

EQuery: Enable Event-driven Declarative Queries in Programmable Network Measurement

Yongyi Ran*, Xiaoban Wu[†], Peilong Li*, Chen Xu[†], Yan Luo*

Department of Electrical and Computer Engineering

University of Massachusetts Lowell

*{yongyi_ran, peilong_li, yan_luo}@uml.edu,

[†]{xiaoban_wu, chen_xu}@student.uml.edu

Liang-min Wang

Intel Corporation

Hudson, MA 01749

liang – min.wang@intel.com,

Abstract—Network measurement is critical in network management such as performance monitoring, diagnosis, and traffic engineering. However, conventional network measurement solutions are limited by simple and fixed functionalities as well as coarse-grained statistics which often fail to precisely illustrate network conditions. In this paper, we propose an event-driven declarative query language, EQuery, for programmable network management in order to design sophisticated measurement tasks and enable event mechanism to avoid human intervene. Furthermore, we design a compiler to support the query language on the EQuery Controller, which drives the chaining query workflow with nondeterministic finite automaton (NFA), and translates measurement jobs into low-level rules/states on the physical devices. Finally, we evaluate the effectiveness of our EQuery framework on a nation-wide operational network with real-time network statistics.

Index Terms—Network Management; Programmable Network Measurement; Event-driven Programming; NFA; NetASM;

I. INTRODUCTION

Network measurement is instrumental to understanding network behavior, fault diagnosing, attack prevention and traffic engineering. While conventional measurement tools support basic network flow metrics (e.g. NetFlow [5], sFlow[9]), the hardware and software to build these measurement units are often proprietary and with fixed functionality, which imposes tremendous obstacles to satisfying on-demand and ad hoc measurement tasks.

Furthermore, though the emerging software defined networking (SDN) and programmable switches promise various measurement tasks including per-flow and traffic metrics, the intrinsic simple counting primitives limit the usage scope. Specifically, network operators need to understand too much of the network details [6] to conduct network trouble-shooting. As a result, several domain-specific languages [4], [11], [7], [6], [8] emerge to descend the bar for straightforward yet efficient network management. But there are still major challenges awaiting solutions in the prior arts.

First and foremost, the lack of an event-driven declarative query language to express various network mea-

surement tasks. Any meaningful measurement task is composed of a collection of measurement primitives, and “event” plays the key role in chaining the primitives to fulfill the complete measurement functionality. For example, to diagnosis packet loss and the root cause along the packet traversal path, one need to first count the packet loss (a primitive) at certain point of the network. And based on the packet loss statistics (event), subsequent measurement primitives such as link throughput checking (another primitive) may need to be conducted to pin-point the failure. “Event” is critical in this case to avoid human intervene and enhance measurement efficiency, as it is the glue logic to trigger the chaining primitives. Moreover, “declarative” query allows network operators to effectively focus on “what” they want to measure as opposed to “how to” in “imperative” query.

Additionally, the need of a compiler to analyze the dependencies within a composite measurement task. Measurement tasks are both resource intensive and time sensitive. The execution of measurement functions require substantial amount of memory and CPU resources to store and update intermediate values along with network states. If not properly analyzed and optimized before instantiation, query jobs from the end users can easily overwhelm the network hardware and lead to management malfunction [7], [8]. Therefore, it is critical for the compiler to 1) understand the temporal and spatial dependencies among the atomic queries in a composed measurement task; and 2) optimize dependency graph to reduce redundant hardware consumption.

To solve the aforementioned challenges, we present EQuery, a network measurement framework that consists of a query language and the associated toolset for event-driven declarative queries in programmable network measurement. This query language supports both the specification of functionalities and temporal dependencies of measurement tasks and enables network monitoring and diagnosing effectively. Specifically, we made the following contributions:

(1) We present an event-driven declarative query language for programmable network measurement. System users can declaratively express their measurement tasks by using a

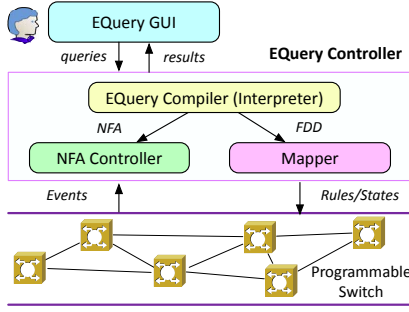


Fig. 1. EQuery system overview.

SQL-like high-level language, independent of the hardware details to derive the entire network conditions as well as individual link/device status. More importantly, we build “events” into the query language to trigger subsequent queries according to the occurrence of specified situations.

(2) We design a compiler to support the proposed query language. The compiler translates the temporal and logical dependencies among atomic queries into an nondeterministic finite automaton (NFA) located on the EQuery controller for maintaining runtime status and managing the triggers of events. The compiler also maps the declarative queries into low-level rules and states onto the network devices.

(3) We design a web interface (cherry.uml.edu) for measurement task submission, and perform a collection of real-world measurement tasks by using EQuery on a nationwide network testbed to demonstrate the effectiveness of the proposed language.

II. THE EQUERY FRAMEWORK OVERVIEW

The proposed EQuery framework in Fig. 1 is composed of 3 major EQuery components: language, controller and GUI. System users submit measurement queries via the **EQuery GUI** with the **EQuery language**. The **EQuery Controller** will then interpret the submitted queries by leveraging the front-end **Compiler**. The compiler analyzes the query dependencies to generate necessary information for the **NFA Controller** and **Mapper** to control the workflow and to configure the underlying network devices. The network events and the measurement results will be reported back to the EQuery Controller for processing.

The EQuery Language. With this query language, network operators can specify what areas of interest should be measured, e.g., the fields in a packet header (such as `src_ip` and `dst_ip`) along with device parameters (such as switch id and in-port). In addition, three other elements can be specified: a filter (to sift specific attributes), an aggregator (for statistics aggregation) and a trigger (to initiate an event). We explain the details in Section §III.

The EQuery Compiler. The EQuery front-end compiler serves for two purposes: 1) analyze the measurement tasks and generate a query dependency and execution graph called NFA for the NFA Controller; and 2) understand

the query semantics to generate “if-else-then” structures, and then constructs each “if-else-then” code-let into a Forwarding Decision Diagram (FDD) [11], [4] for the back-end compiler.

The NFA Controller. The NFA controller takes the NFA from the compiler and controls the workflow of the entire measurement task. The NFA controller tracks the *States* (atomic queries) and transitional *edges* (events) in the current measurement task.

The Mapper. The Mapper is in charge of translating the high-level measurement tasks down to the rules and states of the network hardware in order to filter packets and record statistics. The mapper converts the FDDs from the compiler to a set of switch-level intermediate representations described by NetASM [10], which comes with programmable switches capable of executing those instructions.

Events. Event-driven queries can issue immediate event trigger for pre-defined situations, reducing human intervene in the measurement process while simultaneously avoiding unnecessary traffic on the link to the EQuery controller. An event often indicates a special network situation that requires further action in terms of packets dropped, delayed, or delivered by the network, and can range in topological scope from a specific flow, to traffic aggregates. In this paper, we define two types of events: periodic events (PE) and conditional events (CE). A PE will return measurement results to the EQuery controller on a pre-defined schedule periodically, while a CE will return an event identity along with measurement results to the EQuery controller once a specified network condition occurs. In the EQuery language, a PE is indicated by a time frequency clause in the temporal domain while a CE is specified as a trigger.

In the following sections, we present each of the aforementioned components in greater detail.

III. THE EQUERY LANGUAGE

It is ideal to provide end users a declarative way of expressing “what” they want to measure rather than “how to”. Specifically, a declarative query language should: 1) be able to gather network-wide information, including per-packet, per-flow, and per-link status; 2) support atomic query composition and events mechanism to trigger atomic queries. To achieve the above features, we design an SQL-like query language, EQuery, to express *what* metrics should be measured, *how* to aggregate the data, and *when* to trigger subsequent measurement queries.

A. Primary Clauses

The syntax of EQuery is illustrated in Fig. 2 and the explanations are as follows.

Select: A $Select(h, s, f(h, s))$ clause is used to express what data the user wants to query. Here the arguments are defined as “measurement fields”, including the fields of a packets header, h , the parameters of network devices,

Query q	$:=$	Select($h, s, f(h, s)$) Where(p) Groupby(h, s) When(e) Start(t) Within(i) Every(i)
Pkt hdr h	$:=$	$src_ip \mid dst_ip \mid src_pt...$
Switch s	$:=$	$sw_id \mid pt_in \mid pt_out...$ $\mid q_id \mid q_in \mid q_out \mid q_size...$
Agg $f(h, s)$	$:=$	function of h, s (§III-B)
Event e	$:=$	$(q : g(q.h, q.s, q.f) \bowtie \text{threshold}) \mid (q : \text{true})$
Time t	$:=$	start time
Interval i	$:=$	integer in s or ms
Predicate p	$:=$	$((h \mid s \mid f) \bowtie \text{value}) \mid p \& p \mid (p \mid p) \mid \sim p$
Optr \bowtie	$:=$	$> \mid < \mid >= \mid <= \mid \sim$

Fig. 2. Measurement query language syntax.

s , as well as the aggregated fields, $f(h, s)$, (e.g. COUNT, SUM and customized aggregation fields). More details are introduced in §III-B.

Where: A *Where*(p) clause applies the predicate p to filter the query results. Only those packets satisfying the predicate p will be analyzed for statistical purposes. The basic predicate uses the syntax $((h \mid s \mid f) \bowtie \text{value})$ to test the selected measurement field. In this paper, a test on an aggregated field ($f \bowtie \text{value}$) actually imposes a test on a data plane state. “ \bowtie ” is a series of operators where “ \sim ” indicates “unequal”. More complicated filtering predicates can be constructed using natural set-theory operations such as intersection ($p \& p$), union ($p \mid p$) and complementation ($\sim p$).

Groupby: A *Groupby*(h, s) clause can be used in a “Select” statement to collect measurement data across multiple records and group the results by one or more columns (h, s). The “Groupby” clause is often combined with aggregate functions (§III-B).

When: A *When*(e) clause is employed to impose a trigger on a query. A query driven solely by a “When” clause will be executed once event e occurs (§III-C).

Others: A *Start*(t) clause indicates that the start time of the submitted query is t (in date time format). The default value is the time when the query is submitted. A *Within*(i) clause specifies the maximum execution time for a query. An *Every*(i) clause groups packets that arrive within the same time window, where i can be specified in seconds or microseconds. In turn, the results will be returned every i seconds (or microseconds). The “Every(i)” clause is also used to express a PE in a query.

B. Measurable Fields

There are three categories of measurable fields: packet fields, switch fields and aggregated fields. As shown in Fig. 2, all three categories can be applied for querying target packets as in a “Select” statement, or filtering target packets with “Where”.

Packet Fields. Packets traversing a network can be parsed by network devices to extract the header fields, including the source IP, destination IP, source port, destination port, source MAC, destination MAC, etc. Based on the primary header fields, network operators can query per-packet and per-flow information.

Switch Fields. In addition to the packet fields, network operators are often interested to know: what is the bandwidth utilization of a network link? Where does packet loss happen? What are the one-hop and end-to-end latency? All these questions involve querying the status and performance along a specified link or path. In this paper, we use sw_id to identify a switch or a router while pt_in, pt_out are used to specify the in-port and out-port of a packet traversing a switch respectively. A link l can be represented by: $l \triangleq (sw_id : pt_out \rightarrow sw_id : pt_in)$.

In our design, we employ the queue metadata [7] technique to manipulate switch fields. The metadata q_id identifies a specific queue on a specific switch at which the current packet is observed. The metadata q_in and q_out are timestamps corresponding to the arrival and departure of the packet with respect to a queue. We let $q_out = \infty$ in the event that a packet is dropped from a queue. The field q_size is the queue length seen by the packet when it is enqueued.

Aggregated Fields. Aggregated fields are applied to obtain statistical values on a group of packets that satisfy some pre-defined conditions. Each aggregated field is associated with one aggregate function. In order to calculate the required aggregated field, the aggregate function is used to perform operations across the packets filtered by the “Where” clause, or by the entire packets when “Where” is omitted. We provide 5 basic built-in aggregate functions similar to SQL languages: COUNT, SUM, MIN, MAX and AVG.

In the sophisticated network measurement context, the built-in aggregate functions are far less than enough. EQuery therefore provides customizable aggregate functions - detailed syntax is shown below.

```

ex ∈ Expr := v | h | s | f |  $\vec{e}x$ 
x ∈ Pred := [h,s,f]  $\bowtie$  ex    #field test
                    st[ex]  $\bowtie$  ex    #state test
a ∈ Action := st[ex] += ex #state-
                    st[ex] -= ex #modification
 $\bowtie$  ∈ Optr := = | > | < | >= | <= |  $\sim$  #operators

```

We generalize the definition of an aggregate function as follows.

```

create aggregate agg_name(agg_st, [h, s]):
    if x then a1 else a2

```

C. Triggering a Subsequent Query

Event e . A PE is indicated by the “Every” clause whereas a CE is more complicated. Throughout the remainder of this paper, we refer to an *Event* as a CE unless specified otherwise. An event often indicates a network situation that may require further actions, or in the context of

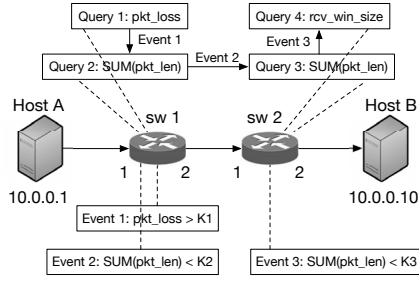


Fig. 3. Example network for event mechanism.

measurements, additional queries. As shown in Fig. 2, an event e is defined based upon the results of another query q_1 in terms of a test on the measured fields within the corresponding query. If the test evaluates to “true”, then the query q_2 would execute. The expression $(q : \text{true})$ implies that query q_2 will be executed once query q_1 finishes, and is independent of any specified events. Conversely, a query with a “When(e)” clause will not be executed unless the event e occurs.

Example: use events to diagnose TCP packet loss along a flow path. As shown in Fig. 3, in order to detect packet loss and its root causes along the path from Host A to Host B, we can describe the process as follows.

```

q1:Select 5-tuple, pkt_loss Groupby 5-tuple
    Where 5-tuple-predicate
q2:Select sw_id, SUM(pkt_len) Where sw_id=sw1 & pt_in=1
    When q1:pkt_loss > K1
q3:Select sw_id, SUM(pkt_len) Where sw_id=sw2 & pt_in=1
    When q2:SUM(pkt_len) =< K2
q4:Select 5-tuple, win_size Where 5-tuple-predicate
    When q3:SUM(pkt_len) =< K3

```

Fig. 4. Example of event-driven queries (the events are in bold).

D. Query Composition

Atomic queries can be composed together either sequentially or concurrently to fulfill sophisticated measurement tasks. We call this process query composition, which can be either “simple compositions” or “event-driven compositions”. For the sake of convenience, we use the following terminology:

Atomic query (a-query): An a-query indicates a single Query q described in Fig. 2, which contains only a single “Select($h,s,f(h,s)$)” clause. In our compilation stage (§IV), an a-query will be used as the minimum granularity to construct an NFA and generate the low-level rules/states.

Composite query (c-query): A sophisticated measurement task is regarded as a c-query that comprises a set of a-queries $\{q_1, q_2, \dots, q_k\} (k > 1)$. These a-queries can be specified with or without temporal order or event dependencies. A c-query will be analyzed and decomposed to a-queries, which can be executed sequentially, concurrently, or a combination of both.

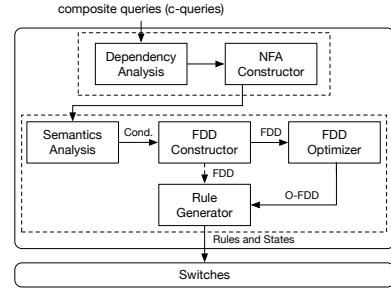


Fig. 5. Overview of compiling phases.

1) *Simple Composition: $q_1 \gg q_2$:* For sequential composition we employ the syntax $q_1 \gg q_2$ to annotate that query q_1 will be executed first, while query q_2 will execute upon the completion of q_1 . In order to identify the sequential dependency between q_1 and q_2 , we add a special clause “**When q_1 :true**” into the query q_2 . These two queries are executed in different time durations, thus there are no conflicts to write/read the same data plane states.

q_1 & q_2 : For parallel composition we employ the syntax q_1 & q_2 to mean that q_1 and q_2 will execute in parallel. In this case, there is no dependency between q_1 and q_2 . For such a c-query, q_1 and q_2 generally will not read/write the same data plane state on the same switch concurrently. If q_1 and q_2 work on the same state, we can merge these two queries. If two queries are submitted without specifying a special dependency, they will execute concurrently. Based on these two combinations, we can derive more complex compositions, e.g., $q_1 \gg (q_2 \& q_3)$, $q_1 \gg q_2 \gg q_3$.

2) *Event-driven Composition: $q_1 \gg q_2 - e$:* Event-driven sequential composition $q_1 \gg q_2 | e$ is similar to $q_1 \gg q_2$. The only difference is that q_2 will start and q_1 will terminate once event e occurs. If event e does not occur during the execution time of q_1 , q_2 will never execute. In order to identify the event-driven sequential dependency between q_1 and q_2 , we apply the clause “**When(e)**” within the query q_2 .

$q_1 \gg (q_2 \& q_3) - e$: According to the combinations specified above, we can derive a c-query whereby q_1 executes first followed by the parallel execution of q_2 and q_3 , once event e occurs.

By the means of composition, we can achieve even more complex queries that combine a-queries in different ways.

IV. COMPILATION

We design a front-end compiler to analyze the temporal and spatial properties of a c-query by breaking it down into two stages. The first stage translates a c-query into a non-deterministic finite automaton (NFA) representing a set of a-queries and their dependencies. The NFA is used by the controller to record and control the workflow for the query operations. The second stage utilizes forwarding decision diagrams (FDD) to create an intermediate representation

of the query operations in NetASM, which is compatible with programmable network devices, e.g., FPGAs, RMT, Intel's FlexPipe, NPUs.

A. Dependency Analysis

Given a c-query, the compiler first performs dependency analysis in order to determine the ordering constraints and triggering conditions among the decomposed a-queries. There are two types of dependencies.

Time dependency: A time dependency indicates that two a-queries $q1$ and $q2$ can be combined by the clause “When $q1:\text{true}$ ”. $q2$ temporally depends on $q1$, such that $q2$ cannot be executed before $q1$ finishes. The execution time of $q1$ is specified by the clause “Within(i)”, with a default duration of 5 minutes. Two queries will execute with independent time durations, so that $q1$ and $q2$ can read/write the same data plane state without conflicts.

Event-based dependency: An event-based dependency implies that two a-queries, $q1$ and $q2$, can be combined by the clause “When (e)”. In this case, $q2$ depends on the event e , in the way that $q2$ will be executed only when event e occurs. Specially, “When $q1:\text{true}$ ” describes a scenario that we use event-based dependency to implement time dependency.

B. Constructing the NFA

With the dependencies implied by a c-query, we can leverage the NFA model to explain its formal semantics. We choose NFA because our language semantics possesses the exact characteristics of a NFA: a finite set of states, the input alphabet, the transition functions, and the start/-final states. With the NFA model, each c-query can be represented by a composition of automata to record the execution states and control the workflow.

Formally, an NFA automaton, $A = (Q, E, \theta, qs, qf)$, consists of a set of states, Q , a set of directed edges, E , a set of formulas, $\theta: Q \times E \rightarrow P(Q)$, that label the edges to determine the state transition, a start state, $qs \in Q$, and a set of final states, $qf \subseteq Q$. We present the compilation rules for automatically translating a c-query into an NFA as follows.

States: For an NFA automaton, each state $qi (i = 1, 2, \dots)$ represents the running a-query qi included within a c-query. The start state, $qs \in Q$, is where the c-query begins. When the state is qi , it implies that the a-query qi is being executed. When qi has finished or an event is triggered, the state will transition to the next state qj , which implies that the system is ready to run the next a-query. The final states, $qf \subseteq Q$, represent the completion of the c-query, resulting in the creation of the final measurement data.

Edges: Each state is associated with a number of edges, representing the “events” that can occur at the state. Every state (except the start and final states) has a self-looping edge whereby no event has occurred. Note that the special ε -edge does not consume any input conditional events (CEs) but simply attempts to proceed when the

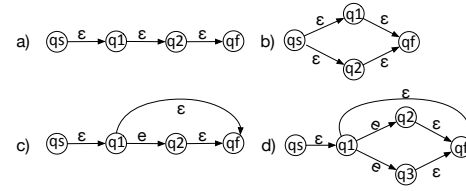


Fig. 6. NFAs of different compositions. a) $q1 \gg q2$, b) $q1 \& q2$, c) $q1 \gg q2|e$, d) $q1 \gg (q2 \& q3)|e$

current query has finished. In other words, an ε -edge only specifies a temporal dependency as represented by the clause “When $q: \text{true}$ ”. In this paper, the selected event for the start state is an ε -edge, which implies that once the c-query starts, the state will transition from the start state to the first state.

Transition function: An NFA will move from state to state according to the transition function θ , which can be defined by a state transition table. The transition rules can be derived from dependency analysis in §IV-A.

Example: Fig. 6 shows the NFA structures of the basic compositions proposed in §III-D. With reference to Fig. 6d), once the c-query begins, the state will transition from the start state to the first state $q1$. If event e occurs during the execution of $q1$, the state will move from $q1$ to $q2$ and $q3$, simultaneously. Otherwise, the state will transition to the final state qf once query $q1$ has finished. In the former case, the state will move from $q2$ and $q3$ to the final state qf when the queries $q2$ and $q3$ are respectively finished.

C. Semantics Analysis

In this paper, queries are constructed by utilizing the proposed EQuery language, which is a high-level abstract language and cannot be directly installed and executed on switches. Such queries must be translated into low-level rules and states in order to monitor network performance. Firstly, we parse each a-query contained within a c-query and translate it into a set of explicit conditionals, e.g. “if t then $d1$ else $d2$ ”. Afterwards, these conditionals will be used to construct an FDD as an intermediate representation of the a-query before deriving the final low-level rules/states on network switches.

D. Converting to an FDD

For a graph-based FDD [11], [4], each root node represents a test on a set of packet fields, packet locations, or state variables. The leaf nodes represent a set of action sequences, rather than merely a ‘true’ or ‘false’ condition, as is the case with a binary decision diagram (BDD) [3]. Each interior node has two successors: ‘true’, which determines the rest of the forwarding decision process for inputs that pass the test, and ‘false’ for failed cases. Furthermore, each lead node represents a set of actions. Formally, an FDD is either a branch ($x?d1 : d2$), where x is a test and $d1, d2$ are FDDs, or a set of actions $\{a1, \dots, an\} \in \text{Action}$. Each branch can be regarded as a

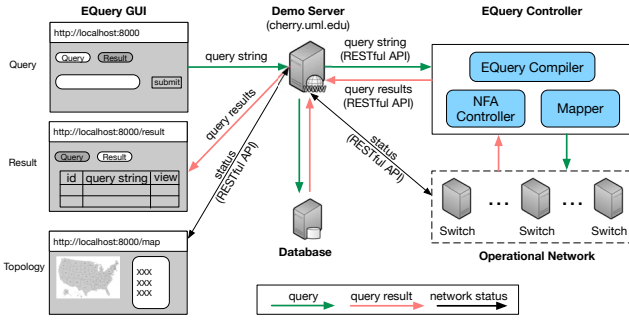


Fig. 7. The EQuery evaluation testbed

conditional. If the test x holds true for a given packet, pkt , then the FDD continues processing packet pkt using $d1$. If x does not hold true, then the FDD continues processing packet pkt using $d2$. As a result, we simply map the derived conditionals of each a-query into an FDD.

V. EVALUATION

In this section, we discuss the EQuery framework implementation and demonstrate the use of EQuery in real-world network measurement scenarios.

A. Implementation

We implement the EQuery language and the EQuery controller in Python. The compiler's output for each switch is a set of switch-level instructions in NetAMS intermediate representation. The EQuery GUI as shown in Fig. 7 is implemented with HTML and Javascript and the demo server runs MySQL database. The demonstration website can be found at cherry.uml.edu. The query page is the interface for users to submit their measurement tasks (c-queries); the result page returns the query results; and the topology page demonstrates real-time status of the measurement hardware (programmable switches or x86 servers).

B. Testbed Network

As shown in the topology demo page cherry.uml.edu/map, the current testbed network consists of 3 working measurement spots at three critical network exchange points: StarLight [2] at Northwestern University, University of Kentucky, and AMPATH [1] at Florida International University. We deploy measurement hardware at these 3 places and actively collect real-time network data controlled by EQuery measurement jobs.

C. Use Cases

We want to demonstrate the effectiveness of EQuery framework solving real-world problems by introducing a collection of measurement tasks: 1) detect flow volume over a link; 2) detect heavy hitters; 3) check flow volume and link throughput sequentially; 4) detect packet loss; 5) pin-point packet loss position along a path; 6) detect DDoS attacks. Though this is not a complete list of measurement

tasks, it covers a great dimension of trouble-shooting scenarios for a typical operational network. The detailed demonstration code by using EQuery can be found in the Appendix, and the query results can be obtained by submitting the EQuery code through the query page. It worth note that we implement all use cases with less than 10 lines of EQuery code for each.

VI. CONCLUSION

In this paper we presented EQuery, an programmable framework suitable for achieving event-driven network measurement tasks. EQuery renders a declarative query language to derive network-wide information and trigger chained network tasks. In addition, the EQuery compiler can automatically translate the declarative queries into low-level rules and states onto network devices with temporal and spatial optimizations. We have implemented the EQuery framework as a portion in an NSF-funded project which involves six institutions. EQuery has demonstrated its effectiveness in the improvement of network fault diagnosis, attack prevention, and traffic engineering.

ACKNOWLEDGMENT

This work is supported in part by the National Science Foundation (No. 1738965, No. 1547428, No. 1541434, No. 1440737 and No. 1450996).

REFERENCES

- [1] Ampath: The international exchange point for research and education networking in miami.
- [2] Starlight: The optical star tap.
- [3] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.
- [4] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.
- [5] B. Claise. Cisco systems netflow services export version 9. 2004.
- [6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.
- [7] S. Narayana, A. Sivaraman, V. Nathan, M. Alizadeh, D. Walker, J. Rexford, V. Jeyakumar, and C. Kim. Hardware-software co-design for network performance measurement. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 190–196. ACM, 2016.
- [8] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling path queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 207–222, Santa Clara, CA, 2016. USENIX Association.
- [9] P. Phaal, S. Panchen, and N. McKee. Immon corporation's sflow: A method for monitoring traffic in switched and routed networks. Technical report, 2001.
- [10] M. Shahbaz and N. Feamster. The case for an intermediate representation for programmable data planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 3:1–3:6, New York, NY, USA, 2015. ACM.
- [11] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha. A fast compiler for netkat. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 328–341, New York, NY, USA, 2015. ACM.

APPENDIX

A. Use Case Demonstration

TABLE I

EXAMPLES OF USE CASES (BASED ON THE TOPOLOGY OF FIG. 3).

Example	Query code	Composition	Description
Flow volume over a link	q1: Select ipv4_src, ipv4_dst, AVG(pkt_len) Where sw_id=sw1 & in_pt=1 Groupby ipv4_src, ipv4_dst	N	Query per-flow average volume over a specified link.
Heavy hitter detection	q2: Select src_ip, dst_ip Where hh_counter >1G Groupby src_ip, dst_ip create aggregate hh_counter (agg_hh): agg_hh[src_ip][dst_ip] += pkt.pkt_len	N	Source-destination IP address pairs whose traffic volume is more than a pre-specified threshold.
Flow volume & link throughput, sequentially	q3: Select src_ip, dst_ip, SUM(pkt_len) Where src_ip=10.0.0.1, dst_ip=10.0.0.10 Groupby ipv4_src, ipv4_dst q4: Select SUM(pkt_len) Where sw_id=sw1 & in_pt=1 When q3:true	q3 \gg q4	Execute two queries sequentially.
Packet loss detection	q5: Select src_ip, dst_ip, pkt_loss Groupby src_ip, dst_ip q6: Select src_ip, dst_ip, window_size Groupby src_ip, dst_ip When q5:pkt_loss>1000 q7: Select src_ip, dst_ip, SUM(pkt_len) Groupby src_ip, dst_ip When q5:pkt_loss>1000 create aggregate pkt_loss (agg_pl, agg_seq) if pkt.seq < agg_seq then agg_pl++ else agg_seq = pkt.seq + pkt.payload_len	q5 \gg (q6&q7)—e	When the packet loss of TCP flows exceed a threshold, TCP window size and volume of the flow will be measured.
Loss localization along a path	q8: Select src_ip, dst_ip, pkt_loss Where ipv4_src=10.0.0.1 & ipv4_dst=10.0.0.10 Groupby src_ip, dst_ip q9: Select src_ip, dst_ip, SUM(pkt_len) Where sw_id=sw1 & in_pt=1 Groupby src_ip, dst_ip When q8: pkt_loss>1000 q10: Select src_ip, dst_ip, SUM(pkt_len) Where sw_id=sw2 & in_pt=1 Groupby src_ip, dst_ip When q8: pkt_loss>1000	q8 \gg (q9&q10)—e	Measure throughput in different locations along a path when the packet loss of a TCP flow exceeds a threshold.
DDoS detection	q11: Select dst_ip, COUNT(SYN) Groupby dst_ip q12: Select dst_ip, diff_syn_ack Groupby dst_ip When q11:COUNT(SYN)>1000 create aggregate diff_syn_ack (agg_diff,agg_syn,agg_ack): if pkt.SYN=1 & pkt.ACK=0 then agg_syn++ if pkt.SYN=1 & pkt.ACK=1 then agg_ack++ agg_diff = agg_syn - agg_ack	q11 \gg q12—e	Measure the asymmetry between incoming TCP packets with SYN flags and outgoing TCP packets with SYN and ACK flags.