IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 20, 2020, accepted December 1, 2020, date of publication December 4, 2020,
date of current version December 16, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3042739

Boosting Big Data Streaming Applications in Clouds With BurstFlow

PAULO RICARDO RODRIGUES DE SOUZA, JR.,[1], (Graduate Student Member, IEEE), KASSIANO J. MATTEUSSI[1], (Member, IEEE), ALEXANDRE DA SILVA VEITH[2], BRENO F. ZANCHETTA[1], VALDERI R. Q. LEITHARDT[3,4,5], (Member, IEEE), ÁLVARO L. MURCIEGO[6], EDISON PIGNATON DE FREITAS[1], (Member, IEEE), JULIO C. S. DOS ANJOS[1], AND CLAUDIO F. R. GEYER[1]

[1]Institute of Informatics, Federal University of Rio Grande do Sul, UFRGS/PPGC, Porto Alegre 91501-970, Brazil
[2]Department of Computer Science, University of Toronto, Toronto, ON M5S 2E4, Canada
[3]COPELABS, Universidade Lusófona de Humanidades e Tecnologias, 1749-024 Lisbon, Portugal
[4]Departamento de Informática, Universidade da Beira Interior, 6201-001 Covilhã, Portugal
[5]VALORIZA Research Center, Instituto Politécnico de Portalegre, 7300-555 Portalegre, Portugal
[6]Faculty of Science, Expert Systems and Applications Laboratory, University of Salamanca, 37008 Salamanca, Spain

Corresponding author: Paulo Ricardo Rodrigues De Souza, Jr. (paulosouzjunior@gmail.com)

This work was supported by the ''SmartSent'' under Grant 17/2551-0001 195-3, CAPES (Finance Code 001), PNPD program, CNPq, PROPESQ-UFRGS-Brasil and FAPERGS Project ''GREEN-CLOUD—Computação em Cloud com Computação Sustentável'' under Grant 16/2551-0000 488-9, partially supported by Project Smart following systems, Edge Computing and IoT Consortium, CONSORCIO TC_TCUE18-20_004, CONVOCATORIA CONSORCIOTC. PLAN TCUE 2018−2020. Project managed by Fundación General de la Universidad de Salamanca and co-financed with Junta de Castilla y León and FEDER funds, in part by the postdoctoral fellowship from the University of Salamanca and Banco Santander, and in part by the ''Fundação para a Ciência e a Tecnologia'' under projects UIDB/04111/2020 and FORESTER PCIF/SSI/0102/2017.

ABSTRACT The rapid growth of stream applications in financial markets, health care, education, social media, and sensor networks represents a remarkable milestone for data processing and analytic in recent years, leading to new challenges to handle Big Data in real-time. Traditionally, a single cloud infrastructure often holds the deployment of Stream Processing applications because it has extensive and adaptative virtual computing resources. Hence, data sources send data from distant and different locations of the cloud infrastructure, increasing the application latency. The cloud infrastructure may be geographically distributed and it requires to run a set of frameworks to handle communication. These frameworks often comprise a Message Queue System and a Stream Processing Framework. The frameworks explore Multi-Cloud deploying each service in a different cloud and communication via high latency network links. This creates challenges to meet real-time application requirements because the data streams have different and unpredictable latencies forcing cloud providers' communication systems to adjust to the environment changes continually. Previous works explore static micro-batch demonstrating its potential to overcome communication issues. This paper introduces BurstFlow, a tool for enhancing communication across data sources located at the edges of the Internet and Big Data Stream Processing applications located in cloud infrastructures. BurstFlow introduces a strategy for adjusting the micro-batch sizes dynamically according to the time required for communication and computation. BurstFlow also presents an adaptive data partition policy for distributing incoming streams across available machines by considering memory and CPU capacities. The experiments use a real-world multi-cloud deployment showing that BurstFlow can reduce the execution time up to 77% when compared to the state-of-the-art solutions, improving CPU efficiency by up to 49%.

INDEX TERMS Big data, stream processing applications, multi cloud, micro-batches, data partition.

I. INTRODUCTION

The advent of the Internet of Things (IoT) has led to new challenges in the Big Data era due to the limitations of storage, computation, and communication of existing devices since IoT devices generate massive amounts of data that require processing to support the decision-making process. The data processing happens in two ways: Batch Processing (BP), which explicitly manipulates large datasets of historical data with high latency, and Stream Processing (SP), which

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

performs (near) real-time analysis over continuous data flows using in-memory processing.

In a traditional approach, IoT devices located at the edge of the Internet produce data at overwhelming rates. This data traverses high distance links to be consumed on faraway cloud providers by applications of the financial market, call services, automation in Industry 4.0, etc [1]–[4]. In SP time-sensitive applications, this approach can produce high overheads making it hard to achieve (near) real-time analytics. It happens because SP applications often handle data (*i.e.*, event) one at a time, leading to adding the network latency to each of the events with regular protocols. The generated latency is the major problem for critical decision-making applications, for instance, in the intrusion detection system where a hacker intrusion can mean financial damage [5]. Furthermore, SP applications can have unpredictable data bursts leading to bandwidth contentions [6], [7].

SP frameworks such as Apache Flink [8] and Apache Spark [9] cannot orchestrate communication across multiple data centers leading to high application latency and low throughput. However, some solutions overcome this limitation by supporting applications in Multi-Cloud (MC) infrastructure and exploring the locality of micro data centers. These micro data centers also allow creating batches of data to add the network latency in data transfers to a set of events rather than to each event [10]. Micro-batch is an attractive method in SP because it permits to achieve better throughput and application latency [11]. Previous works attempt to employ micro-batch strategies using an absolute number of events [12], [13] or time [14] for creating the batches. These solutions neglect how to partition the micro-batches across the heterogeneous computing resources where the SP framework is deployed [15] – leading to higher execution time, demonstrating the lack of solutions that couple the micro-batch policy and the SP framework setup.

This work describes BurstFlow, a tool for dynamically handling data bursts in a geographical infrastructure and distributing the input data across multiple SP operation partitions. BurstFlow improves the communication between data centers at the network edge and cloud computing by transmitting data into micro-batches. The micro-batch size is defined by considering the message life time in the whole system. Furthermore, BurstFlow introduces an adaptive method to distribute incoming streams in the SP framework. Real-world experiments show an improvement of over 9% in the execution time, over 49% better CPU and memory utilization compared to methods applied to data partitioning in Apache Flink and state of the art.

The contributions of this work are:

- BurstFlow, a tool for orchestrating Big Data SP applications in MC infrastructure using monitored information from the application data flow and the resources (Section IV-B);
- BurstFlow's Execution Time-Aware Micro-Batch Strategy (ETAMBS) extends the standard negotiation method of buffer sizes in the Big Data engine communication, embedding a dynamic adjustment to determine the micro-batch sizes dynamically to overcome the overhead of the network latency (Section IV-C);
- BurstFlow's Resource-Aware Partition Policy (RAPP) distributes the incoming micro-batches across the running operator replicas assigning micro-batches by considering memory and CPU (Section IV-D); and
- A prototype and a performance evaluation comparing BurstFlow against the state-of-the-art solutions (Section V).

This work is structured as follows: Section II presents a review of the related work, followed by Section III that defines the evaluated problem in this work; Section IV details the BurstFlow and strategies to create micro-batches and the adopted partition policy; Section V presents the experiment setup, the prototype, the adopted methodology and then discusses the achieved results and threats to validity. Finally, Section VI concludes the paper by providing directions for future work.

II. STATE-OF-THE-ART IN ADAPTIVE STREAM PROCESSING

Cloud computing is a robust environment to perform large-scale and complex computations as it provides security, efficiency, flexibility, pay-as-you-go billing structure, and a scalable data storage [16], [17]. As a result, many organizations have explored it to support data-intensive applications and services in the most diverse domains.

One of the explored contexts is (near) real-time SP applications that require low-latency processing over unpredictable and continuous flows of data [18]. Data arrives from data sources spread in geographically distributed areas. In this process, each generated data requires delivery guarantees to prevent data loss. To achieve these requirements, many SP systems are created on top of a stack of components. For instance, the data source produces data to a queue in a Message Queue System (MQS), and the SP framework consumes from the queue. As a result, each component of the system is often located in a different data center.

The baseline approach implemented in cloud-based frameworks such as Apache Spark, Apache Flink, and Apache Heron [19] consider a single cloud data center to run the MQS and the SP framework. Furthermore, the SP applications consume one message at-at-time from the MQS because they assume the data source is co-located in the data center, neglecting the existing network latency. Another issue is that cloud-based frameworks count on homogeneous workloads when distributing data in the SP framework and neglect performance metrics such as data ingestion, memory, or network utilization. Consequently, applications suffer from resource-related interference problems (e.g., memory and network contention) due to the lack of control and management, incurring low throughput and failures. In the following, there is an analysis of works representing state of the art in adaptive SP for heterogeneous and distributed systems.

In JetStream [12], the approach uses available communication channels allowing multiple stream routes among cloud sites for supporting the largest batch size transfer possible. To achieve this goal an algorithm computes both data transfers and the batch sizes to find one convergence point between the transference of batches seeking the lowest execution time. The data transmission strategy use multicast algorithm (*i.e.* transmits a single message that is replicated to all nodes in a group). The orchestrator uses a oracle that decide when to send a message to the transfer module. The evaluations consider the aggregation of up 1000 events and measure the latency response of data transfer between different cloud data centers. Therefore, JetStream selects the best route – lowest latency – to transfer batches. However, it takes into account only the communication and batch size and lacks on data distribution to optimize processing throughput.

Das *et al.* [14] offer an adaptive batch sizing algorithm for SP systems based on a fixed-point iteration strategy. It is a well-known numerical optimization technique that allows the system to adapt to the window size when data ingestion varies too much. The approach defines a minimum batch size to achieve execution stability. Thus, it is possible to minimize end-to-end latency while keeping the system stable based on batch statistics. This strategy allows better use of system resources because it avoids processing delays and low performance, which occurs due to load spikes that lead the processing to built-up queue conditions. The deployment defines a job controller to manage the batch size and a job processor that provides feedback about job execution status. This approach is implemented in the Apache Spark and does not offer neither indirect communication or load balance.

Zhang *et al.* [20] study batch and block interval as the most important factors affecting the performance of Spark Streaming, such as the application latency problem. The work discusses how long waiting time affects the correctness of the latest completed batch interval statistics, letting it out of date because they usually cannot reflect the workload conditions. A heuristic algorithm is built using the waiting time and an isotonic regression to dynamically adapt the batch and block interval to workloads and operating conditions. The algorithm quickly converges to the desired batch and block interval and continuously adjust both based on previous data rates, processing time, and block interval. Unfortunately, this work is restricted for Reduce and Join workloads and presents a small testbed with few processing nodes and low workload throughput (≈4MB/s). This approach does not address the real-time processing conditions.

Anjos *et al.* [21] identify the relation between stream rate of income and time processing variation of Apache Flink. The experiments simulate an orchestrator that provides the data dispatches to virtual machines in accordance with its computational capacity. The machines are grouped by computational capacity similarity. These experiments confirm that the data generation has low communication impacts in comparison with high network latency. The study demonstrates the best performance is achieved with greater data blocks, but in comparison with BurstFlow it does not provide a dynamic solution to the stream processing adjusts.

Fernandez *et al.* [22] proposed the Liquid a data integration system to provide low latency for data access to batch applications. The implementation provides the incremental processing and keeps producers and consumers decoupled with higher availability. The message layer use the RAM cache of the operational system to achieve high-throughput writes. The orchestrator is the Apache Zookeeper [23] that uses Apache Kafka to message management and the Apache Samza [24] to manage stream processing. The system provides a message approach of at-least-once delivery semantics for idempotent updates. The low latency increments the processing using annotations with metadata. Nowadays the implementation uses Microsoft Azure.

Gulisano *et al.* [25] investigate the resource contention problem associated with the auto-parallelization of running queries in distributed SP frameworks. The authors present the execution of query tasks on worker nodes that may lead to shared resource contention. The proposed solution contains an adaptive feedback controller based on Model Predictive Control (MPC). It ensures the elastic allocation, as observed by the improvement of 15% in total servers resource utilization with an average reduction of 14% in tuple latencies. The proposed solution also reduces the Quality of Service (QoS) violation incidents by 123% (maximum 207%) compared to the round-robin heuristic, which uses all available resources in the cluster farm.

Another aspect of key importance to improve performance in SP applications is memory management. Zhao *et al.* [26] propose a resource-aware cache management solution for in-memory processing. The design of this solution aims to enhance the cache utilization of executors by using a heuristic method based on sub-modular optimization theory and data dependency information to evict and prefetch data blocks from memory appropriately. It improves memory access; however, it does not assume data distribution, which in a distributed environment will not oversee latency constraints in applications.

Xiu *et al.* [27] aim to improve memory allocation for storage and execution areas. The main idea is to optimize and reduce cache loss and memory overflow to improve task execution efficiency. Similarly, Tang *et al.* [28] introduce a Dynamic Spark Memory-Aware Task Scheduler (DMATS) algorithm to treat memory and network I/O through feedback information to utilize these resources optimally. However, both solutions do not improve data consumption and do not acknowledge latency and data partition inside the SP.

Table 1 summarizes the main techniques for adaptive SP found in the Related Work. SP has been applied in cloud environments to perform processing independently on multiple sites. Moreover, MC have important characteristics to consider, as heterogeneity and resource variation, such as network bandwidth, memory, and CPU pools. For instance, network latency can significantly vary depending on the geographic location and the processing flow of the

TABLE 1. Techniques for Adaptative Stream Processing.

		Infrastructures					Strategies															Proc	
	Author	Geo-Distributed	Cloud Infrastructure	Cluster	Multi-Cloud	Hybrid Infrastructure	Window Size	Time Interval	Throughput Evaluation	Application Evaluation	Query Evaluation	Computation Capacity	Memory Management	Cache Management	Iteration Strategy	Incremental processing	Dynamic Approach	On Demand Approach	Data Movement	Adaptive Approach	Group Strategy	Batch Monitoring	Batch	Streaming
Approaches	JetStream [12]				x				x	x										x		x		x
	Das *et al.* [14]		x				x								x					x		x	x	
	Fernandez *et al.* [22]		x						x							x							x	
	Gulisano *et al.* [25]			x						x	x							x		x				x
	Zhang *et al.* [20]		x					x									x					x		x
	Anjos *et al.* [21]	x				x			x			x							x		x		x	
	Zhao *et al.* [26]		x											x	x		x		x					x
	Xiu *et al.* [27]		x										x											x
	Tang *et al.* [28]		x						x				x				x							x
	BurstFlow	x	x		x		x		x				x				x			x		x	x	x

application [12]. Still, the number of CPUs or memory can considerably differ, requiring well-defined scheduling techniques to support workload variations.

BurstFlow diverges in a key point from strategies presented in Table 1: the data aggregation due to flow partition control based on the memory and the application throughput management. Unlike the works found in the literature, BurstFlow uses an adaptive and dynamic model to estimate the number of events per message based on feedback loops that monitor the batch size in the memory of workers to further forward data. This approach leads to overcoming contention scenarios while maintaining network stability.

Typically, computational resources are shared simultaneously among multiple users, applications, and mixed workloads. BurstFlow addresses an actual concern in Big Data processing in MC, and it manages resources appropriately. Regardless, SP applications tend to process in memory to increase events throughput. Unfortunately, the amount of memory is limited, and as it is used for both in-memory processing and storage, it can be a further bottleneck in the system. BurstFlow explores more efficiently the memory avoiding swaps from memory to disk and vice-versa.

This section investigated current concerns about maintaining system stability and high event throughput. Nevertheless, the analysed state-of-the-art solutions reveals a research gap between adaptive solutions to improve network issues and memory management for SP in heterogeneous and distributed systems. Thus, BurstFlow explores this gap to propose a flow partition approach.

III. PROBLEM STATEMENT

Sensors located at the edge of the Internet produce data at burst rates to be consumed by SP applications placed on cloud servers. These servers are often faraway from the data sources requiring to traverse the Internet via high latency links subject to package loss [29]. To manage communication, SP applications need to use a set of tools to provide a stable data transmission infrastructure such as MQSs [30] and MC. For instance, a MQS is used to manage the transfer and guarantee message delivery from different sensors to a micro data center or the cloud service provider. MQS provide delivery semantics such as at-most-once, at-least-once, or exactly-once.

SP application retrieves messages from the MQS utilizing consumer clients. Consumer clients often connect to the MQS via TCP. Hence, the MQS offers multiple settings to the users in order to achieve better performance. This happens because the user can work either on Local Area Network (LAN), Wide Area Network (WAN) or both. The consumer buffer size is essential because permits to configure how much data the sockets use when reading/writing data – the buffer size depends on latency and the available bandwidth. The consumer often uses cache buffers to maintain the storage messages in memory, speeding up the process of reading done by the SP framework.

Furthermore, SP frameworks often handle messages *one-at-a-time*, which can turn infeasible to work with MC as the network link latency is appended to each data transfer. An attractive solution is to use micro-batches where messages are accumulated in batches before being transferred, adding the network latency overhead a single time. Nonetheless, most SP applications are time-sensitive, thus the processing time for each message matters for the system. For instance, an incorrect configuration to the batch size of the micro-batch can drive to high processing times and comprising real-time constraints.

Figure 1 shows a bar plot to represent the impact of each micro-batch size where the micro-batch size is shown in the x-axis and the execution time in seconds in the y-axis. The used infrastructure and application configuration are provided in Section V. The results present a high execution time for the *one-at-a-time* data transfer (micro-batch size equals one). This happens because message executions are subject to network latency and processing time. In contrast,

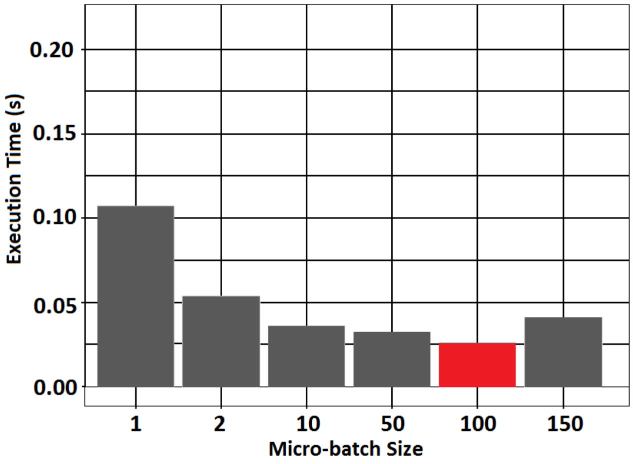


FIGURE 1. Analysis of different micro-batch sizes. The red color shows the better micro-batch size.

a large micro-batch size demands a long wait time to collect the target number of messages due to the production rate of messages in the sensors and higher execution time to process the whole batch in the SP framework. According to the bar, the micro-batch size with 100 is the best execution time. Since the network latency and the processing requirements can vary, then a dynamic policy to establish the batch size is required.

The policy to determine the batch sizes must consider the information of the network as well as application metrics and configurations of the SP framework. A good policy for the batch sizes results in better throughput and mitigate latency in the SP application. This is a challenge due to a lack of solutions that orchestrate SP applications in MC. This work addresses both problems by introducing an orchestration tool to determine a batch size policy that considers the stack of tools required to run SP applications in the MC environment.

This work also addresses the problem of high processing times due to the unbalance of data consumption by multiple consumer clients of SP applications. An SP application is often structured as a directed graph whose vertices are operators that perform transformations over the incoming data and edges representing the data streams between operators. When the SP application is submitted to the framework, the user can inform the parallelism degree to the application operators. Each operator replica connected to the MQS creates a consumer client, which will consume messages *one-at-a-time* in a round-robin fashion (*i.e.*, Baseline) from a given queue. The Baseline was conceived for clusters of homogeneous computing resources in LANs. However, this method applied to heterogeneous infrastructure leads to performance depreciation as computing and communication have different speeds and can change along time.

IV. BurstFlow: A MECHANISM TO BOOST THE THROUGHPUT OF SP APPLICATIONS IN MULTI CLOUD

This section details BurstFlow and its strategies proposed for placing creating micro-batches onto MC infrastructure while improving the application throughput and the aggregate message latency.

A. SYSTEM OVERVIEW

This work considers a scenario where data sources (e.g., sensors) located at the edge of the Internet send their produced messages in micro-batches to an MQS placed on a micro data center closer to the data sources. An SP framework located on a different cloud service provider than the MQS consumes the micro-batches from the MQS. BurstFlow orchestrates the communication between each data source and the SP framework by deploying a set of services (Section IV-B) to determine the size of micro-batches and distribute the workload across the operator replicas dynamically. Each data source runs a BurstFlow service called Data Orchestrator, which has a holistic view of the execution time of a micro-batch – *i.e.*, the execution time is the difference between the timestamp when the last message of the micro-batch is processed by the SP framework and the timestamp when the first message of the micro-batch was created by the data source.

Furthermore, BurstFlow computes the size of the micro-batches using the ETAMBS (Section IV-C) to achieve better throughput and per-message latency, either considering the communication and computation times. BurstFlow also runs the RAPP (Section IV-D) along with the SP framework, which receives the micro-batches and distributes them across the operator replicas by taking into account the resource utilization of each machine of the SP cluster. When the operator replica receives the micro-batch, it parses the micro-batches and then ingests the messages into the application dataflow.

B. BurstFlow ARCHITECTURE

BurstFlow was designed to be a multi-resource dispatcher for geographically distributed and heterogeneous MC environments. Its main goal is to observe the micro-batch lifespan and make scheduling decisions to reduce its execution time. The scheduling decisions use algorithms to establish the size of the micro-batches and balance the workload across operator replicas that consume data from the MQS.

The design also considers a stack of frameworks and services for processing a micro-batch – *i.e.*, data sources, MQS, and SP framework. Hence, BurstFlow has components strategically deployed in infrastructure in order to capture individual latencies from the time where a data source generates a message up to the time in which the SP framework completely processes it. This structure permits the BurstFlow to adjust the micro-batches in each data source to reduce their overall processing time.

Figure 2 introduces BurstFlow architecture. The messages generated by a data source arrive at the *Data Orchestrator* co-located with the data source. The *Data Orchestrator* runs the ETAMBS (Section IV-C), which determines the number of messages to be accumulated in the micro-batch. ETAMBS interacts with the *Resource Watch* to be able to compute the micro-batch size by analyzing how previous micro-batch sizes behaved. Hence, messages are retained in a micro-batch

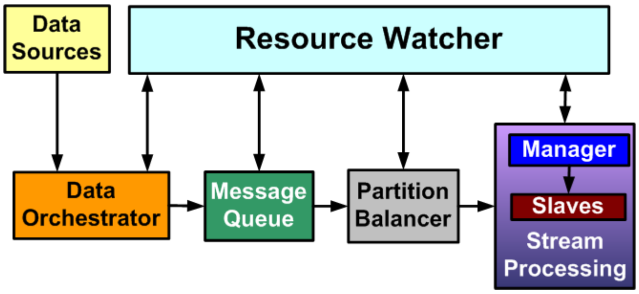


FIGURE 2. BurstFlow architecture.

Algorithm 1 Aggregation Algorithm

```
1: procedure send($msg$)
2:     if $buffer.size = 0$ then
3:         $buffer.setID()$
4:         $buffer.setTimestamp(msg.GetTimestamp())$
5:     $buffer.append(msg)$
6:     if $buffer.size() >= GetMaxBufferSize()$ then
7:         $send\_micro\_batch(buffer)$
8:         $buffer.clear()$
```

before achieving its target size. Then, the micro-batch is sent to a queue in the MQS.

The MQS consumers in the *SP Framework* drain the messages from the MQS queue. When the message arrives at the *SP Framework*, the *Master* decides how to distribute the messages across the operator replicas (*i.e*, *Slaves*) using the RAPP (Section IV-D) from the *Partition Balancer*. The strategy gets available and used memory and CPU of each operator replica's resource to decide where to send a micro-batch. After the decision, the *Manager* sends the micro-batch message and then SP application consumes the individual messages of the micro-batch.

C. Execution Time-Aware Micro-Batch Strategy (ETAMBS)

BurstFlow adjusts dynamically the size of a micro-batch in the data source side taking into account the whole flow of communication and computation. Each data source accumulates messages in a micro-batch before sending it to the MQS.

Aggregation Algorithm 1 presents how BurstFlow handles each message's arrival when forwarding a micro-batch to the MQS. The algorithm maintains a global buffer where it accumulates the messages for the micro-batch. This buffer has properties such as ID and Timestamp, which allows the system to keep track of its execution time. When including the first message, the system attributes a unique ID and sets the timestamp when the received message was created (lines 2-4). In the next step, the algorithm appends the message to the buffer (line 5). Then, the number of messages in the buffer is compared to the *GetMaxBufferSize* function (line 6). If the number of messages is higher or equal to the output of the function then the micro-batch is sent to the MQS and the buffer is reinitialized (lines 7-8).

The *GetMaxBufferSize* function starts returning an aggregation of 2 messages for the micro-batch. This function interacts with the Resource Watcher by checking the execution time required to process the micro-batch. The execution time of a micro-batch is retrieved using a unique ID. The execution time permits to create of a rate by dividing the micro-batch size and the execution time. The rate is stored in vector as a tuple ($\boldsymbol{x}$, $\boldsymbol{r}$) where $\boldsymbol{x}$ denotes the evaluated micro-batch size, and $\boldsymbol{r}$ expresses the average message time. During the convergence phase, each function call increments one to the micro-batch size, and the rate is stored in the rate vector. If the current rate is higher or equal to the previous micro-batch size, then the algorithm picks the micro-batch size from the previous position of the vector leading to conclude the convergence phase. For example, if the current micro-batch size is 6 with rate 3, and the previous micro-batch size is 5 with rate 2 then the *GetMaxBufferSize* function returns 5, and in the next call of the function, it will return the same value.

ETAMBS implements a service, which controls the time required to build a micro-batch. This happens because the ingestion rate in the data sources can change. If the time to achieve the micro-batch size is higher than its rate, then the micro-batch is sent to the MQS and the convergence phase is reinitialized. If the variance between the current and the last saved rate is higher than 30% then the convergence phase is also reinitialized. This trigger permits to the detection of anomalies in the execution of the micro-batches, such as higher or lower network latency or computing capacity for the MQS and SP framework.

D. Resource-Aware Partition Policy (RAPP)

The RAPP distributes the incoming micro-batches across the running operator replicas assigning micro-batches by employing a ratio between memory and CPU of each machine where each operator replica is running. The Resource Watcher provides the CPU and memory. RAPP creates an affinity list identifying the most idle machine by paring the ratio and the operator replica ID. In order to simplify the micro-batch assignment, RAPP orders the affinity list by the ratio in a decent manner. If there exist operator replicas with the same ratio, then RAPP puts the operator replica with the most powerful machine in a higher position. Thus, the partition algorithm does not unpack the micro-batch when it arrives. First, it picks the operator replica at the top of the list to assign the micro-batch. At last, the micro-batch arrives at the operator replica, where it is unpacked using a parsing function, and the operator consumes each message.

RAPP computes the affinity list every five seconds to get the system changes and distribute the workload fairly across the operator replicas. Updating the affinity list starts by interacting with the Resource Watcher to get the total and available memory, and the average CPU consumption percentage of each operator replica's machine. RAPP transforms the incoming metrics in the percentage of available memory and CPU. Then, the ratio is computed by the average of the percentage of available memory and CPU.

V. PROTOTYPE, EXPERIMENTAL SETUP AND PERFORMANCE EVALUATION

This section presents the setup and methodology of the experiments, the implementation of a prototype, and discusses the obtained results with BurstFlow.

A. BurstFlow PROTOTYPE

The software stack of the BurstFlow prototype is shown in Figure 3. The Data Orchestrator was developed using Python with open source libraries to open connections with the MQS. The MQS is the Apache Kafka 1.0.0 since it provides high-level API, scalability, system log, replication, and repartition. The Kafka API allows managing partitions of each topic and setting up the consumption using the partition manager. The used SP framework is Apache Flink 1.1.5 with Hadoop 2.4, which has proven to be an efficient and the state-of-the-art solution to process SP applications, providing further enhancements in data distribution and management. Moreover, the prototype utilizes the Hadoop Distributed File System (HDFS) as a Distributed File System (DFS) component and it runs along with the Apache Flink's Master.

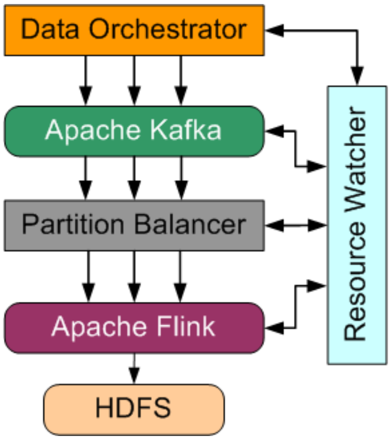


FIGURE 3. **BurstFlow prototype.**

The system provides the Java$^{\text{TM}}$ Platform, Standard Edition Development Kit (JDK$^{\text{TM}}$) 8.181, which is used by the core of Apache Flink and Kafka framework. It also comprises the JMX (Java Management Extensions) framework for the serialization schema and logging. The *Partition Balancer* was implemented by implementing a Partitioner[1] function in Apache Flink and calling it using the partitionCustom function in the data stream. It is partially written in Python and Java. It is required to be in Java since Flink API does not offer the physical management of partitions in any other language than Java. The *Partition Balancer* is configurable in the dispatcher deployment and applies the partitioning algorithms. It is statically defined and requires to redeploy to change the algorithm. Although in Apache Flink, it is possible to use multiple algorithms to partition data streams inside the cluster in each operator.

The Resource Watcher consumes the logs files from the Apache Flink monitor, using the JMX monitor, like a service that provides insight into the cluster's states and conditions (*i.e.*, network throughput, memory and CPU consumption, etc.). It is written in Python and performs as a client that consumes information from the JMX monitor. It collects information regarding the resources and makes it available to make decisions accordingly.

B. EXPERIMENTAL SETUP

The experiments were performed over distributed data centers of the Microsoft Azure Cloud. The setup was designed to analyze the viability of the solution in an MC geographically distributed infrastructure as depicted in Figure 4, where:

- **Data Sources:** The messages are produced by 10 A3 instances in Brazil's South and each one produces data using 100 threads to mimic 100 sensors. Each thread runs a Data Orchestrator to create the micro-bathes and to forward them to the MQS.
- **MQS:** The MQS runs Apache Kafka in a A3 instance in the East US data center for temporarily storing messages in queues that are later consumed by the SP framework.
- **SP Framework:** The SP framework is composed by 5 Virtual Machiness (VMs) instances located in the West US data center with different resource capabilities: 1 A8 instance for the Master, 1 D11 instance for the Slave1, 1 A4 instance for the Slave2, 1 A2 instance for the Slave3 and 1 A3 instance for the Resource Watcher.

The details of each VM instance are shown in the Table 2. The Mirosoft Azure's VM sizes were designed and optimized for compute-intensive and network-intensive applications by Azure as well as, we follow and reproduce well-defined scenarios observed in the related work section. Each VM instance has the Intel Xeon E52670 2.6GHz processor, DDR3 1600 MHz RAM and the Operational system is Ubuntu Server 16.04. The clock of all VMs are synchronised using Network Time Protocol (NTP). Finally, we provided a real world scenario by setting up a heterogeneous environment with varied configurations in terms of hardware specification (VM sizes) and network round-trip latency,[2] see Table 3.

1) EVALUATED APPLICATION

The evaluation considers the sentiment analysis application, shown in Figure 5, which implements a typical SP application of sentiment analysis. The application classifies incoming tweets into two classes, positive and negative. The first operator receives the incoming data and parses it to an understand-

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.1/api/java/org/apache/flink/api/common/functions/class-use/Partitioner.html

[2] A full table can be found here: https://docs.microsoft.com/en-us/azure/networking/azure-network-latency

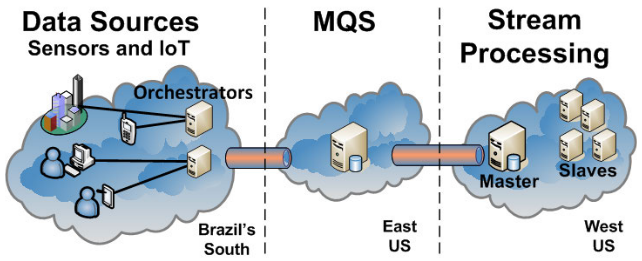


FIGURE 4. Experimental MC infrastructure.

TABLE 2. VM instance characteristics.

Size	#CPU Cores	Memory (GB)	Max Available Storage (GiB)	Network Bandwidth (Mbps)
A2	2	3.5	40	1000
A3	4	7	80	1000
A4	8	14	100	1000
A8	8	56	1023	2000
D11	2	14	200	1000

TABLE 3. Round-trip latency between Microsoft Azure regions.

Who	From	To	Latency (ms)
Data Sources	Brazil's South	East US	117
MQs	East US	West US	65
SP Framework	West US	West US	-

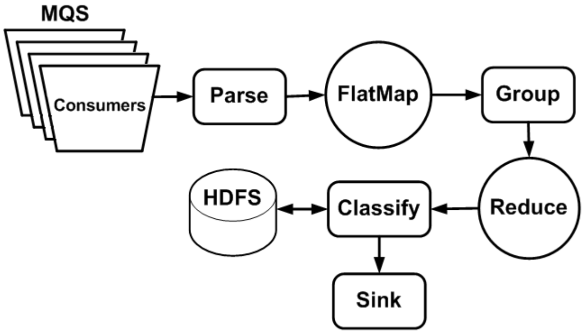


FIGURE 5. Sentiment Analysis application.

able format. Later, the next operator can perform a FlatMap to split all the words from every tweet into sets for each tweet. The next operator is responsible for grouping the data by keyword, and then the reduce operator sums up each word's total frequency in its individual tweet. The classification performs a standard naïve Bayes algorithm that analyzes the tweet word's frequency in a tweet against a positive and negative database. Therefore, the higher frequency defines the class as positive or negative, sending the result to the sink operator, which pushes the message to a message queue.

The used dataset[3] comprises *tweets* collected on-line using Twitter API between July 13, 2016, and November 10, 2016, related to the 2016 US election. Each tweet size is equivalent to 224 bytes at maximum (because the tweet size can vary), the input data size can achieve a 2.5 KB maximum by one micro-batch size [12]. The dataset is divided into slices of 2 GB, and each slice is assigned in a round-robin fashion across the 100 threads of the Data Source. This data distribution leads to having 200 GB in each experiment.

2) COMPARISON

BurstFlow is compared against the **Baseline** approach to demonstrate the benefits of ETAMBS regarding employing micro-batch methods. The Baseline is the state-of-the-art solution, and it is widely employed by SP applications because it handles messages *one-at-a-time* fashion allowing the system to achieve real-time analytics. This approach includes the network latency to each message, which can cause problems in MC. The BurstFlow's RAPP is compared against the state-of-the-art algorithms implemented in the standard version of Apache Flink. The **data partition algorithms** are described as follows:

- **Baseline:** This solution only considers *one-at-a-time* messages and distributes them in a round-robin fashion to the operator replicas;
- **Flink's Broadcast:** This solution sends each message to all operator replicas and picks the one that finishes first;
- **Flink's Random (Shuffle):** Distributes incoming micro-batches randomly across the operator replicas according to a uniform distribution for the next operation; and
- **Flink's Rebalance:** Distributes incoming micro-batches in a round-robin fashion, creating equal load per operator partition.

The **performance metrics** comprise:

- **Execution Time:** the time required to process the whole dataset.
- **Throughput:** each machine of the Apache Flink cluster has its outputs measured in messages and MB. The throughput is measured after the sink, which is usually the final step of an SP application.
- **Event Time:** the time per-message to traverse the whole infrastructure. This time is computed by dividing the micro-batch size and the Execution Time.
- **Used Memory and CPU:** the CPU and memory consumption of each machine of the Apache Flink's cluster.

The metrics were selected from the most commonly chosen for measuring SP systems and load balance algorithms. The throughput is measured using the JMX monitor. The JMX is a Java framework for monitoring Java applications, and it provides easy integration with multiple systems and Big Data frameworks from the Apache family. While the CPU and memory consumption are measured using dstat,[4] which is a tool for generating system resource statistics.

[3] A similar dataset can be found in < https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/PDI7IN >

[4] Available in https://linux.die.net/man/1/dstat

C. PERFORMANCE EVALUATION

The experiments were defined based on the evaluation of the design patterns proposed on Jain Methodology [31] where each set of experiment is carried out 30 times. The performance evaluation consists of a direct comparison between BurstFlow and the most common approach in state of the art, and an evaluation considering BurstFlow's RAPP and the state-of-the-art solutions for data partitioning.

1) BurstFlow VS. BASELINE

This experiment compares the performance of BurstFlow's ETAMBS and the Baseline. BurstFlow's ETAMBS uses the same policy the the Baseline (*i.e.*, round-robin) to distribute data across the operator replicas. The main objective is to demonstrate the achieved throughput in the application inside the cluster in comparison with baseline.

Figure 6 introduces the overall view of the throughput. The *y-axis* measures throughput in MB and the *x-axis* measures the execution time in seconds. At a first observation, the BurstFlow's ETAMBS begins with lower throughput than the Baseline because messages must be accumulated in the micro-batch. In contrast, the Baseline sends messages to the MQS as they are generated. However, when the micro-batch arrives at the SP framework, BurstFlow increases its throughput. BurstFlow's ETAMBS determines the micro-batch sizes dynamically, making the throughput more stable – low variance in the throughput (an average of 1.5GB/s) between the 500 seconds up to the end of the experiment. This happens because BurstFlow's ETAMBS includes a single network latency to a set of messages, while the Baseline includes every message.

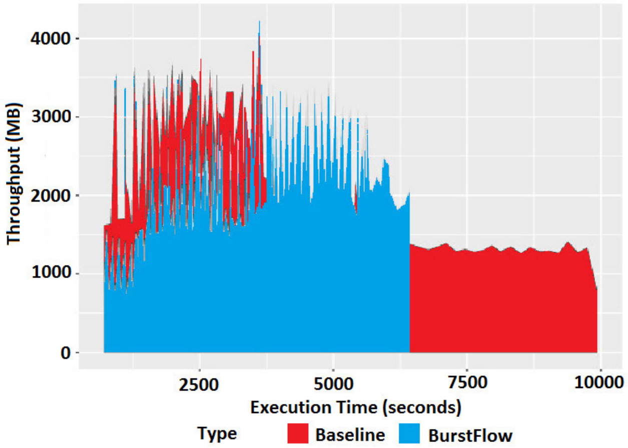


FIGURE 6. Throughput comparison between Baseline and BurstFlow.

2) THROUGHPUT, EXECUTION TIME AND RESOURCE CONSUMPTION EVALUATIONS

This set of experiments evaluates the impacts on throughput and CPU and memory consumption for consuming messages from the MQS and distributing them across operator replicas. The experiments adopt the default configuration on Apache Flink, where each core corresponds to one slot. Therefore, there are 24 available slots – master machine provides 8 slots, Slave1 2 slots, Slave2 8 slots, and Slave3 2 slots. The parallelism degree in Apache Flink is static as dynamic adjustments can not be made on-the-fly without stopping and restarting the whole system. The default configuration in Apache Flink deploys one pipeline per slot – *i.e.,* the whole sequence of operators of the Sentiment Analysis application. The experiments consider the state-of-the-art policies implemented in Apache Flink, BurstFlow's RAPP, and Baseline.

a: EXECUTION TIME AND THROUGHPUT

The first evaluation comprises an analysis of the execution time with and without BurstFlow's ETAMBS, as presented in Table 4. The results demonstrate the improvements when utilizing the proposed solution for determining the micro-batch sizes dynamically. BurstFlow's ETAMBS allows the system to reduce the execution time by over 8%. The benefits vary between the different data partition policies as each one distributes differently the incoming workloads. BurstFlow's ETAMBS permits operator replicas to receive micro-batches with several tweets. In contrast, no BurstFlow's ETAMBS requires each operator replica to consume one tweet at-a-time from the MQS. This data transfer adds an average of 65 ms to each tweet retrieval because of the network latency between the East US data center and the East US data center.

TABLE 4. Execution time in seconds with and without ETAMBS.

Algorithms	w/o ETAMBS	w/ ETAMBS	Gain (%)
BurstFlow's RAPP	6276	5400	16
Baseline	7380	7380	-
Flink's Broadcast	6364	5880	8
Flink's Random	10200	9180	11
Flink's Rebalance	10756	9540	13

Table 5 presents a comparison between BurstFlow's RAPP and state-of-the-art solutions by considering ETAMBS, except Baseline which considers one at-a-time. The table contains the execution time in seconds and the percentage of improvement that BurstFlow's RAPP achieved compared to the other solutions (% of Gain). The improvement is evident with any algorithm. In particular, the Flink's Broadcast sends a copy of each micro-batch to each operator replica, and it picks the outcomes from the operator replica who finishes first. This allows the Flink's Broadcast to achieve only % higher execution time than BurstFlow's RAPP. However, this performance improvement requires a higher effort in the network and computing resources because of wasting time processing spare copies of the micro-batches. Nevertheless, BurstFlow's RAPP is 37%, 70%, and 77% faster than Baseline, Flink's Random, and Flink's Rebalance respectively because it implements an algorithm to deal with heterogeneous workloads.

TABLE 5. Average execution time duration.

Algorithms	Execution time (s)	Gain (%)
BurstFlow's RAPP	5400	-
Flink's Broadcast	5880	9
Baseline	7380	37
Flink's Random	9180	70
Flink's Rebalance	9540	77

SP clusters often comprise homogeneous computing resources, and current solutions cannot manage the distribution of incoming workloads in heterogeneous clusters. This happens because existing data partition solutions, even for stateless operators, neglect the required time for processing an incoming workload into resource with low throughput. The workload in each operator replica varies mainly because each micro-batch has a different number of messages requiring different times for parsing the micro-batches in the Sentiment Analysis application. The Classify operator also leads to a different workload as it requires access to a database in the HDFS in order to check the frequency of positive and negative words in each tweet.

Figure 7 summarizes the throughput per node when applying BurstFlow's RAPP and the state-of-the-art solutions, where the y-axis is the number of processed messages while the x-axis is the execution time in seconds. Flink's Rebalance distributes the workload in a round-robin fashion, neglecting the required time to process each micro-batch. Likewise, Flink's Random divides the workload using a uniform distribution. In contrast, BurstFlow's RAPP creates a ratio between memory and CPU per operator replica, which is updated periodically. This ratio leads to 2.5 times less execution time because it assigns a micro-batch to the most available machines, avoiding waiting times resulted from queued workloads.

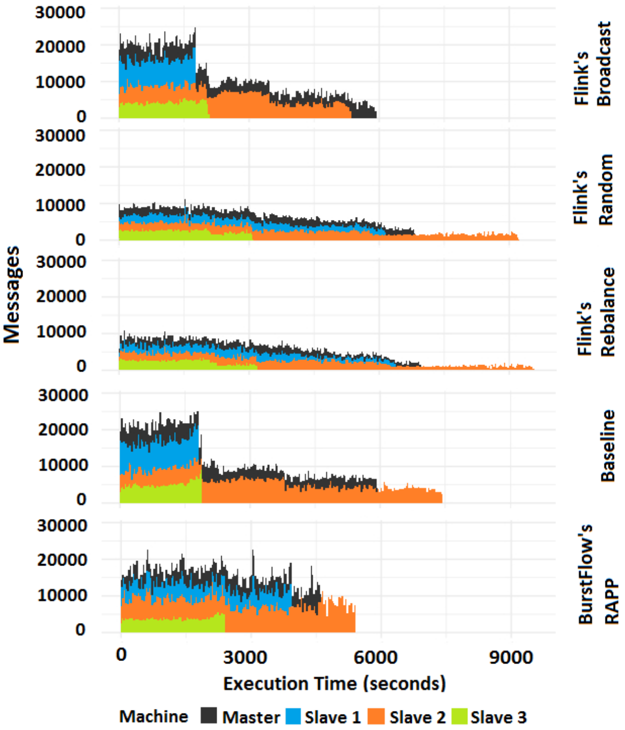


FIGURE 7. Throughput per node.

The BurstFlow's RAPP algorithm provides an almost fair distribution, with a more homogeneous throughput in comparison with the Flink's Broadcast in most of the execution time as seen in Figure 7. Master and Slave 2 have more free resources to task execution than Slave 3, with two cores and a 3.5 GB memory. In this aspect, the RAPP algorithm promotes 1.9 times more computational resource usage than Flink's Broadcast. Flink's Broadcast also transmits messages for all computing resources, promoting data deluge in the internal memory buffer and decreasing CPU consumption but a higher computational cost. In the next experiments, this resource consumption is evaluated in detail.

b: RESOURCE CONSUMPTION

The resource consumption evaluation enables identifying bottlenecks that impact the application performance and the solution as a whole. This subsection analyses the Central Processing Unit (CPU) and memory use collected during the execution time of applications. The results are the average execution time to each experiment.

Figure 8, in the top, shows the results of the experiment evaluating the average CPU usage for each machine in the cluster for the algorithms applied to the BurstFlow. The CPU usage is measured in percentage at the *y-axis*, and the *x-axis* shows the machines. This experiment evaluates the influence of BurstFlow on CPU usage.

In comparison, side by side, the presence of the ETAMBS approach with the BurstFlow's RAPP is up to 49%, in mean, more efficient in exploring the available CPU resources in opposition without its use. This result is due to the BurstFlow with ETAMBS approach having more data to process at the same time compared to the approach without resizing the data flow. The *one-at-a-time* approach uses, in the mean, 40% fewer CPU resources to a higher delay of 36% of execution time. Thus it produces lower execution throughput and requires more communication, which results in the application delay.

The CPU usage reflects in the use of memory for the same scenarios. Figure 8 in the bottom displays the average memory use of each node. The *y-axis* measures the memory use in GB and the *x-axis* indicates the machines. There are some contrasts in this chart because the BurstFlow's RAPP with ETAMBS seeks to optimize use of the memory. The memory is determinant for stream processing because the raw data is temporarily persisted in memory to be computed. Thus, more homogeneous memory use is expected as well as more CPU use since there is more data to process due to the flow re-partitioning of data.

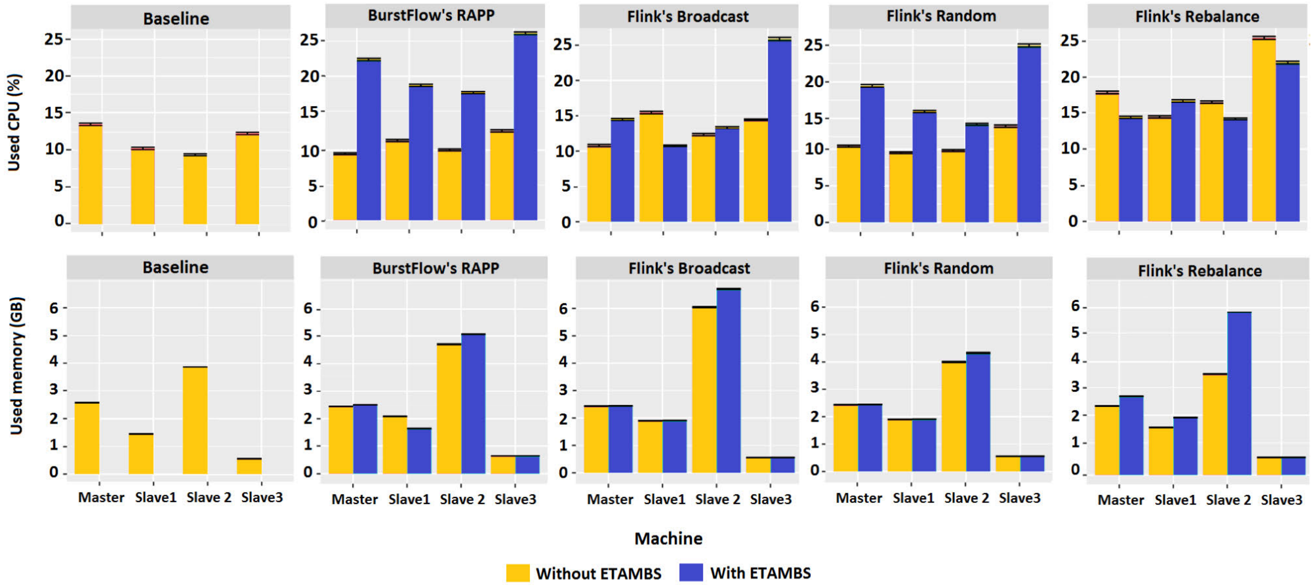


FIGURE 8. Average per node.

BurstFlow's RAPP explores better the available CPU resources without to compromise the memory use. It is clear that the BurstFlow's RAPP uses more CPU and memory in comparison with baseline execution, either by a minimal overhead in resizing the input data either by having more data to process at same time. Other important details are that Flink's Broadcast compared with BurstFlow's RAPP has a 32% lower use of processor and 36% higher consumption of memory on the average.

In general, CPU usage achieves more availability with BurstFlow, and owing to this, the performance increases. It may not be ideal in some cases, such as using Flink's Random and Flink's Rebalance algorithms when the data distribution approach is to create equal load per partition or seeking a uniform distribution.

D. THREATS TO VALIDITY

The experiments evaluation was performed in the Microsoft Azure Cloud Computing Platform, and due to this, the model has the following assumptions:

1) The machines are stables without failures and with a high SLA level;
2) The *Data Orchestrator*, Apache Kafka, and Apache Flink are safe and not exposed to the Internet directly, because they are behind the Microsoft Azure Firewall, and therefore the model is not subject to hackers or malicious users.

VI. CONCLUSIONS AND FUTURE WORK

This paper introduced BurstFlow, a promising tool for enhancing communication between multiple cloud providers. The proposed solution overcomes existing orchestration issues presented in cloud-based stream processing frameworks. For instance, Apache Flink permits to deploy applications on a single cloud provider while the proposed tool manages the computing resources dynamically in a geographically distributed MC infrastructure. The proposed solution also improves the application latency and throughput by automatically adjusting the size of micro-batches using a feedback loop to collect metrics and make decisions.

BurstFlow also enables to control the distribution of data in each operator replica by employing ad-hoc partitioning policies. This flexibility leverages BurstFlow to be applied to multiple scenarios without compromising memory usage, avoiding the swap context between storage and memory, and leading to low application latency. Our solution was evaluated in an MC deployment utilizing a real-world application. The proposed solution was compared to the state-of-the-art methods for data partitioning and policies used in Apache Flink and cloud-based solutions. Results have shown that BurstFlow reduces the execution time by 77% in the best case, improves the CPU and memory usage by up to 49%, and delivers the throughput of approximately 1.5GB/s.

For future work, BurstFlow intends to include methods to estimate the micro-batches sizes by considering the information of stream processing operators. For example, a big part of stream processing applications has stateful operators. These operators often process incoming data according to a counter or time window. Considering the requirements to build a window, we plan to adjust how to compute the size of the micro-batches that will be transferred between cloud providers. Also, it is feasible to evaluate our tool using benchmarks from multiple fields in order to demonstrate all the existing benefits of BurstFlow.

REFERENCES

[1] M. Hilbert, ''Big data for development: A review of promises and challenges,'' *Develop. Policy Rev.*, vol. 34, no. 1, pp. 135–174, Jan. 2016, doi: 10.1111/dpr.12142.

[2] N. Miloslavskaya and A. Tolstoy, ''Application of big data, fast data, and data lake concepts to information security issues,'' in *Proc. 4th Int. Conf. Future Internet Things Cloud Workshops*, Apr. 2016, pp. 148–153, doi: 10.1109/W-FiCloud.2016.41.

[3] T. Mohammed, A. Albeshri, I. Katib, and R. Mehmood, ''Ubipriseq—Deep reinforcement learning to manage privacy, security, energy, and qos in 5g iot hetnets,'' *Appl. Sci.*, vol. 10, no. 20, pp. 1–18, 2020, doi: 10.3390/app10207120.

[4] N. Janbi, I. Katib, A. Albeshri, and R. Mehmood, ''Distributed artificial intelligence-as-a-service (daiaas) for smarter ioe and 6g environments,'' *Sensors*, vol. 20, pp. 1–28, May 2020, doi: 10.3390/s20205796.

[5] M. T. Tun, D. E. Nyaung, and M. P. Phyu, ''Performance evaluation of intrusion detection streaming transactions using apache kafka and spark streaming,'' in *Proc. Int. Conf. Adv. Inf. Technol. (ICAIT)*, Nov. 2019, pp. 25–30, doi: 10.1109/AITC.2019.8920960.

[6] J. Abawajy, ''Comprehensive analysis of big data variety landscape,'' *Int. J. Parallel, Emergent Distrib. Syst.*, vol. 30, no. 1, pp. 5–14, Jan. 2015, doi: 10.1080/17445760.2014.925548.

[7] K. J. Matteussi, B. F. Zanchetta, G. Bertoncello, J. D. D. Dos Santos, J. C. S. Dos Anjos, and C. F. R. Geyer, ''Analysis and performance evaluation of deep learning on big data,'' in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2019, pp. 1–6, doi: 10.1109/ISCC47284.2019.8969762.

[8] A. Katsifodimos and S. Schelter, ''Apache flink: Stream analytics at scale,'' in *Proc. IEEE Int. Conf. Cloud Eng. Workshop (IC2EW)*, Apr. 2016, p. 193, doi: 10.1109/IC2EW.2016.56.

[9] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, ''Apache spark: A unified engine for big data processing,'' *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016, doi: 10.1145/2934664.

[10] J. Berliáska and M. Drozdowski, ''Comparing load-balancing algorithms for MapReduce under zipfian data skews,'' *Parallel Comput.*, vol. 72, pp. 14–28, Feb. 2018, doi: 10.1016/j.parco.2017.12.003.

[11] E. Alomari, I. Katib, and R. Mehmood, ''Iktishaf: A big data road-traffic event detection tool using Twitter and spark machine learning,'' *Mobile Netw. Appl.*, vol. 2020, pp. 1–16, Aug. 2020, doi: 10.1007/s11036-020-01635-y.

[12] R. Tudoran, A. Costan, O. Nano, I. Santos, H. Soncu, and G. Antoniu, ''JetStream: Enabling high throughput live event streaming on multi-site clouds,'' *Future Gener. Comput. Syst.*, vol. 54, pp. 274–291, Jan. 2016, doi: 10.1016/j.future.2015.01.016.

[13] M. Welsh, D. Culler, and E. Brewer, ''SEDA: An architecture for well-conditioned, scalable Internet services,'' *ACM SIGOPS Operating Syst. Rev.*, vol. 35, no. 5, pp. 230–243, Dec. 2001, doi: 10.1145/502059.502057.

[14] T. Das, Y. Zhong, I. Stoica, and S. Shenker, ''Adaptive stream processing using dynamic batch sizing,'' in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13, doi: 10.1145/2670979.2670995.

[15] A. Aral and T. Ovatman, ''A decentralized replica placement algorithm for edge computing,'' *IEEE Trans. Netw. Service Manage.*, vol. 15, no. 2, pp. 516–529, Jun. 2018, doi: 10.1109/TNSM.2017.2788945.

[16] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. Ullah Khan, ''The rise of big data on cloud computing: Review and open research issues,'' *Inf. Syst.*, vol. 47, pp. 98–115, Jan. 2015, doi: 10.1016/j.is.2014.07.006.

[17] C. Rista, D. Griebler, C. A. F. Maron, and L. G. Fernandes, ''Improving the network performance of a container-based cloud environment for Hadoop systems,'' in *Proc. Int. Conf. High Perform. Comput. Simul. (HPCS)*, Jul. 2017, pp. 619–626, doi: 10.1109/HPCS.2017.97.

[18] J. C. S. Dos Anjos, K. J. Matteussi, P. R. R. De Souza, G. J. A. Grabher, G. A. Borges, J. L. V. Barbosa, G. V. Gonzalez, V. R. Q. Leithardt, and C. F. R. Geyer, ''Data processing model to perform big data analytics in hybrid infrastructures,'' *IEEE Access*, vol. 8, pp. 170281–170294, 2020, doi: 10.1109/ACCESS.2020.3023344.

[19] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, ''Twitter heron: Stream processing at scale,'' in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 239–250, doi: 10.1145/2723372.2742788.

[20] Q. Zhang, Y. Song, R. R. Routray, and W. Shi, ''Adaptive block and batch sizing for batched stream processing system,'' in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Jul. 2016, pp. 35–44, doi: 10.1109/ICAC.2016.27.

[21] J. C. S. dos Anjos, K. J. Matteussi, P. R. de Souza, A. S. da Veith, G. Fedak, J. L. V. Barbosa, and C. F. R. Geyer, ''Enabling strategies for big data analytics in hybrid infrastructures,'' *Proc. Int. Conf. High Perform. Comput. Simulation*, 2018, pp. 869–876, doi: 10.1109/HPCS.2018.00140.

[22] R. Fernandez, P. R. Pietzuch, J. Kreps, N. Narkhede, J. Rao, J. Koshy, D. Lin, C. Riccomini, and G. Wang, ''Liquid: Unifying nearline and offline big data integration,'' in *Proc. 7th Biennial Conf. Innov. Data Syst. Res.*, 2015, pp. 1–8.

[23] (2020). *Zookeeper*. [Online]. Available: https://zookeeper.apache.org/

[24] Z. Zhuang, T. Feng, Y. Pan, H. Ramachandra, and B. Sridharan, ''Effective multi-stream joining in apache samza framework,'' in *Proc. IEEE Int. Congr. Big Data*, Jun. 2016, pp. 267–274, doi: 10.1109/BigDataCongress.2016.41.

[25] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, ''StreamCloud: An elastic and scalable data streaming system,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012, doi: 10.1109/TPDS.2012.24.

[26] Z. Zhao, H. Zhang, X. Geng, and H. Ma, ''Resource-aware cache management for in-memory data analytics frameworks,'' in *Proc. IEEE Intl Conf Parallel Distrib. Process. with Appl.*, Dec. 2019, pp. 364–371, doi: 10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00060.

[27] W. Xiu, J. Guo, and Y. Li, ''A memory management strategy based on task requirement for in-memory computing,'' in *Proc. Asia–Pacific Conf. Image Process., Electron. Comput. (IPEC)*, Apr. 2020, pp. 406–412, doi: 10.1109/IPEC49694.2020.9115157.

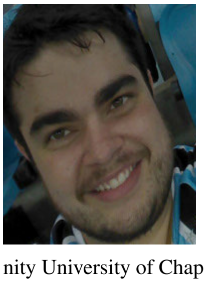
[28] Z. Tang, A. Zeng, X. Zhang, L. Yang, and K. Li, ''Dynamic memory-aware scheduling in spark computing environment,'' *J. Parallel Distrib. Comput.*, vol. 141, pp. 10–22, Jul. 2020, doi: 10.1016/j.jpdc.2020.03.010.

[29] K. Cao, Y. Liu, G. Meng, and Q. Sun, ''An overview on edge computing research,'' *IEEE Access*, vol. 8, pp. 85714–85728, 2020, doi: 10.1109/ACCESS.2020.2991734.

[30] R. Buyya and A. V. Dastjerdi, *Internet of Things: Principles and Paradigms*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2016.

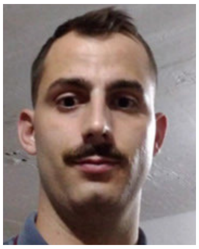
[31] R. Jain, *The Art of Computer Systems Performance Analysis—Techniques for Experimental Design, Measurement, Simulation, and Modeling*, 2nd ed. Hoboken, NJ, USA: Wiley, 1991.

PAULO RICARDO RODRIGUES DE SOUZA, JR. (Graduate Student Member, IEEE) received the bachelor's degree in computer science from the University of Passo Fundo (UPF) and the master's degree in computer science from the Federal University of Rio Grande do Sul (UFRGS), in 2018. He is currently pursuing the Ph.D. degree in computer science with Univ Rennes, Inria, CNRS. His areas of interests include big data, stream processing systems, distributed and parallel programming, load balance, and job scheduling.



KASSIANO J. MATTEUSSI (Member, IEEE) received the bachelor's degree in computer science from the Regional Community University of Chapecó (Unochapecó) and the master's degree in computer science from the Pontifical Catholic University of Rio Grande do Sul (PUCRS), in 2015. He is currently pursuing the Ph.D. degree in computer science with the Federal University of Rio Grande do Sul (UFRGS). He is certified in software engineering from the Regional Community University of Chapecó (Unochapecó). His areas of interests include big data, cloud computing, virtualization, distributed systems, high-performance computing, blockchain, FOG, and EDGE computing.

ALEXANDRE DA SILVA VEITH received the Ph.D. degree in computer science from the Ecole Normale Superieure (ENS), Lyon, and the University of Lyon. He is currently a Postdoctoral Researcher with the University of Toronto, Canada. His interests include reinforcement learning, the IoT, and data stream processing on cloud-edge infrastructure.

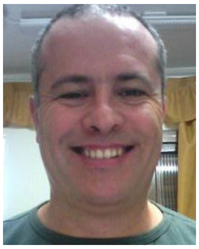


BRENO F. ZANCHETTA received the bachelor's degree of science in computer engineering from the Federal University of Rio Grande (FURG) in 2017. He is currently a Graduate Student in computer science with the Federal University of Rio Grande do Sul (UFRGS). He is also working as a Java Developer for POS solutions at the Rio Grande do Sul, Brazil. His topics of interest and areas of research include edge computing, deep learning, big data, and SOCs evaluation.

VALDERI R. Q. LEITHARDT (Member, IEEE) received the Ph.D. degree in computer science from INF-UFRGS, Brazil, in 2015. He is currently an Adjunct Professor with the Polytechnic Institute of Portalegre and a Researcher Integrated to the VALORIZA Research Group, School of Technology and Management (ESTG). He is also a Collaborating Researcher in the following research groups: COPELABS from Universidade Lusófona de Lisboa, Portugal; Telecommunications Institute of Portugal, IT Branch Covilhã, Portugal; Department of Informatics, University Beira Interior, Covilhã, Portugal; Laboratory of Embedded and Distributed Systems, University of Vale do Itajaí (UNIVALI), Brazil; Expert Systems and Applications Laboratory; and is a Recognized Research Group (GIR) of the University of Salamanca, Spain. The mainline of research is in distributed systems with a focus on data privacy, communication and programming protocols, involving scenarios and applications for the Internet of Things, smart cities, BIG DATA, cloud computing, and Blockchain.

ÁLVARO L. MURCIEGO received the master's degree in intelligent systems and the Ph.D. degree in computer engineering from the University of Salamanca, in 2015 and 2019, respectively. He is currently working on the Expert Systems and Applications Laboratory Research Group, Computer Science Department, University of Salamanca, as a Postdoctoral Researcher. Throughout his training, he has followed a well-defined line of research, focused on machine learning, route optimization, IoT sensors, and edge computing.

EDISON PIGNATON DE FREITAS (Member, IEEE) received the M.Sc. degree in computer science from the Federal University of Rio Grande do Sul (UFRGS), Brazil, in 2007, and the Ph.D. degree in computer science and engineering from Halmstad University, Sweden, in 2011. He is a Computer Engineer at the Military Institute of Engineering, Brazil, in 2003. He currently holds an Associate Professor with UFRGS, where he was also the Secretary for International Relations, developing research in the Graduate Programs in computer science and in electrical engineering. His research interests are mainly in the following areas: computer networks, the Internet of Things, real-time systems, massively distributed systems, multiagents systems, and unmanned systems.



JULIO C. S. DOS ANJOS received the bachelor's degree in electrical engineering with an emphasis on electronics from the Pontifical Catholic University of Rio Grande do Sul (PUC/RS), in 1991, the master's degree in computer science from the Federal University of Rio Grande do Sul, in 2012, and the Ph.D. degree in computer science from the Federal University of Rio Grande do Sul (UFRGS/RS), in 2017. He has been holding a postdoctoral position in computer science with UFRGS since 2017. He is currently a Researcher in the areas of big data, MapReduce and streaming processing, distributed and heterogeneous systems, hybrid infrastructures, and big data analytics with machine learning and deep learning.

CLAUDIO F. R. GEYER received the M.Sc. degree in computer science from UFRGS/RS in 1986, and the Ph.D. degree in informatics from the Université de Grenoble I, in 1991. He was a Mechanics Engineer at UFRGS/RS in 1978, where he is also a Tenured Professor in computer science. His areas of interest are pervasive and ubiquitous computing, grid and volunteer computing, scheduling and data-intensive computing, and big data.

• • •