Scalable Hybrid Stream and Hadoop Network Analysis System

Vernon K. C. Bumgardner
Department of Computer Science
University of Kentucky
Lexington, Kentucky, USA
cody@uky.edu

Victor W. Marek
Department of Computer Science
University of Kentucky
Lexington, Kentucky, USA
marek@cs.uky.edu

ABSTRACT

Collections of network traces have long been used in network traffic analysis. Flow analysis can be used in network anomaly discovery, intrusion detection and more generally, discovery of actionable events on the network. The data collected during processing may be also used for prediction and avoidance of traffic congestion, network capacity planning, and the development of software-defined networking rules. As network flow rates increase and new network technologies are introduced on existing hardware platforms, many organizations find themselves either technically or financially unable to generate, collect, and/or analyze network flow data. The continued rapid growth of network trace data, requires new methods of scalable data collection and analysis. We report on our deployment of a system designed and implemented at the University of Kentucky that supports analysis of network traffic across the enterprise. Our system addresses problems of scale in existing systems, by using distributed computing methodologies, and is based on a combination of stream and batch processing techniques. In addition to collection, stream processing using Storm is utilized to enrich the data stream with ephemeral environment data. Enriched stream-data is then used for event detection and near real-time flow analysis by an in-line complex event processor. Batch processing is performed by the Hadoop MapReduce framework, from data stored in HBase BigTable storage. In benchmarks on our 10 node cluster, using actual network data, we were able to stream process over 315k flows/sec. In batch analysis were we able to process over 2.6M flows/sec with a storage compression ratio of 6.7:1.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network Management, Network Monitoring; C.2.4 [**Distributed Systems**]: Distributed applications

ICPE'14, March 22–26, 2014, Dublin, Ireland.

ACM 978-1-4503-2733-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2568088.2568103.

Keywords

NetFlow, SDN, Stream Processing, Hadoop, Complex Event Processing

1. INTRODUCTION

The University of Kentucky, a large public university in Lexington, KY has extensive IT operations, serving to over 30,000 students (both undergraduate and graduate students), and over 16,000 faculty and staff. Both the instruction of students, research of the faculty and significant administrative functions of the university must be supported. Significant network operations have to be supported, administered and supervised to provide 24/7 smooth operation of the university. An additional burden on the IT is created by the Medical Campus, a number of research hospitals and clinics that provide one of the main facilities for the Eastern part of the Commonwealth of Kentucky.

Since a significant number of students resides in the university housing, the university must provide uninterrupted service 24/7, with the only periods of decreasing demand during parts of vacation period (although additional sessions are still meeting), and inter-session breaks. On any weekday one expects of up to 17,000 separate networks (most of them small) within the large Campus community. The number of wireless access points on Campus exceeds 5000.

With the rapid growth of student personal computing equipment (laptops, tablets, and intelligent phones), networking operations are under constant demand (varying over time with significant peaks and valleys). The equipment is used often during the instruction, creating an additional demand. Faculty support is also time-of-the-day dependent, concentrating primarily over the working hours.

For that reason the IT operations have to maintain an adequate and up-to-date picture of the demand and, in a bigger detail, the information about the traffic patterns on the campus network. The amount of traffic is significant; We estimate it at 282GB/sec (of the order 1PB/hour) and the storage capabilities and processing power required for off-line processing do not allow to store the entire traffic even for limited period of time. In particular only limited forms of an audit of the traffic are possible. Moreover, the data collected (flows) does not provide the information *ex post* to react to the events occurring within the network.

Instead, we report on the distributed processing and analysis of the network traffic. Such analysis is done in *real-time*, using the distributed computation within MapReduce/Hadoop paradigm. The flows collected during a specific period of time are analyzed and the results used for assignment of re-

sources to the networks that need them. Additionally, some auditing capabilities that are, for all practical reasons, also *real-time* become available to react to events that may interfere with normal IT operations.

By collecting and analyzing the data the university is able to abstract from specific short-time events and create a more complete picture of the data processing on campus and the associated network traffic. Having such data allows to predict the future trends and thus needs of the network needs that are required by the university community.

The main part of this paper, Section 2.1, describes the procedure and technical means applied in our research. The problem of collecting the data and the obstacles that appear in the process are described in Section 2.1.1. The main software tool used in this phase of the generation collection, and processing is *Fprobe* [8]. The collection of flows is described in Section 2.1.2. Custom code has been written to receive flows generated by Fprobe and submit flow bundles to an AMQP-compatible queue. These operations are executed on the host, with the flow generation, collection and message serving on the same physical machine.

The stream processing included in our experiments is described in Section 2.2. A variation on MapReduce suitable for stream processing, Storm [24] is used to process flows. The workflow of the stream processing is described as a spout/bolt process with the specific steps used in the process described in some detail. Specifically, the AMPQ Spout translates the AMPQ queue data into a format that is used by Storm Bolts. This stream is then processed by the Combiner Bolt, which merges multiple queues into a single stream. The Resolver Bolt consumes the merged stream and injects state information into the records. A Complex Event Processing Bolt using ESPER [5], provides discovery of security incidents and the traffic information. Additional bolts allow for reporting and storing the data in HBase.

Time-related data, in particular time series is stored in HBase and due to the nature of the flow information (ip-addresses, ports and router information) is stored in compressed form. Availability of MapReduce framework (Hadoop) allows for parallel processing of information. We report the experimental results found in the process and indicate how experimental data found during the processing allows us to get the useful and practically usable information about the flow.

Section 3 describes our perspective on the results of the paper and possibilities of further research.

2. TRAFFIC COLLECTION AND ANALYSIS

There will be a projected [19] 18-fold growth in mobile network traffic from 2011-2016, and by 2017 there will be a predicted [12] five network devices per person. Hardware devices equipped with ASICs will be capable of generating line-rate flow exports, even on very high throughput links. At the time of this writing, no single appliance exist that can collect hundreds of thousands of flows per second. Commercial distributed collectors, claim to collect millions of flows per second. These claims are based on the aggregate processing of the distributed collectors, which does not elevate the limitations of a single flow exporter to a single collection device. Methods of collection and analysis are needed that allow for both distributed processing and central visibility.

Table 1: Network Devices

Device	*Count*
Core	6
Distribution	44
Access	1176
Wireless Controllers	47
Wireless Access Points (AP)	5442
Virtual Switchs	42

2.1 Flow Generation and Collection

Simply generating NetFlows from high traffic links, is in itself, a highly computational task [6]. With the introduction of advance network technologies, such as Multi-protocol Label Switching (MPLS) [17], often embedded hardware does not have the ability to generate discrete flows and sampling methods are used. In our environment, the network topology is based on a hierarchical network design model [15]. In this model, the Access layer operates on the OSI [29] Layer 2 (Data Link) and so NetFlow generation is not possible on these devices. Both Core and Distribution layer devices typically have the ability to generate NetFlows, however in our environment due to the presence and computational overhead of MPLS, NetFlow generation on these routers is not vendor-supported.

While we were not able to generate NetFlows from our network hardware, we were able to generate flows in our virtual environments. NetFlows for all network traffic on our VMware vSphere [27] virtual machine farm, including intra-node communication, is exported by the vSphere virtual distributed switch.

2.1.1 NetFlow Generation from Monitor Ports

NetFlow generation was not possible on our network hardware, so we distributed software-based probes at the Core layer. The probes ingest aggregates of distribution links from the Core routers, effectively monitoring all traffic passing from distribution to distribution. This probe point also allows for the observation of all traffic between cores and on the network Edge. Due to link aggregation the overall potential monitor capacity exceeds the monitoring link, so dropped packets will occur if this limit is exceeded. Additional probes can be added to prevent packet loss due to link aggregation.

The probe boxes run an instance of Fprobe [8] for each monitored network interface. In Fprobe we are able to specify the Link layer header size, so MPLS header information is ignored and a NetFlow is generated from the correct IP diagram. The monitored data is used to generate NetFlows which are directed toward a collector.

2.1.2 NetFlow Collection

NetFlow collection is largely dependent on the host networking stack and how quickly flows can be removed from the UDP buffer. In Linux, the UDP maximum receive buffer size is set as a kernel option. The default buffer size is far too small for high rate flow collection, so on our collectors we increased the receive buffer to 16MB. There are many NetFlow collectors and libraries available in the common domain, unfortunately most collectors have been developed to record flows to either a relational database or a flat file. The available libraries proved to be complete and accurate,

but were either too slow, or the program language made in-application augmentation difficult. To solve those problems we wrote our own NetFlow version 5 and 9 collectors. These collectors run on the probe nodes and stream a pertinent subset of NetFlow information to a Advanced Message Queuing Protocol (AMPQ) [26] queues. Every Fprobe instance has a related collector and every collector has a set of AMPQ queues. There is an AMPQ queue for the data stream and a queue for log information. NetFlows are sequenced by the flow generator, so if a missed flow is detected, a warning message is placed in the log queue specific to that stream. The AMPQ server is hosted on the same node as the collector. We run the flow generator, collector, and AMPQ server all on the same host, however all of these functions can be distributed. This model keeps the data collection fully distributed and allows for horizontal and vertical scaling of probes based on load.

2.2 Complex Event (Stream) Processing

In Section 2.1.1 we described a distributed method of collecting NetFlows, but there is an even larger computational problem in analyzing that distributed aggregation of data. In our campus environment, we average from 5k to 25k flows per second. At that rate we process and record over a billion flows per day. Traditionally, single threaded applications analyzed collections of flow logs. Using traditional methods flow analysis could not possibly keep up with flow generation. Recently, researchers [14, 22] have started using Hadoop [10] to process these massive logs, however this method is still batch in nature. In addition to simply processing flow logs, there are additional benefits to processing streams of flows as they are generated. The obvious benefit of stream processing is the ability to react in near real-time to observed network events. Perhaps not so obvious, is the benefit of injecting state (machine name, network, subnet, etc) information in the flow logs, even if report generation will ultimately be a batch process.

Analysis of NetFlow data can be involved, among other things, for security applications [18, 31, 4, 20], including anomaly and intrusion detection. In addition, performance [25, 13] and planning information can be obtained from this analysis. With the introduction of Software Defined Networking (SDN) [16], we are now able control the network in near real-time. As we analyze streams of flows we can now preempt and react to actionable events as they occur. If a security anomaly is detected, a copy of the anomalous flow can be directed to a payload analyzer for deep packet inspection. Similarly, congestion can be predicted and detected through flow stream analysis. SDN controllers can be reconfigured to avoid and correct performance problems.

To process our aggregate of flows generated from our distributed probes, we use Storm, an complex event processor and distributed computation framework. Storm applications create topologies of interfaces to ingest and transform tuple streams. Similar to MapReduce [3], Storm distributes and processes tuples of information on multiple nodes and processes. However, unlike MapReduce, Storm will process tuples until the job is manually terminated. The primary topology components of Storm are Spouts and Bolts. Spouts, as the name suggest, are used to ingest data streams and emit tuples consumable by the application topology. On the other hand, Bolts read tuples from either Spouts or other Bolts, and also typically emit a tuple stream. Normally, tuple transformations, operations, and external data drains occur in Bolts.

Table 2: AMPQ Spout Tuple

Element Name	*Description*
timestamp	Time of flow creation
srcIp	Source IP address
srcPort	Source Port
dstIp	Destination IP
dstPort	Destination Port
byteCount	Sum of bytes in flow
proto	IP protocol
first_t	Router uptime at flow start
last_t	Router update at flow end
collector	Probe Queue Name

2.2.1 AMPQ Spout

We have developed spouts that subscribe to AMPQ queues on the probe nodes. The AMPQ service, which is provided by RabbitMQ [21], ensures an interoperable, flow-controlled, message passing service with guaranteed delivery. The spout, unpackages bundles of NetFlows generated by the probes and creates a discrete tuple for each flow. Along with building the Storm tuple, the spout also injects a element identifying the originating probe and related coverage area. In effect, AMPQ Spout produces a stream of database records; the attribute names are in the Column 1 of Table 2, AMPQ Spout Tuple, the meaning of these attributes is provided by Column 2.

2.2.2 Combiner Bolt

The combiner bolts accept tuples emitted by the *AMPQ Spouts*. The first three elements of the tuple defined in Table 2 are combined in the new element *srcIp-dstIp-TS* and the source elements are removed. The *srcIp-dstIp-TS* element will be used later as a tuple key. The modified tuple stream with the new flow key is then emitted by the combiner bolt. The purpose of this bolt is to take the output of the probe-specific spouts, combine them in a common output, and assign a key value to the emitted tuple. Output from this bolt is typically consumed by the *Resolver Bolt*.

2.2.3 Resolver Bolt

The resolve bolt is one of the most important components in the flow processing system. This bolt reads tuples from the *Combiner Bolt* and injects known state (internal networks) information into the tuple. At the time of this writing flow state information includes: internal or external network, host subnet router, router interface, and host subnet VLAN for both the source and destination addresses in the flow. These elements are concatenated and injected in the tuple stream under the element names *srcInfo* and *dstInfo*.

In order to trace source and destination network information we must first have a list of subnets with related state information. The next step is to calculate for each subnet on you list to check if source or destination address exist in that subnet. Subnet matching can be calculated in $O(n)$ time. One needs to realize, however, we have over 17k subnets on our campus, so near real-time processing was challenging. As stated earlier, our flow rate averages from 5k to 25k flows per second, so at that rate we must process a flow on

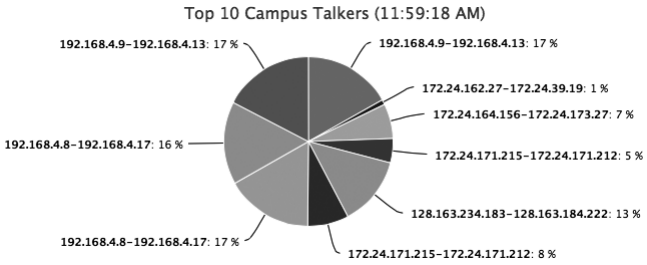

Figure 1: Live CEP Report

average every 0.04ms. Initial tests show that the resolver bolts execution latency is 2ms per bolt process. In an attempt to improve performance we increased the number of processes to 20 (across 10 computers) and sorted the subnet list on update, which placed most used subnets in the front of the list. While these changes improved performance, we still could not keep up with the flow rate. In our test we found that the majority of the execution overhead involved in the creation of subnet objects used to compare flow addresses with known subnets. The bolt was rewritten using the Google Guava [9] object caching libraries. The Guava-cached *Resolver Bolt* had an execution latency of 0.08ms, so over 20 processes that gave us a theoretical limit of 250k resolutions per second. The *Resolver Bolt* consumes all tuples and elements emitted by the *Combiner Bolt* and injects state elements. Output from this bolt is typically consumed by all other downstream bolts.

2.2.4 Reducer and CEP Bolts

While distributed remote call procedures (RPC) are possible in Storm, often it is easier to simply reduce a subset of tuples to a single process. *Reducer bolts* can either receive all tuples emitted by a large number of bolts or they can receive a reduced tuple stream based on a field-grouping filter. Most often these bolts are used in near real-time reporting or Complex Event Processing (CEP) where a specific subset of system wide elements is needed.

Complex Event Processing (CEP) [2] is the term used to describe a collection of methods used in the analysis of unbounded streams of information. A CEP engine will continuously process information streams in an attempt to identify, and react to, meaningful events. In the *CEP Bolt* we have implemented the ESPER [5] event stream processing (ESP) and event correlation engine (CEP). With this bolt we can detect any event that can be defined using the ESPER event processing language (EPL). We have implemented several *CEP Bolts* including bot detection, network scan detection, top talkers, top connections, highest transfer rates, lowest transfer rates, total flows per second, and total bandwidth per second. *CEP Bolts* are specific to a single EPL query, so they most often take input from a *Reducer Bolt* and emit a value specified by the query, on a user specified interval.

2.2.5 Report Bolt

Due to the distributed nature of the Storm framework there is no method to "query" the application topology for information. Luckily, we can get information out of the system the same way we put it in, by making use of AMPQ queues. RabbitMQ, our AMQP server, provides a Simple Text-Orientated Messaging Protocol (STOMP) [28] plugin,

Table 3: Stream Process Rates

Source → Destination	*Processed Flows/sec*
AMPQ → Spout	318672
AMPQ → ResolverBolt	315208
AMPQ → DrainBolt	233864

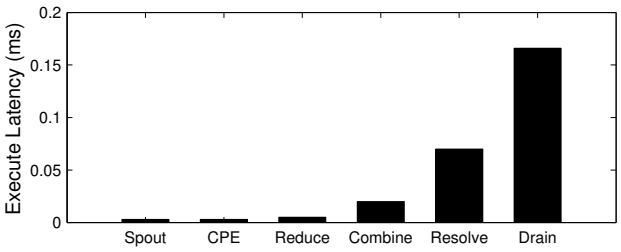

Figure 2: Topology Component Latency

which is directly consumable by web browsers using WebSockets [7]. The near real-time bi-directional capabilities of WebSockets, allows us to observe CEP events as they occur in our application topology. An example of this type of reporting is found in the CEP "Top Talkers" bolt, shown in Figure 1.

2.2.6 Drain Bolt

Once all state elements have been injected into the tuple stream and all CEP events have been calculated for a given tuple, we are ready to record the tuple. We have implemented an HBase [11] client into the *Drain Bolt*, which allows us to keep a running log of all flows processed by the system. All flows are recorded in a single HBase table for post-processing. In Section 2.3.1 we describe the benefits of using Hbase tables for this process. The *Drain Bolt* records all output emitted from the *Resolver Bolt* and does not emit a tuple stream.

2.2.7 Stream Load Testing

Due to the queue-based architecture of our flow collection system we have a good way to load-test the overall system using real data. If we disable or kill the Storm application topology there will be nothing to clear the AMPQ queues on the probe boxes and they will continue to grow. In our test we disabled our topology for several hours allowing millions of flows to be queued on the probe servers. We then enabled our topology, processed the awaiting queues, and calculated the transfer rates across the topology as shown in Table 3. Under load the topology actually performed 20% better than we had predicted, based on calculations in Section 2.2.3 *Resolver Bolt*.

2.3 Hadoop Processing

Not all information about networks can be extracted from streaming CEP. The application topologies we develop to analyze flows of network data are continuous by nature, however a large number of CEP detection rules are time-dependent. Due to resource constraints, even in a distributed system, the window of time available for in-memory processing is often insufficient for event modeling.

We have developed a method of storing and analyzing network flow data using Hadoop and HBase. Hadoop is an

ideal system for log processing. We take advantage of HBase for flow storage and Hadoop MapReduce for data processing.

2.3.1 HBase

HBase is particularly suited for sparse, time-dependent, structured, and highly compressible data. For data collection, a single database is used. HBase is capable of storing multiple records with the same key, as long as the timestamp is different, however our database it limited to a single row per key. Our database is configured with no block level caching, since random repeated reads are unlikely. Unlike file-based batch processing methods, we can retrieve HBase data based on a specific time range. This greatly improves job startup time since all data does not have to be scanned to process a specific range. To do this efficiently, we must take steps to avoid monotonically increasing row keys [30] due to the time series nature of our data. This is caused by using a increasing value, like a timestamp, for the first part of the key. When this occurs data piles up on certain nodes, which prevents the efficient distribution and processing of data. When storing time series information in an HBase key, one should stick with the form *[metric_type][event_timestamp]* to prevent performance problems related to data distribution. As described in Section 2.2.2, our key (*srcIp-dstIp-TS*) is constructed as a concatenation of address information and timestamp, which balances well across nodes.

The data we are storing is highly structured and we know that there is a relatively small amount of fixed information (ip addresses, tcp/udp ports, and router information) that we will be recording many times. This type of data compresses very well, so we have configured our database to use Google's Snappy [23] compression. Our sample HBase table contains over a billion records which relates to a database size of 118GB. If we dump our database to sequence files on the HDFS file system, the resulting sequence file size is 789GB. Under a compressed HBase table a flow generates 125 bytes of data as opposed to 836 bytes for the sequence file, which is a 6.7X reduction in size. The HDFS file system and batch flow processors outside of Hadoop also provide several compression options [1]. HBase has the advantage of automatic table compression updates, which will periodically evaluate and compact the entire table.

2.3.2 MapReduce

The MapReduce framework, provided by Hadoop, is well suited to process large datasets in parallel on distributed clusters of commodity servers. We have developed several MapReduce jobs as shown in Table 4, to analyze flows based on count, size, and rate. Since our flow data has been enriched with environmental state data, we have a rolling historical record of network utilization. Not only can we analyze traditional source and destination ip traffic, but we can analyze information about network routers, interfaces, subnets, and VLANs. With this additional information, analysis can be extended from our campus network topology to its building geography. Information such as, under provisioned wireless areas, can be determined by relating the interface flow rate with the service access point, at the time the flow was generated. As shown in Table 5, our MapReduce jobs were able to process between 1.5 - 2.6M flows/sec at a rate of 1.2 - 2.1GB/sec, depending on job type. For highly parallel tasks, such as flow processing, the aggregate throughput is largely a sum of the through of the individual parts. It

Table 4: MapReduce Jobs

Job Type	*Description*
1	Flow count per router subnet
3	Flow count between router subnets
4	Observed unknown subnet count
5	IP $Src \rightarrow Dst$ count
6	IP $Src \rightarrow Dst$ Bytes
7	Router subnet $Src \rightarrow Dst$ Bytes
8	Bytes per router subnet
9	Bytes per router
14	IP $Src \rightarrow Dst$ Bytes/sec
15	Router subnet $Src \rightarrow Dst$ Bytes/sec

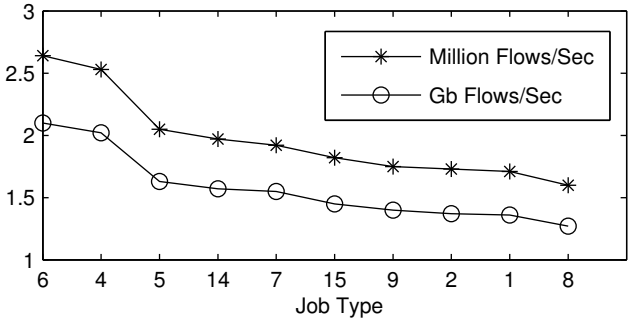

Figure 3: Performance information for MapReduce Jobs

is worth note that we achieved a per node throughout of 211MB/sec compared to 72MB/sec in a similar [14] Hadoop based flow processing system.

3. CONCLUSIONS

In this paper we reported the results of our work on processing the flow information at a medium-size educational institution. We showed how the paradigm of Storm can be used to process the flow data through the sequence of spouts and bolts. One benefit of this approach is that the data can be processed further with MapReduce on HBase. Our results have a practical aspect; the data thus obtained can be used in a variety of applications that trace both security of the system and also provide the data that can be used for predictions of the future IT needs of the university.

[1]FPS: Flows Per Second FPN: Flows Per Node,MBS: Total Throughput/Sec in MB, MBS: Node Throughput/Sec in MB

Table 5: Hadoop Process Rates

Job	*FPS*	*FPN*	*MBS*	*MBN*
1	1708906	170891	1363	136
3	1726783	172678	1377	138
4	2533070	253307	2020	202
5	2045258	204526	1631	163
6	2641174	264117	2106	211
7	1915686	191569	1528	153
8	1596946	159695	1273	127
9	1759167	175917	1403	140
14	1972324	197232	1573	157
15	1822925	182293	1454	145

[1]

We are in the process of developing *CEP Bolts* that interface with SDN controllers for security and performance functions. In the future, we plan on adding server performance metric information to our stream, when values are known. We will further develop analytical methods to make informed decision around the choice of movements of workloads or movements of data, based on network, system, and job profile data.

4. ACKNOWLEDGMENTS

The system described in this document, was initially developed to analyze network traffic in support of *NSF Grant OCI-1246332, U. of Kentucky, "CC-NIE Integration: Advancing Science through Next Generation SDN Networks", PI: V. Kellen, co-PI J. Griffioen.*

5. REFERENCES

[1] Y. Chen, A. Ganapathi, and R. H. Katz. To compress or not to compress-compute vs. io tradeoffs for mapreduce energy efficiency. In *Workshop on Green networking*, pages 23–28. ACM, 2010.

[2] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Comm. of the ACM*, 51(1):107–113, 2008.

[4] L. Ertoz, E. Eilertson, A. Lazarevic, P.-N. Tan, V. Kumar, J. Srivastava, and P. Dokas. Minds-minnesota intrusion detection system. *Next Generation Data Mining*, pages 199–218, 2004.

[5] Esper. `http://esper.codehaus.org/`, 9 2013.

[6] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better netflow. *SIGCOMM Comput. Commun. Rev.*, 34(4):245–256, Aug. 2004.

[7] I. Fette and A. Melnikov. The websocket protocol. *IETF*, 2011.

[8] Fprobe. `http://sourceforge.net/projects/fprobe`, 9 2013.

[9] Guava, `http://code.google.com/p/guava-libraries/`, 9 2013.

[10] Hadoop. `http://hadoop.apache.org/`, 8 2013.

[11] Hbase. `http://hbase.apache.org`, 9 2013.

[12] C. V. N. Index. Forecast and methodology, 2012–2017. *White Paper*, 2013.

[13] M. A. Kolosovskiy and E. N. Kryuchkova. Network congestion control using netflow. *arXiv preprint arXiv:0911.4202*, 2009.

[14] Y. Lee and Y. Lee. Toward scalable internet traffic measurement and analysis with hadoop. *ACM Comp. Comm.Rev.*, 43(1):5–13, 2012.

[15] L. Li, D. Alderson, W. Willinger, and J. Doyle. A first-principles approach to understanding the internet's router-level topology. *SIGCOMM Comput. Commun. Rev.*, 34(4):3–14, Aug. 2004.

[16] N. McKeown. Software-defined networking. *INFOCOM keynote talk, Apr*, 2009.

[17] MPLS. `http://www.cisco.com/en/US/products/ps6557/products_ios_technology_home.html`, 9 2013.

[18] J.-P. Navarro, B. Nickless, and L. Winkler. Combining cisco netflow exports with relational database technology for usage statistics, intrusion detection, and network forensics. In LISA 2000, pages 285–290, 2000.

[19] Cisco. Global mobile data traffic forecast update, 2012–2017. `http://www.cisco.com/en/US/solutions/.../white_paper_c11-520862.html`, 2013.

[20] T.-L. Pao and P.-W. Wang. Netflow based intrusion detection system. In *Networking, Sensing and Control, 2004 IEEE International Conference on*, volume 2, pages 731–736. IEEE, 2004.

[21] RabbitMQ. `http://www.rabbitmq.com`, 9 2013.

[22] RIPE. `https://labs.ripe.net/Members/wnagele/large-scalepcap-data-analysis-using-apache-hadoop`, 10 2011.

[23] Snappy `http://code.google.com/p/snappy/`, 9 2013.

[24] Storm. `http://storm-project.net/`, 9 2013.

[25] T. Telkamp. Traffic characteristics and network planning. In *Proc. Internet Statistics and Metrics Analysis Workshop*, 2002.

[26] S. Vinoski. Advanced message queuing protocol. *Internet Computing, IEEE*, 10(6):87–89, 2006.

[27] Vmware vsphere. `https://www.vmware.com/products/vsphere/`, 9 2013.

[28] V. Wang, F. Salim, and P. Moskovits. *The Definitive Guide to HTML5 WebSocket*. Apress, 2012.

[29] D. Wetteroth. *OSI Reference Model for Telecommunications*. McGraw-Hill Professional, 2001.

[30] T. White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[31] X. Yin, W. Yurcik, M. Treaster, Y. Li, and K. Lakkaraju. Visflowconnect: netflow visualizations of link relationships for security situational awareness. In *ACM workshop on Visualization and data mining for computer security*, pages 26–34. ACM, 2004.