A Framework for Data Stream Applications in a Distributed Cloud

Matteo Nardelli

Department of Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata, Italy
nardelli@ing.uniroma2.it

Abstract The ever increasing diffusion of sensing and computing devices enables a new generation of data stream processing (DSP) applications that operate in a distributed Cloud environment. Despite this, most of the existing solutions, such as Apache Storm, are designed to run in a local cluster. In this paper we present our extension of Storm, which provides distributed monitoring, scheduling and management capabilities. Exploiting these new functionalities, the system can improve its performance and react to internal and external changes. Finally, we analyze open challenges of placing and adapting DSP applications.

Keywords: Data Stream Processing, Adaptation, Placement, Apache Storm

1 Introduction

With the disruptive diffusion of sensing devices (e. g., smartphones, cars, monitoring stations), the almost ubiquitous Internet connection, and the Fog Computing [13] paradigm, urban environments are today permeated by an ever increasing number of diffused and networked sensing and computing devices. All these sensing devices continuously produce streams of data that can be collected by distributed data stream processing (DSP) applications, to timely extract valuable information about many fundamental aspects of the environment we live in (e. g., urban mobility, public decision making, energy management). As data increases, we cannot push it toward the core of Internet. To increase scalability and reduce latency, a possible solution is to rely on distributed and near-edge computation. Furthermore, determining the computational resources that should host and execute each operator of the DSP application, i. e., solving the *operator placement* problem, is challenging because the characteristics of computational tasks are not known a-priori, the properties of the input streams change continuously, and the load imposed has to be sustained for long provisioning times. Therefore, we extended Storm [14], an open source DSP system, with policies and mechanisms that allow to find a placement that optimizes a utility function and to continuously adapt the placement when changes occur in the execution environment.

The main contributions of this paper are as follows: a) we describe how our extension implements the MAPE (Monitor, Analyze, Plan, and Execute) reference model for autonomic systems (Sect. 4); b) we show its benefits when the placement is determined according to the distributed policy proposed by Rizou et al. [12] (Sect. 5); and c) we illustrate some of the open challenges for DSP systems when they are executed in distributed environments (Sect. 6).

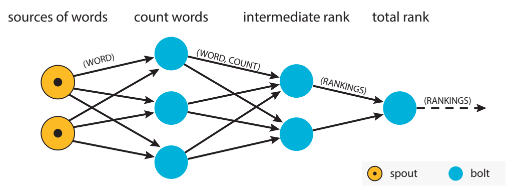
2 Related Work

As technologies and needs evolve in time, the DSP paradigm has experienced different generation of architectures [7], where the last one relies on Cloud-based resources. Despite this, most DSP systems are still designed to run in a local cluster, where the often homogeneous nodes are interconnected with negligible network delays (e. g., [14,15,17]). These assumptions do not hold any more when the DSP system runs in geographically distributed and dynamic environments, where a great heterogeneity of devices are interconnected with not-negligible network latencies. Storm, a framework of the last generation, is attracting increasing interests. However, most of the proposed Storm extensions are all centralized solutions (e. g., [1]), implicitly designed for clustered environments, which do not scale well as the number of applications increases. Our extension, instead, provides distributed monitoring, scheduling and management capabilities [3].

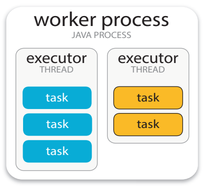
A great variety of placement algorithms have been proposed in literature. Lakshmanan et al. [10] provide a comprehensive overview of them, but, as the authors show, they differ each other on assumptions and optimization goals. Being interested in a network-aware solution, in our previous work [3] we implemented the Pietzuch's algorithm [11], and here we evaluate the strategy proposed by Rizou et al. [12]. Both the solutions minimize network usage, however the authors of [12] claim that their formulation has better convergence properties and works better in a distributed environment than [11].

Recently, another framework for large-scale processing is gaining interest: Spark [17]. It extends and improves the MapReduce approach (batch processing), and, using the Spark Streaming module, can reduce the size of each batch and process streams of data (micro-batch processing). This alternative is throughput oriented, whereas Storm, which is a pure DSP system, can further minimize the application latency, therefore is preferred in latency sensitive scenarios. Apache Flink[1] proposes a unified framework for batch and stream processing. Similarly to Storm, Flink has been originally designed to run in a cluster environment and shows the drawbacks we discuss in Sect. 4.1. DSP systems are also offered as Cloud services. Google Cloud Dataflow[2] provides a unified programming model to process batch and streaming data on top of Google cloud platform. Amazon offers Kinesis[3], which resembles an evolved publish-subscribe system, suitable to process near real-time streams of data. Both these Cloud-based services abstract

[1] `https://flink.apache.org/`
[2] `https://cloud.google.com/dataflow/`
[3] `https://aws.amazon.com/kinesis/`



(a) An example of Storm topology. The depicted application identifies the most frequent words (`http://bit.ly/1QxMn71`)



(b) Execution model

Figure 1: Storm abstractions

the underlying infrastructure, but it is reasonable to believe that they execute in a centralized data center, conversely to the context investigated in this paper.

3 Apache Storm

Storm[4] is an open source and scalable DSP system maintained by the Apache Software Foundation. It provides an abstraction layer where event-based applications can be executed over a set of worker nodes interconnected by an overlay network. A *worker node* is a generic computational resource, whereas the overlay network comprises the logical links between these nodes. In Storm, an application is represented by its *topology*, 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 data and generates new outgoing streams, or a final information consumer. A *stream* is an unbounded sequence of *tuples*, which are key-value pairs. We refer to spouts and bolts as operators. Figure 1a shows an example of a DSP application. Storm uses three types of entities with different grain to execute a topology. A *task* is an instance of an operator in charge of a share of its incoming streams. An *executor* can execute one or more tasks related to the same operator. A *worker process* is a Java process that runs a subset of executors of the *same* topology. As represented in Fig. 1b, there is a hierarchy among these entities: 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), Storm includes two centralized components: Nimbus and ZooKeeper. *Nimbus* coordinates the topology execution and defines the placement of its operators on the available worker nodes. This assignment plan is communicated to all the worker nodes through *ZooKeeper*, which is a shared memory service that enables distributed coordination. Since each worker node can execute one or more worker processes, a *Supervisor* component on the node starts and terminates worker processes on the basis of the Nimbus decisions.

[4] `http://storm.apache.org/`

4 Distributed Storm

4.1 From Cluster to Distributed Cloud: A Gap to Close

Storm has been originally designed to run in a local cluster, where network delays are negligible. If we deploy Storm in a distributed Cloud, it shows poor performances, because of the assumption that data can quickly move between computational nodes. We can summarize the limitations that Storm shows in this new environment as follows: 1) it is unaware of QoS attributes (e. g., resource utilization, network delays) of computational and network resources; 2) its placement decision is static, therefore the system cannot adapt to internal (i. e., application) and external (i. e., environmental) changes; and 3) if we create a custom centralized scheduler that collects the QoS attributes for each node and periodically evaluates the placement of each application, it will not scale well as the number of applications and network resources increases. In a geographically distributed environment, we would like to have a framework that considers network delays and resource heterogeneity while determining placement decisions.

4.2 Distributed Scheduling in Storm

We have extended the Storm architecture to run distributed, adaptive, and QoS-aware scheduling algorithms [3]. The newly introduced components, illustrated in orange in Fig. 2, are: the AdaptiveScheduler, the QoSMonitor, and the WorkerMonitor. We preserved the centralized scheduler, named BootstrapScheduler, which defines the initial placement of the application. The *AdaptiveScheduler* is the distributed scheduler that coordinates the MAPE control cycle. It executes on each Supervisor together with the *QoSMonitor*, an infrastructure level monitoring component. The *WorkerMonitor* is an application level monitor and runs on each worker process. Exploiting the feedback control loop, the distributed scheduler can react to internal and external changes of the operating conditions. In a single loop iteration, it monitors the environment and the locally executed executors,

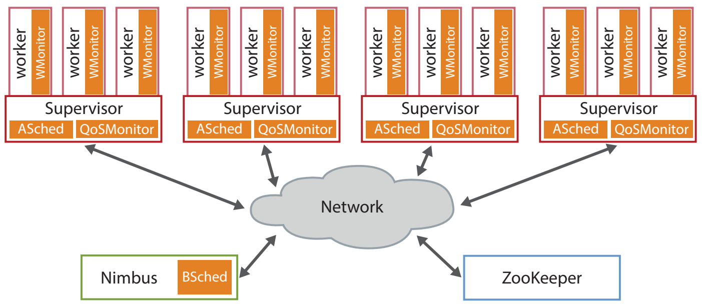


Figure 2: Storm architecture with new components in orange: AdaptiveScheduler (abbreviated as ASched), WorkerMonitor (WMonitor), and BootstrapScheduler (BSched).

analyzes if there are candidate executors for a new reassignment, and, in positive case, plans and executes the corresponding repositioning actions.

Monitor. The AdaptiveScheduler acquires the information on computational resources and on executors that run locally through the QoSMonitor and the WorkerMonitors respectively. The QoSMonitor provides the QoS awareness to each distributed scheduler, thus it is responsible of obtaining intra-node information (i. e., utilization and availability) and inter-node information (i. e., network delays). For the latter, it resorts on a network coordinates system [5] that provides an accurate estimate of the delays between any two computational nodes without the need of an exhaustive probing. The WorkerMonitor computes the exchanged data rate for each executor that runs on the node.

Analyze and Plan. A distributed scheduling policy drives these two phases. Our previous work [3] relies on the Pietzuch's placement algorithm [11]. In this paper, we use the scheduling solution designed by Rizou et al. [12], which places the application minimizing the network usage (i. e., sum of bandwidth-delay product for each application link). Implementing the Rizou's algorithm within the extended Storm requires just few changes. Basically, it needs to account for the specific Storm application model, where a processing operator can be instantiated in one or more executors and pinned operators are not modeled. Furthermore, the algorithm can readily obtain QoS information (i. e., latency, bandwidth) relying on the monitoring components.

Execute. Finally, if a new assignment must take place, the executor is moved to the new candidate node. The new assignment decision is shared with the involved worker nodes through ZooKeeper. We note that in Storm an executor reassignment does not preserve its state; thus, the executor is stopped on the previous worker node and started on the new one.

Thanks to the adaptation cycle, the distributed scheduler can manage changes that may occur both in the infrastructure layer (e. g., a worker node appears or fails) and application layer (e. g., data rate fluctuations).

The source code of our extension is available at `http://bit.ly/extstorm`.

5 Experimental Results

We show the improvements and the self-adaptation capabilities of our distributed scheduler equipped with the Rizou's algoritm (named as dRizou) with respect to the centralized and default EvenScheduler of Storm (named as cRR). For a better evaluation, we also indicate the behaviour of dQoS, that is the distributed scheduler equipped with the Pietzuch's algoritm (further details in [3]). dRizou and dQoS place operators exploiting QoS attributes, whereas cRR uses a round-robin policy. The evaluation uses a cluster of 8 worker nodes (each can host at most 2 worker processes) and 2 further nodes for Nimbus and ZooKeeper. We emulated wide-area network latencies among the Storm nodes applying to outgoing packets a Gaussian delay with mean and standard deviation in the ranges $[12, 32]$ ms and $[1, 3]$ ms, respectively. The DSP application is composed of a source, which generates 10 tuples/s, followed by a sequence of 5 operators before

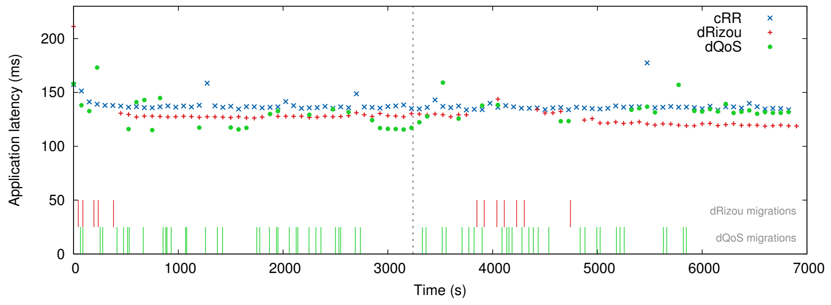


Figure 3: Performance of the tag-and-count topology when the nodes' utilization changes

reaching the final consumer. The placement of source and consumer is fixed. The other operators are unpinned and replicated (i. e., two executors are assigned to each of them). Figure 3 shows the evolution of the application end-to-end latency; on its bottom, we indicate the run-time reassignments performed by the distributed schedulers (cRR does not intervene during the execution). We start the application and, after 3240 s, we artificially increase the load on a subset of three nodes using the Linux tool `stress`. The subset is composed by one worker node running some application executors and two free worker nodes. This event is represented in Fig. 3 with a vertical dotted line. As the distributed scheduler (both dQoS and dRizou) perceives the change, it moves the application operators on lightly loaded nodes. cRizou reduces the application latency with respect to cRR of about 12.6 % (measured between 5000 s and the end of the experiment). Furthermore, differently from dQoS, dRizou converges with a lower number of reassignments, increasing the application availability.

6 Open Challenges

Although our extension enables the execution of the Storm, as a generic DSP system, in a distributed environment, the peculiarities of Cloud computing require an efficient management of scalability, elasticity and fault tolerance. With no claim of completeness, we summarize some of the needed mechanisms.

Stateful Migration: an operator is stateful if its behavior depends also on its internal state. Therefore, moving a stateful operator requires an efficient relocation of its internal, possibly extremely large, state across the network. In literature, the general tendency is to use the strategy *stop-move-play*, which stops the incoming streams, moves the operator and its state, and redirects the streams to the new operator location (e. g., [4, 6]). Wu et al. [16] improve this technique by aggressively dividing the application-level state in computation slices, which are asynchronously checkpointed to remote machines, enabling parallel state migrations between nodes. However, most of the existing techniques do not fit well in a latency sensitive scenario, because they do not explicitly consider QoS attributes of communication links and computational nodes. A fast, live, and QoS-aware migration strategy could bring important improvements to these systems.

Elastic Replication: the ability of the system to autonomously adapt the number of replicas for each operator. This mechanism can increase non-functional attributes of the applications (e. g., availability) with the penalty of a higher cost and resource overhead. Bellavista et al. [2] present a prototype that allows to trade-off monetary cost and active replication. An alternative to active replication is upstream backup, which achieves fault-tolerance using an upstream server that stores a copy of the operator state. However, since this technique imposes a higher recovery time, it is used as a second-class mechanism. For example, Heinze et al. [8] combine these two mechanisms to reduce the overall resource consumption with respect to a recovery time threshold.

Elastic Sharding: the ability of automatically scaling in and out the number of shards for an operator based on the incoming load. Each shard of an operator is in charge of a partition of its incoming stream; therefore, this mechanism can increase the application scalability by handling a growing workload. As a consequence, the system can acquire and release resources when needed, without resorting in over- or under-provisioning (i. e., resource elasticity [9]). Increasing the number of shards is critical for stateful operators, because the system needs to preserve the consistency of the operations. In literature different works investigate this issue. Some solutions define a-priory the maximum number of shards [14], expose some API to manually manage the state [4], or automatically determine the optimal number of state partitions to be used [6].

Solutions to the above mentioned issues are almost consolidated in a clustered environment, however the emerging distributed Cloud scenario imposes a new perspective. In a distributed environment, aside the number of replicas or shards, the scheduler should also define their optimal placement with respect to some QoS metrics (e. g., latency, bandwidth, reliability), considering the heterogeneity of applications and resources. For example, replicas should be placed in different availability zones; different shards of the same operator should let users experience similar response times.

7 Conclusion

The ever increasing diffusion of sensing and computing devices enables a new generation of DSP systems. Starting from the major drawbacks of an existing framework to the execution in distributed and dynamic environments, we developed an extension of Apache Storm that provides distributed monitoring, scheduling and management capabilities. The evaluation results showed that our extension of Storm is suitable to operate in a distributed environment, where QoS-awareness and adaptation capabilities can be truly beneficial to the application performances. Finally, we highlighted some core mechanisms that can improve performances of DSP systems when executed in a distributed Cloud.

As future work, we will provide a formal definition of the placement problem for DSP applications and design a new placement algorithm that better leverages the potentialities of a distributed Cloud model.

Acknowledgments. Thanks to the anonymous reviewers for the valuable comments, to Valeria Cardellini, and to Gabriele Scolastri for the implementation of the Rizou's algorithm.

References

1. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in Storm. In: Proc. of ACM DEBS '13. pp. 207–218 (2013)
2. Bellavista, P., Corradi, A., Kotoulas, S., Reale, A.: Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. In: EDBT. pp. 85–96 (2014)
3. Cardellini, V., Grassi, V., Lo Presti, F., Nardelli, M.: Distributed QoS-aware Scheduling in Storm. In: Proc. of ACM DEBS '15. ACM (2015)
4. Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: Integrating scale out and fault tolerance in stream processing using operator state management. In: Proc. of ACM SIGMOD'13. pp. 725–736. ACM (2013)
5. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: A decentralized network coordinate system. SIGCOMM Comput. Commun. Rev. 34(4), 15–26 (2004)
6. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. 25(6), 1447–1463 (2014)
7. Heinze, T., Aniello, L., Querzoni, L., Jerzak, Z.: Cloud-based data stream processing. In: Proc. of ACM DEBS '14. pp. 238–245 (2014)
8. Heinze, T., Zia, M., Krahn, R., Jerzak, Z., et al.: An adaptive replication scheme for elastic data stream processing systems. In: Proc. of ACM DEBS '15. pp. 150–161. ACM (2015)
9. Hochreiner, C., Schulte, S., Dustdar, S., Lecue, F.: Elastic stream processing for distributed environments. Internet Computing, IEEE 19(6), 54–59 (2015)
10. Lakshmanan, G.T., Li, Y., Strom, R.: Placement strategies for internet-scale data stream systems. Internet Computing, IEEE 12(6), 50–60 (2008)
11. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., et al.: Network-aware operator placement for stream-processing systems. In: Proc. of IEEE ICDE '06 (2006)
12. Rizou, S., Durr, F., Rothermel, K.: Solving the multi-operator placement problem in large-scale operator networks. In: Proc. of IEEE ICCCN 2010 (2010)
13. Satyanarayanan, M., Schuster, R., Ebling, M., Fettweis, G., et al.: An open ecosystem for mobile-cloud convergence. IEEE Communications 53(3), 63–70 (2015)
14. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., et al.: Storm@Twitter. In: Proc. of ACM SIGMOD '14. pp. 147–156 (2014)
15. Urbani, J., Margara, A., Jacobs, C., Voulgaris, S., et al.: AJIRA: a lightweight distributed middleware for MapReduce and stream processing. In: Proc. of IEEE ICDCS '14. pp. 545–554 (2014)
16. Wu, Y., Tan, K.L.: Chronostream: Elastic stateful stream computation in the cloud. In: IEEE Int'l Conf. ICDE 2015 (forthcoming) (2015)
17. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., et al.: Spark: Cluster computing with working sets. In: Proc. of USENIX HotCloud'10. p. 10 (2010)