Published in Proceedings of the 2016 International Conference on High Performance Computing & Simulation (HPCS 2016),
https://doi.org/10.1109/HPCSim.2016.7568388

Elastic Stateful Stream Processing in Storm

Valeria Cardellini, Matteo Nardelli, Dario Luzi
Department of Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata, Italy
Email: cardellini@ing.uniroma2.it, nardelli@ing.uniroma2.it, luzidario@gmail.com

***Abstract*—The advent of the Big Data era and the diffusion of Cloud computing have renewed the interest in Data Stream Processing (DSP) applications, which can timely extract useful information from distributed data sources. Due to the unpredictable rate at which the sources may produce data, DSP applications demand high dynamism. Storm has emerged as a widely adopted DSP system, which, although having many desirable features, shows some limitations due to the lack of adaptation capabilities.**

In this paper, we extend Storm with two mechanisms that support the run-time adaptation of DSP applications. Specifically, we introduce new components that allow automatic elasticity and stateful migration of the application components. The experimental results show the benefits of the newly introduced functionalities that, albeit equipped with proof of concept policies, allow to properly cope with workload variations while improving the resource utilization of the underlying infrastructure.

I. Introduction

Data Stream Processing (DSP) applications are widely used to process unbounded streams of data and timely extract valuable information. The key feature of DSP applications is the ability of processing data on-the-fly (i.e., without storing them), moving them from an operator to the next one, before reaching the final consumers of the information. Usually, these are long-running applications subject to periodic or unpredictable workload variations, which can exploit the *data parallelism* to process great volumes of data. Scaling the application with data parallelism consists in increasing or decreasing the number of parallel instances for the operators, so that each instance can process a subset of the incoming data flow in parallel (e.g., [1], [2]).

For the execution of DSP applications, users commonly rely on DSP systems (or frameworks), such as Apache Storm [3], Spark [4], and Flink[1], that offer simple programming interfaces, abstracting away the underlying infrastructure and complexity of distributing the operators. To date, most of these systems, although including numerous features, do not provide automatic scaling capabilities and only support a static or manual definition of the operator parallelism. Therefore, they deploy the application on a fixed number of computing nodes and do not fully exploit the Cloud computing principles, which promote the elastic usage of on-demand resources. As a consequence, to support workload fluctuations, the user determines the number of parallel instances for the operators on the expected maximal workload, achieving either an average under-utilization of the system, because load peaks can rarely occur, or being unable to manage a bursty workload with unexpected fluctuations. Among the open source DSP systems, Apache Storm has received increasing interest in the last few years and in the literature different works have proposed extensions, defined new scheduling policies, and build applications on top of it (e.g., [5], [6], [7], [8]).

In this paper, we extend Storm by introducing two mechanisms that support the run-time adaptation of DSP applications: automatic elasticity and stateful migration. The *elasticity* mechanism implements scaling decisions at the framework level, i.e., it allows to automatically adapt the number of parallel instances for each application operator, according to a scaling policy. Different scaling policies can be defined; as proof of concept, we propose a simple threshold-based policy that elastically changes the number of parallel instances of each operator according to the incoming workload. Once equipped with elasticity at the framework level, Storm can be then properly coupled with a lower-level scaling system that realizes elasticity at the infrastructure level by acquiring and releasing the computational nodes as needed, therefore encompassing the on-demand resource principle of Cloud computing. The *stateful migration* mechanism supports the relocation of the operator internal state on a different node and enables Storm to change the application deployment at run-time, without compromising the application integrity in terms of extracted information. This mechanism operates at fine granularity with respect to the execution model adopted by Storm and allows multiple and concurrent migrations.

The main contributions of our work are as follows.

- We extend Storm with an automatic *elasticity* mechanism that changes the number of parallel instances for the operators at run-time; we realize, as proof of concept, a threshold-based scaling policy that aims at maximizing the system utilization (Section IV).
- We enhance Storm with stateful operator migrations, which enable the self-adaptation capabilities of DSP applications in a non-destructive way, preserving the operators state (Section V).

A set of experiments run with the enhanced Storm shows the benefits and overhead of the newly introduced mechanisms (Section VI). Our extension is fully modular and loosely coupled from the existing architecture of Storm, therefore existing solutions based on or proposed for Storm in the literature can transparently reuse the new functionalities. To this end, we publicly release our source code to the community[2].

[1]Apache Flink: https://flink.apache.org/

[2]Elastic Storm is available on GitHub: http://bit.ly/1oUjZAi

II. Apache Storm

Storm is an open source, real-time, and scalable DSP system maintained by the Apache Software Foundation. It provides an abstraction layer where DSP applications can be executed over a set of worker nodes interconnected by an overlay network. A *worker node* is a generic computing resource (e.g., a physical host, a virtual machine, a mobile device), whereas the overlay network comprises the logical links between these nodes.

In Storm, we can distinguish between an abstract application model, which is defined by the user, and an execution application model, which is used by Storm to run the application. In the abstract model, an application is represented by its *topology* q, which is a directed acyclic graph with spouts and bolts as vertices and streams as edges. A *spout* is a data source that feeds the data into the system through one or more streams. A *bolt* is either a processing element, which extracts valuable information from incoming tuples, or a final information consumer; a bolt can also generate new outgoing streams, like spouts do. A *stream* is an unbounded sequence of *tuples*, which are key-value pairs. We refer to spouts and bolts as operators and denote an operator with $op \in OP(q)$, where $OP(q)$ is the set of operators of the topology q. In the execution model, Storm transforms the topology q by replacing each operator op with its tasks $T(op)$. A *task* is an instance of an application operator (i.e., spout or bolt) that is in charge of a share of the operator incoming stream. Therefore, if the operator has some internal state (i.e., it is a stateful operator), a task handles a partition of it. In Storm, the number of tasks for an operator is statically defined. For the execution, one or more tasks of the *same* operator op are grouped into executors $E(op)$. An *executor* is the smallest schedulable unit, and Storm can process great volumes of data in a timely way by launching multiple executors for each operator. The relationship $|E(op)| \leq |T(op)|$ must hold among executors and tasks of op. From an operational perspective, Storm implements the executors with threads and also introduces the *worker process*, that is a Java process, to run a subset of executors of the *same* topology q. The resulting execution model of Storm shows the following hierarchy: a group of tasks runs sequentially in the executor, which is a thread within the worker process that serves as container on the worker node.

Besides the computational resources (i.e., worker nodes), the architecture of Storm includes two components: Nimbus and ZooKeeper. *Nimbus* is the centralized entity in charge of coordinating the topology execution; it uses its *scheduler* to define the placement of the application operators on the pool of available worker nodes. The assignment plan determined by the scheduler is communicated to all the worker nodes through *ZooKeeper*, which is a shared memory service for managing configuration information and enabling distributed coordination[3]. Since each worker node can execute one or more worker processes, a *Supervisor* component on the worker node starts or terminates the worker processes on the basis of the Nimbus assignments. Each worker node can concurrently run a limited number of worker processes, based on the number of available *worker slots*.

[3]Apache ZooKeeper: http://zookeeper.apache.org/

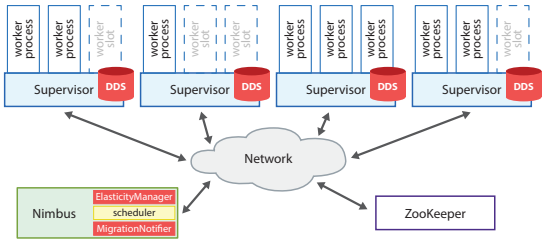


Fig. 1: Extended Storm architecture.

III. Solution Overview

Neither stateful migrations nor elastic scaling decisions are supported by the current architecture of Storm. However, Storm provides the `rebalance` function which allows to manually change the number of executors for the topology operators. We extend Storm to support elasticity and stateful migration, aiming at providing mechanisms that can be reused by the Storm-based extensions in the literature, including those focusing on the operator placement policies (e.g., [5], [6], [7], [8], [9]). Figure 1 illustrates in red the newly introduced components, namely the ElasticityManager, the MigrationNotifier, and the Distributed Data Store (DDS).

The *ElasticityManager* is located on the centralized component Nimbus and is in charge of evaluating scaling decisions for the topologies managed by Storm. A scaling decision changes the operator parallelism, i.e., it increases or decreases the number of executors for an operator, improving resource utilization in relationship to the incoming data rate. Since the newly added executors have to be placed on the worker nodes, the ElasticityManager is executed before the Storm scheduler, so that the latter can analyze and define the placement of the newly introduced executors.

The *MigrationNotifier* is executed after the scheduler; it initiates the migration by notifying the tasks of the executors that have changed their placement to save their internal state. The MigrationNotifier will resume the execution as soon as all the migrating tasks can be terminated without loss. Afterwards, the new assignment plan, defined by the scheduler, will become effective and the migrating tasks can be restored on the new worker nodes.

The *Distributed Data Store* (DDS) is introduced on each worker node and allows the migrating tasks to save their internal state before terminating their execution. This data store acts as a repository for the migrating tasks, which can retrieve and restore their state as soon as they are instantiated on the new worker nodes. The presence of a locally available data store allows us to minimize the amount of state moved across the network during a migration.

IV. Elasticity

The *ElasticityManager* dynamically reconfigures a topology by scaling horizontally (i.e., scaling out and in) the number of

executors for an operator. A *scale-out* decision increases the number of executors when the operator needs more computing resources. A *scale-in* decreases the number of executors when the operator under-uses its resources. Recalling the execution model of Storm (Section II), a scaling operation changes how tasks are grouped into executors, thus leading to a possible relocation of the operator state on a different worker node.

A. Design Overview

The *ElasticityManager* is designed as a loosely coupled component of Storm, which works independently from the scheduling policy. Periodically, every T_{nbs}, Nimbus activates the ElasticityManager; the latter analyzes each of the topologies running in Storm by possibly taking scaling decisions for at most a single topology at a time. This limitation allows to evaluate the effects of scaling decisions on the other running topologies. We describe the simple yet effective policy adopted by the ElasticityManager to compute the scaling decisions in Section IV-B. The resulting decisions are then implemented exploiting the `rebalance` function provided by Storm, which allows to change the set of executors for the operators of a topology and to adapt the Storm execution environment. As side effect, `rebalance` suspends the execution of the topology spouts until the scheduler defines a new placement for the topology. To avoid stressing a topology with frequent scaling decisions that may lead to instability, after a `rebalance` we let the topology enter in a cooldown state for the next EM_{cld} invocations of the ElasticityManager.

B. Scaling Policy

We design a reactive and threshold-based policy: the ElasticityManager takes scaling decisions comparing the fraction of CPU time utilized by each executor against some thresholds. Specifically, U_e measures the fraction of CPU time used by the executor $e \in E(op)$ of the operator $op \in OP(q)$ in the topology q[4]. That is, U_e is the CPU utilization per executor e. The ElasticityManager considers a single topology q at a time and decides first the scale-out actions, then the scale-in ones.

Scale-out. It adds a new executor for each one in overload. Formally, for each $e \in E(op)$, with $op \in OP(q)$, such that

$$U_e > ScaleOutThr$$

where $ScaleOutThr$ is the upper usage threshold, one new executor is added to $E(op)$. According to this policy, the number of executors for *op* can be at most doubled in a run of the ElasticityManager. All the operators subject to a scale-out operation compose the set $OP^{sc}(q)$.

Scale-in. It halves the number of executors of an operator, if all those existing are underloaded. Scale-in decisions are evaluated for all the operators not already under a scaling-out operation. Formally, for each operator $op \in OP(q) \backslash OP^{sc}(q)$ such that

$$U_e < ScaleInThr \quad \forall e \in E(op)$$

[4]Since an executor is a Java thread, the CPU time for the thread can be obtained relying on the `ThreadMXBean` class.

where $ScaleInThr$ is the lower usage threshold, the number of executors is halved or set to $minExec(op)$, which is the configurable minimum for operator *op*.

This scaling strategy doubles or halves the number of executors for an operator. Therefore, we conveniently define the scaling thresholds in a such a way that $ScaleOutThr, ScaleInThr \in [0, 1]$ and we can guarantee a stability gap $S \in [0, 1]$ between them such that $ScaleOutThr > 2 \times ScaleInThr + S$. The higher the values for S, the more conservative are the scaling decisions.

C. Elasticity and Placement

When a topology is subject to a scaling decision, the ElasticityManager conveniently marks it with a special label. The Storm scheduler can thus adopt a specific placement policy, which assigns only the added or changed executors by minimizing, for example, the amount of relocated operator state. Since in Storm the number of tasks for each operator is defined a-priori and cannot change at run-time (see Section II), the elasticity changes how the tasks are grouped into executors. If the operator is stateful, a scaling decision leads to the relocation of a partition of the operator state, which could be costly or negatively impact the application performance. Therefore, we implement a simple placement policy for those topologies subject to scaling decisions, which places the new or updated executors by minimizing the number of tasks that should be relocated with respect to the previous configuration. This strategy assumes that the operator state is uniformly distributed across its partitions (i.e., tasks); if this condition does not hold true, more sophisticated strategies can perform better. Furthermore, this strategy consolidates the application executors on fewer worker nodes.

V. Stateful Migration

After either a scaling operation or the definition of a new application placement, some executors may be relocated on the worker nodes. If the executor is stateless (i.e., it contains tasks of a stateless operator), the relocation can be easily performed by terminating the executor on the old location, moving its code to the new location, and restarting it. On the other hand, if the executor is stateful (i.e., it contains tasks of a stateful operator), we also need to efficiently *migrate* its internal state, so to preserve the integrity and consistency of the outputted streams. As a consequence, this kind of migration can involve a sophisticated cooperation among several components.

A. Overview

We propose a stateful migration solution that uses a pause-and-resume approach [10], which extracts the current state from the old instance and replays it within the new instance. To this end, the executor needs to be paused to ensure a semantically correct migration. Since in Storm a task represents the smallest entity that handles a partition of the operator state, we extend Storm to support *task-level*, or fine-grained, migrations. Note that this kind of migration covers the needs

not only of scaling decisions, which define a new tasks-to-executors mapping, but also of replacement decisions, which move already existing executors. Furthermore, thanks to the fine granularity, we can parallelize the migrations towards different computing locations, for example when an executor is split in tasks that will be relocated in different locations.

B. Extended Architecture

The design of the migration protocol aims at satisfying the following requirements: (1) to be transparent to and reusable by the other existing Storm components; (2) to preserve the operator semantics by avoiding tuple loss and tuple reordering; and (3) to minimize the amount of data transferred using the network. Our intent is also to minimize the impact on the existing Storm architecture, reducing the amount of new code and reusing properly the Storm functionalities. The key idea is to enhance the tasks with the ability of exporting the operator state from the old worker node and of importing it to the new one. We realize this idea thanks to the cooperation of the following new components of Storm (see Figure 1):

- a Distributed Data Store (DDS) which enables to decouple the operator state from the related task during the migration;
- an extension of the Storm API that allows to define the code of spouts (i.e., data sources) and bolts (i.e., operators and final consumers). Specifically, we introduce the *StatefulSpout* and *StatefulBolt* classes, which enrich the tasks with the ability of storing and retrieving the partition of the operator state from the DDS in a user-transparent way;
- a centralized *MigrationNotifier* that orchestrates the migration and prevents Storm from propagating new placement decisions until all the tasks involved in a migration have saved their state to DDS. Indeed, in the official release of Storm, after a new placement decision the executors that have changed location are restarted, making the tasks loose their state.

The proposed solution also exploits ZooKeeper, which provides a coordination and synchronization service that simplifies the cooperation among the distributed components.

DDS. Each worker node is equipped with a data store, which is accessible to all the other worker nodes. This allows a migrating task to save its state on a local storage, so to minimize the amount of data transferred on the network. The data store is implemented as an in-memory caching system with Hazelcast[5]. We do not use ZooKeeper for this purpose, because it is not designed to hold large data values.

StatefulSpout and StatefulBolt. These classes support and execute the stateful migration protocol presented in the next section and should be used when the topology has at least one stateful operator. That is, also stateless operators should be defined leveraging on these new classes. Differently from the default implementation of spouts and bolts, these new classes are enhanced to: (1) provide a common interface that defines the task state and allows to export and import it with `getState()` and `setState()`, respectively; and (2) define two execution modes for the task, namely the *operational mode* and the *migration mode*. The former represents the traditional execution mode of the task, which runs the operator logic that is defined by the user. The latter is used when the task or a neighbor (i.e., upstream, downstream) task is migrating; it allows to safely stop the task execution, save and restore the internal state, and avoid tuple loss and tuple reordering — thus preserving the operator integrity. Moreover, since the coordination among tasks relies on ZooKeeper, the StatefulSpout and StatefulBolt classes include a Watcher component that asynchronously observes and retrieves information published by the MigrationNotifier or by other tasks.

MigrationNotifier. This component executes on the centralized entity Nimbus. Basically, the MigrationNotifier intercepts the scheduler assignment plan, notifies the tasks that should export their internal state, and waits until each of them has correctly completed this operation (i.e., it reaches the first synchronization barrier, as presented next). When the MigrationNotifier resumes the execution, Nimbus can disseminate the new placement decisions to the worker nodes; the latter will terminate and launch the worker processes (and related executors) according to the new placement.

C. Migration Mode and Migration Protocol

When the MigrationNotifier communicates, through ZooKeeper, the set of tasks involved in a migration, these tasks, as well as the tasks that precede and follow them in the topology, enter in the migration mode. Each task relies on the Watcher component to asynchronously check for this kind of notifications. The migration mode runs the migration protocol, which specializes the task behaviour with respect to the role played during the migration. We identify three roles: the task precedes a migrating task in the topology, the task itself is migrating, and the task follows a migrating task in the topology. For short, we refer to them as *upstream* task, *migrating* task, and *downstream* task, respectively.

To avoid tuple loss or their reordering, we need to pause the streams directed to a migrating task and save the tuples in transit, so to replay them as soon as the migration is completed. To this end, we introduce an *OutputBuffer* that resides on the upstream task and allows to temporary store the tuples directed towards a migrating task that could change its location. The upstream task explicitly notifies the last tuple sent on a stream, leveraging a special end-of-stream (EOS) message. For sake of efficiency, we introduce also an *InputBuffer*, which resides on the migrating task and on the downstream ones. Using this buffer, the migration protocol can stop the execution of the operator logic and rapidly retrieve the incoming tuples from the communication link.

We now present the migration protocol according to the role played by a task during a migration: upstream task (of a migrating one), downstream task (of a migrating one), and migrating task. If a task plays different roles during a migration (e.g., upstream and downstream task), the following

[5]Hazelcast: https://hazelcast.com/

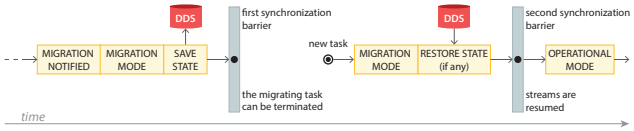


Fig. 2: Sequence of operations performed by a migrating task.

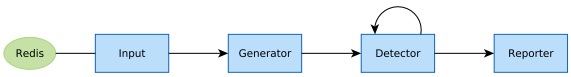


Fig. 3: Frequent Pattern Detection topology.

procedures are combined. As depicted in Figure 2, the migration protocol is characterized by two synchronization barriers, which indicate respectively that a task has correctly saved its state and has completely recovered it.

Upstream Task. Entering the migration mode, the upstream task stops (only) the streams directed to the migrating task: it emits the EOS message and stores the subsequently produced tuples for that stream on the OutputBuffer. As soon as all the downstream migrating tasks reach the second synchronization barrier, i.e., they complete the migration, the upstream task emits the tuples stored within the OutputBuffer and switches back to the operational mode.

Downstream Task. Entering the migration mode, the downstream task stops the computation and rapidly downloads the streams coming from the migrating task; all the received tuples that precede the EOS message are stored into the InputBuffer. Afterwards, it switches back to the operational mode and resumes the computation.

Migrating Task. Although logically sequential, the execution of the migration mode of the migrating task is divided in two parts, namely *save state* and *restore state*, which can be executed on two different computing locations.

a) Save State. Entering the migration mode, the migrating task stops the computation, emits the EOS message to its downstream tasks, and buffers the incoming tuples on the InputBuffer until the EOS message is received from its upstream tasks. As soon as these operations are completed, the task pushes on the local DDS the operator state and the InputBuffer. The operator state is extracted relying on the `getState()` function that the user implements. Since computation is stopped, the OutputBuffer is always empty. At this point, the task is ready to be safely terminated, reaches the first synchronization buffer, and disseminates this information to the other tasks using ZooKeeper.

b) Restore State. When a new task is launched, it automatically enters the migration mode and checks on ZooKeeper if it is involved in a migration. If so, it contacts the DDS on the old worker node and recovers the operator state together with the InputBuffer. The operator state is imported using the `setState()` function that the user implements. Afterwards, the migration is considered as completed, the task reaches the second synchronization barrier and spreads this information to the other tasks using ZooKeeper. When all the tasks involved in a migration reach the second synchronization barrier, the paused streams will be resumed and the application will continue the execution. In the meanwhile, the task retrieves and processes the tuples from the InputBuffer and, then, switches to the operational mode.

The proposed protocol supports concurrent migrations, because each task can autonomously relocate its state.

VI. Experimental Results

To analyze the benefits and overhead introduced by the new elastic scaling and stateful migration mechanisms, we run a set of experiments executing a stateful DSP application on Storm, by first enabling and then disabling the new mechanisms.

A. Experimental Setup

We run the experiments using Storm 0.9.3 on a cluster of 5 worker nodes and one further node for Nimbus and ZooKeeper. Each node is a *m4.xlarge* AWS EC2 virtual machine with 4 vCPUs on Intel Xeon E5-2676 and 16 GB of RAM.

In our experiments, Storm determines the operators placement using the scheduler designed by Xu et al. [8], which assigns the operators on the worker nodes in descending order of incoming and outgoing traffic exchanged using the network so to minimize the inter-node traffic. Differently from the default round-robin scheduling policy, this one triggers executor reassignments if it finds a new configuration that reduces the inter-node traffic. Moreover, the policy ensures an even distribution of the executors on all the worker nodes, because a node can run at most E/N executors, where E is the total number of executors and N the number of worker nodes. To avoid inter-process communication overhead, the scheduler by Xu et al. uses only a worker slot per node. Since its source code is not publicly available, we implement the scheduler according to the description in [8] and integrate it with the special placement policy for the new or updated executors that we presented in Section IV-C.

The ElasticityManager is executed together with the scheduler every $T_{nbs} = 10$ s, and *ScaleOutThr* and *ScaleInThr* are set to 0.7 and 0.2, respectively. After a scaling decision, the topology enters in a cooldown state for the next $EM_{cld} = 12$ invocations of the ElasticityManager (i.e., for 120 s), where no new scaling decision can be applied. In preliminary experiments, not reported for space limits, we found that this setting ensures a stable behavior of the system.

As testing application we implement the Frequent Pattern Detection (FPD) [11], which analyzes tweets from Twitter and retrieves the most frequent patterns (i.e., those that occur more than 20 times) on a sliding window of 60 s. Figure 3 shows the FPD topology and Table I reports the configuration of its operators. The topology spout reads input data from Redis, a shared memory, and emits them according to an exponential distribution with parameter λ.

Each experiment comprises five sequential phases, each lasting 900 s, and with input data rate according to the sequence $\lambda = \{120, 350, 900, 250, 120\}$ tweets/s. This workload stresses

TABLE I: Number of executors and tasks, and minimum number of executors for each operator of the FPD application.

Operator op	$\|E(op)\|$	$\|T(op)\|$	$minExec(op)$
Input	1	1	1
Generator	2	2	1
Detector	5	20	1
Reporter	1	1	1

the infrastructure of Storm, which has to repeatedly evaluate the application placement at run-time. In the following figures, the beginning of each phase is represented with a vertical dotted line with a rhombus on the extremities. A scaling decision, which changes the number of executors for the topology, is represented with a vertical dash line, whereas a scheduling decision, which changes the placement of the executors, is represented with a vertical dot-dash line and a symbol "+" on top. The performance metrics are collected through the Storm metric system, which every 5 s provides an average value computed on a sliding window of 600 s, made of samples harvested every 5 s. Since this metric system is stateless, after a migration the samples are lost and thus the following figures will show some zero values.

B. On the Elastic Scaling Mechanism

Figure 4 shows the application latency, i.e., the average latency experienced to traverse the entire FPD topology. When the elastic scaling mechanism is deactivated (referred to as "w/o E+SM" in Figure 4), the application ability of handling the incoming data rate depends on the parallelism that is statically defined by the user at design time. In this case, the application can manage the data source with $\lambda = 120$ tweets/s, but cannot keep the pace when the data rate reaches $\lambda = 350$ tweets/s; the system becomes unstable, as confirmed by the continuous increase of the application latency after 1500 s. When the input data rate reaches 900 tweets/s, the application latency grows up to 1500 ms. When the source reduces its data rate to $\lambda = 250$ tweet/s, the system can properly handle the buffered elements, until 3300 s, when all the buffers are empty, and the experienced application latency is below 90 ms. The elastic scaling and stateful migration mechanisms

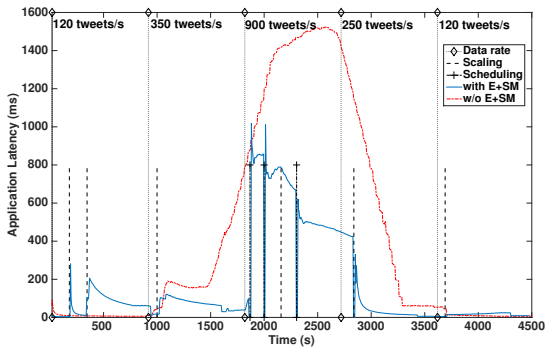


Fig. 4: Application latency with and without elasticity and stateful migration.

(referred to as "with E+SM" in Figure 4) improves significantly the application performance. When the data source emits 350 tweets/s, a scaling decision allows to efficiently handle the incoming load by doubling the number of executors for the Detector operator. Similarly, when the input data rate reaches 900 tweets/s, three reassignments and a scaling decision lead to an application latency lower than 900 ms. Soon after the occurrence of either a scaling or scheduling decision, we observe a transient period, where the application latency increases due to the processing of the collected buffers and the overhead imposed by the extended Storm to restart the executors on the new worker nodes.

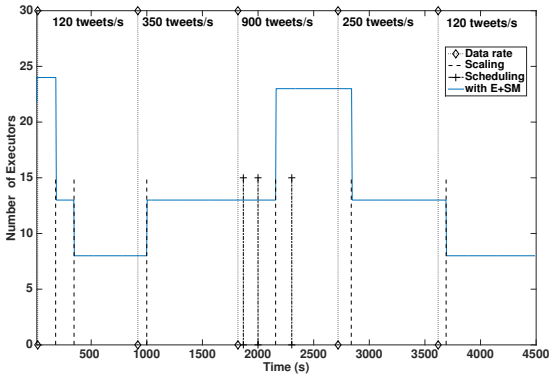
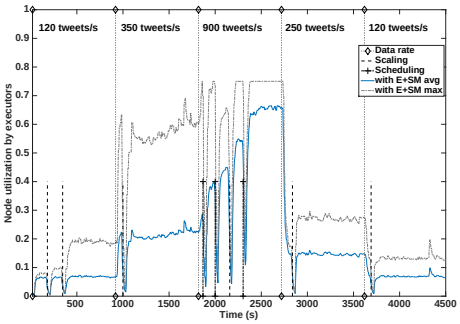


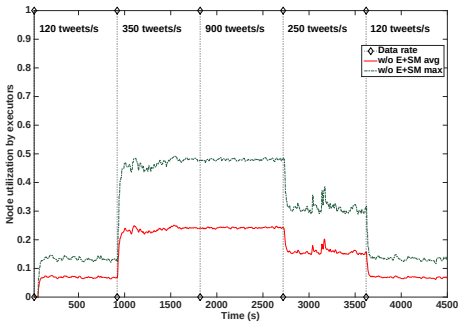
Fig. 5: Effects of the elasticity on the number of executors.

Figure 5 shows how the ElasticityManager scales in or out the number of running executors during the experiment. In the first phase, the unneeded executors are terminated, ensuring that the utilization of the running executors is higher than the $ScaleInThr$ threshold. When the load imposed to the system increases, new executors are launched up to the third phase, when 23 executors run concurrently. As the incoming load starts decreasing, the number of executors decreases as well and, at the end of the experiment, only 8 executors are running. Except for the Generator operator whose parallelism is halved in the first phase, only the Detector operator is scaled out and in, since it constitutes the bottleneck of the FPD application. The ElasticityManager changes the Detector parallelism degree in the following sequence: $\{20, 10, 5, 10, 20, 10, 5\}$.

When the elastic scaling and stateful migration mechanisms are both active, the application can better exploit the available resources. Figure 6a represents the average and maximum node utilization by the application executors. We can observe that the benefits of the scaling and scheduling decisions are complementary. The former allow to change the set of executors in order to better exploit the available computing resources, whereas the latter allow to balance the load among the worker nodes, enabling a more efficient usage of the available resources. When the elasticity and stateful migration mechanisms are both disabled, as results from Figure 6b, the system utilization is low and the application cannot process the increasing load in a timely way. This happens because the fixed number of executors overloads a subset of the computing resources, whereas the remaining subset of computing resources

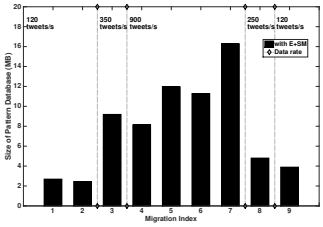


(a) With elasticity and stateful migration.

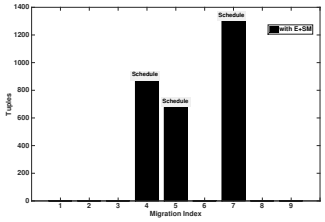


(b) Without elasticity and stateful migration.

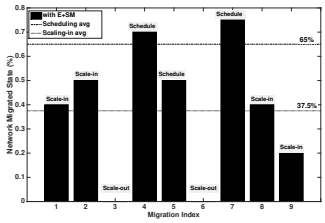
Fig. 6: Node utilization by executors.



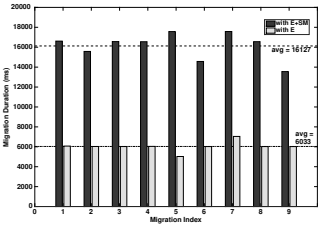
(a) Size of migrated state



(b) Number of migrated tuples



(c) Percentage of migrated state on network



(d) Overhead of stateful migration

Fig. 7: Analysis of stateful migration.

is free and not utilized. For example, with $\lambda = 900$ tweets/s, on the worker node with the maximum utilization (of about 50%), the half of the CPU cores with running executors is overloaded, whereas the other half is almost idle.

C. On the Stateful Migration Mechanism

We now analyze in detail the stateful migrations occurred during the experiments, with a special focus on two characteristics: the amount of state transferred using the network and the overhead introduced by the stateful migration.

The system has performed 9 migrations, 6 of which are related to a scaling decisions and 3 to placement decisions. In the following, we refer to each of them using the *migration index*, a progressive number that reflects the chronological order when the migration has been performed. We first analyze the amount of state transferred using the network. Figures 7a and 7b illustrate the operator state and the InputBuffer that *a migrating task* saves on the DDS. As expected, Figure 7a shows that there is a general tendency which links the size of the migrating operator with the incoming data rate: the higher the load, the larger the saved state. From Figure 7b, we can clearly see that the InputBuffers are always empty after the scaling decisions; this happens because the time elapsed between the `rebalance` command, that suspends the spouts activity, and the beginning of the migration is enough to consume the tuples already emitted. Figure 7c shows the percentage of saved state that is relocated during each migration. Specifically, after a scheduling decision on average 65% of the state is migrated. For the scaling decisions, the placement policy proposed in Section IV-C reduces this percentage and obtains a relocation of only 37.5% and 0% for the scale-in and scale-out decisions, respectively. To analyze the overhead introduced by the stateful migration, we run again the same experiment by enabling the elastic scaling mechanism but disabling the stateful migration. The result is shown in Figure 7d, where we compare the time elapsed to perform the stateful migration with the time needed to reassign the executors in Storm (i.e., stateless reassignment). On average the stateful migration introduces an overhead of about 10 s comprising: (1) the time to save the state on the DDS; (2) the time to retrieve and replay the state from the DDS; and (3) the time waited on the synchronization barriers. In this experiment, the third component dominates on the others, while the size of the operators state (from 2 to 16 MB, see Figure 7a) does not affect the migration overhead.

VII. Related Work

Elasticity and stateful migration for DSP systems are two features that have recently received an increasing attention. We first review the approaches that enable elasticity by reacting to changes observed in some monitored performance metric. Some works, e.g., [12], [13], [14], exploit a threshold-based elastic policy based on the measured CPU utilization of the system nodes. Gulisano et al. [12] define an upper threshold for the load variance among all the nodes, while Fernandez et al. [13] define upper and lower thresholds for each individual host. Heinze et al. [14] propose to adapt the thresholds using a reinforcement learning approach, which allows to gain in adaptivity. Other works, e.g., [1], [2], [15], [16], use more complex policies to determine the scaling decisions. Lohrmann et al. [16] propose a strategy that enforces latency constraints by relying on a predictive latency model based on queueing theory; nevertheless, their solution manages only stateless DSP applications. Heinze et al. [15] propose a model

to estimate the latency spike created by a set of operator movements and use it to define an elastic operator placement algorithm that minimizes the latency violations. This work is further extended in [2] with an online parameter optimization approach that avoids the manual tuning of the used thresholds. Similarly to the approaches in [2], [14], [16], our scaling policy is reactive and threshold-based; however, we decouple the elasticity mechanism from the corresponding policy, which can be easily changed. De Matteis and Mencagli [17] present an interesting elasticity policy, which relies on a proactive and control-theoretic method that takes into account a limited future time horizon to choose the reconfigurations to execute. The most popular open-source DSP frameworks, i.e., Storm, Spark Streaming, and Flink, do not support elasticity. Yang and Ma [18] investigate the internal architecture of Storm and propose different strategies for relocating stateless executors, achieving a reduction of the application latency degradation. Since we introduce the new mechanisms on top of the existing architecture of Storm, enhancements on its internal components are beneficial also for our extension.

While scaling out stateless operators can be achieved by just starting a new operator instance with a blank "memory", elasticity of stateful operators requires state migration to preserve the consistency of the operations [1]. Operator state migration is a challenging task, because it should be application-transparent and with a minimal footprint (i.e., amount of migrated state). The most common solutions, as seen in [10], are the pause-and-resume approach (that we adopt in this paper) and the parallel track approach, where the old and the new operator instances run concurrently until the state of both is synchronized. To identify the portion of state to migrate, Fernandez et al. expose an API to let the user manually manage the state [13], while Gedik et al. automatically determine, on the basis of a partitioning key, the optimal number of state partitions to be used and to migrate [1]. In our approach, the minimum unit of migratable state is defined by the user through the Storm execution model. ChronoStream [19] natively supports stateful migrations and uses a lightweight protocol that leverage on distributed checkpoints to minimize the amount of state relocated during a migration. Conversely, our work is driven by the existing Storm architecture. Storm also includes Trident, which provides high-level processing abstractions such as joins, aggregations, and filters. Differently from our extension, Trident can persist a state which is obtained by applying a sequence of Trident transformations on the input data. However, this approach requires to play the stream as a sequence of micro-batches, processed in a commit-like fashion, thus causing a constant latency overhead. Finally, Flink, although checkpointing stateful operators for fault tolerance, does not yet support stateful migrations.

VIII. Conclusions

In this paper, we designed and implemented two mechanisms, i.e., elasticity and stateful task migration, that allow Storm to address at runtime the highly dynamic nature of DSP applications. The proposed solution is modular and loosely coupled with the existing Storm architecture as well as transparent and fully reusable by other Storm-based solutions in literature. The experimental results show that Storm can elastically increase or decrease the number of operator executors as needed, improving resource utilization of the underlying infrastructure and properly processing the incoming workload.

As future work, we plan to design proactive and adaptive scaling policies and integrate in Storm a cross-layer scaling solution that exploits elasticity also at the infrastructure level so to enable Storm to effectively run in a distributed Cloud. We also plan to further investigate the stateful migration mechanism in order to reduce its overhead and the negative impact on performance of the synchronization barriers.

Acknowledgments

Publication supported by COST Action ACROSS (IC1304).

References

[1] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, 2014.

[2] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proc. of ACM SoCC '15*, 2015, pp. 276–287.

[3] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel *et al.*, "Storm@Twitter," in *Proc. of ACM SIGMOD '14*, 2014, pp. 147–156.

[4] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. of ACM SOSP'13*, 2013, pp. 423–438.

[5] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in Storm," in *Proc. of ACM DEBS '13*, 2013, pp. 207–218.

[6] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Distributed QoS-aware scheduling in Storm," in *Proc. of ACM DEBS '15*, 2015, pp. 344–347.

[7] L. Fischer, T. Scharrenbach, and A. Bernstein, "Scalable linked data stream processing via network-aware workload scheduling," in *Proc. of SSWS '13*, 2013.

[8] J. Xu, Z. Chen, J. Tang, and S. Su, "T-Storm: traffic-aware online scheduling in Storm," in *Proc. of IEEE ICDCS '14*, 2014, pp. 535–544.

[9] A. Chatzistergiou and S. D. Viglas, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in *Proc. of ACM CIKM '14*, 2014.

[10] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, "Cloud-based data stream processing," in *Proc. of ACM DEBS '14*, 2014, pp. 238–245.

[11] J. Ding, T. Z. J. Fu, R. T. B. Ma, M. Winslett, Y. Yang *et al.*, "Optimal operator state migration for elastic data stream processing," *CoRR*, vol. abs/1501.03619, 2015.

[12] V. Gulisano, R. Jiménez-Peris, M. P. no Martínez, 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, 2012.

[13] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. of ACM SIGMOD '13*, 2013.

[14] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proc. of IEEE ICDEW '14*, 2014, pp. 296–302.

[15] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proc. of ACM DEBS '14*, 2014, pp. 13–22.

[16] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proc. of IEEE ICDCS '15*, 2015, pp. 399–410.

[17] T. De Matteis and G. Mencagli, "Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing," in *Proc. of ACM PPoPP '16*, 2016.

[18] M. Yang and R. T. Ma, "Smooth task migration in Apache Storm," in *Proc. of ACM SIGMOD '15*, 2015, pp. 2067–2068.

[19] Y. Wu and K. L. Tan, "ChronoStream: Elastic stateful stream computation in the cloud," in *Proc. of IEEE ICDE '15*, 2015, pp. 723–734.