Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

MODELING AND SIMULATING STREAM PROCESSING PLATFORMS

Alonso Inostrosa-Psijas

Escuela de Ingeniería Informática
Universidad de Valparaíso
General Cruz 222
Valparaíso, CHILE

Roberto Solar
Mauricio Marin

Universidad de Santiago de Chile and
Centro de Bioinformática y Bioingeniería, CeBiB
Av. Libertador Bernardo O'Higgins 3363
Santiago, CHILE

Verónica Gil-Costa

Universidad Nacional de San Luis and
CONICET
Ejército de Los Andes 950
San Luis, D5700HHW, ARGENTINA

Gabriel Wainer

Dept. Of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON K1S 5B6, CANADA

2023 Winter Simulation Conference (WSC) | 979-8-3503-6966-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/WSC60868.2023.10407840

ABSTRACT

Stream processing platforms allow processing and analyzing real-time data. Several tools have been developed for these platforms to guarantee that the applications running on them are scalable, fast, and fault-tolerant and that they can be deployed on many processors. However, determining the proper number of processors suitable to hold a given stream processing-based software application is challenging, especially if the application is intended to serve a large user community. In this paper, we propose to model and simulate stream processing platforms for performance evaluation purposes. In our case study, we simulated a commonly used application for the analysis of Twitter streams with Storm. We evaluate its performance under different workloads. Our simulator supports profiling to measure various aspects of the application's performance. Results show that the simulator can replicate the metrics reported by the application running on a real platform with minimal error.

1 INTRODUCTION

Stream processing platforms are devised to manage substantial amounts of data from diverse sources, whether these are feeds in a social network, advertisement networks, businesses, or scientific applications. Usually, these systems require a real-time view of the data so human users can understand them. However, the high volume of data arriving from diverse sources makes it impossible to store these data, such as model-based on a data warehouse. In other words, stream processing platforms need to process large amounts of data within a few seconds, with very low latency and high throughput, to produce some knowledge or information out of the data. Therefore, the processing speed must be kept up with the incoming data rate while providing high-quality analysis of results as fast as possible. Additionally, the application components and the infrastructure must be fault-tolerant.

Stream processing is a distributed computing method that supports gathering and analyzing large volumes of a heterogeneous data stream to assist real-time decision-making as explained in Andrade et al. (2014). Stream programs use message-passing in a collection of computers connected by data channels or

links. Each computer includes a variety of processing units or processing elements (PEs) that are used to process and transfer data from its input message queue and produce results on its output queue. Each PE has an independent address space, while all dependencies between the processing units are made explicitly by message channels or links. Several stream processing platforms like Storm, Spark, Heron, Flink, Samza, and more have been developed.

In this work, we model and simulate stream processing platforms to evaluate their performance under different situations. We used Apache Storm as a case study. Storm is a widely used stream processing engine. Storm focuses on extremely low latency and requires near real-time processing. As a result, it can process enormous quantities of data and deliver results with less latency than other solutions. Although Twitter (the company that originally developed Storm) replaced it with Heron, the Storm platform has been successfully used by other well-known companies like Spotify, Flipboard, the Weather Channel, WebMD.com, among many others. Furthermore, research is still ongoing on Storm, including scheduling algorithms and resource allocation mechanisms. See the work presented by Muhammad et al. (2021), Hoseiny Farahabady et al. (2021a), Muhammad and Aleem (2021), Hoseiny Farahabady et al. (2021b). Although this research focuses on Storm, our model and simulation tool can be adapted to other platforms because we use a modeling methodology based on the design of individual interconnected components that can be replaced according to the characteristics of the platforms, the simulated hardware that supports them. Similarly, the connections between the components are defined by the task flow.

Our case studies use a Storm application of a Twitter stream analysis consisting of eight PEs, and communication among them is performed bottom-down. We aim to evaluate the platform's performance under different workloads to detect bottlenecks. To this end, we present the SimStream tool for simulating stream processing applications running on a stream processing platform, in this case, Apache Storm. The scope of our simulator includes Apache Storm, the hardware of the multi-core cluster where it is deployed, and the communication network. Next, we validate our simulation tool against a real application implemented on Storm. Finally, we evaluate the utilization of each PE as we vary the arrival rate of the tweets and replicate the more computationally intensive processes.

Results show that our simulator can be used as a testbed for predicting the performance of an application running on top of a stream processing engine with different levels of replication.

This paper is organized as follows. Section 2 describes stream processing platforms, particularly the Apache Storm platform. Section 3 presents related works. Our proposed tool, SimStream, is presented in Section 4. Section 5 presents a case study and Section 6 presents the experimental results. Finally, we present our conclusions in Section 7.

2 BACKGROUND

Stream programming has been proposed as a flexible approach for processing data originating at various sources on a distributed cluster. However, the data stream arrives at unpredictable times and may have dynamic variations in traffic intensity. In this context, storing and organizing the incoming data conveniently to process them in batches can be very costly, given the massive volume of data and the number of computational resources required for processing them. Nevertheless, even if this is feasible, it is often desirable or even imperative to process the data as soon as it is detected, and to deliver results in real-time.

To process data "as soon as it arrives", stream processing aims to process data in real-time in a fully integrated way to provide information and outcomes for consumers and end users. Also, it aims to integrate added information to support decision-making in the medium and long term. There are many stream processing platforms such as SPC (Stream Processing Core) as presented in Amini et al. (2006), Storm proposed by Toshniwal et al. (2014), Esc of Satzger et al. (2011), D-Stream of Zaharia et al. (2013) and, more recently, Heron presented by Kulkarni et al. (2015). In the following sections, we describe the main components of the Storm stream processing platform, which we used in our research.

2.1 Storm

Apache Storm, described in Toshniwal et al. (2014), is a real-time, fault-tolerant, and distributed stream processing system developed by Twitter. Ever since its open-source release in 2011, the Apache Foundation have incubated Storm. Even though Twitter replaced Storm and it is now using Heron presented in Kulkarni et al. (2015), the platform has been successfully used by well-known companies like Spotify, Flipboard, the Weather Channel, WebMD.com, and others (Storm 2015). Similarly, researchers still devote their efforts to improving Storm's efficiency. TOP-Storm is a scheduler for Storm that optimizes resource utilization and increases throughput by performing resource-aware task allocation (Muhammad et al. 2021). A3-Storm presented in Muhammad and Aleem (2021) is a scheduler implemented on top of Storm that optimizes resource usage in heterogenous clusters based on the topology of the applications and their traffic. The work of Hoseiny Farahabady et al. (2021a) focuses on a controlling strategy for optimizing the energy consumption of running applications. Hoseiny Farahabady et al. (2021b) have developed an elastic solution for resource sharing in Storm.

Storm provides a set of general primitives for defining applications for performing real-time computation. A Storm application comprises five major components: tuples, streams, spouts, bolts, and topologies. A **tuple** is an ordered list of elements. For example, in Twitter, a tuple might be a text composed of the tweet name, followed by the tweet itself (e.g., < *"I3M2018", "Deadlineapproaching"* >). A **stream** is a sequence of tuples, potentially unbounded. There is a sequence of arriving tuples, which form a stream, and these tuples are processed potentially one at a time.

Storm applications are called **topologies**. A topology is a directed graph whose vertices correspond to processing elements or operators (called spouts and bolts), and edges represent the data flow among them. Topologies can contain loops, but in this case, the programmer must be careful not to create infinite loops where tuples can go around the system in endless processing.

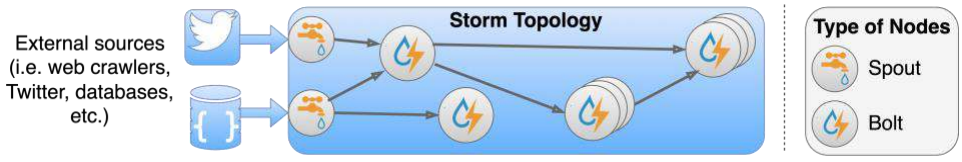

Figure 1: Logical layers of a Storm application.

In a Storm topology, **spouts** correspond to the source of streams. Usually, tuples are captured from an external source like a crawler or a database. A spout can generate multiple streams simultaneously. On the other hand, **bolts** process streams by performing a given task or operation on each tuple. They process a large amount of data and process it fast. Typically, each bolt processes one input stream, but they can process multiple input streams as well, generating an output stream used to feed other bolts. Some operations that a bolt can perform are:

- Filter: Forwards a tuple only if it satisfies a condition.
- Joins: When receiving multiple streams, say A and B, it computes a cross product of all the tuples and streams A and B, and for every pair of tuples, one from A and one from B, it returns the results of the process if the pair of tuples satisfy a given condition.
- Apply/transform: Modifies a tuple according to a function.
- Ticker: Control tuples are emitted periodically at regular intervals.

Figure 1 shows an example of a logical layer of a Storm topology. Here we have two spout processing units and four bolts processing units. The two spouts gather streams from various sources. Then, they send the incoming tuples forming streams to different bolts. The links between the processing units represent the flow of the data stream.

Several replicas - running in parallel into multiple processes or tasks - can be deployed to speed up bolt execution. Thus, incoming streams can be distributed among these tasks. The assignment of tuples among the bolt replicas (or tuple routing) is decided by a grouping strategy. Some of the most popular grouping strategies supported by Storms are the Shuffler Grouping, the Fields Grouping, and the All Grouping. The first distributes the tuples in a round-robin fashion over the bolt's replicas. The second one groups a stream by a subset of its fields and sends each tuple to its corresponding group using a hash function. The last one, where all the tasks in the bolt receive all the input tuples, is helpful for the join operation.

2.1.1 Storm Cluster

In a Storm cluster, topologies are submitted to a head node that runs a daemon called **Nimbus**. The Nimbus is responsible for distributing and coordinating the execution of the topology. Then, Nimbus distributes the code of the application around the cluster so when a bolt is started in a computer of the cluster, Nimbus is responsible for sending the code for that bolt to that computer. Also, when a bolt is split up into multiple tasks, Nimbus decides which tasks run on which computer.

Additionally, Nimbus is responsible for failure detection to potentially restart the tasks or the bolts on other computers of the cluster. A tuple is considered failed when its topology of resulting tuples fails to be processed within a given timeout. That is, when a tuple is emitted by the spout, the tuple has a *message-id* that will be used to identify the tuple later. Then, the tuple flows through the graph of bolts forming the topology of the application and when a tuple is fully processed, an ack method is called on the originating spout task with the message id. Likewise, if the tuple times out, a failure method is called on the spout and the tuple will be replayed. Note that a tuple will be acknowledged or failed by the exact same spout task that created it.

The worker nodes run a daemon called Supervisor which listens for work assigned to its computer. The worker nodes execute the topology's components (spouts and bolts). Each worker runs one or more *executors* inside a Java Virtual Machine, and executors are formed by one or more tasks. Note that tasks are executed by the actual work of a bolt or a spout. Each worker node keeps track of the tasks that are running at itself, so if any of these tasks crash, it can be either re-started, or the Supervisor can ask Nimbus for a new task that needs to be run in its place. Finally, Storms also use the Zookeeper system, which maintains the state of the cluster and coordinates the Nimbus and the Supervisors.

3 RELATED WORK

Simulation and stream processing platforms has been studied in the research literature in the area. In Amarasinghe et al. (2018), the authors present a simulator named ECSSim to place processing operators of the platform either on cloud nodes or edge devices. This simulator is extended in Amarasinghe et al. (2020), where they present a simulation toolkit named ECSNeT++ for stream processing platforms. The simulator is implemented on top of the network simulator OMNeT++. In both cases, the simulators are devised for Edge computing and IoT. Thus, it is not possible to use commodity switch-based networks like the Fat-Tree. In Kroß and Krcmar (2019), the authors propose a domain-specific language (DSL) to model the characteristics of Spark and YARN applications. Their proposal automatically extracts the model's specifications and transforms them into an evaluation tool.

In Gil-Costa et al. (2016), the authors proposed an asynchronous simulation protocol that can be executed in the S4 stream processing platform. This simulator was developed to simulate the execution of user queries in a Web search engine. The proposed simulator controls the virtual time advance at each Logical Process (LP), employing two barriers. The first barrier consists of a time window that allows the processing of events with timestamps within the time window. The second barrier is based on an oracle time barrier used to adaptively compute the value of the time window across the simulation. The two barrier-based mechanisms reduce the number of straggler events - in an asynchronous simulation - across

the LPs. Even though the case study of Gil-Costa et al. (2016) simulates a Web Search Engine, the authors mention the possibility of modeling and simulating a Stream Processing platform using this simulator.

Flow, presented in Park et al. (2010), is a parallel and distributed platform for simulating stream processing applications. It does not include hardware or software components of the stream processing system. Instead, it is designed to capture the data flow at the application level. Flow uses a hybrid conservative synchronization approach, where local events are processed in LBTS (lower bound time stamp) order, whereas events spanning across super-processes resort to conservative synchronization.

There are simulators for some stream processing cloud environments like CEPSim presented by Higashino et al. (2016). CEPSIm was used to study the scalability and performance of complex event processing (CEP) systems and the effects of different query processing strategies on cloud platforms. It is focused on the processing of queries. CEPSim was built on top of the CloudSim tool. RStorm (Kaptein 2014) is an R package for developing and evaluating streaming algorithms. RStorm is a simulation package intended to help developers analyze and evaluate their streaming algorithms easily and without the difficulties of actual implementation to a given stream processing platform. RStorm resorts to Storm's terminology and concepts, providing a graphical representation of streaming algorithms. However, this package analyzes the algorithm and does not include the associated hardware costs (like communication, multiple threads per node, etc.).

4 SIMSTREAM SIMULATION TOOL

The simulation model is implemented as a process-oriented simulator. Processes represent bolts/spouts in charge of tuples processing. Resources are artifacts such as the data of the incoming messages, global variables like the input queue of each process, the CPU, and the communication network. The simulation program is implemented using libCppSim (Marzolla 2004), where each process is implemented by a coroutine that can be blocked and unblocked during simulation.

4.1 Simulation

The operations *hold(), passivate(),* and *activate()* are used for this purpose. Thus, a coroutine C_i can be paused for a given amount of time δ_t -which represents the duration of a task. Once the simulation time δ_t has expired, the coroutine C_i activates itself if a *hold()* operation was previously executed. Otherwise, the coroutine C_i is activated by another coroutine C_j using the *activate()* operation. This last case allows the representation of the interaction among the different components of the simulated platform.

Our simulator includes the nodes of the topology. These nodes transfer tuples (data) from one to another as defined by the topology. Each node can play the role of a bolt or a spout. The nodes can be allocated into the same physical processor or into different processors. The allocation process is transparent to the application but involves different communication costs. Nodes located in the same physical processor communicate at a lower cost than nodes hosted by different processors. Each processor has cores, Ram, and Cache memory. The least recently used (LRU) replacement policy is used for both memories. The sizes of the memories are set by the parameters of the simulator. A function called *schedule_processing()*, is used to schedule the tasks of the bolts/spouts into the processor's cores.

Each processor has a network interface that divides the messages into packages before sending them through the communication network. Likewise, the network interface gathers the packages belonging to the same message upon reception. In this work, we simulate the switches of a Fat-Tree network described in Al-Fares et al. (2008), however, it can be easily replaced by other networks.

The following characteristics are considered in our simulator:

- Different topologies can be simulated simultaneously. Different users may deploy their applications into the same platform at the same time.

- Spouts and/or bolts can be deployed to the same machine or to different machines according to their communication pattern.
- The machines of the simulated cluster can be heterogeneous. They may have different memory sizes and different numbers of cores, among others.
- The communication between two processing elements (bolts/spouts) located in different machines is performed via a simulated Fat-Tree network.
- A First Come First Service policy is used when choosing the next spout/bolt to be executed.
- Tuples communicated among bolts follow the publish-suscribe pattern. In case of saturation, no tuples are dropped, instead they are queued at the publisher bolt.

4.2 Communication Network

We simulate the Storm platform deployed in a datacenter composed of N racks holding k processors each, which communicate to each other by means of a set of switches composing a Fat-Tree network topology as shown in Figure 2. This is a three-level network where switches and processing nodes are organized into PODs (Points of Delivery) which can be regarded as racks. Each POD consists of $(k/2)^2$ processing nodes and two layers of $k/2$ k-port switches from the two bottom layers.

At the top, we have the Core switches, which provide inter-POD communication. At the bottom are the Edge switches, where the processing nodes are connected to the network. Finally, in the middle, there are the Aggregation switches. This type of network allows an elevated level of parallelism, avoids congestion, and allows fault tolerance by providing different routes for sent messages depending on their destination by means of a two-level routing table.

4.3 Simulation Parameters

Three configuration files are needed to set up the parameters of a simulation: one for the topologies, another for specifying the parameters of each spout/bolt, and a third optional file can be used to define different tuple arrival rates during the simulation. The topologies files consist of pairs of spout/bolts source and target of tuples. For specifying the parameters of each spout/bolt, the file must contain the name of each node of the defined topology and the node type (spout/bolt), the replication level, the grouping type, and the IP address of the machine where it is hosted (according to the Fat-Tree). If the node is a spout, the previous values are followed by the name of a random generation number function and its corresponding parameters for representing the average service time and, finally, the average arrival rate of tuples. If the node is a bolt, the following values are the number of output tuples for each tuple that is processed and the name of a random generation number function and its corresponding parameters for representing its average service time. The third file includes the arrival parameter and the simulation time when it should start.

More details regarding this simulation model, configuration files format, and its code implementation can be found in our public SimStream GitHub repository.

5 CASE STUDY

In this paper, we simulate a popular real-time processing application example (Marçal 2017).The topology consists of two simple components**: kafka producer** - a simple producer in charge of reading tweets from the Twitter Streaming API and storing them into Apache Kafka -, and **twitter processor** - a Storm topology that reads tweets from Apache Kafka and processes them in a parallel fashion in order to perform, on one hand, compute *sentiment analysis* and, on the other hand, compute *top-k hashtag*.

The Storm topology consists of the following components (shown in Figure 3): **kafka spout** – an Storm-specific adapter in charge of reading tweets from *Kafka* into the Storm topology, **twitter filter** –

bolt in charge of filtering out all non-English language tweets in order to properly apply the *sentiment analysis* algorithm, **text sanitization** - bolt in charge of text normalization in order to properly process

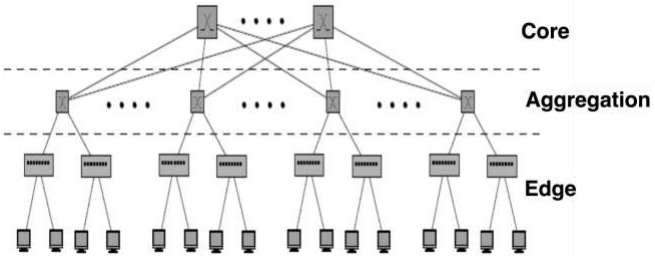

Figure 2: 3-Layer Fat-Tree topology.

tweets by the sentiment analysis algorithm, **sentiment analysis** - bolt in charge of scoring the tweet by each of its words using *SentiWordNet* classifier, **sentiment analysis to cassandra** – a bolt in charge of storing the tweets and its sentiment score into a *Cassandra* database, **hashtag splitter** – a bolt in charge of splitting the different hashtags and emitting a tuple per hashtag to the next bolt (hashtag counter), **hashtag counter** - bolt in charge of counting the number of occurrences of a hashtag, **top hashtag** bolt in charge of performing a ranking of the top-*k* hashtags by means of a sliding windows algorithm, and **top hashtag to cassandra** - bolt in charge of storing the top-*k* hashtags into a *Cassandra* database. Cassandra is a widely used open-source, distributed, scalable and elastic NoSQL database.

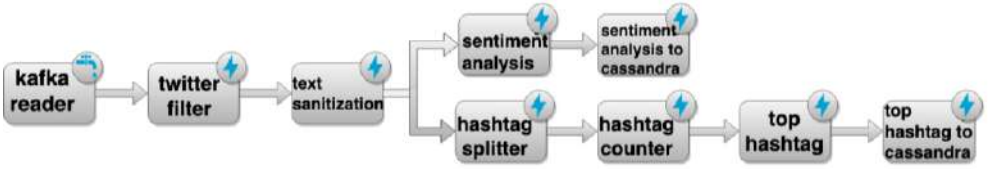

Figure 3: Storm topology of the Twitter streams analyzer.

6 EXPERIMENTAL RESULTS

In this section, we present the evaluation of our proposed Storm simulator (`SimStream`) in terms of throughput, response times, and utilization. Experiments were performed in an Intel Xeon CPU E5-2650 v2 2.60 GHZ (with 32 cores) and 128 GB of Ram, running Ubuntu 14.04.5 LTS. The simulator was implemented on top of the libcppsim library version 0.2.5 presented by Marzolla (2004), and it was compiled with GCC V.4.8.5. All the simulated bolts have been properly instrumented to measure and record tweet arrival times, bolts service times and the number of output tuples per bolt.

The process executed by the stream application on the Storm platform has been deployed using the following technologies: **messaging system** - Apache Kafka (version 2.12 − 1.0.1), **stream processing system** - Apache Storm (version 1.1.1), and **storage** - Apache Cassandra (version 3.11.2). All components run over a Zookeeper (version 3.4.11) cluster.

Figure 2.(a) and Figure 2.(b) show the tweet response times and the throughput (number of tweets processed per second), respectively. Both figures show the results of three Storm real executions compared to results obtained from our simulator (`SimStream`). We show the average results obtained with different execution of the simulations. Furthermore, `SimStream` allows generating random numbers by using two different mechanisms: **curve fitting** - which fits the given data (data sizes) by applying a *maximum likelihood estimation* method, and **spline interpolation** - which fits the given data by means of a piecewise polynomial function. Both mechanisms have been fed with logs obtained from isolated executions of the Storm topology.

In Figure 2.(a), we observe that real and simulated tweet response times have a similar tendency. These results can be contrasted with mean squared error (MSE) reported in Table 1. On one hand, the **curve fitting** approach reports MSEs close to 0.00001 with a mean of 0.00013474 and a standard deviation of 0.00009726. On the other hand, the **spline interpolation** approach shows a resemble behavior with a mean of 0.00013237 and a standard deviation of 0.00009635. In Figure 2.(b), we observe that real and simulated throughput have a similar behavior through time. Notice that the throughput is almost constant, that is because the available resources to process the data is limited. Thus, even though the arrival stream changes (increases), the resources will be saturated and the output of the system will reach its maximum. Furthermore, these results can be contrasted with MSE reported in Table 2. Both approaches, **curve fitting** and **spline interpolation**, show a mean close to 0.00001 and a standard deviation close to 0.00009. Therefore, the results show that our `SimStream` tool accurately simulates the Storm platform's behavior.

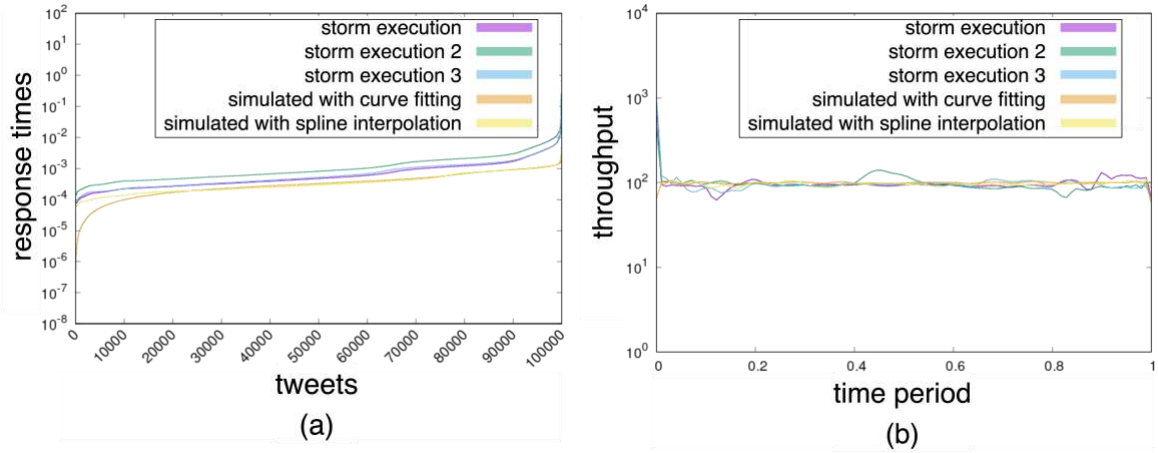

Figure 2: (a) Tweet processing response time. (b) Throughput.

Table 1: Mean squared error obtained for the tweet processing time.

Storm execution	Curve fitting		Spline interpolation	
1	0.0000626465		0.0000609966	
2	0.0001996262		0.0001967055	
3	0.0001419408		0.0001394227	
	Mean	**Stddev**	**Mean**	**Stddev**
	0.00013474	0.00009726	0.00013237	0.00009635

Table 2: Mean squared error obtained for the throughput.

Storm execution	Curve fitting		Spline interpolation	
1	0.0001636884		0.0001242584	
2	0.0003988999		0.0004130434	
3	0.0001734087		0.0001712610	
	Mean	**Stddev**	**Mean**	**Stddev**
	0.00024533	0.00018821	0.00023619	0.00021914

6.1 Level of Bolt Utilization without Replication

The tweet arrival rate, represented by a *negative exponential distribution*, has been parametrized by a location parameter and a non-negative scale parameter. The initial scale parameter value has been extracted from a *Storm log* by applying a *maximum likelihood estimation* method. The value of the scale

parameter has been progressively decreased to emulate changes in the tweet arrival rate. Figure 3 shows the level of utilization per bolt varying the tweet arrival rate. The utilization of a bolt is simply the ratio of the average time the bolt was busy and the total period of observation, thus the utilization is calculated as $U_b = B_b/R_b/T$, where B_b is the sum of time in which the bolt was busy, R_b is the number of replicas of the bolt, and T is the total simulated time.

In Figure 3, we can observe that the **twitter filter** bolt tends to be saturated as the tweet arrival rate increases. This behavior occurs because of the high number of incoming processing requests per time unit arriving at this bolt, turning it into a bottleneck. Furthermore, we can see that the **sentiment analysis to cassandra** bolt is tightly coupled with the **twitter filter** bolt since as the level of utilization of the **twitter filter** bolt is increased the level of utilization of the **sentiment analysis to cassandra** bolt is increased as well (close to 46% for high tweet arrival rates). We must consider that not all tweets processed by the **twitter filter** bolt are emitted to the next bolt and that is why the **sentiment analysis to cassandra** bolt is not fully saturated, but for being a disk access bolt is a bottleneck by itself. Unlike the **sentiment analysis to cassandra** bolt, the **top hashtag to cassandra** bolt does not write on disk on demand since disk accesses are managed by *tick tuples* (the database is periodically updated each Δt), and that is why its level of utilization is maintained as the tweet arrival rate is increased.

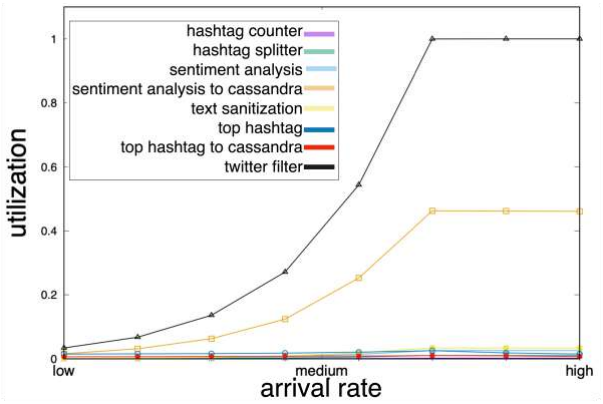

Figure 3: Utilization per bolt using the standard configuration (no replication).

6.2 Level of Bolt Utilization with Replication

Figure 6 shows the level of utilization per bolt varying the tweet arrival rate and the number of **twitter filter** bolt replicas. Figure 4.(a) shows the impact of two **twitter filter** bolt replicas over the topology. We observe that the **twitter filter** bolt and the **sentiment analysis to cassandra** bolt tend to have similar behavior in terms of the level of utilization, close to 100% for high tweet arrival rates. Figure 4.(b) shows the effect of eight **twitter filter** bolt replicas over the topology. We observe that the level of utilization of the **twitter filter** bolt decreased to a value lower than 27% for all cases. Nevertheless, we also observe that the level of utilization of the **sentiment analysis to cassandra** bolt is drastically increased in comparison to the level of utilization of the **twitter filter** bolt, which indicates the need for replication of the **sentiment analysis to cassandra** bolt. In general, as we add **twitter filter** bolt replicas, the level of utilization of **twitter filter** bolt is decreased. Still, the level of utilization of the **sentiment analysis to cassandra** bolt tends to be increased. This behavior is produced because as we increase the number of **twitter filter** bolt replicas, the number of incoming tweets to be processed by the next bolts is incremented proportionally to the number of replicas.

Figure 5 shows the level of utilization per bolt varying the tweet arrival rate and the number of **sentiment analysis to cassandra** bolt replicas, and with four replicas of the **twitter filter** bolt. Figure 5.(a) shows the level of utilization per bolt of the Storm topology with two **sentiment analysis to**

cassandra bolt replicas. The level of utilization of the **sentiment analysis to cassandra** bolt decreases in comparison to results reported in Figure 4. Nevertheless, the level of utilization of the **sentiment analysis to cassandra** bolt is still close to 100% for high tweet arrival rates. Figure 5.(b) shows the level of utilization per bolt of the Storm topology with eight **sentiment analysis to cassandra** bolt replicas, respectively. Results show that the level of utilization of the **sentiment analysis to cassandra** bolt is decreased to a value close to 23%. Even so, the level of utilization of the **twitter filter** bolt is maintained close to 100% for high tweet arrival rates.

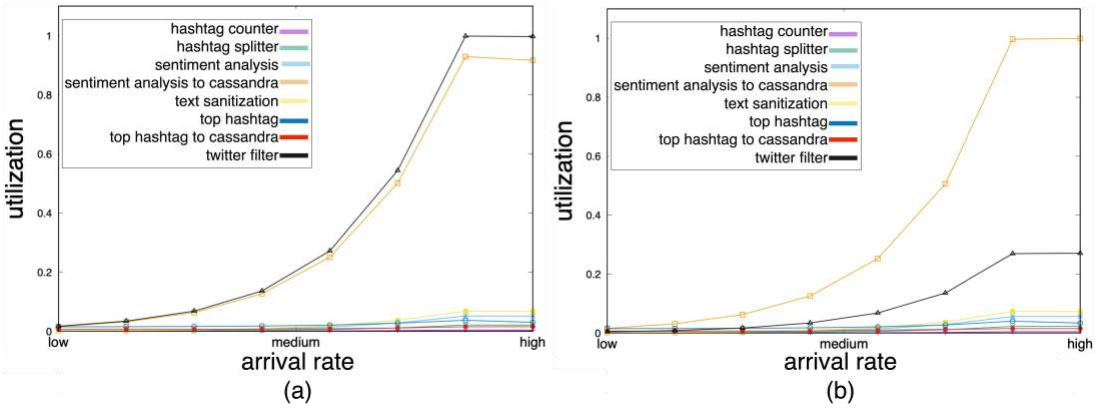

Figure 4: Utilization per bolt with different levels of **twitter filter** bolt replication (a) 2-times replicated and (b) 8-times replicated.

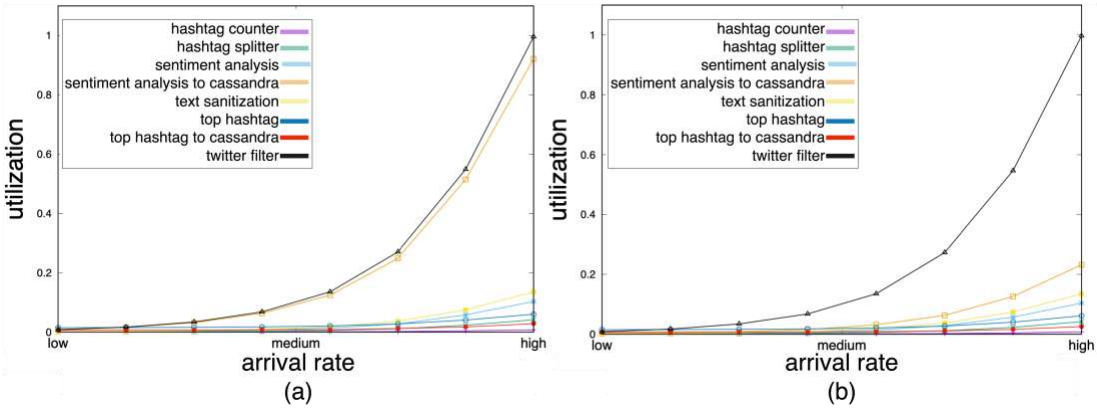

Figure 5: Utilization per bolt with 4-times replicated **twitter filter** bolt (fixed) and different levels of **sentiment analysis to cassandra** bolt replication: (a) 2-times replicated, (b) 8-times replicated.

Figure 6 shows the level of utilization per bolt varying the tweet arrival rate and the number of **sentiment analysis to cassandra** bolt replicas and fixing the number of **twitter filter** bolt replicas to eight. Figure 6.(a) shows a reduction in terms of the level of utilization of the **sentiment analysis to cassandra** bolt by using two replicas but is still close to 100% for high tweet arrival rate. Figure 6.(b) shows how the level of utilization of the **sentiment analysis to cassandra** bolt is decreased to values close to 24% with eight replicas.

Figure 3, Figure 4, Figure 5 and Figure 6 indicate that if we want to maintain the level of utilization close to 40%, in order to prevent sudden peaks in the tweet arrival rate, the number of **twitter filter** bolts must be close to 8 replicas and the number of **sentiment analysis to cassandra** bolts must be close to 8 replicas as well.

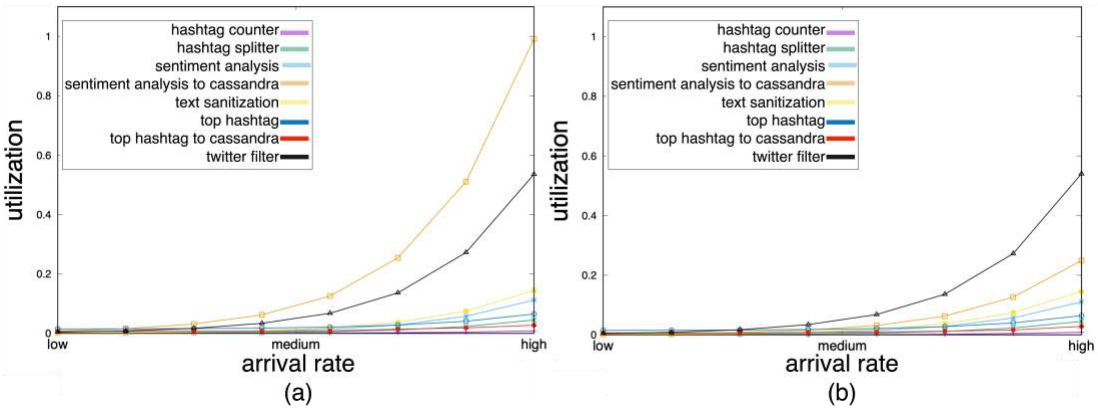

Figure 6: Utilization per bolt with 8-times replicated twitter filter bolt (fixed) and different levels of sentiment analysis to cassandra bolt replication: (a) no replication; (b) 8 replicas.

7 CONCLUSIONS

In this work, we simulated and evaluated a sentimental analysis application running on top of the Storm stream processing platform. We selected the Storm platform as a case study because it has many use cases and it is easy to integrate with other technologies, like database technologies. It also supports all the desirable characteristics of a stream processing platform, such as fault tolerance, and it offers scalability.

We model each component of the application and how they interact to represent the task processing flow. The application components are deployed on a distributed platform; therefore, we also simulated the communication network, particularly a Fat-Tree network.

Our simulator is designed in a modular way, so each component representing a specific task to be processed in the distributed platform can be easily replaced by other components. In the same way, we can simulate different network topologies. We evaluate the performance of the application running on a Storm platform. Experiments showed that our simulator is capable of reproducing the results achieved in a real system. Moreover, our simulator shows how the application behaves as we increase the number of replicas for specific components to avoid bottlenecks.

In our future work, we plan to simulate different topologies at the same time (already supported, though) and include a fault injector to allow the evaluation of different fault tolerance strategies.

ACKNOWLEDGMENTS

This work has been partially supported by Fondecyt de Iniciación 11230961 from ANID, Chile. Part of this work has been funded by INICI-UV UVA 20993 program from Universidad de Valparaíso, Chile.

REFERENCES

Al-Fares, M., A. Loukissas, and A. Vahdat. 2008. “A Scalable, Commodity Data Center Network Architecture”. *ACM SIGCOMM computer communication review* 38(4):63–74.

Amarasinghe, G., M. D. de Assunção, A. Harwood, and S. Karunasekera. 2018. “A Data Stream Processing Optimisation Framework for Edge Computing Applications”. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, 91–98.

Amarasinghe, G., M. D. de Assuncao, A. Harwood, and S. Karunasekera. 2020. “ECSNeT++: A Simulator for Distributed Stream Processing on Edge and Cloud Environments”. *Future Generation Computer Systems* 111:401–418.

Amini, L., H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. 2006. “SPC: A Distributed, Scalable Platform for Data Mining”. *DMSSP ’06*, 27–37. New York, NY, USA: Association for Computing Machinery.

Andrade, H. C. M., B. Gedik, and D. S. Turaga. 2014. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. New York, NY, USA: Cambridge University Press.

Gil-Costa, V., E. Tapia, and M. Marin. 2016. "Asynchronous Approximate Simulation Algorithm for Stream Processing Platforms (WIP)". In *Proceedings of the Summer Computer Simulation Conference, SCSC '16*, 52:1–52:6.

Higashino, W. A., M. A. Capretz, and L. F. Bittencourt. 2016. "CEPSim: Modelling and Simulation of Complex Event Processing Systems in Cloud Environments". *Future Generation Computer Systems* 65:122–139.

Hoseiny Farahabady, M., J. Taheri, A. Y. Zomaya, and Z. Tari. 2021a. "Energy Efficient Resource Controller for Apache Storm". *Concurrency and Computation: Practice and Experience*:e6799.

Hoseiny Farahabady, M. R., J. Taheri, A. Y. Zomaya, and Z. Tari. 2021b. "Graceful Performance Degradation in Apache Storm". In *Parallel and Distributed Computing, Applications and Technologies: 21st International Conference, PDCAT 2020*, Shenzhen, China, December 28–30, 2020, Proceedings 21, 389–400. Springer.

Kaptein, M. 2014. "RStorm: Developing and Testing Streaming Algorithms in R". *The R Journal* 6(1):123–132.

Kroß, J., and H. Krcmar. 2019. "Pertract: Model Extraction and Specification of Big Data Systems for Performance Prediction by the Example of Apache Spark and Hadoop". *Big Data and Cognitive Computing* 3(3):47.

Kulkarni, S., N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. 2015. "Twitter Heron: Stream Processing at Scale". In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15*, 239–250. New York, NY, USA: Association for Computing Machinery.

Marzolla, M. 2004. "LibCppSim: A SIMULA-like, Portable Process-Oriented Simulation Library in C++". In *ESM 2004*.

Marçal, Serrate 2017. Analysis of Twitter Streams with Kafka and Storm. https://serrate.net/2016/01/05/analysis-of-twitter-streams-with-kafka-and-storm/, Accessed on April 12th, 2023.

Muhammad, A., and M. Aleem. 2021. "A3-Storm: Topology-, Traffic-, and Resource-Aware Storm Scheduler for Heterogeneous Clusters". *The Journal of Supercomputing* 77:1059–1093.

Muhammad, A., M. Aleem, and M. A. Islam. 2021, mar. "TOP-Storm: A Topology-Based Resource-Aware Scheduler for Stream Processing Engine". *Cluster Computing* 24(1):417–431.

Park, A. J., C.-H. Li, R. Nair, N. Ohba, U. Shvadron, A. Zaks, and E. Schenfeld. 2010. "Flow: A Stream Processing System Simulator". In *Proceedings of the 2010 IEEE Workshop on Principles of Advanced and Distributed Simulation, PADS '10*.

Satzger, B., W. Hummer, P. Leitner, and S. Dustdar. 2011. "Esc: Towards an Elastic Stream Computing Platform for the Cloud". In *2011 IEEE 4th International Conference on Cloud Computing*, 348–355. IEEE.

SimStream, SimStream GitHub repository. https://github.com/alonso-inostrosa/simstream, Accessed on April 12th, 2023.

Storm, Apache Software Foundation 2015. Companies Using Apache Storm. https://storm.apache.org/Powered-By.html, Accessed on April 12th, 2023.

Toshniwal, A., 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. 2014. "Storm@Twitter". In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, 147–156. New York, NY, USA: ACM.

Zaharia, M., T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. 2013. "Discretized Streams: Fault-Tolerant Streaming Computation at Scale". In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13*, 423–438. New York, NY, USA: Association for Computing Machinery.

AUTHOR BIOGRAPHIES

ALONSO INOSTROSA-PSIJAS received the Ph.D. degree from Universidad de Santiago, Chile. He is currently an Associate Professor at the School of Informatics Engineering at Universidad de Valparaiso, Chile. His research interests are discrete-event and parallel/distributed simulation. He can be contacted at alonso.inostrosa@uv.cl.

ROBERTO SOLAR holds a Ph.D. in High-Performance Computing from Universitat Autonoma de Barcelona, Spain. His research interests include discrete-event simulation, agent-based modeling and simulation, and similarity search in metric spaces. His e-mail address is. His email address is roberto.solar@usach.cl.

MAURICIO MARIN is a former researcher at Yahoo! Labs Santiago hosted by the University of Chile and currently a Full Professor at the University of Santiago, Chile. He holds a Ph.D. in Computer Science from the University of Oxford, UK. His research work is on parallel computing and distributed systems with applications in query processing and capacity planning for Web search engines. His email address is mauricio.marin@usach.cl.

VERONICA GIL-COSTA received her Ph.D. in Computer Science, both from Universidad Nacional de San Luis (UNSL), Argentina. She is a former researcher at Yahoo! Labs Santiago. She is currently an Associate Professor at the University of San Luis and a researcher at the National Research Council (CONICET) of Argentina. Her email address is gvcosta@unsl.edu.ar.

GABRIEL WAINER received the Ph.D. degree from Université d'Aix-Marseille III. He is a Full Professor at Carleton University. His current research interests relate to modeling methodologies and tools, parallel/distributed simulation, and real-time systems. His e-mail is gwainer@sce.carleton.ca.