A Performance Comparison of Open-Source Stream Processing Platforms

Martin Andreoni Lopez, Antonio Gonzalez Pastana Lobato, Otto Carlos M. B. Duarte
Universidade Federal do Rio de Janeiro - GTA/COPPE/UFRJ - Rio de Janeiro, Brazil
Email: {martin, antonio, otto}@gta.ufrj.br

***Abstract*—Distributed stream processing platforms are a new class of real-time monitoring systems that analyze and extract knowledge from large continuous streams of data. These type of systems are crucial for providing high throughput and low latency required by Big Data or Internet of Things monitoring applications. This paper describes and analyzes three main open-source distributed stream-processing platforms: Storm, Flink, and Spark Streaming. We analyze the system architectures and we compare their main features. We carry out two experiments concerning threats detection on network traffic to evaluate the throughput efficiency and the resilience to node failures. Results show that the performance of native stream processing systems, Storm and Flink, is up to 15 times higher than the micro-batch processing system, Spark Streaming. However, Spark Streaming is robust to node failures and provides recovery without losses.**

I. Introduction

Sensor monitoring, network traffic analysis, cloud management [1], and security threats detection are applications that generate large amount of data to be processed in real time. These stream applications are characterized by an unbounded sequence of events or tuples that continuously arrive [2]. The advent of the Internet of things (IoT) increases the need of real-time monitoring. The estimate number of sensors networked by 2025 is around 80 billion [3]. Hence, data with this order of magnitude cannot always be processed centrally.

The main method to analyze big data in a distribute fashion is the MapReduce technique with Hadoop open-source implementation. Nevertheless, the platforms based on this technique are inappropriate to process real-time streaming applications. Applications processed by Hadoop correspond to queries or transactions performed in a stored and passive database and without real-time requirements, data elements are synchronized having an exact answer. Real-time monitoring applications require distributed stream processing that substantially differs from current conventional applications processed by distributed platforms. Monitoring normally requires the analysis of multiple external stream sources, generating alerts in abnormal condition. The real-time feature is intrinsic to stream processing applications and a big amount of alerts is normally expected. The stream data are unbounded and arrive asynchronously. Besides, the stream analysis requires historical data rather than only the latest reported and the data arrive. In cases of high entrance rates is common to filter the most important data discarding others and, therefore, approximate solutions are required, such as sampling methods. Hence, to meet these applications requirements, distributed processing models have been proposed and received attention from researchers.

Real-time distributed stream processing models can benefit traffic monitoring applications for cyber security threats detection [4]. Current intrusion detection and prevention systems are not effective, because 85% of threats take weeks to be detected and up to 123 hours for a reaction after detection to be performed [5]. New distributed real-time stream processing models for security critical applications is required and in the future with the advancement of the Internet of Things, their use will be imperative. To respond to these needs, Distributed Stream Processing Systems have been proposed to perform distributed processing with minimal latency and open-source general-purpose stream processing platforms are now available, meeting the need of processing data continuously. These open-source platforms are able to define custom stream processing applications for specific cases. These general-purpose platforms offer an Application Programming Interface (API), fault tolerance, and scalability for stream processing.

This paper describes and analyzes two native distributed real time native stream processing systems, the Apache Storm [6] and the Apache Flink [7], and one micro-batch system, the Apache Spark Streaming [8]. The architecture of each analyzed systems is discussed in depth and a conceptual comparison is presented showing the differences between these open-source platforms. Furthermore, we evaluate the data processing performance and the behavior of systems when a worker node fails. The experiments consider a real-time threat detection application developed by the authors [9]. The results are analyzed and compared with the conceptual model of each evaluated platform.

The remainder of this paper is organized as follow. In Section II we describe related work. We detail the Stream processing concept in Section III. Analyzed Platforms are presented in Section IV. Experimental results are shown in Section V. Finally, Section VI concludes the paper.

II. Related Work

Distributed real-time stream processing systems is a recent area and performance evaluations and comparisons between systems are fairly unexplored in the scientific literature. A few number of older and discontinued stream processing systems such as Aurora [10], and the project developed by Yahoo Apache S4 [11] served as base for most current systems. Google Millwheel [12], InfoSphere Streams IBM [13], among others are examples of proprietary solutions, but, in this paper, we focus on open-source systems.

Hesse and Lorenz compare the Apache Storm, Flink, Spark Streaming, and Samza platforms [14]. The comparison is restricted to description of the architecture and its main elements. Gradvohl *et. al* analyze and compare Millwheel, S4, Spark Streaming, and Storm systems, focusing on the fault tolerance aspect in processing systems [15]. Nevertheless, these two cited paper are restricted to conceptual discussions without experimental performance evaluation. Landset *et. al* perform a summary of the tools used for process big data [16], which shows the architecture of the stream processing systems. However, the major focus is in batch processing tools, which use the techniques of MapReduce. Roberto Colucci *et. al* show the practical feasibility and good performance of distributed stream processing systems for monitoring Signalling System number 7 (SS7) in a Global System for Mobile communications (GSM) machine-to-machine (M2M) application [17]. They analyze and compare the performance of two stream processing systems: the Storm and Quasit, a prototype of University of Bologna. The main result is to prove Storm practicability to process in real time a large amount of data from a mobile application.

Nabi *et. al* compare Apache Storm with IBM InfoSphere Streams platform in an e-mail message processing application [18]. The results show a better performance of InfoSphere compared to Apache Storm in relation to throughput and CPU utilization. However, IBM owns InfoSphere system and the source code is unavailable. Lu *et. al* propose a benchmark [19] creating a first step in the experimental comparison of stream processing platforms. They measure the latency and throughput of Apache Spark and Apache Storm. The paper does not provide results in relation to Apache Flink and the behavior of the systems under failure.

Dayarathna e Suzumura [20] compare the throughput, CPU and memory consumption, and network usage for the stream processing systems S, S4, and the Event Stream Processor Esper. These systems differ in their architecture. The S system follows the manager/workers model, S4 has a decentralized symmetric actor model, and finally Esper is software running on the top of Stream Processor. Although the analysis using benchmarks is interesting, almost all evaluated systems are already discontinued or not currently have significant popularity.

Unlike the above-mentioned papers, the following sections describe the architectural differences of open-source systems Apache Storm, Flink, and Spark Streaming. Moreover, we provide experimental performance results focusing on the throughput and parallelism in a threat detection application on a dataset created by the authors. They also evaluated the response and tolerance of the systems when one worker node fail. Finally, we conducted a critical overview of the main characteristics of each analyzed systems and discussed how the characteristics influence in the obtained results.

III. The Stream Processing

Data stream processing is modeled by a graph that contains data sources emitting continuously unbounded samples. The sources emit tuples or messages that are received by Processing Elements (PE). Each PE receives data on its input queues, performs computation using local state and produces an output to its output queue. The set of sources and processing nodes creates a logical network connected in a Directed Acyclic Graph (GAD). GAD is a graphical representation of a set of nodes and the processing tasks.

Stonebraker *et.al.* [2] highlight the most important requirements for distributed stream processing platforms. The data mobility identifies how it moves through the nodes. In addition, data mobility is a fundamental aspect to maintain low latency, since blocking operations, as done in batch processing platforms, such as Hadoop, decrease data mobility. Due to the large volume, data should be separated in partition to treat it in parallel. High availability and fail recovery are also critical in stream processing systems. In low latency applications, the recovery should be quick and efficient, providing processing guarantees. Thus, the stream processing platforms must provide resilience mechanisms against imperfections, such as delays, data loss, or out of order samples, which are common in data stream. Moreover, processing systems must have a highly optimized execution engine to provide real-time response for applications with high data rates. Thus, the ability to process millions of messages per second with low latency, within microsecond, is essential. To achieve a good processing performance, platforms shall minimize communication overhead between distributed processes in data transmission.

A. Methods of Data Processing

Data processing is divided in three main approaches: batch processing, micro-batch processing and stream processing. The analysis of large sets of static data, which are collected over previous time periods, is done with batch processing. However, this technique has large latency, with responses greater than 30 seconds, while several applications require real-time processing, with responses in microsecond order [21]. Despite, this technique can perform near real-time processing by doing micro-batch processing. Micro-batch treats the stream as a sequence of smaller data blocks. For small time intervals, the input is grouped into data blocks and delivered to the batch system to be processed. On the other hand, the third approach, stream processing, analyzes massive sequences of unlimited data that are continuously generated.

In contrast to batch processing, stream processing is not limited by any unnatural abstraction. Further, latency of stream processing is better than micro-batch, since messages are processed immediately after arrival. Stream processing performs better in real time, however, fault tolerance is more costly, considering that it must be performed for each processed messages. In batch and micro-batch processing, some functions, such as *join* and *split* are hard to implement, because the system handles an entire batch at time. However, unlike stream processing, fault tolerance and load balancing are much simpler, since the system sends the entire batch to a worker node, and if something goes wrong, the system can simply use another node.

B. Fault Tolerance

High availability is essential for real-time stream processing. The stream processing system should recover from a failure rapidly enough without affecting the overall performance. Hence, guaranteeing the data to be processed is a major concern in stream processing. On large distributed computing systems, various reasons lead to failure, such as node, network, and software failure. In batch processing systems, latency is acceptable and, consequently, the system does not need to recover quickly from a crash. However, in real-time systems without failures prevention, failures mean data loss.

Exactly once, *at least once*, and *at most once* are the three types of message delivery semantics. These semantics relate to the warranty that system gives to process or not a sample. When a failure occurs, the data can be forwarded to other processing element without losing information. The simplest semantic is *at most once* in which there is no error recovery, that is, either the samples are processed or lost. In *at least once* semantic, the error correction is made jointly for a group of samples, this way, if an error occurs with any of these samples, the entire group is repeated. This semantics is less costly than *exactly once*, which requires an acknowledgment for each sample processed.

IV. Analyzed Platforms

A. Apache Storm

Apache Storm [6] is a real-time stream processor, written in Java and Clojure. Stream data abstraction is called tuples, composed by the data and an identifier. In Storm, applications consists of topologies forming a directed acyclic graph composed of inputs nodes, called spouts, and processing nodes, called bolts. A topology works as a data graph. The nodes process the data as the data stream advance in the graph. A topology is analog to a MapRedude Job in Hadoop. The grouping type used defines the link between two nodes in the processing graph. The grouping type allow the designer to set how the data should flow in topology.

Storm has eight data grouping types that represent how data is sent to the next graph-processing node, and their parallel instances, which perform the same processing logic. The main grouping types are: *shuffle*, *field*, and *all grouping*. In *shuffle* grouping, the stream is randomly sent across the bolt instances. In *field* grouping, each bolt instance is responsible for all samples with the same key specified in the tuple. Finally, in *all grouping*, samples are sent to all parallel instances.

Figure 1 shows the coordination processes in a Storm cluster. The manager node, Nimbus, receives a user-defined topology. In addition, Nimbus coordinates each process considering the topology specification, i.e., coordinates spouts and bolts instantiation and their parallel instances. Each Worker node runs on a Java Virtual Machine and execute one or more

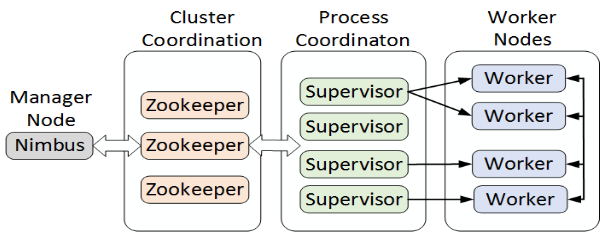


Figure 1: Nimbus receives topologies and communicates to Supervisors which coordinate process in workers, all the coordination between Nimbus and Supervisor is made by *Zookeeper* who store the cluster state.

tasks, also called processes. The Supervisors monitor the processes of each topology and inform the state to Nimbus using the Heartbeat protocol. *Zookeeper* is used as a coordinator between Nimbus and Supervisors. Nimbus and Supervisor are stateless, granting to *Zookeeper* all state management.

Apache Storm uses an upstream backup and acknowledgments mechanism to ensure that tuples are re-processed after failure. For each processed tuple, an acknowledgment (ACK) is sent to the acker bolt. Every time a tuple enters the topology, the spout add an *id*, then send the *id* to the acker bolt. The acker bolt saves all *ids* and when a bolt processes the tuple, the bolt sends an ACK to the acker bolt. When tuples exit the topology, the acker bolt drops the *ids*. If a fault occurs, not all acknowledgments have been received or the ACK timeout expired. This guarantee that each tuple is either processed once, or re-processed in the case of a failure, known as delivery *at-least-once*.

B. Apache Flink

Apache Flink [7] is a hybrid processing platform, supporting both stream and batch processing. Flink core is the stream processing, making batch processing a special class of application. Analytics jobs in Flink compile into a directed graph of tasks. Apache Flink is written in Java and Scala. Figure 2 shows Flink architecture. Similar to Storm, Flink uses a master-worker model. The job manager interfaces with clients applications, with responsibilities similar to Storm master node. The job manager receives client applications, organizes the tasks and sends them to workers. In addition, the job manager maintains the state of all executions and the status of each worker. The workers states are informed through the mechanism Heartbeat, like in Storm. Task manager has a similar function as a worker in Storm. Task Managers perform tasks assigned by the job manager and exchange information with other workers when needed. Each task manager provides a number of processing slots to the cluster that are used to perform tasks in parallel.

Stream abstraction is called DataStream, which are sequences of partially ordered records. DataStreams are similar to Storm tuples. The DataStreams receive stream data from external sources such as message queues, sockets, and others. The DataStreams support multiple operators or functions, such as *map*, *filtering* and *reduction*, which are applied incremen-

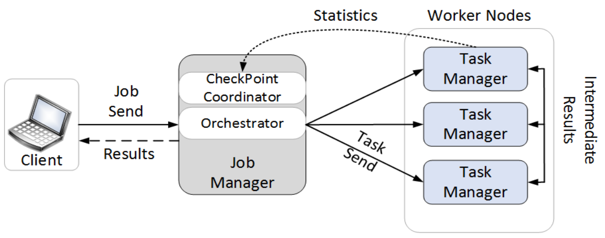


Figure 2: Architecture of Flink system. The Job manager receives jobs from clients, divides the jobs into tasks, and sends the tasks to the workers. Workers communicate statistics and results.

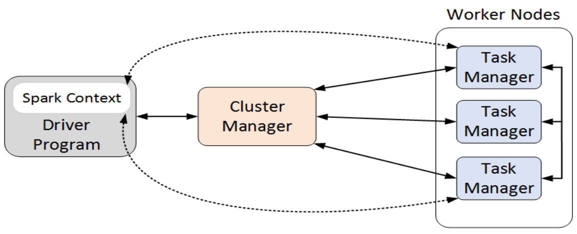


Figure 3: Cluster architecture of Spark Streaming system.

tally to each sample generating a new DataStreams. Each operator or function can be parallelized, running instances on different partitions of the respective stream. This method allows distributed execution on the streams.

Flink fault tolerance approach is based on snapshot over distributed checkpoints that maintain the status of jobs. Snapshots act as consistency checkpoints to which the system can return in case of failure. Barriers are injected in source elements and flow through the graph together with the samples. A barrier indicates the beginning of a checkpoint and separates records into two groups: records that belong to current snapshot and others that belongs to the next snapshot. Barriers trigger new snapshots of the state when they pass through operators. When an operator receives a barrier, it stores the status of the corresponding stream snapshot and sends the checkpoint coordinator to the job manager. In case of software, node, or network failure, Flink stops the DataStreams. The system immediately restarts operators and resets to the last successful checkpoint stored. All records processed in the restart of a stream are guaranteed not to have been part of the previous checked state, ensuring delivery of *exactly-once*.

C. Apache Spark Streaming

Spark is a project initiated by UC Berkeley and is a platform for distributed data processing, written in Java and Scala. Spark has different libraries running on the top of the Spark Engine, including Spark Streaming [8] for stream processing. The stream abstraction is called Discrete Stream (D-Stream) defined as a set of short, stateless, deterministic tasks. In Spark, streaming computation is treated as a series of deterministic batch computations on small time intervals. Similar to MapReduce, a job in Spark is defined as a parallel computation that consists of multiple tasks, and a task is a unit of work that is sent to the Task Manager. When a stream enters Spark, it divides data into micro-batches, which are the input data of the Distributed Resilient Dataset (RDD), the main class in Spark Engine, stored in memory. Then the Spark Engine executes by generating jobs to process the micro-batches.

Figure 3 shows the layout of a Spark cluster. Applications or jobs within the Spark run as independent processes in the cluster which is coordinated by the master or Driver Program, responsible for scheduling tasks and creating the `Spark Context`. The `Spark Context` connects to various types of cluster managers, such as the Spark StandAlone, Mesos or Hadoop YARN (*Yet Another Resource Negotiator*). These cluster managers are responsible for resource allocation between applications. Once connected, Spark executes task within the task managers, which perform processing and data storage, equivalent to Storm workers, and results are then communicated to the `Spark Context`. The mechanism described in Storm, in which each worker process runs within a topology, can be applied to Spark, where applications or jobs are equivalent to topologies. A disadvantage of this concept in Spark is the messages exchange between different programs, which is only done indirectly such as writing data to a file, worsen the latency that could be around seconds in applications of several operations.

The Spark has Streaming delivery semantics *exactly-once*. The idea is to process a task on various worker nodes. During a fault, the micro-batch processing may simply be recalculated and redistributed. The state of RDDs are periodically replicated to other worker nodes, in case of node failure. Tasks are then discretized into smaller tasks that run on any node without affecting execution. Thus, the failed tasks can be launched in parallel evenly distributing the task without affecting performance. This procedure is called Parallel Recovery. The semantics of *exactly-once* reduces the overhead shown in upstream backup in which all tuples are acknowledge like in Storm. However, micro-batch processing has disadvantages. Micro-batch processing takes longer in downstream operations. The configuration of each micro-batch may take longer than conventional native stream analysis. Consequently, micro-batches are stored in the processing queue.

Table I presents a summary of features underlined in the comparison of the stream processing systems. The programming model can be classified as compositional and declarative. The compositional approach provides basic building blocks, such as Spouts and Bolts on Storm and must be connected together in order to create a topology. On the other hand, operators in the declarative model are defined as higher-order functions, that allow writing functional code with abstract types and the system will automatically create the topology.

V. Results

Fault tolerance and latency requirements are essential in real-time stream processing. This section evaluates the processing throughput and the behavior during node failure of the three presented systems: Apache Storm version 0.9.4, Apache Flink version 0.10.2 and Apache Spark Streaming version 1.6.1, with micro-batch size established in 0.5 seconds.

Table I: Overview of the comparison between Stream Processing Systems.

	Storm	**Flink**	**Spark Streaming**
Stream Abstraction	Tuple	DataStream	DStream
Build Language	Java/Closure	Java/Scala	Java/Scala
Messages Semantic	At least once	Exactly one	Exactly one
Failure Mechanism	Upstream Backup	Check-point	Parallel Recovery
API	Compositional	Declarative	Declarative
Failures Subsistem	Nimbus, Zookeeper	No	No

We perform the experiments in an environment with eight virtual machines running on a server with Intel Xeon E5-2650 processor at 2.00 GHz and 64 GB of RAM. The experiment topology configuration is one master and seven worker nodes for the three evaluated systems. We calculate the results with confidence interval of 95%. To enter data at high rates in the stream processing systems, we use a message broker that operates as a publish/subscribe service, Apache Kafka in version 0.8.2.1. In Kafka, samples or events are called messages, name that we will use from now on. Kafka abstracts message stream into topics that act as buffers or queues, adjusting different production and consumption rates.

The dataset used in the experiments is the one from an threat detection application created by the authors [9]. The dataset is replicated to assess the maximum processing throughput at which the system can process. The application tested was a threat detection system with a neural network classifier programmed in Java.

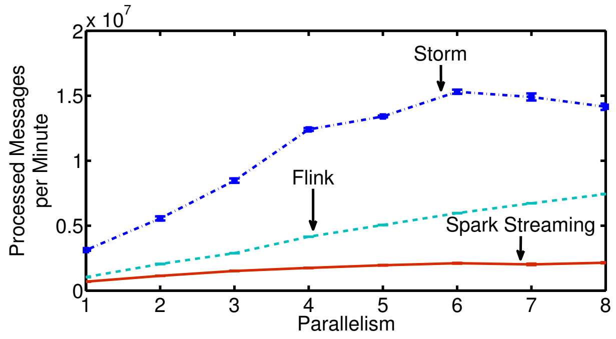
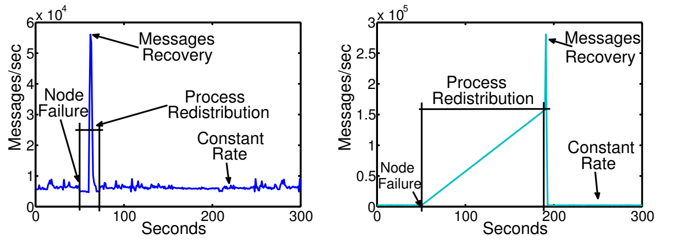


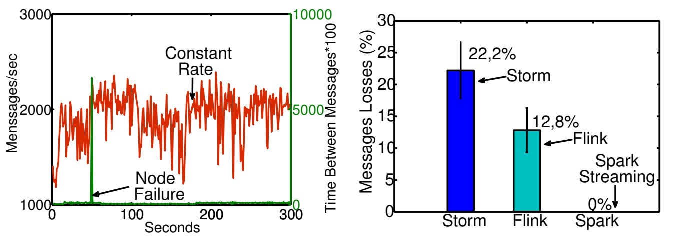
Figure 4: Throughput results of the platforms in terms of number of messages processed per minute in function of the task parallelism.

The first experiment measures the performance of the platforms in terms of processing throughput. The data set is injected into the system in its totality and replicated as many times as necessary. The experiment calculates the consumption and processing rate of each platform. It also varies the parallelism parameter, which represents the total number of cores available for the cluster to process samples in parallel. Figure 4 shows the results of the experiment. Apache Storm has a better throughput than the others do. For a single core, i.e., without parallelism, Storm already shows better performance compared to Flink and Spark Streaming in at least 50%. Flink has a completely linear growth, but with values always below the Apache Storm. The processing rate of the Apache Spark Streaming, when compared to the Storm and Flink, is much lower, this is due to the use of micro-batch, as each batch is grouped before processing. Apache Storm behavior is linear until the parallelism of four cores. Then, the processing rate grows until parallelism of six, where the system saturates. This behavior was also observed in Apache Spark Streaming with the same six-core parallelism.



(a) Storm behavior under node failure. (b) Flink behavior under node failure.

Figure 5: A failure is produced at 50 seconds. a) Storm and b) Flink system behavior after detecting the failure and consisting of process redistribution and message recovery procedures.



(a) Spark behavior under node failure. (b) Messages losses during node failure

Figure 6: a) The Spark system behavior under failure, indicating that it keeps stable and does not lose messages. b) Percentage of message losses.

The second experiment shows the system behavior when a node fails. Messages are sent at a constant rate to analyze the system behavior during the crash. The node failure is simulated by turning off a virtual machine. Figures 5a, 5b and 6a show the behavior of the three systems before and after a worker node failure at 50 seconds. Apache Storm takes some time in the redistribution processes after the fault was detected. This time is due to communication with the Zookeeper. Zookeeper has an overview of the cluster and reports the state for Nimbus in Storm, which reallocates the processes on other nodes. Soon after this redistribution, the system retrieves Kafka messages at approximately 75 seconds. Although the system can quickly recover from node failure, during the process there is a significant message loss. A similar behavior is observed in Apache Flink. After detecting the failure at approximately 50 seconds, the system redistributes the processes for active nodes. Flink does this process internally without the help of any subsystem, unlike Apache Storm that uses Zookeeper.

Figure 5b shows that time period in which Flink redistributes processes is much greater than the time spent in Apache Storm. However, message recovery is also higher, losing some messages during the process redistribution. Figure 6a shows Spark streaming behavior during a failure. When a failure occurs at approximately 50 seconds, the system behavior is basically the same as before. This is due to the use of tasks with micro-batch that are quickly distributed without affecting performance. Spark Streaming shows no message loss during fail. Thus, despite the low performance of Spark Streaming, it could be a good choice in applications where resilience and processing all messages are necessary.

Figure 6b shows the comparison of lost messages between Storm, Flink and Spark. It shows that Spark had no loss during the fault. The measure shows the percentage of lost messages by systems, calculated by the difference of messages sent by Apache Kafka and messages analyzed by the systems. Thus, Apache Flink has a smaller loss of messages during a fault with about a 12.8% compared to 22.2% in Storm. We obtain the result with 95% confidence interval.

VI. Conclusion

This paper describes and compares the three major open source distributed stream processing systems: Apache Storm, Apache Flink, and Apache Spark Streaming.The systems are similar in some characteristics, such as the fact that tasks run inside a Java Virtual Machine and the systems use master-worker model. A performance analysis of stream systems in a threat detection experiment by analyzing network traffic was carried out. The results show that Storm has the highest processing rate when the parallelism parameter, in number of processing cores, is changed, getting shorter response time, up to 15 times lower.

We also performed another experiment to show the behavior of the systems the during node failure. In this case, we show that Spark streaming, using micro-batch processing model, can recover the failure without losing any messages. Spark Streaming stores the full processing state of the micro-batches and distributes the interrupted processing homogeneously among other worker nodes. On the other hand, the stream processing native systems, Storm and Flink, lose messages despite using more complex recovery mechanisms. Apache Flink, using a checkpoint algorithm, has a lower message loss rate, about a 12.8% during the redistribution process after a failure. Storm loses 10% more, about 22.2% of messages since it uses a subsystem, *Zookeeper*, for nodes synchronization. Therefore, in order to select a platform, the user should take into account the application requirements and balance the compromise between high processing rates and fault tolerance.

Acknowledgment

This work was supported by CNPq, CAPES, and FAPERJ.

References

[1] B. Dab, I. Fajjari, N. Aitsaadi, and G. Pujolle, "VNR-GA: Elastic virtual network reconfiguration algorithm based on genetic metaheuristic," in *IEEE GLOBECOM*, Dec 2013, pp. 2300–2306.

[2] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[3] P. Clay, "A modern threat response framework," *Network Security*, vol. 2015, no. 4, pp. 5–10, 2015.

[4] M. Andreoni Lopez, D. M. F. Mattos, and O. C. M. B. Duarte, "An elastic intrusion detection system for software networks," *Annals of Telecommunications*, pp. 1–11, 2016.

[5] I. Ponemon and IBM, "2015 cost of data breach study: Global analysis," www.ibm.com/security/data-breach/, may 2015, accessed: 16/04/2016.

[6] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 147–156.

[7] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *Computing Research Repository (CoRR)*, vol. abs/1506.08603, 2015.

[8] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *XXIV ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[9] A. Lobato, M. A. Lopez, and O. C. M. B. Duarte, "An accurate threat detection system through real-time stream processing," Grupo de Teleinformática e Automação (GTA), Univeridade Federal do Rio de Janeiro (UFRJ), Tech. Rep. GTA-16-08, 2016.

[10] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: A new class of data management applications," in *28th International Conference on Very Large Data Bases*, 2002, pp. 215–226.

[11] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2010, pp. 170–177.

[12] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013.

[13] C. Ballard, O. Brandt, B. Devaraju, D. Farrell, K. Foster, C. Howard, P. Nicholls, A. Pasricha, R. Rea, N. Schulz *et al.*, "IBM infosphere streams," *Accelerating Deployments with Analytic Accelerators, ser. Redbook. IBM*, 2014.

[14] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *IEEE 21st International Conference on Parallel and Distributed Systems*, 2015, pp. 797–802.

[15] A. L. S. Gradvohl, H. Senger, L. Arantes, and P. Sens, "Comparing distributed online stream processing systems considering fault tolerance issues," *Journal of Emerging Technologies in Web Intelligence*, vol. 6, no. 2, pp. 174–179, 2014.

[16] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin, "A survey of open source tools for machine learning with big data in the hadoop ecosystem," *Journal of Big Data*, vol. 2, no. 1, pp. 1–36, 2015.

[17] R. Coluccio, G. Ghidini, A. Reale, D. Levine, P. Bellavista, S. P. Emmons, and J. O. Smith, "Online stream processing of machine-to-machine communications traffic: A platform comparison," in *IEEE Symposium on Computers and Communication (ISCC)*, June 2014, pp. 1–7.

[18] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas, "Of streams and storms," *IBM White Paper*, 2014.

[19] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 2014, pp. 69–78.

[20] M. Dayarathna and T. Suzumura, "A performance analysis of system S, S4, and Esper via two level benchmarking," in *Quantitative Evaluation of Systems*. Springer, 2013, pp. 225–240.

[21] M. Rychly, P. Koda, and P. Smrz, "Scheduling decisions in stream processing on heterogeneous clusters," in *Eighth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, Jul. 2014, pp. 614–619.