Stream-monitoring with BlockMon: convergence of network measurements and data analytics platforms*

Davide Simoncelli
University of Brescia – CNIT
netcelli.tux@gmail.com

Maurizio Dusi
NEC Laboratories Europe
maurizio.dusi@neclab.eu

Francesco Gringoli
University of Brescia – CNIT
francesco.gringoli@ing.unibs.it

Saverio Niccolini
NEC Laboratories Europe
saverio.niccolini@neclab.eu

ABSTRACT

Recent work in network measurements focuses on scaling the performance of monitoring platforms to 10Gb/s and beyond. Concurrently, IT community focuses on scaling the analysis of big-data over a cluster of nodes. So far, combinations of these approaches have targeted flexibility and usability over real-timeliness of results and efficient allocation of resources. In this paper we show how to meet both objectives with BlockMon, a network monitoring platform originally designed to work on a single node, which we extended to run distributed stream-data analytics tasks. We compare its performance against Storm and Apache S4, the state-of-the-art open-source stream-processing platforms, by implementing a phone call anomaly detection system and a Twitter trending algorithm: our enhanced BlockMon has a gain in performance of over 2.5x and 23x, respectively. Given the different nature of those applications and the performance of BlockMon as single-node network monitor [1], we expect our results to hold for a broad range of applications, making distributed BlockMon a good candidate for the convergence of network-measurement and IT-analysis platforms.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Performance, Measurement, Experimentation

Keywords

Performance analysis, data analysis, distributed computing

1. INTRODUCTION

To cope with the growth of data exchanged over the Internet, the network measurement community has been focusing on scaling up capture and real-time processing of packets to 10Gb/s and beyond: platforms like CoMo [2] have emerged, as well as high-performance packet-capturing mechanisms like Netmap, PFQ [3]. At the same time, IT community has been designing platforms for scaling the analysis of big data over clusters of general-purpose machines. However, the proposed frameworks are mostly built on the MapReduce logic [4], which is oriented to the off-line processing of (batches of) data [5, 6, 7] previously stored on a distributed filesystem like HDFS. Due to this design choice, combination of high-performance network probes and off-line data-analysis frameworks is still a challenging target: the "store-and-analyze" paradigm has dominated, neglecting performance and real-timeliness of results for scale out and flexibility.

To overcome this issue, we apply a stream-processing paradigm to data monitoring. According to this model, an application is seen as a network that processes, routes or splits data across a topology of computing nodes for speeding up the analysis and facing failures. Indeed, the application processes data as it is produced, never stops and does not need to store any (intermediate) result. Moreover, this model allows to run algorithms which have shown to poorly fit into a map-reduce logic [7], e.g., algorithms where the reduce tasks cannot make use of the combiner at the map tasks, thus causing a large overhead in memory, network and disk.

In this context two new frameworks have recently emerged for the analysis of unbound streams of data: Storm [8] and Apache S4 [9] target scalability and high availability of the computing topology together with flexibility, by providing multi-language support and automatic allocation of resources (cores and nodes) to tasks. Although this flexibility may reduce the development time, we have experienced that it affects pure performance or, at worst, can leave nodes underutilized: as a consequence, to increase performance more nodes have to be added, thus raising the operational cost.

To investigate *what are the capabilities of the current data-analytics platforms in processing unbound stream of data*, in this paper we analyze the performance we achieve when running applications on the two stream-processing platforms Apache S4 and Storm, and on BlockMon [1], a platform originally designed for running packet processing operations on a single multi-core node, which we extend to execute applications which are distributed across machines. To this end, we implement on all platforms two stream-monitoring applications: *VoIPSTREAM* [10], a phone anomaly detection system, and *Twitter trending*, a system that monitors topics discussed by Twitter users over time. Our results point out some pitfalls in the existing platforms, which are not due to the use of mechanisms for high availability, and show that both applications perform best on BlockMon: for instance, to achieve the best processing rate of *VoIPSTREAM* on Storm, BlockMon needs only half of the machines required by Storm. We believe that our results, together with the performance of single-node BlockMon when functioning

*The research leading to these results has received funding from the European Union under the FP7 Grant Agreement n. 318627 (Integrated Project "mPlane")

as network-measurement platform [1], qualifies our enhanced BlockMon as a good candidate for the convergence of distributed network-measurement and IT-analysis platforms.

The main contributions of this paper are: (i) extending BlockMon to run distributed applications; (ii) comparing three open-source stream-processing platforms; (iii) evaluating their performance with stream-monitoring applications.

The rest of this paper is organized as follows. In Section 2 we describe BlockMon, Storm and Apache S4. In Section 3 we detail the applications we implemented over the three platforms. Section 4 and Section 5 show our testbed and discuss the results of our performance tests. In Section 6 we report the related work. Section 7 concludes the paper.

2. STREAM-PROCESSING PLATFORMS

Here we provide an overview of the three stream-processing systems readily available to the best of our knowledge: our enhanced BlockMon, Storm and Apache S4.

2.1 BlockMon

BlockMon is a modular system developed within the EU FP7 project *DEMONS* [11], for *flexible, high-performance traffic monitoring and analysis*, implemented in C++11 and available open-source under the BSD license [12]. In its original shape, users can run on a single computer a *composition* of processing *blocks* which exchange *messages* through their *gates*: users can develop efficient monitoring applications by interconnecting blocks that capture traffic from high-speed NIC with blocks that run algorithms for packet analysis. BlockMon allows also to add, update and remove blocks from a running composition without the need of stopping it.

In the BlockMon architecture, blocks implement at least two methods: *configure*, which sets configuration parameters passed via XML; and *receive_msg* which receives data (messages). A block can use the *send_out* method to enqueue messages on the output gates. Blocks that generate messages (e.g., packet capture threads) may use the *do_async* method to perform high-frequency non-periodic asynchronous work. Blocks are executed either via direct invocation, where messages are passed via method call; or indirect invocation, where messages are passed via a lock-free rotating queue [3]. When invoked directly, the downstream block runs within the thread of the upstream one; otherwise it runs in a separate thread. Users set the block invocation via the XML file.

Finally BlockMon schedules work in thread pools: each block is assigned to a pool via the XML, and pools can be pinned to specific cores. This model allows flexibility in terms of which block is executed on which CPU core.

2.1.1 BlockMon for distributed stream-processing

We extend BlockMon with interfaces for connecting blocks istantiated on different machines allowing *distributed compositions* to span over a network of nodes. The improvement with respect to the standard BlockMon architecture is twofold: first, it enables a fine-grained allocation of tasks on specific nodes (e.g., capturing-oriented nodes send data to processing-oriented nodes) by simply passing nodes' IP addresses via the XML description of the composition; second it opens for scalability, as a *distributed application* can take advantage of an heterogeneous set of nodes within a cluster.

A distributed application requires users to implement a *serialize* method for exporting data from a block of a node, and a *build_same* method for importing data on a block located on another node. Exporters send messages to importers using TCP as transport protocol, with the IP address and port of the destination specified in the XML file. Importer blocks handle incoming data: the types of message to expect and the input port are specified in the XML file. At the current state, we assume i) sessions established between a given exporter-importer pair are persistent; and ii) neither application (message) level acknowledgment nor any flow control technique is implemented, since TCP supplies for them natively and sessions are persistent. We plan to address fail safety, with reliability and high availability, in future work. We release improved BlockMon to the public [12].

2.2 Storm

A distributed application on Storm is called *topology* and it is composed of interconnections of *spouts* and *bolts*. Spouts read data from external sources (e.g., files or live streams) through the *nextTuple* method; bolts implement an *execute* method to process incoming data. Spouts and bolts send data out by means of an *emit* method. Data are exchanged in form of *tuples* through ZMQ [13] sockets, which handle the transmission of data locally or remotely on top of TCP.

Storm relies on Nimbus [14] for distributing code around the cluster and assigning tasks to machines. Users have no control on how resources are allocated within the cluster; they only set, for a given component, the number of instances to create, each of them running in a separated thread. Within a machine, *workers* and *executors* handle the execution of the elements for a given topology.

Storm runs on Java Virtual Machine, is written in Clojure and Java and supports multi-language programmability. We used the stable release of the software *v0.8* [1], available under the EPL license [8].

2.3 Apache S4

An Apache S4 application is made of processing *elements* that exchange *events*, i.e, messages holding (key, value) pairs. Every element processes only events with the same value of the key, through the *onEvent* method, and a new element is spawned for every new key. An external *adapter* converts data into Apache S4 events and injects them into the cluster through the *put* method: elements then exchange events on top of TCP sessions (UDP is supported as well). A task periodically deletes elements that have not received events within a time interval: though this prevents a machine from running out of resources, it causes to lose the state associated to the deleted elements.

Given a node, Apache S4 allows to allocate only one thread per element; furthermore, a single thread de-serializes data from the network into events and dispatches them to elements: this approach creates a serial point within the node in the Apache S4 architecture. The (de)queuing of events is based on blocking calls. Users can launch more processes within the same physical machine to increase the scalability of their application, at the expense of using more resources.

Apache S4 is written in Java. We used its latest available release (*v0.5*) at the time of the analysis, available under the Open Source Apache 2.0 license [15].

[1]We did not use the latest version (*v0.8.1*) as we verified that its wait strategy mechanism severely affects performance.

3. STREAM-MONITORING APPLICATIONS

Monitoring applications are built upon two basic elements: probes, which are responsible for getting data, and aggregation points, which correlate the data from the probes.

The number of probes and aggregators within an application depends on the application itself and on the topology of the network where it is deployed. However, they all generate by combining the following two basic configurations:

1. multiple probes and one aggregation point: it covers the case where the aggregation point can keep up with the amount of data being received;

2. one probe and multiple aggregation points: it covers the case where the tasks on the aggregation point are so computational demanding that require the application to parallelize them to process data on-the-fly.

We implemented an application for each of the two scenarios above on the three platforms. In the former case, we focus on how such platforms handle data transfer over the topology. In the latter case, we investigate how they help speed up the execution of applications that need high-computational power. As a matter of fact, multiple probes and multiple aggregation points might concur to form any service-monitoring application: by assessing how the distributed streaming platforms perform in each separated scenario, we aim at gaining insight on how such platforms perform in a broad range of applications.

The following of this section describes the design of the two applications. As several designs are indeed possible, to carry out a fair comparison we opted for designs that allowed to run each task on a dedicated machine, exploiting the same number of nodes, tasks and threads on all the platforms; moreover, machines were connected with links of the same capacity. This way, we prevent Apache S4 from assigning multiple high-computational tasks to the same machine, as the platform itself lacks of mechanisms to control it.

3.1 Twitter trending

The *Twitter trending* application extracts hashtags from Twitter tweets, and keeps a count of how many times people cite them. This application, that is the default use-case for Storm and Apache S4, aims at monitoring and ranking on-the-fly topics discussed by Twitter users over time.

Twitter data is provided by companies such as GNIP [16], which sell and send tweets to customers' probes as a collection of data in the JSON format. To count hashtags, first a parser has to extract them from within the JSON records.

The JSON parsing is the most-computational demanding operation within the application and a single aggregation point (*Hashtag Counter*) can cope with several sources: a quick analysis revealed that approximately only 8% of tweets contains hashtags (i.e., they are passed to the *Hashtag Counter*), which in turn account for 0.7% bytes on average within a single JSON data record. As JSON parser, we used *jsmn* and *jackson* for the C++ and Java implementation, respectively: a test revealed that the two JSON parsers, when used standalone, exhibit the same performance.

As a consequence, we opted for a design with multiple parsers (*Hashtag Finder*), one per each *Tweet Source*, and a single *Hashtag Counter* as reported in Figure 1: each element is a task running on a dedicated machine of the cluster.

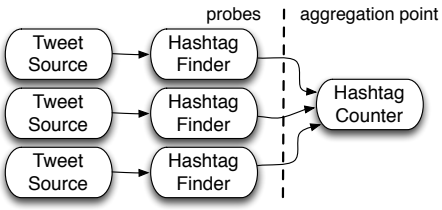

Figure 1: *Twitter trending*: design. Probes parse tweets and send the hashtags to a central counter.

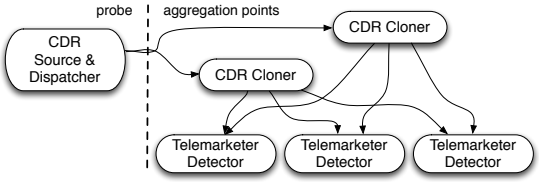

Figure 2: *VoIPSTREAM*: design. Probes extract from CDRs features for telemarketer detectors. Cloners ensure that CDRs from given users go to the same detector.

3.2 VoIPSTREAM

VoIPSTREAM is a phone anomaly detection application which has proven to be an effective method for detecting potential telemarketers in real-time while protecting the privacy of normal (i.e., non-telemarketing) users [10].

VoIPSTREAM takes a continuous stream of Call Data Records (CDR) as input, and associates a score to each user based on the calls that such user makes or receives, their duration and response code. Scores are computed over a configurable sliding time-window interval, and are continuosly updated for every new call through the use of time-decaying Bloom filters [10]: a threshold-based algorithm states if the user is acting like a telemarketer within a given time-window. As *VoIPSTREAM* emits a score for every new call, it is an example of applications that poorly fit into a map-reduce logic, due to the lack of a data-reduction stage.

Figure 2 outlines our implementation of distributed *VoIPSTREAM*. As here the bottleneck is the computational power of the Bloom filters, we distributed the dataflow over multiple telemarketer detectors. We implemented Bloom filters as native library, and wrote a binding for the Java implementation, so that *VoIPSTREAM* makes use of the same library on all platforms. As the algorithm must dispatch CDRs based both on the caller and on the callee, we implemented cloners within our application, which replicate the CDR in case the hash of the source and the hash of the destination differ, i.e., in case they are sent to different telemarketer detectors.

4. EXPERIMENTAL ANALYSIS

This section evaluates the performance of BlockMon, Storm and Apache S4 when executing distributed applications for (i) anomaly detection and (ii) trend analysis. Given the target we assume no failure during the experiments and we focus on scalability and costs in terms of CPU and memory usage, leaving as future work the analysis of node failure (fault tolerance) and data loss (reliability).

For every experiment, we assigned only one task per machine, and run the controller of the platform under test on a

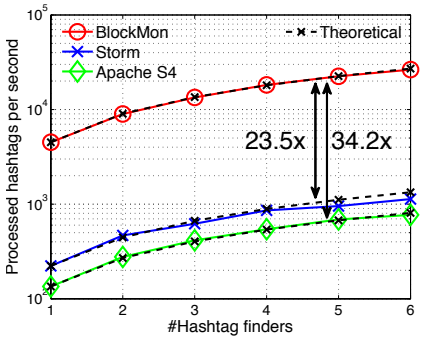

Figure 3: *Twitter trending*: scalability (y-axis is in log scale).

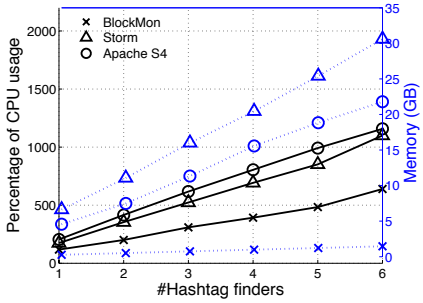

Figure 4: *Twitter trending*: total CPU (solid line) and memory (dotted line) usage.

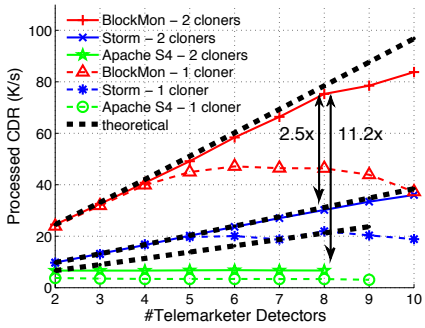

Figure 5: *VoIPSTREAM*: scalability.

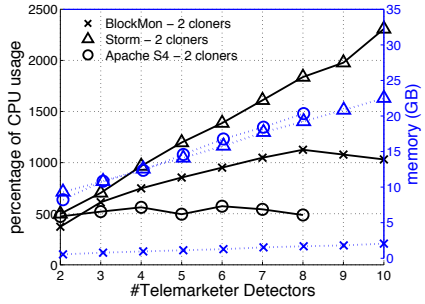

Figure 6: *VoIPSTREAM*: total CPU (solid line) and memory (dotted line) usage.

dedicated machine, to avoid the load introduced by the controller to affect our measurements. Moreover, we discarded the start-up time of the platforms from our measurements.

4.1 Testbed and Datasets

Our testbed is composed of 14 commodity machines, each one hosting two AMD Opteron(tm) Processors 246 (single core) and 4GB RAM. A 16-port switch connects the 1GbE interfaces of all machines. We estimate the value of each machine to be around $1000 on today's market.

For the *Twitter trending* application, we considered a dataset composed of around 20 millions of tweets, in the JSON format as provided by GNIP.

As for *VoIPSTREAM*, we used a dataset composed of few tens of million of anonymized Call Detail Records (CDR) collected over a period of several consecutive weeks, thanks to the collaboration of a small European telecom operator.

4.2 Performance tests

Figure 3 shows the performance of *Twitter trending* as the number of the hashtag finders (HF) increases. On all the platforms, the application scales linearly, with a gain in performance of 23.5x (34.2x) when we use BlockMon compared to Storm (Apache S4). Note that this scaling behavior is expected: as by design we increase the number of both tweet sources and HFs in the topology (see Figure 1), the rate of processed hashtags must increase linearly as long as network capacity towards the hashtag counter is not a bottleneck. In the figure, we report the theoretical behavior for each platform as a dashed black line. Figure 4 shows the total cost of memory and CPU required by the three platforms to run *Twitter trending*. As all machines in our testbed are identical, we computed the overall memory and CPU usage as the sum of the resources used on each machine. The CPU load of the hashtag counter is even in the worst case (with six HFs on Storm) always below 4%, thus suggesting that one is enough to cope with multiple HFs, which account in turn for around 75% of the total CPU resources.

Figure 5 shows the performance of *VoIPSTREAM* when using one source and one cloner, and increasing the number of telemarketer detectors (TD). In BlockMon, the application first scales up to four TDs with a processing rate of 40K CDR/s, which we determined is the maximum throughput that a single cloner can manage. With more than four TDs, the CPU load on the single cloner is such that prevents the cloner to fill the queues of the TDs fast enough: this leads to inactivity periods on the TDs and consequently to performance degradation. By adding one cloner the application still scales. Same considerations hold for Storm, even if the processing rate is up to 2.5x slower than with BlockMon. As for Apache S4, we observed that the bottleneck is due to the communication between the adapter and the cluster where the application runs, which prevents the application from scaling at all: in this case, *VoIPSTREAM* on BlockMon runs up to 11.2x faster. Interestingly, we were not able to run Apache S4 on the whole testbed: under high-memory consumption cases, the communication between the node and ZooKeeper hungs, thus partitioning the cluster. Developers of Apache S4 are aware of this issue.

The figure also reports the theoretical behavior of *VoIPSTREAM* as the number N of TDs increases, showing how BlockMon and Storm scale accordingly to it. A cloner replicates a CDR when the hash of the source and the hash of the destination mismatch, which happens with the probability $p = \frac{N-1}{N}$. Hence, given M CDRs emitted by the source, the cloner outputs $M \cdot (1+p)$ CDRs which are evenly distributed

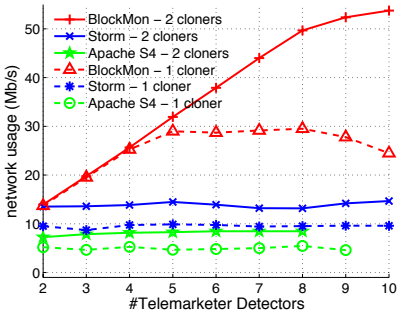

Figure 7: *VoIPSTREAM*: total network usage.

across the TDs. As all TDs run the same algorithm, we can assume that they finish to process CDRs at the same time, giving a total theoretical processing rate of $\frac{M \cdot (1+p)}{N}$.

Finally, Figure 6 shows memory and CPU usage required by the three platforms to run *VoIPSTREAM*, and BlockMon needs at best $\frac{1}{11}$ of the memory of Storm. Note that the CPU usage in Apache S4 is almost constant and lower than any other platforms due to its inability to scale, whereas its memory cost increases as we add JVM into the cluster.

To show the gain in using BlockMon, we measured the resources required to achieve the best processing rate that Storm is able to handle. In our experiments, *VoIPSTREAM* on Storm performs at best 36K CDRs/s, consuming in total 22GB of memory and 2308% of CPU on 13 machines (one source, two cloners, ten TDs). At a rate of 39.8K CDRs/s, BlockMon consumes 662MB of memory and 539% of CPU on 6 machines (one source, one cloner and four TDs).

5. DISCUSSION

To investigate pitfalls of the platforms under test and to stress their internals, in our experiments we removed (i) unfairness due to the use of different development languages and run-time systems, and (ii) I/O and network bottlenecks.

In favour of a fair comparison, in *VoIPSTREAM* we used on all the platforms the same C-library to handle the Bloom filters; in *Twitter trending* instead we measured that the two libraries for the JSON parsing exhibit the same performance.

To avoid bottlenecks due to the I/O and the network we instrumented the source blocks to pack more records (either CDRs or JSON formatted tweets) before sending the data to the exporters. We verified that the rates at which our blocks were processing data were lower than the link capacity which instead was fully exhausted when disabling data processing. Furthermore, we carried out experiments by running the applications on a single twelve-core machine. Although in this case blocks exchange data by using the internal CPU or motherboard bus (whose speed is order of magnitude higher than 1Gb/s) we measured performance rates similar to the ones reported in Section 4.2: this demonstrates that the network speed of 1Gb/s was not a limiting factor.

Therefore, we conclude that bottlenecks are intrinsic of the design of the platforms, and of the way they handle message passing through threads and queues. On Apache S4, we noted that the source of *Twitter trending* represents already a limiting factor. A look at the source code showed that sending out an event requires a series of nested copies, thus affecting performance: the rate of one tweet source on Storm, the other Java-based platform, is ten times faster.

Both in Storm and Apache S4, each node has a dedicated thread for dispatching messages from the network to the queues of the processing blocks. Moreover, we experienced that Storm can become unstable when the application on top runs for long period of time (hours in our case). We believe the choice of ZMQ queues may have an impact here. As they are not dropping, they keep growing in case the instantiated bolts cannot cope with the rate at which data is sent: as a consequence, bolts eventually run out of memory and are relaunched with empty queues, hence losing tuples. Users should take this pitfall into account in applications such as *Twitter trending*, where moving data among nodes is crucial. On the contrary, in BlockMon queues are automatically blocking thanks to the design choice of using TCP only, which also improves the overall stability of the platform. To show the behavior of the queues in each platform, Figure 7 reports the network usage in the *VoIPSTREAM* case: the flat behavior in Storm and Apache S4, combined with the increasing processing rate for Storm in Figure 5, proves that queues are not blocking and can increase indefinitely, in contrast with the linear increase of BlockMon.

5.1 Fault tolerant mechanisms and task assignment

During our experiments we verified that the presence of fault tolerant mechanisms on Storm and Apache S4 such as ZooKeeper (BlockMon currently lacks this capability), does not affect the performance of the overall execution of the topology: the overhead of communication between the master and the slaves is negligible in terms of CPU consumption (<3%) on all nodes – note that no failure happened during the experiments. Moreover, we experience this consideration to hold also when enabling the ack mechanism on Storm[2] – disabled during our experiments as we had no failures.

It is worth pointing out that all considered platforms currently statically assign tasks to machines, regardless of in the first instance they do it automatically (Storm and Apache S4) or ask the users to do it manually (BlockMon). In the former case, should a node fail, it is up to Zookeeper and Nimbus to realize that a failure happened, stop the topology, reschedule tasks and restart everything from scratch. For this reason, while a topology is running, no mechanisms for improving reliability or dynamically re-routing messages can slow down Storm performance when compared to BlockMon. We believe this is a key point when dealing with performance: existing stream processing architectures require some re-design (and BlockMon is the demonstration) to boost their performance, as this gap in performance is not due to mechanisms for high availability and dynamic message routing.

Additionally, Storm affects by itself the scalability of the cluster. Let us assume an application requires two tasks to exchange data over a 10Gbps link. As there is no way for users to assign tasks to machines, to satisfy this requirement *all* machines must be equipped with such a network card. Storm partially overcomes this problem with the use of Kestrel servers [17], which allows users to set the data source node and enable data reliability. The drawback is that enabling Kestrel servers decreases the performance of the application: in the case of *VoIPSTREAM*, we experienced that BlockMon becomes up to 20x faster (not shown

[2]To control and react to any loss of tuples, users must implement an *ack* and a *fail* methods within their application.

here). Alternatively, Storm provides a pluggable scheduler, though it is quite un-flexible: Storm assigns a fixed number of slots (four by default) to a machine for running workers, and the developer has to make sure that there are available slots on a given machine to run tasks – note that other topologies could run simultaneously on a given machine, and having slots already assigned.

6. RELATED WORK

The authors of [18] report the solutions adopted by the companies Twitter, Facebook and LinkedIn: only the first one has a platform specifically designed for streaming processing, i.e., Storm, whereas the others combine a distributed messaging system with a Hadoop cluster, in accordance with the "store-and-analyze" paradigm. Scribe [19] is a server for aggregating streaming log data, and is used by Facebook to feed a MapReduce clusters over a Hadoop Distributed File System. Kafka [20] is a distributed messaging system that provides an infrastructure for the analysis of both fresh and historical data: it is used by LinkedIn to feed an Hadoop-based system with data for batch processing.

The authors of [6] integrate online-aggregation into a MapReduce framework, thus allowing users to see "early returns" from a job as it is being computed. Examples of other platforms are Cloud MapReduce [21], HStreaming [22], DataStax Brisk [23] and RapidMiner Streams Plugin [24]. All these platforms are built on top of an already-existing batch processing programming model, such as Hadoop or RapidMiner. Recently, new platforms based on in-memory computation have been proposed, such as Spark [7], yet they are not oriented to the stream-processing of data.

7. CONCLUSIONS

In this paper we applied a stream-processing approach to data monitoring. To this end, we extended BlockMon, originally a packet-processing platform which run on a single node, to run distributed general-purpose applications. We compared our platform against the state-of-the-art of open-source stream-processing platforms, namely Storm and Apache S4, by implementing a phone anomaly detection system and a Twitter trending mechanism on all of them. Our results show that BlockMon performs better than the other platforms by at least 2.5x (and up to 34.2x).

This work points out that existing stream-processing platforms have serious issues when it comes to performance, which are not due to mechanisms for high availability or dynamic message routing: improving performance is possible, and our enhanced BlockMon showed that. We believe that our findings can help improve existing architectures to target stream data processing for network *stream monitoring*. As future work, we plan to include the performance of Hadoop-based platforms [25] into the comparison, and address the design of fail tolerant mechanisms into BlockMon.

8. REFERENCES

[1] A. di Pietro, F. Huici, N. Bonelli, B. Trammell, P. Kastovsky, T. Groleat, S. Vaton, and M. Dusi. Blockmon: Toward high-speed composable network traffic measurement. In *Proceedings of the IEEE Infocom Conference (mini-conference)*, 2013.

[2] G. Iannaccone. Fast prototyping of network data mining applications. In *Proceeding of the Passive and Active Measurement Conference*, 2006.

[3] N.Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. On multi–gigabit packet capturing with multi–core commodity hardware. In *Proceedings of the Passive and Active Measurement Conference*, 2012.

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

[5] Apache Hadoop. http://hadoop.apache.org (accessed 2012-11-10).

[6] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the USENIX NSDI Conference*, 2010.

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX NSDI conference*, 2012.

[8] Storm. http://storm-project.net (accessed 2012-11-10).

[9] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the International Conference on Data Mining Workshops*, 2010.

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

[11] FP7 Demons Project. http://fp7-demons.eu (accessed 2012-11-10).

[12] BlockMon. http://blockmon.github.com/blockmon (accessed 2012-11-10).

[13] The 0MQ Project. http://www.zeromq.org.

[14] The Nimbus Project. http://www.nimbusproject.org.

[15] Apache S4. http://incubator.apache.org/s4 (accessed 2012-11-10).

[16] GNIP. http://gnip.com.

[17] Kestrel Queues. https://github.com/robey/kestrel.

[18] D. Eyers, T. Freudenreich, A. Margara, S. Frischbier, P. Pietzuch, and P. Eugster. Living in the present: on-the-fly information processing in scalable web architectures. In *Proceedings of the ACM International Workshop on Cloud Computing Platforms*, 2012.

[19] Scribe. https://github.com/facebook/scribe.

[20] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the International Workshop on Networking Meets Databases*, 2011.

[21] Cloud MapReduce. http://code.google.com/p/cloudmapreduce.

[22] HStreaming. http://www.hstreaming.com.

[23] Brisk. http://www.datastax.com/products/enterprise.

[24] C. Bockermann and H. Blom. Processing data streams with the rapidminer streams-plugin. In *Proceedings of the RapidMiner Community Meeting and Conference*, 2012.

[25] Y. Lee and Y. Lee. Toward scalable internet traffic measurement and analysis with hadoop. *Comput. Commun. Rev.*, 43(1):5–13, Jan. 2013.