George Pavlou (UCL)
Managing networks is hard

**Complex systems challenging to manage**

Demand **agility, flexibility, adaptability and reactivity**
Advances in networking have brought promises enable reactive behaviour in response to emerging requirements.

New opportunities for effective network resource management:

**Quickly** adapt and react to network and traffic dynamics

Configuration **flexibility**

Express **high-level operators’ policies**
A key functionality: monitoring

Goal: provide efficiency

Management System

- Monitoring
- Traffic load-balancing
- Anomaly detection
- Energy-saving management

Collect statistics from the network resources
Apply new configurations in the network
What is efficient monitoring?

Detailed information

  Identify congestion, DDoS attacks, unresponsive servers, ...

Timely reports

  Detect short-lived episodes, support fast resource reconfigurations, ...
Well, easy?...

Reality: hard to produce *detailed* and *timely* reports

- Hardware & resource constraints
- Large scale settings
- Massive and dynamic network traffic

On top of that constraints on the monitoring system

- Scalability requirements
- Good accuracy/resource usage trade-offs

Firestone *et al.* "a physical core sells for $0.10-0.11/hr, [...] a maximum potential revenue of around $900/yr" [FirestoneNSDI18]

Designing efficient monitoring for software-based networks
Traffic monitoring in software dataplane

Monitoring States
State 1: Count
State 2: Count + Heavy Hitter
State 3: Count + Heavy Hitter + Retransmission
State 4: Count + Heavy Hitter + Retransmission + Bursty Flow
State 5: Count + Heavy Hitter + Retransmission + Bursty Flow + Latency Change

Measurement Task: involves a set of operations (e.g., sum bytes)
Traffic monitoring in software dataplane

- Network Card
- Packet Buffer
- Monitoring Pipeline (Processing)

- Packets
- Flow 5-tuple
- Monitoring State
- Statistics Buffer
Traffic monitoring in software dataplane

Limited total time budget available

=> Potential packet loss at the buffers in case of bottlenecks

<table>
<thead>
<tr>
<th>Monitoring States</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>26 ns for processing</td>
</tr>
<tr>
<td>S2</td>
<td>87 ns for processing</td>
</tr>
<tr>
<td>S3</td>
<td>96 ns for processing</td>
</tr>
<tr>
<td>S4</td>
<td>122 ns for processing</td>
</tr>
<tr>
<td>S5</td>
<td>163 ns for processing</td>
</tr>
</tbody>
</table>
Traffic monitoring in software dataplane

How to reconfigure monitoring operations at run-time to cope with emerging conditions?

Monitoring States
- S1: 26 ns for processing
- S2: 87 ns for processing
- S3: 96 ns for processing
- S4: 122 ns for processing
- S5: 163 ns for processing

Limited total time budget available => Potential packet loss at the buffer(s) in case of bottlenecks
Solution: adapting monitoring at run-time

Detect changes in operating conditions in a timely manner

Dynamically reconfigure (per flow) measurement operations

Monitoring States
S1
S2
S3
S4
S5

Adjust packet processing time

No need of overprovisioning
Yes, but how do you...

1. Preserve monitoring report accuracy?

2. Avoid packet starvation (i.e., all packets processed in time)?

3. Guarantee low computation overhead (no more than 1% CPU-time)?
Key idea: estimating per-packet processing time

Total expected time of a packet in the monitoring pipeline

\[(1 - \lambda_f)[T_r^H P + T_r^M (1 - P) + T_p^{target}] + \lambda_f T_i = 1/\lambda_{pkt}\]

- Retrieval of existing flow entry information based on whether this is in cache (Hit) or memory (Miss)
- Insertion of a new flow entry
- Targeted per packet time
- Packet rate
Time profiling and estimation

Benchmarking of retrieval and insertion times

Based on sampling
Two strategies

**Greedy:** adaptation applied to random sets of flow-entries

**Low-States-First (LSF):** downgrade in priority flows mapped to less advanced monitoring states (except s1)
Objective:

To re-adjust flow allocations in order to satisfy a global accuracy threshold for all tasks

Two main steps:

1. To quantify the effect of adaptations on report accuracy
2. To recover accuracy gaps by re-adjusting flow allocation
Task generic online accuracy estimation

Accuracy estimated through the recall

\[
Recall_{iw} = \frac{N_{iw}^{\text{Found}}}{N_{iw}^{\text{Found}} + N_{iw}^{\text{Miss}}}
\]

Let’s expand $N^{\text{Miss}}$

\[
N_{iw}^{\text{Miss}} = F_{iw}^{\text{Miss}} \cdot E[X_{iw}^{\text{Miss}}]
\]

- $N_{iw}^{\text{Miss}}$: unknown
- $F_{iw}^{\text{Miss}}$: # flows dropped for task $i$ from adaptation
- $E[X_{iw}^{\text{Miss}}]$: # events for a missing flow

**Objective:** find an estimator for $X^{\text{Miss}}$
Task generic online accuracy estimation (con’t)

Our solution: risk minimization strategy

\[ R(\hat{x}^{Miss}) = \sum_{l=0}^{\infty} L(x_l^{Miss}, \hat{x}^{Miss}) \text{Prob}(X_{iw}^{Miss} = x_l^{Miss}) \]

Best estimator for \(X^{Miss} = \) one minimizing risk function \(R\)

\[ \hat{x}_{Best}^{Miss} = \arg\min_{\hat{x}^{Miss}} R(\hat{x}^{Miss}, L) \]
Solution: the Richs give to the Poors

Re-allocation of flows from monitoring states with rich tasks to monitoring states with poor tasks

Algorithm 2: Recover Accuracy Gaps

1: function UPDATESTEP_SIZE(x, S_x)
2:     Compute accuracy decrease D = A_{x,w-1} - A_{x,w}
3:     Update residual accuracy H = A_{x,w} - threshold
4:     if D > H then return INCREASE(S_x)
5:     else return DECREASE(S_x)
6: function REBALANCEBY_STEP(s_{Rich}, s_{Poor}, S)
7:     Compute \Delta^- = S/n_p, where n_p number of poor states
8:     Retrieve E[t^p_{Poor}] from T_{Poor,j}, j \in 1, .., n
9:     Compute \Delta^+ from equilibrium condition (9)
10: return \Delta^-, \Delta^+
11: procedure RECOVERYGAPS(A_w, M, T_w)
12:     Find set of rich, poor states \{s_{Rich}\}, \{s_{Poor}\} using A_w
13:     if \{s_{Poor}\} == \emptyset or \{s_{Rich}\} == \emptyset then return
14:     for each x in \{s_{Rich}\} do:
15:         S_x = UPDATESTEP_SIZE(x, S_x)
16:     for each (x, y) with x \in \{s_{Rich}\}, y \in \{s_{Poor}\} do:
17:         REBALANCEBY_STEP(x, y, S)
MONA implementation

- Implemented in C
  Generic monitoring pipeline based on a single flow-table

- Flow-table realized as a hash-table
  Table size = $2^{20}$ entries to limit hash collisions

- Flow-entry size = 64 bytes (fit within a single cache)

- Packet trace generated based on reported flow statistics in Facebook data centers [RoySIGCOMM15]
MONA evaluation setup

Evaluation with four measurement tasks

- Heavy Hitter detection (HH)
- Bursty flow detection (Bursty)
- Latency Change detection (LatChange)
- ReTransmission detection (RTx)

Two monitoring state configurations
How do we validate MONA?

1. How does adaptive traffic monitoring perform in terms of packet loss risk and adaptation responsiveness?

2. What is the impact of monitoring adaptation and accuracy control on the measurement tasks?

3. What are the throughput limiting factors for MONA?

4. What is the overhead of our solution?
MONA prevents packet loss and preserves packet balance
MONA maintains monitoring report accuracy
MONA throughput limiting factors

Impact of hash collisions

Impact of uniform traffic (DoS attack)

(a) Hash collision and pkt loss ratio  
(b) Average task accuracy (recall)

(a) DoS-induced packet loss  
(b) Accuracy (recall) reductions
MONA enables short timescale re-configurations (every 10ms), with no additional processor core(s) and minimal CPU-time overhead (~1-2%).
Concluding remarks

1. Timely and accurate resource monitoring fundamental for any network management system

2. Self-adaptive monitoring approaches as drivers to responsiveness and flexibility

3. Our solution: MONA
   - Adaptive monitoring framework offering resilience to bottlenecks
   - Preservation of monitoring accuracy


