# Toward Composable Network Traffic Measurement

Andrea di Pietro* Felipe Huici§ Nicola Bonelli* Brian Trammell‡
Petr Kastovsky* Tristan Groleat† Sandrine Vaton† Maurizio Dusi§

*University of Pisa, §NEC Europe Ltd., ‡ETH Zurich
†Institut Telecom, *INVEA-TECH

***Abstract*—As the growth of Internet traffic volume and diversity continues, passive monitoring and data analysis, crucial to the correct operation of networks and the systems that rely on them, has become an increasingly difficult task. We present the design and implementation of Blockmon, a flexible, high performance system for network monitoring and analysis. We present experimental results demonstrating Blockmon's performance, running simple analyses at 10Gb/s line rate on commodity hardware; and compare its performance with that of existing programmable measurement systems, showing significant improvement (as much as twice as fast) especially for small packet sizes. We further demonstrate Blockmon's applicability to measurement and data analysis by implementing and evaluating three sample applications: a flow meter, a TCP SYN flood detector, and a VoIP anomaly-detection system.**

## I. Introduction

Two salient trends have dominated Internet-scale monitoring and measurement over the past decade: the continued growth of the Internet, both in terms of attached nodes and total data transferred; and the diversification of devices attached to and applications running on the network. These trends are evidence of the Internet's success as a platform for communication and innovation, but they continue to increase the difficulty of the monitoring, measurement, and traffic analysis activities crucial to ensure security, quality of service, and future operational planning.

These challenges point to the need for a high-performance, yet easily-extensible solution. In this paper we present Blockmon, a system for supporting high-performance *composable measurement*: building network measurement applications out of small, discrete blocks. [1]

Our contributions include (1) a new, flexible design for composable network measurement and data analysis; (2) enabling parallelization of measurement processing and full use of modern multi-core hardware, ensuring high performance for line-rate measurement in user-space; (3) introduction of a multi-slice memory allocator that reduces CPU utilization and increases throughput when compared to the standard Linux 3.0 allocator; (4) application of a wait-free rotating queue mechanism and new C++11 features to minimize lock contention and copying overhead, further increasing performance; and (5) adapter code to support fast, 10Gb packet capture technologies such as PF_RING, PFQ and COMBO hardware cards.

This work was partly funded by the EU FP7 DEMONS (257315) project.
[1]Blockmon is available at http://blockmon.github.com/blockmon

### A. Related Work

The modular principles in Blockmon were inspired by the Click modular router [13]. However, Click focuses on packets and so cannot easily do more advanced types of processing such as maintaining TCP connections, inter-acting with databases, or any number of other actions relevant to monitoring and data analysis.

While lots of measurement tools exist, few are flexible enough to adapt to new application areas or changing traffic. One such programmable tool is CoMo [9], which introduced the concept of a monitoring plugin by which a monitoring application is written as a set of callback functions to be called by the framework. Blockmon generalizes this concept to a message-passing architecture, described in section II. ProgME [17] specifies a runtime-programmable network flow aggregator, configured using a declarative language based upon set algebra. RTC-Mon [7] provides a framework for building monitoring applications like CoMo, but its architecture is similarly limited to the use cases for which it was designed. In section IV we present results to show how Blockmon compares to some of these.

Performance work in network measurement has largely focused on getting packets off the wire as fast as possible, as in [2] and PF_RING [6]; this latter even contains a basic, if inflexible, programmable measurement system. For 10Gb/s and faster links, packet capture is often enhanced through hardware acceleration as on Endace DAG or Napatech cards. NetworkDVR [3] decides early in the capture process which packets to capture and which to ignore; this can also be performed by packet capture and offload cards such as INVEA-TECH's COMBO platform. More recently, advances in commodity hardware have made 10Gb/s capture in software possible: Netmap [16], PF_RING DNA [4] and PFQ [15] are examples of this.

## II. Base System

Blockmon provides a set of units called *blocks*, each of which performs a certain discrete processing action, for instance parsing a DNS response, or counting the number of distinct VoIP users on a link. The blocks communicate with each other by passing *messages* via *gates*; one block's output gates are connected to the input gates of other blocks, which allows runtime indirection of messages. A set of interconnected blocks implementing a measurement application is called a *composition*.

Compositions are defined using an XML format which lists the blocks and their configuration parameters, then lists the connections among their gates. The Blockmon core and the blocks themselves are implemented in C++, and the system is controlled at runtime using a simple, Python-based command-line interface (CLI). Blockmon also comes with a GUI that allows users to draw compositions and that can automatically generate and install XML compositions from users' input.

### A. Blocks

As mentioned, a block performs a discrete processing action. All blocks are derived from a common superclass. New blocks simply inherit from this class and implement at least two methods: `configure`, which receives XML representing the block's configuration parameters, and `receive_msg` which is called when a message arrives at the block. Blocks can also be invoked on periodic or one-shot timers via the `handle_timer` method, and can perform high-frequency but non-periodic asynchronous work in the `do_async` method; this last method is mainly provided for source blocks (e.g., packet capture or message import via IPFIX), which send messages but do not receive them.

### B. Gates and Scheduling

A gate is essentially a named point on a block to allow connections between blocks at configuration time: compositions are built by defining connections between specific gates on one block and a specific gate on another. Blocks send messages via output gates, and receive messages via input gates.

There are in essence two types of input gates, which lend Blockmon its scheduling flexibility. Blockmon supports *direct* and *indirect* message passing. In the former, the sending block directly calls the receiving block's `receive_msg` method: the input gate is in this case essentially a function call. This is fast but inflexible: the receiving block runs in the sending block's thread, which will be busy with the receiving block until it finishes.

The alternative is indirect message passing, which is mediated by a wait-free, rotating queue described in section III-B. With indirect message passing, each block is separately scheduled in different thread pools on different CPU cores; this allows truly parallel processing on multi-core systems without blocking or locking overhead, key to Blockmon's performance, as we will show in section IV.

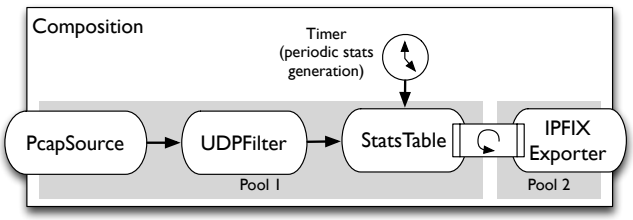


Fig. 1. Example composition showing block invocation

To make things more concrete, figure 1 shows an example of how the different block and scheduling types are used in a simple composition. In this case, *PcapSource* is a source block and therefore indirectly scheduled, capturing packets from a network interface in its `do_async` method, and sending Packet messages directly to *UDPFilter*. This block filters for UDP packets, and sends results directly to *StatsTable*, which keeps statistics for received packets. All three of these blocks run in the same thread pool (see section III-A); that is, when a packet is received, PcapSource creates a Packet message and invokes `receive_msg` on *UDPFilter*, which invokes `receive_msg` on *StatsTable* if the packet is a UDP packet.

*StatsTable* registers a periodic timer on configuration; Blockmon's scheduler then periodically calls `handle_timer` on *StatsTable*, sending a message containing statistics to *IPFIXExporter*. The latter is indirectly scheduled, and runs in a separate thread pool from the source/counter in order to isolate network export from packet capture and counting. Periodically, the scheduler will dequeue the pending messages from the rotating queue associated with the input gate on *IPFIXExporter*, and invoke `receive_msg` for each.

### C. Messages

The actual communication between blocks is in terms of messages. These, like blocks, are derived from a common superclass; pointers to messages are passed via the gates. The **Message** class provides a basic interface for identifying message types, and for supporting import and export of messages in order to connect compositions across nodes. Messages are constant in order to ensure that they can be shared without contention among multiple blocks concurrently, and provide a tagging mechanism to allow Blocks to add small bits of data to a message in a thread-safe manner without incurring too large a performance overhead.

## III. Performance Mechanisms

### A. Thread Pools and CPU Pinning

Blockmon is multi-threaded in order to take advantage of multi-core CPUs. The assignment of activities to threads and threads to CPU cores can have a large impact on performance [5]. To leverage this, Blockmon schedules work in thread pools. Each block is assigned to a pool via the composition, and pools can be pinned to specific cores.

This model allows flexibility in terms of which block is executed on which CPU core. Since the amount of code in a single block is relatively small [2], this model gives fine granularity and control in terms of what runs on which core.

### B. Wait-Free Rotating Queues

Since indirect invocation, the key to parallelization in Blockmon, relies on passing messages from one thread to another quickly and with minimal lock contention, the design of the queues for the input gates of indirectly scheduled blocks is crucial for performance. To this end, we apply the queue design described in [15]. The gates are implemented as two queues, with one queue for writing (by the sender's thread) and one queue for reading (by the receiver's thread). The queues

[2]large blocks can be broken down; the message-passing overhead, as we will show, is small.

are rotated using atomic primitives provided by C++11 each time the Blockmon scheduler invokes an indirect block. Blockmon's wait-free rotating queues are wait-free for producers, which is crucial to keeping up with bursts of packets at line rate. Consumers only have to wait for any pending writes to complete after a swap. Experimentation shows this is a low-probability occurrence with negligible impact on performance.

### C. Memory Allocation

Since each packet captured requires a Blockmon message, dynamic memory allocation overhead [8] is a major component of the time used by compositions which read packets at line rates. Blockmon reduces this overhead by batching memory allocations of buffers used by each message into larger chunks. The batch allocations are reference-counted, such that they are automatically freed with the destruction of the last buffer in the batch. Note that this optimization is only possible with the new shared ownership constructor of the shared pointer class supported by C++11 [12], which avoids the allocation of a reference-count metadata structure for each buffer.

In addition, Blockmon provides a multi-slice allocator: each message containing nontrivial amounts of data (e.g., Packets with full payload) uses chunks allocated from layered slices, and the slice layering is chosen such that common access patterns will exploit cache locality. The allocator is templated, making it easy to create different slices (e.g., an allocator could have an IP header slice and a cap lengthed' payload slice). In section IV we present the improved packet rates resulting from this, and show that the multi-slice allocator outperforms the Linux 3.0 allocator both in terms of rates but also in terms of reduced CPU load.

### D. Efficient Message Transfer

Messages are passed using C++11 `std::shared_ptr`, so that the same message can go through different processing paths in a composition without spurious allocations or copies, with automatic reference counting. However, copying a shared pointer involves atomically decrementing and incrementing the reference counter, which can lead to high contention when a message moves from core to core. Therefore, Blockmon adopts the new C++11 object-move semantic, which allows for the transfer of shared pointers without reference count updates. This small change results in a significant performance benefit, as shown in section IV.

## IV. Evaluation

### A. Experimental Setup

We use a pair of servers, one running a traffic generator and the other one running Blockmon, directly connected via 10Gb wired interfaces. Each server costs about $2000, has a 2.66Ghz 6-core Intel Xeon X5650 with HyperThreading enabled, 12GB of DDR3 RAM, an Intel 82599EB network interface, and runs Linux kernel version 2.6.39. For packet capture we use the *PFQSource* block which yields the best performance out of the source blocks currently available in Blockmon. Unless otherwise stated, all experiments in this section are performed with 64-byte packets to maximize strain on the system. Throughout we use Mp/s to mean millions of packets per second.

### B. Comparison to Existing Measurement Systems

Before evaluating the different aspects of Blockmon's performance we wanted to measure how it stacked up against other existing programmable measurement systems. For this comparison we chose CoMo (version 1.5) since it is perhaps the most recognized system in this space; and Click (version 2.0.1), since Blockmon takes inspiration from its modular architecture. In order to compare such disparate systems, we needed to pick an application that would be simple enough to implement in each, but that would stress the systems' performance. As a result, we implemented a simple meter that keeps per-flow byte and packet counts. For further comparison, we also put YAF [10] version 2.1.2, a flow meter, to the test.

Although some of the measurement platforms that we used as a benchmark can make use of multiple threads, they do not support capture parallelization through explicit use of multiple hardware queues. Therefore, in order to be fair to them, we used an experimental layout where multiple kernel contexts fetch packets from multiple hardware queues and hand them over to a single capturing socket: in practice, this increases performance by using multiple kernel threads to feed packets to a single user space thread. For this, we adopted the optimal scheme described in [14], where the kernel context runs on all of the physical cores but one, which is devoted to the user-space monitoring process.

For this evaluation, we sent a synthetic stream of packets representing 1024 simultaneous UDP flows at line rate via a 10Gb/s Ethernet interface, varying the size of packets from 64 to 1500 bytes, and measured the data rate reported by each system.

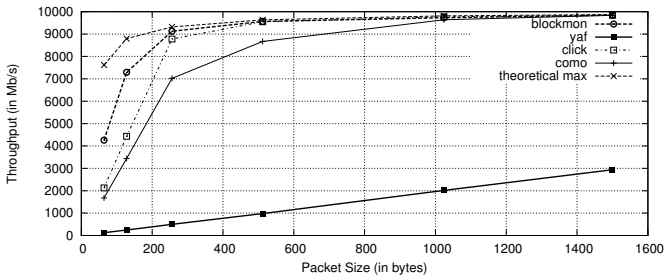


Fig. 2. Performance comparison between Blockmon and other systems.

As shown in figure 2, Blockmon outperforms the other systems for minimum-sized packets up to 256-byte ones, including a rate twice as fast as Click's for 64-byte packets. Blockmon also performs better than CoMo even for larger packet sizes, and roundly outperforms YAF, which is flow-table limited on this workload.

### C. Performance Mechanisms

To test the effect of the batch allocation and rotating queue optimizations, we created the single counter composition shown in figure 3. Each of the multiple capture blocks services

one of the hardware queues on the Intel NIC, and feeds packets into a single counter block. We further assigned one CPU core to each capture block, and one for the counter. The single counter creates a bottleneck that allows us to measure the effects of the optimization.

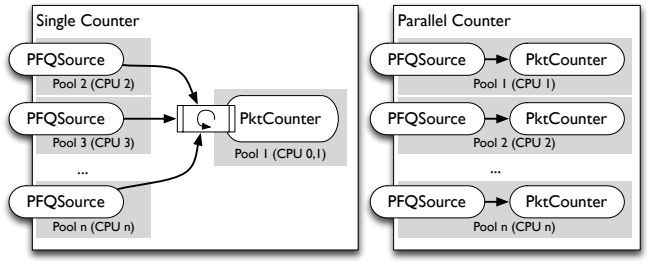


Fig. 3. Compositions used in testing Blockmon performance mechanisms

Running this set-up produces the results in figure 4. Two logical cores are reserved for the counter block. We start our measurements with 2 cores (i.e., 2 sniffers) since setting RSS [11] on the Intel NIC to 1 causes the driver to use all of the NIC's hardware queues instead of a single one.

Applying each of the two optimizations in turn, as is the case for the "batch only" and "rotating only" curves, results in sub-optimal performance, as each optimization removes only one of the two performance bottlenecks: performance is still bound by the remaining bottleneck. Thus, the rotating queue optimization curve does not show much improvement as the number of sniffers increases and contention on the single counter becomes more severe since the memory allocation bottleneck remains. Conversely, the batch optimization curve holds steady but decreases slightly as the contention on the packet counter's queue increases with the number of capture blocks. This effect is confirmed by the top curve: removing both bottlenecks provides a significant bump in performance. Note that the slight dip in performance at 6 cores is due to the fact that the last 6 cores are not physical CPUs, but rather emulated by means of the Intel HyperThreading technology. As a result, their contribution is lower with respect to that of actual cores.

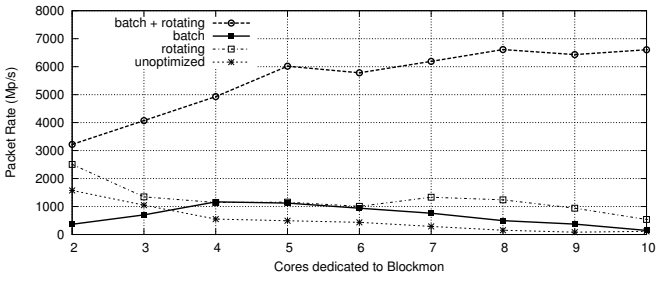


Fig. 4. Effect of batch allocation and rotating queue optimizations

Parallelizing packet counting (figure 3), removes the single counter bottleneck, and allows us to isolate the impact of the batch allocation optimization. Batch allocation alone speeds up Blockmon by about 30% depending on the number of cores.

To show the dependence of performance on packet size, we tested the parallel composition in figure 3 with varying packet sizes. As shown in figure 5, four cores are sufficient to capture and count 256-byte packets at offered rate; and eight cores for 128-byte packets. Minimum-sized packets can be processed at 12 Mp/s, close to line rate.

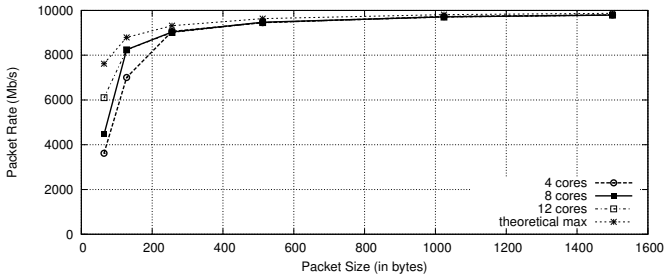


Fig. 5. Performance for different packet sizes using the parallelized composition and the batching optimization.

To quantify the overhead introduced by Blockmon itself, we created a simple stand-alone test application that uses the PFQ engine to capture and count packets, and compared it to Blockmon using the parallel composition in figure 3. The results show that for this simple packet application, Blockmon's flexibility has a cost of about 10% - 15%, depending on the number of cores dedicated to measurement.

We assessed the improvement brought by c++11 move semantics by writing a separate test program (due to the deep integration of move semantics in the Blockmon core, running this test on Blockmon itself would have been prohibitively difficult). The test program emulates the Message lifecycle: allocation on one core, queueing (long enough for data to be swapped out of the cache) and deallocation on a set of different cores. Comparing move-based to copy-based message passing results in up to a 20% reduction of message processing time, depending on concurrency.

### *D. Multi-Slice Allocator*

As mentioned, Blockmon comes with a novel multi-slice allocator. To test its performance, we used a simple composition consisting of twelve of the following block chains (one per processor on our system): *PFQSource* → *MessageCounter* → *FlowMeter*, where the measurements are done by *MessageCounter*.

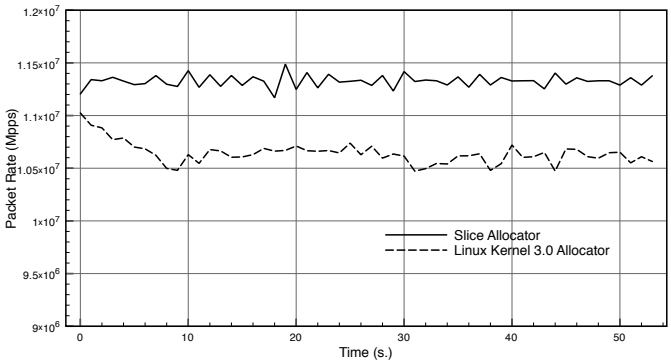


Fig. 6. Multi-Slice and Linux 3.0 Allocator Performance

We conducted this experiment for both the multi-slice allocator and included the Linux 3.0 allocator for comparison purposes. When offered minimum-sized packets at line rate, the multi-slice allocator utilizes the CPU cores roughly 30%

less than the Linux allocator. Unsurprisingly, this translates to better packet rates, about 0.6 Mp/s more on average (figure 6).

## V. Applications

In order to test the performance of Blockmon under more realistic workloads, we implemented three different applications over it; due to space constraints we only give a brief description of each.

**Heavy Hitter Statistics**: A simple application that keeps per-flow byte and packet counts, periodically exporting information about flows whose counts go over a configurable threshold. Sending synthetic traffic generated at 6.1 Gb/s using only minimum size packets (an unrealistic scenario), Blockmon was able to process packets at a rate of 3.8Gb/s. A hardware-accelerated composition using an INVEA-Tech COMBO card performed this task at line rate.

**SYN Flooding Detection**: An application that detects SYN flooding attacks by using a Count Min Sketch (CMS) to store the number of TCP SYN packets sent to each IP address, and the multi-channel NP-CUSUM algorithm to watch all the values of the sketch and detect any abrupt changes in the number of TCP SYN packets sent to a particular IP address. Even with 10% of all traffic being TCP SYN packets (more than one million SYN packets per second), Blockmon was able to perform anomaly detection at a rate of approximately 5.5 Gb/s while detecting all of the (synthetic) attacks we introduced.

**VoIP Anomaly Detection**: To demonstrate that existing applications can be easily ported to Blockmon and derive performance improvements from it, and to show its flexibility in using something other than packet- and flow-based data sources, we ported VoIPSTREAM [1], a system for SIP telephony abuse detection, to Blockmon; the implementation was done by an engineer not part of the Blockmon core team in just two weeks. The port resulted in a 51% increase in performance.

## VI. Conclusions and Future Work

We have presented Blockmon, a modular, high-performance, composable network traffic analysis system and its performance evalution. The results are promising, with Blockmon able to process most packet sizes up to line rate with nontrivial applications and outperforming existing systems in the area. Blockmon allows very flexible scheduling with its direct and indirect block invocation as well as threadpools. However, this flexibility means that it can be difficult to decide how to choose the "right" scheduling such that a given composition yields good performance. Metrics such as queue occupancy, and CPU, memory, and cache load could be used to automatically (and adaptively depending on traffic patterns) select the best scheduling and pooling configuration; deriving an algorithm to do this is future work.

## References


[1] Bianchi, G., d'Heureuse, N., and Niccolini, S. On-demand time-decaying bloom filters for telemarketer detection. *SIGCOMM Comput. Commun. Rev. 41* (September 2011), 5–12.

[2] Braun, L., Didebulidze, A., Kammenhuber, N., and Carle, G. Comparing and improving current packet capturing solutions based on commodity hardware. In *Proceedings of the 10th annual conference on Internet measurement* (New York, NY, USA, 2010), IMC '10, ACM, pp. 206–217.

[3] Chang, C.-W., Gerber, A., Lin, B., Sen, S., and Spatscheck, O. Network DVR: A programmable framework for application-aware trace collection. In *Proceedings of the Passive and Active Measurement Conference (PAM) 2010* (Zürich, Switzerland, mar 2010).

[4] Deri, L. Direct NIC Access. http://www.ntop.org/products/pf_ring/dna/, December 2011.

[5] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., and Mathy, L. Towards high performance virtual routers on commodity hardware. In *Proceedings of ACM CoNEXT 2008* (Madrid, Spain, December 2008).

[6] Fusco, F., and Deri, L. High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th annual conference on Internet measurement* (New York, NY, USA, 2010), IMC '10, ACM, pp. 218–224.

[7] Fusco, F., Huici, F., Deri, L., Niccolini, S., and Ewald, T. Enabling high-speed and extensible real-time communications monitoring. In *IM'09: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management* (Piscataway, NJ, USA, 2009), IEEE Press, pp. 343–350.

[8] Han, S., Jang, K., Park, K., and Moon, S. Packetshader: a gpu-accelerated software router. In *Proceedings of ACM SIGCOMM 2010* (New Delhi, India, September 2010).

[9] Iannaccone, G. Fast prototyping of network data mining applications. In *Passive and Active Measurement Conference 2006* (Adelaide, Australia, mar 2006).

[10] Inacio, C., and Trammell, B. YAF: Yet Another Flowmeter. In *Proceedings of the 24th USENIX Large Installation System Administration Conference (LISA '10)* (San Jose, California, nov 2010), pp. 107–118.

[11] Intel. Receive side scaling on Intel Network Adapters. "http://www.intel.com/support/network/adapter/pro100/sb/cs-027574.htm", July 2011.

[12] ISO/IEC JTC1/SC22/WG21. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, February 2011.

[13] Kohler, E., Morris, R., Chen, B., Jahnotti, J., and Kasshoek, M. F. The click modular router. *ACM Transaction on Computer Systems 18*, 3 (2000), 263–297.

[14] N.Bonelli, Pietro, A. D., Giordano, S., and Procissi, G. Packet capturing on parallel architectures. In *IEEE workshop on Measurements and Networking* (2011).

[15] N.Bonelli, Pietro, A. D., Giordano, S., and Procissi, G. On multi–gigabit packet capturing with multi–core commodity hardware. In *Passive and Active Measurement conference (PAM)* (2012).

[16] Rizzo, L., and Landi, M. netmap: memory mapped access to network devices. In *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 422–423.

[17] Yuan, L., Chuah, C.-N., and Mohapatra, P. ProgME: towards programmable network measurement. *SIGCOMM Comput. Commun. Rev. 37*, 4 (2007), 97–108.