Fast Prototyping Network Data Mining Applications

Gianluca Iannaccone
Intel Research Berkeley
Motivation

- Developing new network monitoring apps is **unnecessarily time-consuming**
- Familiar development steps
  - Need deep understanding of data sets (including details of the capture devices)
  - Need to develop tools to extract information of interest
  - Need to evaluate accuracy and resolution of data (e.g., timestamps, completeness of data, etc.)
- ...and all this happens before one can really get started!
Motivation (cont’d)

- Developers tend to find shortcuts
  - Quickly assemble bunch of ad-hoc scripts
  - Not “designed-to-last”
  - Well known consequences
    - hard to debug
    - hard to distribute
    - hard to reuse
    - hard to validate
    - suboptimal performance

- End result: many papers, very little code
Can we solve this problem by design?

- Yes, and it has been done before in other areas.

- Solution: Define declarative language and data model for network monitoring

- What is specific to network measurements?
  - Large variety of networking devices (i.e. potential data sources) such as NIC cards, capture cards, routers, APs, ...
  - Need native support for distributed queries to correlate observations from a large number of data sources.
  - Data sets tend to be extremely large for which data shipping is not feasible.
Existing Solutions

- AT&T’s GigaScope
- UC Berkeley’s TelegraphCQ and Pier
- Common approach (stream databases):
  - Define subset of SQL adding new operators (e.g., ‘window’ for time bins of continuous query)
  - Gigascope supports hardware offloading by static analysis of the GSQL query
Benefits and Limitations

+ Decouple what is done from how it is done.
+ Amenable to optimizations in the implementation
  - Limited expressiveness.
  - Need workaround to implement what is not in the language losing the advantages above
  - Entry barrier for new users is relatively high.
Alternative Design: The CoMo project

• Users write “monitoring plugins”
  • Shared objects with predefined entry points.
  • Users can write code in C or higher level languages (support for C#, Java, Python, and others)

• The platform provides
  • one single, extensible, network data model.
  • support for a wide variety of network devices.
  • abstraction of monitoring device internals.
  • enforcement of programming structure in the plug-ins to allow for optimization.
Design Challenges

• Fast Prototyping
  • Network Data and Programming Model

• Resource Management
  • Local monitoring node (Load Shedding)
  • Global network of monitors ("Network-wide Sampling")
Network Data Model

• Unified data model with quality and lineage information.
  • Allows the definition of ad-hoc metadata (i.e., labels defined by the users)

• Software sniffers understand native format of each device and translate to our common data model
  • support so far for PCAP, DAG, NetFlow, sFlow, 802.11 w/radio, any CoMo monitoring plug-in.

• Sniffers describe the packet stream they generate
  • Provide multiple templates if possible
  • Describe the fields in the schema that are available
  • Plug-ins just have to describe what they are interested in and the system finds the most appropriate matching
Programming Model

• Application modules made of two components:
  \(<filter>:<monitoring\ function>\)

• Filter run by the core, monitoring function contained in the plug-in written by the user
  • set of pre-defined callbacks to perform simple primitives
  • e.g., update(), export(), store(), load(), print(), replay()
  • callback are closures (i.e., the entire state is defined in the call). they can be optimized in isolation and executed anywhere.

• No explicit knowledge of the source of the packet stream
  • Modules specify what they need in the stream and access fields via standard macros
  • e.g., IP(src), RADIO(snr), NF(src_as)
Hardware Abstraction

• Goals: scalability and distributed queries
  • support large number of data sources and high data rates
  • support a heterogeneous environment (clients, APs, packet sniffers, etc.)
  • allow applications to perform partial query computations in remote locations

• To achieve this we...
  • hide to modules where they are running
  • enforce a programming structure
  • ... basically try to partially re-introduce declarative queries
Hardware Abstraction (cont’d)

- EXPORT/STORAGE can be replicated for load balancing
- CAPTURE is the main choke point
  - It periodically discards all state to reduce overhead and maintain a relative stable operating point
Distributed queries

- Modules behave as software sniffers themselves
  - replay() callback to generate a packet stream out of module stored data
  - e.g., snort module generates stream of packets labeled with the rule they match; module B computes correlation of alerts

- This way computations can be distributed but also modules can be pipelined (to reduce the load on CAPTURE)
Design Challenges

- Fast Prototyping
  - Network Data and Programming Model

- Resource Management
  - Local monitoring node (Load Shedding)
  - Global network of monitors ("Network-wide Sampling")
Resource Management

online

Load Shedding

Network-wide Sampling

offline

Capacity Provisioning

Distributed Indexing

local
global
Predictive Load Shedding

• Building robust network monitoring apps is hard
  • Unpredictable nature of network traffic
  • Anomalous traffic, extreme data mixes, highly variable data rates

• Operating Scenario
  • Monitoring system running multiple arbitrary queries
  • Single resource to manage: CPU cycles

• Challenge:
  “How to efficiently handle overload situations?”
Approach

- Real-time modeling of the queries’ CPU usage
  1. Find correlation between traffic features and CPU usage
     - Features are query agnostic with deterministic worst case cost
  2. Exploit the correlation to predict CPU load
  3. Use the prediction to guide the load shedding procedure

- Main Novelty:
  No a priori knowledge of the queries is needed
  - Preserves high degree of flexibility
  - Increases possible applications and network scenarios
Key Idea

• Cost of maintaining data structures needed to execute a query can be modeled looking at a basic set of traffic features

• Empirical observation
  • Updating state information incurs in different processing costs
    – E.g., creating or updating entries, looking for a valid match, etc.
  • Type of update operations depend on the incoming traffic
  • Query cost is dominated by the cost of maintaining the state

• Our method
  • Find the right set of traffic features to model queries’ cost
Example
Example
System overview

Use multi-resolution bitmaps to extract features (e.g., # of new flows, repeat flows, with different aggregation levels).

Use a variant of FCBF [1] to remove irrelevant and redundant features.

MLR to predict CPU cycles needed by queries to process the batch.

Apply flow/packet sampling on batch to reduce CPU requests. Assume linear relationship CPU/pkts.

Use TSC to measure and feed back actual cycles spent.

Performance: Cycles per batch
Performance: packet losses

- No load shedding
- Reactive
- Predictive
Performance: Accuracy

- Queries estimate their unsampled output by multiplying their results by the inverse of the sampling rate.

<table>
<thead>
<tr>
<th>Query</th>
<th>original</th>
<th>reactive</th>
<th>predictive</th>
</tr>
</thead>
<tbody>
<tr>
<td>application (pkts)</td>
<td>55.38% ±11.80</td>
<td>10.61% ±7.78</td>
<td>1.03% ±0.65</td>
</tr>
<tr>
<td>application (bytes)</td>
<td>55.39% ±11.80</td>
<td>11.90% ±8.22</td>
<td>1.17% ±0.76</td>
</tr>
<tr>
<td>flows</td>
<td>38.48% ±902.13</td>
<td>12.46% ±7.28</td>
<td>2.88% ±3.34</td>
</tr>
<tr>
<td>high-watermark</td>
<td>8.68% ±8.13</td>
<td>8.94% ±9.46</td>
<td>2.19% ±2.30</td>
</tr>
<tr>
<td>link-count (pkts)</td>
<td>55.03% ±11.45</td>
<td>9.71% ±8.41</td>
<td>0.54% ±0.50</td>
</tr>
<tr>
<td>link-count (bytes)</td>
<td>55.06% ±11.45</td>
<td>10.24% ±8.39</td>
<td>0.66% ±0.60</td>
</tr>
<tr>
<td>top destinations</td>
<td>21.63 ±31.94</td>
<td>41.86 ±44.64</td>
<td>1.41 ±3.32</td>
</tr>
</tbody>
</table>

Errors in the query results (*mean ± stdev*)
Limitations

- Current method works only with queries that support packet/flow sampling
  - Working on custom load shedding support

- Results shown when applying same sampling rate across all queries.
  - Need to accommodate for varying needs of queries
  - Maximize the overall system utility by guaranteeing queries a fair access to CPU (and packet streams)

- Consider other resources (e.g., memory, disk)
Resource Management

online

Load Shedding

Network-wide Sampling

capacity

Capacity Provisioning

Distributed Indexing

offline

local

global
Network-wide Sampling

- Given a network of monitors, select the ones that need to participate in a measurement task
  - The task is unknown a priori

- Operating scenario
  - Routing is known. Relationship between pairs of monitoring nodes is known

- Challenge:
  how to configure a network-wide monitoring infrastructure with hundreds of viewpoints?
Our objective

• Given a measurement task and a target accuracy, find a method that:
  • sets the sampling rates on all monitors
  • guarantees optimal use of resources (in terms of processed packets)
  • requires minimum configuration
  • can adapt quickly to changes in the traffic

• Method should apply to a general class of measurement tasks
A case study

- Estimate amount of traffic flowing among a subset of origin-destination pairs
- Common task for traffic engineering apps
Problem formulation

Choose vector of sampling rates $p$ that maximizes

\[
\sum_{k \in \mathcal{F}} M\left(\rho_k(p)\right),
\]

subject to

\[
p_i \geq 0
\]

\[
p_i \leq \alpha_i \quad \text{for all } i \in \mathcal{L}
\]

\[
\sum_{i \in \mathcal{L}} p_i U_i \leq \theta
\]

- Effective sampling rate approximated by sum of sampling rates
- All constraints are linear and define a convex solution space
- Unique maximizer exists as long as $M(\cdot)$ is strictly concave
Algorithm

- Solve system defined by KKT conditions
  - select set active/inactive constraints (equivalent to switching off/on a monitor)
  - use gradient projection method to explore space
  - use KKT conditions to check optimality of solution

- Selection of active/inactive constraints is NP-hard → no guarantee of convergence

- Limit algorithm runs to 2,000 iterations → 98.6% optimum found (for our task)
The utility function

• Measures quality of sampling an OD pair
• “Well behaved” to make the algorithm run fast
• Square relative error good candidate
  • \( \text{SRE} = \left( \frac{X}{p} - S \right) / S \)^2
• Utility is \( 1 - \mathbb{E}[\text{SRE}] \)
  • \( M(p) = 1 - \mathbb{E}[1/S] * (1/p - 1); \)
  • minor tweaking to force it to be zero when \( p = 0 \)
  • needs \( \mathbb{E}[1/S] \) where \( S \) is the size of the OD pair
Evaluation

• Consider NetFlow data from GEANT
  • Collected using Juniper’s Traffic Sampling
  • 1/1000 periodic sampling
  • We scale the measurement by 1000 (we just need a realistic mix of OD pair sizes)

• Results based on one run of the algorithm
  • One five minute snapshot of the network traffic
  • Compute OD pair sizes and link loads
  • Assume E[1/S] is known
Results highlights

• Measuring relative accuracy
  • Defined as one minus relative error (not squared)
  • Allows to validate manipulation of utility function and the use of effective sampling rate

• Accuracy is in the range 89-99%
  • Worst accuracy for JANET – LU (it has just 20 pkts/sec)

• Measurement spread across 10 links

• Max sampling rates is 0.92% (lightly loaded links)
  • Most links are around 0.1%
  • No OD pair is monitored on more than two links
Comparing to “naive” solutions

• Why not just monitoring JANET access link?
  • All the monitored traffic would be relevant!
  • To achieve same accuracy over all OD pairs we need ~1% sampling rate
    → 70% more packets are processed
  • It’s not always possible to monitor both directions of access links

• Why not just monitoring all UK links?
  • There are just 6 links leaving the UK
  • Straightforward algorithm to set sampling rate (each OD pair is present on just one link), but...
Monitoring all UK links

• Why does our method work better?
  • It looks across the entire network to find where small OD pairs manifest themselves without hiding behind large flows
Deployment on real networks

• Two aspects need to be addressed

• Bootstrap:
  What prior knowledge about the network does the method need?
  • need routing information
  • need estimate of E[1/S] for each OD pair

• Adaptation:
  How does the method perform over time?
  • time of day effect change E[1/S] and Ui
  • routing event change path taken by OD pairs
Bootstrapping phase
Adapting to traffic fluctuations

- Three different cases that require different approaches
- Link load increases
  - more sampled packets, exceeding capacity
  → find new sampling rates to enforce target capacity
- OD pair decreases in volume
  - poor accuracy because of bad E[1/S] estimate
  → adapt capacity Q to keep target accuracy
- OD pair traverses different set of links
  - missing entire OD pair
  → monitor routing updates and “re-bootstrap” the algorithm
Fluctuations in OD pairs

- Monitoring accuracy of OD pairs
  - Accuracy is not known.
  - Need to estimate $E[1/S]$ from sampled data.
  - Use simplest method → Current size of OD pair

- Compute new sampling rates when estimated accuracy drops below target

- If the estimated accuracy is still below target, increase capacity by 10%

- Decrease capacity if estimated accuracy is above target for more than one hour
Fluctuations in OD pairs
Fluctuations in OD pairs (cont’d)

![Graph showing fluctuations in OD pairs](image-url)
Conclusions

• The CoMo Project
  • Code available at http://como.sourceforge.net
  • Open source, BSD License
  • Currently in the process of being commercialized by Intel (codename Harris Hill)
  • Used by EU Onelab/Onelab2 (Planetlab Europe)