On QoS-aware Scheduling of Data Stream Applications over Fog Computing Infrastructures

Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, Matteo Nardelli
Department of Civil Engineering and Computer Science Engineering
University of Rome "Tor Vergata", Italy
Email: cardellini@ing.uniroma2.it, vgrassi@info.uniroma2.it, lopresti@info.uniroma2.it, nardelli@ing.uniroma2.it

***Abstract*—Fog computing is rapidly changing the distributed computing landscape by extending the Cloud computing paradigm to include wide-spread resources located at the network edges. This diffused infrastructure is well suited for the implementation of data stream processing (DSP) applications, by possibly exploiting local computing resources. Storm is an open source, scalable, and fault-tolerant DSP system designed for locally distributed clusters. We made it suitable to operate in a geographically distributed and highly variable environment; to this end, we extended Storm with new components that allow to execute a distributed QoS-aware scheduler and give self-adaptation capabilities to the system.**

In this paper we provide a thorough experimental evaluation of the proposed solution using two sets of DSP applications: the former is characterized by a simple topology with different requirements; the latter comprises some well known applications (i.e., Word Count, Log Processing). The results show that the distributed QoS-aware scheduler outperforms the centralized default one, improving the application performance and enhancing the system with runtime adaptation capabilities. However, complex topologies involving many operators may cause some instability that can decrease the DSP application availability.

I. Introduction

In the last decade the benefits introduced by Cloud computing, such as low upfront costs, elasticity and scalability, have pushed computation, control, and storage into few large data centers. On the other hand, the recent idea behind Fog computing [1] is to make the Cloud descending to the network edges by moving computational resources from few large data centers located in the network core to diffused cloudlets or micro Clouds, that serve as a second-class data center with soft state [2]. Urban environments are more and more pervaded by a wide plethora of sensing systems, which produce an exponentially increasing flow of data. To make a city smart, or smarter, useful information should be extracted from these raw data and timely processed to improve urban services and, therefore, the citizen well-being. In the context of smart cities, data stream processing (DSP) applications have been and are proposed in a variety of cross concerning domains, such as urban traffic monitoring and management [1], [3], environmental monitoring, and building energy management.

As presented in [4], we can distinguish at least three generations of DSP systems; the current one is driven by the trend towards Cloud computing, where elasticity and fault tolerance are key architectural features. Despite this, most DSP systems are still designed to run in local clusters, where the often homogeneous nodes are interconnected with negligible latencies (e.g., [5], [6], [7]). Furthermore, their execution in a urban environment, characterized by the Fog computing diffused infrastructure possibly integrated with the more traditional Cloud environment, poses new challenges that include network and system heterogeneity and dynamism, geographic distribution as well as non-negligible network latencies. To address these challenges, we have proposed in a previous work [8] a distributed and self-adaptive QoS-aware scheduler for Storm[1], an open source DSP system. Storm is attracting increasing industrial and academic interests, e.g., [9], [7], [10]. Its default scheduler, which performs a basic even distribution of the processing elements on the computational nodes, aims at load sharing in the cluster, although schedulers driven by other metrics such as the inter-node traffic have been proposed, e.g., [9], [10]. However, the current architecture of Storm as well as the above cited Storm-based works adopting a centralized scheduler are not designed to process streams in a urban sparse distributed environment, such as that envisioned by the convergence of Fog computing, Cloud computing, and IoT, where DSP systems, relying on both distant and proximate computing resources, are requested to scale on a wide-spread infrastructure with many running applications and distributed sources. We designed and implemented a distributed, QoS-aware and self-adaptive scheduler for Storm by adding a few key modules to the standard Storm architecture in a user-transparent way [8]. Our scheduler can scale as the number of applications and network resources increase since it does not require a global knowledge of the system. Moreover, the self-adaptive capability, which is a must in the Fog context, allows to automatically reconfigure the operator placement in a distributed fashion when unpredictable environmental changes occur. To demonstrate the effectiveness of the extended Storm, we implemented a distributed scheduling algorithm that is aware of QoS attributes by adapting the network-aware scheduling algorithm by Pietzuch et al. [11]. The scheduling strategy aims at placing as far as possible the DSP application near to the data sources and the final consumers. A similar idea is also investigated in [12], which relies on mobile devices resources to pre-process data streams generated on neighbors clients. Differently, we do not rely on mobile resources for computation because of their limited battery power and the difficulties introduced by their mobility.

In this paper, we present a thorough experimental evaluation of the proposed solution using a good variety of DSP applications, coming with different processing requirements. Specifically, we consider two sets of DSP applications where the former is characterized by a simple topology with differ-

[1] https://storm.apache.org

978-1-4673-7194-0/15/$31.00 ©2015 IEEE

ent requirements and the latter comprises some well known applications. Our results show that the distributed QoS-aware scheduler outperforms the centralized default scheduler of Storm, improving the application performance and enhancing the system with runtime adaptation capabilities, that allows to react to changes in the operating environment by properly reassigning processing elements in a distributed fashion. However, for complex topologies involving many operators the distributed scheduler may determine some instability that can be detrimental for the DSP application availability and deserves future investigation on the scheduling algorithm.
The rest of this paper is organized as follows. In Section II we briefly introduce the official release of the Storm framework and its DSP model. Then, in Section III we present the design of the extended Storm with the QoS-aware distributed scheduler. We analyze a wide set of experiments run on our Storm prototype in Section IV and conclude with Section V.

II. Data Stream Processing in Storm

A DSP application continuously processes streams of data, generated from multiple distributed sources, to extract valuable information or knowledge as soon as data are collected.

Storm is an open source, real-time, 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, e.g., a physical host, a virtual machine, a mobile device, whereas the overlay network comprises the logical links among nodes. In Storm, a *stream* is an unbounded sequence of *tuples*, which are key-value pairs created and processed in a distributed way. A *spout* is an application 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 a new outgoing stream, like spouts do. We can further distinguish between *pinned* and *unpinned* operators: the former have a fixed physical location, while the latter can be conveniently instantiated at some arbitrary node of the network. A Storm application is represented by its *topology*, which is a directed acyclic graph with spouts and bolts as vertices and streams as edges. Storm uses three types of entities with different grain to execute a topology: tasks, executors, and worker processes. A *task* is an instance of an application operator (i.e., spout or bolt) and represents the smallest schedulable unit. To process large amount of data in a timely manner and exploit parallelism, multiple tasks can be instantiated for an operator; tasks are connected each other according to the link between their related operators in the application. Storm defines various partitioning strategies to send tuples to tasks, including shuffle grouping (i.e., random partitioning) and fields grouping (i.e., hash-based partitioning on a subset of tuple fields). An *executor* can execute one or more tasks related to the same operator. A *worker process* is a Java process that runs one or more executors of the *same* topology; this also means that a topology can be distributed across different worker processes. There is an evident hierarchy among the Storm 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.

The Storm architecture is composed by two centralized components, Nimbus and ZooKeeper, and a cluster of locally distributed worker nodes, which provide the computational resources to the framework. *ZooKeeper*[2] is a shared memory service for managing configuration information and enabling distributed coordination. *Nimbus* is the centralized component in charge of coordinating the topology execution; after receiving a new topology, it uses a software component called *scheduler* in order to deploy the application operators on the pool of available resources. Different scheduling algorithms can be defined for this component; the default one is the *EvenScheduler*, which evenly assigns topologies across worker nodes selecting them with a round-robin strategy. The assignment plan determined by the scheduler is communicated to all the worker nodes through ZooKeeper. Since each worker node can execute one or more worker process, a *Supervisor* component on the worker node starts or terminates 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*.

III. Distributed Scheduling in Storm

We have extended the Storm architecture to run a distributed QoS-aware scheduling algorithm [8]. The newly introduced components, illustrated in orange in Figure 1, are: the *AdaptiveScheduler*, the *QoSMonitor*, and the *WorkerMonitor*. The *AdaptiveScheduler* is located on each Supervisor and represents the distributed scheduler that executes the distributed placement policy. The *QoSMonitor* estimates the network latency with respect to the other system nodes and monitors the QoS attributes of the worker node, namely the node availability and its resources utilization. A *WorkerMonitor* is executed for each Storm worker process and computes the data rate exchanged among the application components. The information collected by these monitors is then used by the QoS-aware scheduling algorithm. Besides introducing the distributed scheduler on each worker node, we also maintain a centralized scheduler, called *BootstrapScheduler*, which is executed by Nimbus and is in charge of defining the initial assignment of the application, monitoring its execution, and rescheduling the application when a worker process fails.

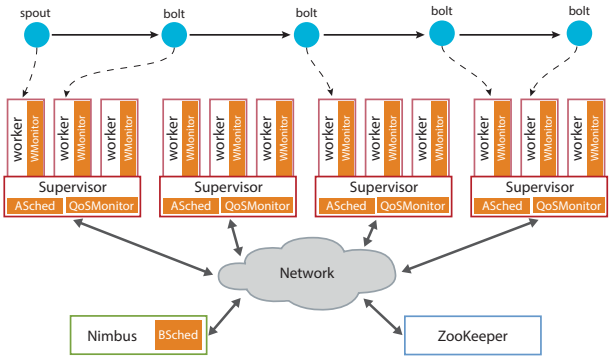


Fig. 1: Extended Storm architecture (new components in orange): AdaptiveScheduler is abbreviated as ASched, WorkerMonitor as WMonitor, and BootstrapScheduler as BSched

[2]http://zookeeper.apache.org/

A. Monitoring Components: QoSMonitor and WorkerMonitor

The QoSMonitor provides the QoS awareness to each distributed scheduler, thus it is responsible of obtaining intra-node (i.e., utilization and availability) and inter-node (i.e., network) node information. For the latter we resort to a network coordinates (NC) system that provides an accurate estimate of the round-trip latency between any two network locations, i.e., computational nodes, without the need of an exhaustive probing. The NC system is maintained through the Vivaldi's algorithm [13], a decentralized algorithm with linear complexity with respect to the number of network locations, thus ensuring scalability. To make each node be informed of all the QoS-related information of the other nodes, we adopts a gossip-based information dissemination scheme. The gossiping protocol is periodically executed: every 30 s each QoSMonitor randomly picks another worker node, and the two monitors exchange their own information. Each QoSMonitor does not further disseminate a newly received information. The *WorkerMonitor* is responsible of obtaining the incoming and outgoing data rate for each executor that runs on the node. This information is stored in a local database to be subsequently used by the distributed scheduler.

B. AdaptiveScheduler

The *AdaptiveScheduler* executes the distributed QoS-aware scheduling algorithm on every worker node. Only the executors assigned to the worker node can be managed by this scheduler, which can reassign them to improve the application performance. This component is orchestrated as a classical adaptive software system, that reacts to internal and external changes of the operating conditions through a feedback control loop. A single loop iteration is executed periodically (every 30 s), and is composed by the Monitor, Analyze, Plan, and Execute phases of the MAPE reference model for autonomic systems [14].

Monitor. During the *Monitor* phase the AdaptiveScheduler acquires the information collected by the QoSMonitor and identifies the set of local executors that could be moved. An executor is movable if it is not pinned and not directly connected to an operator which is going to be reassigned.

The remaining phases of the MAPE loop are executed in sequence for each local executor in the set of movable candidates. That is, for each movable executor, the AdaptiveScheduler analyzes if the executor will be effectively relocated; in the positive case, it plans where and executes the corresponding actions. The rationale to consider each executor at a time is to reduce the effect of multiple relocations that can negatively affect the application performance.

Analyze. In the *Analyze* phase, the AdaptiveScheduler determines if the local executor e_i previously identified as movable candidate will be effectively moved to another position. To perform this task in a distributed fashion, we exploit the elegant approach proposed by Pietzuch et al. [11], which comprises a cost space and a two-step operator placement algorithm. The cost space models the placement problem by transforming the performance metrics of interest into distances in a multidimensional space; we adopt a four dimension space, where two dimensions refer to the latency attribute, while the other two refer to node availability and utilization, respectively. The Analyze phase is in charge of executing the first step of the placement algorithm [11], which determines the placement of the candidate executor e_i into the cost space.

Plan. If the Analyze phase finds that executor e_i needs to be moved to a new position, it triggers the *Plan* phase; otherwise, it analyzes the next executor. The Plan phase is responsible of executing the second step of the placement algorithm, which, relying on the cost space, determines a worker node that will execute the candidate executor e_i. If no worker node is found, the MAPE iteration for executor e_i terminates; otherwise, the Plan phase performs two additional tests and the executor e_i is moved only if both of them are passed. First, the *relative distance* is evaluated in order to avoid moving executors to a new candidate worker node if such change of position does not improve the application performance. Indeed, the migration of an operator has a non negligible cost (i.e., downtime, network and computational cost, loss of state information). Then, the *look-ahead* avoids moving an executor if cyclic reassignments can be generated; indeed the Pietzuch's scheduling algorithm inherently introduces new opportunities to generate them.

Execute. Finally, in the *Execute* phase, if a new assignment must take place, the executor e_i is moved to the new candidate node. The new assignment decision is shared with the involved worker nodes through ZooKeeper. 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. When a reassignment takes place, the topology is considered in a cooldown state for the next K_{cld} invocations of the AdaptiveScheduler; in this state, all its executors cannot be moved. This behavior allows the scheduler to evaluate the effects of a previously taken decision, before planning a new assignment.

Thanks to the adaptation cycle and the multi-dimensional cost space, the AdaptiveScheduler can manage changes that may occur both in the infrastructure layer (e.g., new worker nodes that appear or existing ones that fail) and the application layer (e.g., increase in the source data rate). A more detailed description of the scheduler behaviour can be found in [8].

C. BootstrapScheduler

Nimbus runs a centralized scheduler, that we called *BootstrapScheduler*, which defines the initial assignment of the application, monitors its execution, and restarts failed executors, when, for example, a worker node disappears. The BootstrapScheduler is a centralized version of the QoS-aware scheduling algorithm, which assigns executor pools (i.e., groups of executors), instead of single executors, in order to efficiently use the system resources. Its detailed description can be found in [8].

IV. Experimental Results

In this section we prove that our distributed QoS-aware scheduler can improve the application performance, when the DSP system is deployed on a network with not negligible latency and is subject to changes in the QoS of the nodes. Even if our scheduler can take into account different QoS attributes, in this paper we focus only on network latency and node utilization. We first show how the scheduler reacts to runtime changes by reassigning operators (Section IV-A); then, we investigate its performance under different data stream applications (Section IV-B). We compare the performance of our scheduler (named as **dQoS**) to that of the centralized default EvenScheduler in Storm (named as **cRR**), which has

been presented in Section II. We performed the experiments using Apache Storm 0.9.3 on a cluster composed of 8 worker nodes, each with 2 worker slots, and 2 further nodes for Nimbus and ZooKeeper. Each node is a virtual machine with Ubuntu 14.04 LTS configured with one vCPU on an Intel Xeon E5504 Quad-code and 2 GB of RAM. We emulated wide-area network latencies among the Storm nodes using *netem*[3], which applies to outgoing packets a Gaussian delay with mean and standard deviation in the ranges $[12, 32]$ ms and $[1, 3]$ ms, respectively (see [8] for details). We set the weights for the QoS metrics in the placement algorithm so to weigh twice the latency with respect to the utilization. The other parameters of the distributed QoS-aware scheduler are set as in [8]. As main performance metrics, we consider the average latency experienced by tuples to traverse the entire application topology, and the average inter-node traffic incurred by the topology.

A. Adaptation Capabilities

In this first set of experiments we compare the two schedulers when the load experienced by the worker nodes changes during the application execution due to same external noise event. We use a simple application which tags and counts sentences produced by a data source; its topology is represented in Figure 2 and is composed by a source, which generates 10 tuples/s, followed by a sequence of 5 operators before reaching the final consumer. The source and the consumer are the pinned operators.

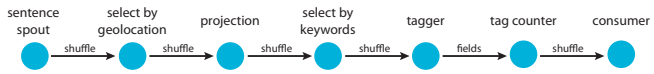
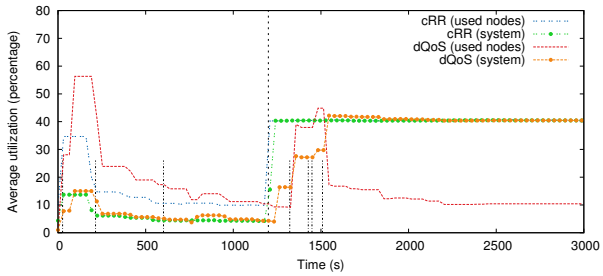
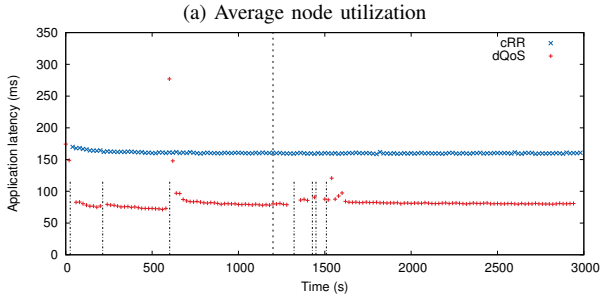


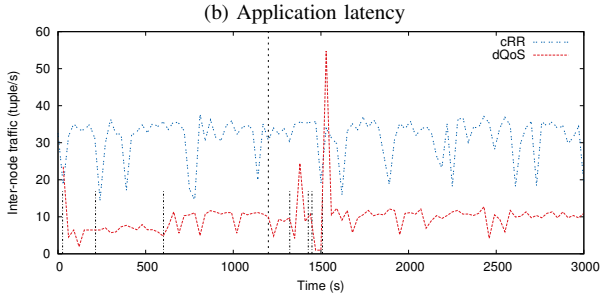
Fig. 2: Tag-and-count topology



(a) Average node utilization



(b) Application latency



(c) Inter-node traffic

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

a) Baseline Case: We use the tag-and-count topology with a single executor for each operator. Figure 3 shows the evolution of the observed metrics; vertical dot-dash lines indicate a new run-time reassignment performed by some dQoS scheduler, while, on the contrary, the cRR scheduler does not intervene during the application execution. We start the application on an idle cluster. As soon as the application metrics are collected by the QoSMonitor components, i.e., the exchanged datarate between operators and the node utilization are available, the dQoS scheduler performs some adjustments to the initial placement decision. After 1200 s, we artificially increase the load on a subset of three nodes using the Linux tool *stress*. This subset is composed by one worker node running two application executors and two free worker nodes. This event is represented with a vertical dotted line in Figure 3. Figure 3a shows the average utilization of the subset of worker nodes that run the application as well as the average utilization of the overall system. Soon after the event, the dQoS scheduler moves the application operators on lightly loaded nodes. The effectiveness of the decision is clearly visible in Figure 3a. Figure 3b shows that the presence of a stressed node does not degrade the application latency; this happens because the processing time of each operator is one order of magnitude lower than the network latency between worker nodes. Overall, the placement strategy of the dQoS scheduler reduces the application latency of about 49% with respect to that achieved by cRR. Furthermore, the inter-node traffic, reported in Figure 3c, shows a transient period for the dQoS scheduler after 1200 s which lasts for about 300 s. This period length depends on the number of reassigned operators and on the time interval each distributed scheduler must wait between two consecutive placement decisions (being $K_{cld} = 5$, this time interval is equal to 150 s).

b) Heavy Application: This second experiment investigates how the scheduler performs when a "heavy" application is submitted to the system. We modify the unpinned operators of the tag-and-count topology in order to waste some CPU time so that each operator completes its execution in about 14 ms, which is one order of magnitude larger than the baseline case. The load stress event is launched at 2450 s. Similarly to the previous experiment, the dQoS scheduler reacts to the load surge by reassigning operators, which impacts positively on the utilization (not shown for space reasons). However, in this case (see Figure 4) the degradation of the application latency for cRR is clearly visible, because the processing time is negatively influenced by the load stress event. On the other hand, dQoS allows to keep the latency low after

[3]Network Emulator: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

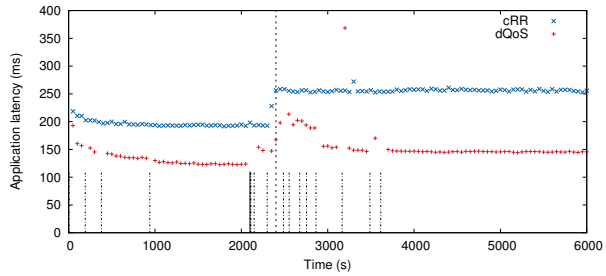
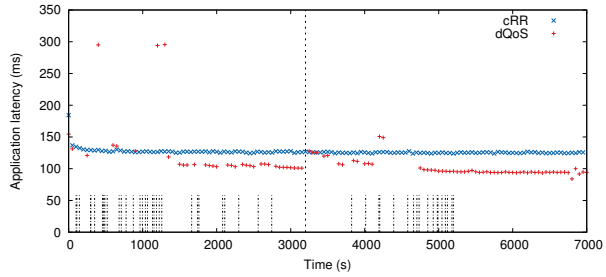


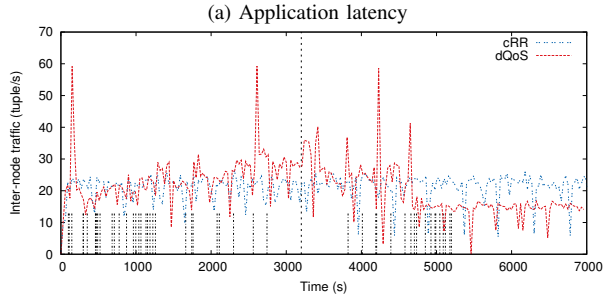
Fig. 4: Average latency of the "heavy" version of the tag-and-count topology when the nodes' utilization changes

a transient period. Between 1000 s and 2000 s, the average application latency experienced with dQoS is reduced by 35%, while between 4000 s and 5000 s it is reduced by 43%. We also observe that the BootstrapScheduler intervenes at 2100 s, because a worker process is erroneously terminated by the system.

c) Replicated Operators: This third experiment, reported in Figure 5, investigates the adaptation capabilities of the system when the unpinned operators of the tag-and-count application are replicated (i.e., two executors are assigned to each unpinned operator). The stress event is launched at



(a) Application latency



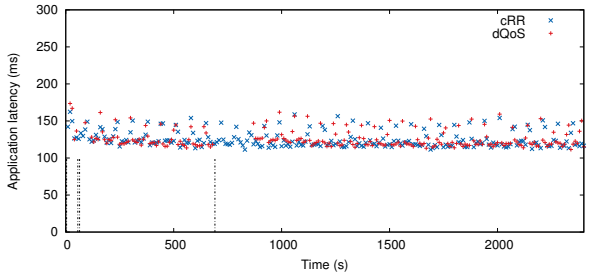
(b) Inter-node traffic

Fig. 5: Performance of the tag-and-count topology with replicated unpinned operators when the nodes' utilization changes

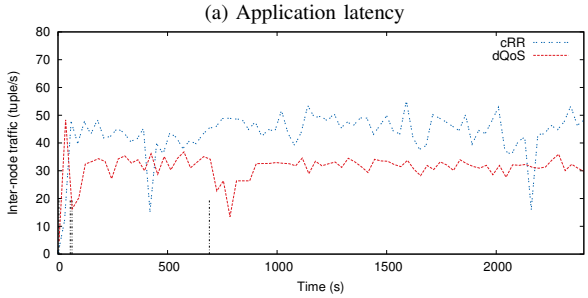
3200 s, and the distributed scheduler reacts to it improving the observed metrics: after the final placement, the application latency is reduced by about 25%. However, this experiment provides us a different insight on the distributed scheduler. First, we observe that the two transient periods (the initial one after the start and that after the load stress event) increase their length, because dQoS must deal with a larger number of operators, which, in turn, are more connected among them. Indeed, the number of logical links between pairs of operators grows from 6 to 20. A decentralized reassignment decision independently taken by one distributed scheduler provokes chain reactions in the other schedulers and the lack of coordination among them can thus originate some instability in the placement decisions. Even if the final application placement can improve the performance in terms of the observed metrics, the more frequent and larger number of operator reassignments and related stop-and-replay of the involved operators can negatively impact on the application availability during the transient periods. Second, the application latency obtained by the cRR scheduler is lower than that measured in the previous experiments. By comparing Figures 3c and 5b we can understand the reason: when the number of executors increases while keeping the number of worker process constant, the probability that the executors of two consecutive operators are on the same worker process increases as well. Therefore, data spend on average less time to traverse the topology.

B. Performance with Well-known Applications

In this second set of experiments, we investigate the behavior of the QoS-aware scheduler when two well-known data processing applications are executed by Storm, namely Word Count (stream version) and Log Stream Processing.



(a) Application latency



(b) Inter-node traffic

Fig. 6: Performance of the Word Count topology

d) Word Count: The Word Count topology is composed by a sequence of a source generating 1 tuple/s, two operators, and a consumer. The first operator splits the sentence into words and feeds the next one, which counts the occurrence of each word; each update of the counters is notified to the consumer. Source and consumer are pinned. We assign two executors to the source and three executors to each other operator. dQoS and cRR schedule the application on the same two worker nodes. As detailed in [8], the QoS-aware scheduler relies on the pinned operators to drive the placement of the executor pools. However, for the current topology the scheduler is constrained by the presence of at least a pinned operator for

each executor pool. Therefore, the application latency shown in Figure 6a is comparable for the two scheduling strategies. Anyway, the dQoS scheduler minimizes the network usage by re-arranging the executors on the worker nodes; the inter-node traffic is reduced of about 31% (see Figure 6b).

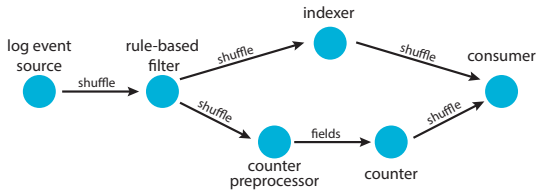
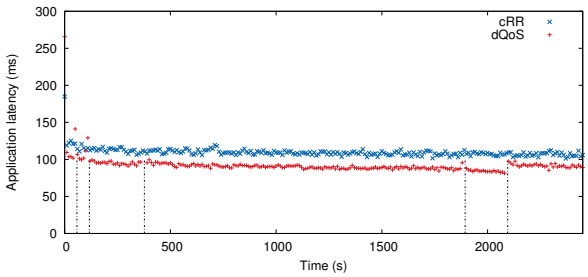
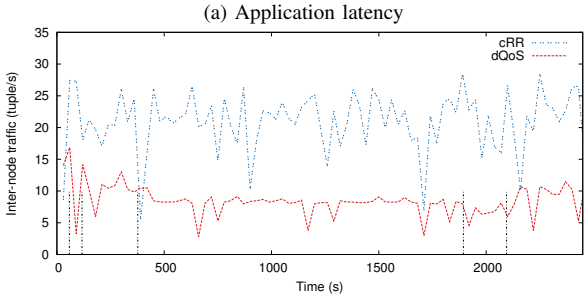


Fig. 7: Log Processing topology

e) Log Processing: We developed a log processing application based on [15], which relies on Redis as external caching service and whose topology is illustrated in Figure 7. The data source emits 10 log events/s that are filtered and sent to two different branches of the topology. The first branch is made of one operator, which indexes log events on Redis, while the latter is made of a sequence of two operators, which collect statistics. Subsequently, both the branches are merged together on the final consumer. The source and the consumer are pinned into the network. We assign one executor to each pinned operator and two executors for unpinned ones. From Figure 8 we observe that the QoS-aware placement improves the application performances, while reducing the network usage: the average application latency and the inter-node traffic are reduced by about 18% and 63% respectively.



(a) Application latency



(b) Inter-node traffic

Fig. 8: Performance of the Log Processing topology

V. Conclusions

In this paper we have extensively evaluated the distributed QoS-aware scheduler for DSP systems based on Storm [8], which is able of operating in a distributed Fog computing environment. We have used two sets of applications: the former relies on a reference topology with different requirements; the latter on some well known applications. The results show that our scheduler outperforms the Storm default one, improving the application performance, and enhancing the system with adaptation capabilities. However, the investigation also pointed out a weakness of the scheduling algorithm: since each placement decision is taken in a reactive, distributed, and independent manner, for complex topologies involving many operators, it can determine some instability that affects negatively the application availability. As future work, we plan to design a new scheduling algorithm which can overcome the highlighted weaknesses, and further leverage the potentialities offered by an even more diffused Fog environment, with mobile devices actively involved in data stream processing and integrated with the capability of controlling network resources through Software Defined Networking.

References

[1] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*, ser. SCI. Springer, 2014, vol. 546, pp. 169–186.

[2] M. Satyanarayanan *et al.*, "An open ecosystem for mobile-cloud convergence," *IEEE Communications*, vol. 53, no. 3, pp. 63–70, 2015.

[3] A. Artikis *et al.*, "Heterogeneous stream processing and crowdsourcing for urban traffic management," in *Proc. of EDBT '14*, 2014.

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

[5] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal, "AJIRA: a lightweight distributed middleware for MapReduce and stream processing," in *Proc. of IEEE ICDCS '14*, 2014, pp. 545–554.

[6] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. of IEEE ICDM Workshops '10*, 2010, pp. 170–177.

[7] A. Toshniwal *et al.*, "Storm@Twitter," in *Proc. of ACM SIGMOD '14*, 2014, pp. 147–156.

[8] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Distributed QoS-aware scheduling in Storm," Univ. Roma Tor Vergata, Tech. Rep. DICII RR-15.7, Mar. 2015, http://www.ce.uniroma2.it/publications/RR-15.7.pdf.

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

[10] 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.

[11] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proc. of IEEE ICDE '06*, 2006.

[12] J. Morales, E. Rosas, and N. Hidalgo, "Symbiosis: Sharing mobile resources for stream processing," in *Proc. of IEEE ISCC 2014*, vol. Workshops, Jun. 2014.

[13] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 15–26, 2004.

[14] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[15] Q. Anderson. (2015) Log topology. [Online]. Available: https://bitbucket.org/qanderson/log-topology/