Poster: Distributed QoS-Aware Scheduling in Storm

Valeria Cardellini
cardellini@ing.uniroma2.it

Vincenzo Grassi
vgrassi@info.uniroma2.it

Francesco Lo Presti
lopresti@info.uniroma2.it

Matteo Nardelli
nardelli@ing.uniroma2.it

Department of Civil Engineering and Computer Science Engineering
University of Rome "Tor Vergata", Italy

ABSTRACT

Storm is a distributed stream processing system that has recently gained increasing interest. We extend Storm to make it suitable to operate in a geographically distributed and highly variable environment such as that envisioned by the convergence of Fog computing, Cloud computing, and Internet of Things.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

Keywords

Distributed event processing, Data streaming, Resource management, Self-adaptive scheduler, Storm

1. INTRODUCTION

Urban environments are more and more permeated by a diffused and networked infrastructure that includes the sensors embedded in human-carried mobile devices, the monitoring systems embedded in public utilities, and many other kinds of sensing systems, i.e., the Internet-of-Things (IoT). All these distributed and heterogeneous data sources continuously produce ever-increasing streams of data that can be collected and processed by distributed data stream processing (DSP) systems. These systems aim at extracting valuable information in a timely way. At the same time, the large adoption of Cloud computing and the appearing of Fog computing, which complement traditional large data centers with widespread resources located in smaller data centers at the edges of the network, are changing the landscape of traditional DSP platforms on which data stream applications are executed. In this scenario, we can envision a distributed DSP system spread among multiple small data centers sparse in the urban environment, which would be close to the sources but with a non-negligible network latency among them. To fully exploit the potential of such infrastructure, the distributed system should be enhanced with a scheduler that allows to place the stream processing elements in a network-aware and scalable fashion, while taking into account other Quality of Service (QoS) attributes.

In this paper, we present a distributed and self-adaptive QoS-aware scheduler for Storm[1], an open source, real-time, distributed and resilient DSP system, that is attracting increasing interests [4]. Existing Storm extensions, e.g., [1, 7], propose only centralized scheduling algorithms that optimize the utilization of the DSP system; furthermore, they are designed for clustered environments, therefore may be unsuitable when Storm is deployed on a infrastructure having non-negligible latencies. The main contributions of our work are as follows: *a)* we extend the Strom architecture by designing and implementing the support for distributed QoS-aware scheduling and run-time adaptivity; *b)* to show the flexibility of the proposed extension, we implement in Storm and evaluate the distributed network-aware scheduling algorithm proposed by Pietzuch et al. [6]. We remark that our extension is completely transparent and thus does not require any modification to existing (and future) Storm applications.

2. QOS-AWARE SCHEDULING

A DSP application can be represented by a directed acyclic graph (DAG), where the vertices are the application operators, and the edges are the streams exchanged between operators. For the execution, a DSP is instantiated in a distributed infrastructure, which consists of a set of worker nodes, i.e., computational resources, interconnected by an overlay network. The *operator placement* problem consists in determining the nodes that should host and execute the DSP operators; this task is performed by a DSP system component called *scheduler*. We implement a distributed scheduling algorithm that is aware of QoS attributes, specifically latency, node utilization, and availability. To this end, we adapt to Storm the network-aware scheduling algorithm proposed by Pietzuch et al. [6]. It comprises a *cost space*, which models the placement problem by transforming the performance metrics of interest into distances in this space, and an *operator placement algorithm*, which places operators in this cost space. It achieves good application performance by minimizing the amount of data that transits the network at a given instant, blending together network delay and network resource consumption: the smaller the link delays, the better the overall application delay; but, at the same time, the larger the data rate between two operators,

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

DEBS '15, June 29 - July 3, 2015, Oslo, Norway.

DOI: http://dx.doi.org/10.1145/2675743.2776766.

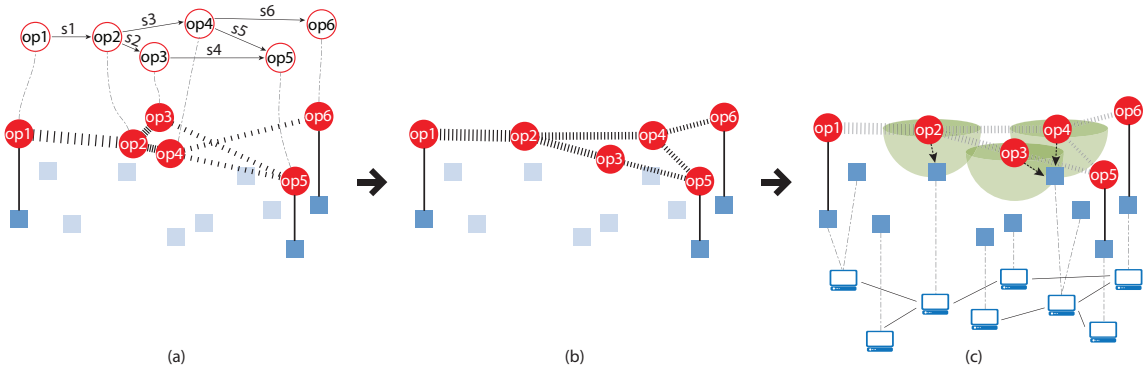


Figure 1: High-level architecture of our solution: (a) pinned and unpinned operators; (b) Virtual Operator Placement; (c) Physical Operator Placement

the closer they should be in the network (possibly co-located in the same physical node). Along with network usage, we consider two other metrics which capture the nodes performance: utilization and availability. The former captures the node processing latency, which is function of the node utilization level. The latter represents the fraction of time a node is up and able to execute operators code.

2.1 Cost Space

A cost space is a metric space where distance between two points *estimates* the cost of routing and processing data between two nodes placed in those two points. We adopt a four-dimension cost space, where two dimensions refer to the latency attribute, while the other two refer to node availability and utilization, respectively. The two latency dimensions of the cost space form a latency sub-space, where the distance between two nodes is an estimate of their network latency. The cost space dynamically adapts to changing network and node conditions as nodes continuously adjust their coordinates through measurements.

2.2 Placement Algorithm

The operator placement comprises two phases: the virtual operator placement and the physical operator mapping. The former determines the placement of the operators in the cost space; the latter maps its decision back to a physical node, see Figure 1. To perform the placement task, we need to distinguish between two kind of operators: a *pinned* operator has a fixed physical location (e.g., *producers*, *consumers*), while an *unpinned* one can be conveniently instantiated at some arbitrary node of the network. The placement algorithm, which is fully decentralized, is periodically executed for all unpinned operators in order to adapt to the ever changing conditions of the environment.

Virtual Placement Algorithm. The idea behind the placement algorithm is to regard the system of operators and links as a collection of massless bodies connected by springs. In this mechanical analogy, the rest position of the springs - which represents the minimum potential energy configuration - is the configuration to which the system naturally converges to, by simply letting each operator op_i *moves* as the results of the force $\vec{F_i}$ applied to it by the systems of springs. As observed in [6], by setting the spring extension equal to the latency, $s_l = \text{Lat}(l)$, and the spring constant to the data rate over that link, $k_l = \text{DR}(l)$, the minimum energy configuration of the spring system corresponds to the minimum network usage configuration of the operators.

Physical Placement Algorithm. Once the Virtual Placement Algorithm terminates, the operator op_i has associated a coordinate $\vec{P_i}$ in the latency space. Since the virtual operator placement uses only the latency dimension, $\vec{P_i}$ has the coordinate associated to availability and utilization equal to 1 and 0, respectively, i.e., $\vec{P_i} = (p_{l1i}, p_{l2i}, 1, 0)$. In the second stage of the placement algorithm, we map the operator op_i to an actual physical node ws_j. Ideally, the best candidate node is the one closest to the operator coordinate $\vec{P_i}$; thus, we choose the node ws_j with coordinate $\vec{P_j}$ with minimal Euclidean distance to $\vec{P_i}$. Differently from the original work [6], we formalize how the metrics interact among them. First, being the distance a trade-off among different dimensions, we need to normalize each coordinate. Availability and utilization are both in the range $[0, 1]$, thus we scale the latency coordinates to the same interval, dividing them by the largest observed delay. Second, since distinct applications can be differently affected by the performance indices, we introduce the weights w_l, w_a and w_u, for the latency, availability and utilization coordinates, respectively, to gauge the relative importance of the different performance indices. The distance between $\vec{P_i} = (P_{l1i}, P_{l2i}, P_{ai}, P_{ui})$ and $\vec{P_j} = (P_{l1j}, P_{l2j}, P_{aj}, P_{uj})$ is computed as follows:

$$d(\vec{P_i}, \vec{P_j}) = \sqrt{\begin{matrix} w_l^2[(\frac{P_{l1i}-P_{l1j}}{Lat_{max}})^2 & +(\frac{P_{l2i}-P_{l2j}}{Lat_{max}})^2]+ \\ w_a^2(P_{ai}-P_{aj})^2 & +w_u^2(P_{ui}-P_{uj})^2. \end{matrix}}$$

3. DISTRIBUTED SCHEDULING IN STORM

We extend the Storm architecture by adding a few key modules, represented in Figure 2. They enable a *distributed* QoS-aware scheduler and enhance the system with adaptation capability. The *AdaptiveScheduler* is located on each worker node and executes the distributed scheduling policy. The *QoSMonitor* estimates the network latency with the other system nodes and monitors the QoS attributes of the worker node, i.e., node availability utilization. It acquires infrastructure-related QoS metrics that are used for the placement task. We also introduce a *WorkerMonitor* for each Storm worker process, which computes and stores incoming and outgoing data rates; this information is used by the scheduler to compute DR(l). Besides introducing the distributed scheduler, we preserve a centralized scheduler, called *BootstrapScheduler*, executed by Nimbus. It defines

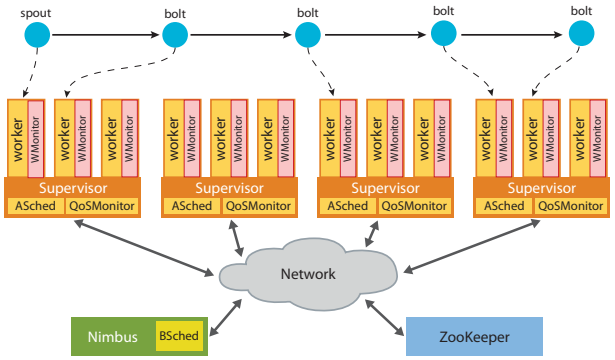


Figure 2: Extended Storm architecture: AdaptiveScheduler is abbreviated as ASched, WorkerMonitor as WMonitor, and BootstrapScheduler as BSched

the initial assignment of the application, monitors its execution and reschedules the application when a worker process fails. Our extension is completely user-transparent, because applications are executed with the new scheduler without requiring any change. More details can be found in [2].

3.1 QoSMonitor

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 delay) 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, without the need of an exhaustive probing. The NC system is maintained through the Vivaldi's algorithm [3], a decentralized algorithm which has linear complexity with respect to the number of network locations. To make each node be informed of all the QoS attributes of the other nodes, we rely on a gossip-based dissemination protocol.

3.2 AdaptiveScheduler

The *AdaptiveScheduler* executes the distributed scheduling policy on every worker node. In principle, any policy can be implemented; in this paper, as a proof of concept, we implemented a known use case, relying on the Pietzuch et al. algorithm presented in Section 2. Since the architecture of our extended Storm is modular and low-coupled, the implementation of this algorithm is straightforward. The algorithm has been adjusted 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.

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 [5].

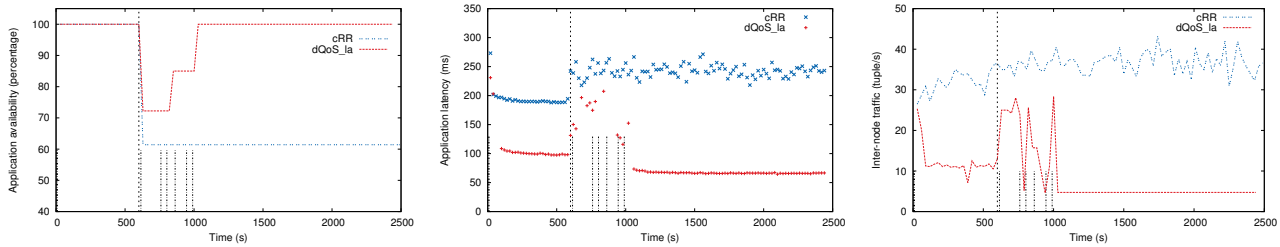
During the *Monitor* phase the AdaptiveScheduler acquires the information collected by the QoSMonitor and identifies the set of local executors that could be moved. An executor is movable if it is not pinned and not directly connected to an operator which is going to be reassigned. The remaining phases of the MAPE loop are executed in sequence for each local executor in the set of movable candidates. For each movable executor, the AdaptiveScheduler analyzes if the executor will be effectively relocated; in the positive case it plans where, and finally it 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. In the *Analyze* phase the AdaptiveScheduler runs the Virtual Placement Algorithm described in Section 2. The goal is to determine if the local executor e_i previously identified as movable candidate will be effectively moved to another position. 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 goal of the *Plan* phase is to determine which worker node will execute e_i. To this end, the planner executes the Physical Placement Algorithm to find the worker node closest to $\vec{P_i}$ which has at least a free worker slot and can thus host e_i. If no worker node is found, the MAPE iteration for executor e_i terminates. 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.

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).

4. EXPERIMENTAL RESULTS

We present two experiments in order to show the self-adaptation capabilities of the distributed QoS-aware scheduler (dQoS_la and dQoS_lu) and the default EvenScheduler of Storm (cRR) during the application execution. The former places operators exploiting QoS attributes as described in Section 2, whereas the latter uses a centralized round-robin policy. The experiments run on a Storm cluster composed of 8 worker nodes, each with 2 worker slots, and 2 further nodes for Nimbus and ZooKeeper. The DSP application is composed of a source, which generates 10 tuples/s, followed by a sequence of 5 operators before reaching the final consumer. The source can resend tuples at most once. More details on the experimental setup can be found in [2].

Exploiting Availability Dimension. The first experiment investigates dQoS_la, which uses network latency and node availability as QoS attributes. We give more weight to availability by setting $w_a = 10$ and $w_l = 1$. Different metrics for this experiment are reported in Figure 3. We start the application with all nodes having 99.999% availability. After 600 s, the availability of an active node suddenly decreases to 85%; this event is represented with a vertical dotted line in Figure 3. Figure 3a shows the overall application availability, which is computed as $\prod_{i \in WN^q} a_i$ being the application q a sequence of operators, where WN^q is the set of worker nodes involved with the execution of q, and a_i is the availability of node i. Since the default scheduler is blind to this metric, when the node availability decreases, the overall availability decreases to 61.41%. As a consequence, the ap-

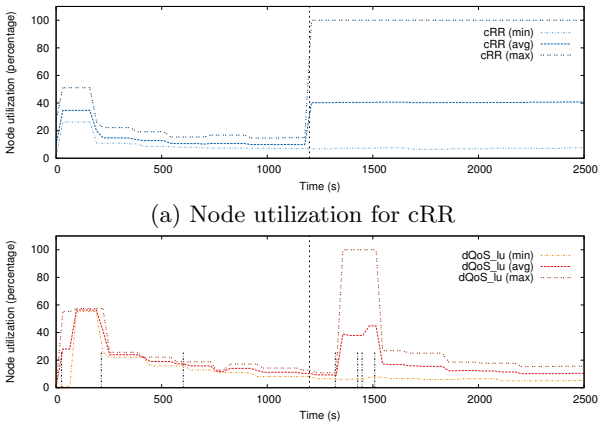


(a) Application availability (b) Application latency (c) Inter-node traffic

Figure 3: Comparison of dQoS_la with the default Storm scheduler (cRR) when the nodes' availability changes

plication end-to-end latency increases, because tuples spend more time in the upstream buffers and some of them are resent from the source (see Figure 3b). On the contrary, our distributed scheduler reassigns the operators to nodes with better availability; a new run-time reassignment performed by some distributed scheduler is indicated with vertical dot-dash lines in Figure 3. We can see that, after 1000 s, the application runs with an overall availability of 99.993%, which reduces the median of the application latency of about 72% with respect to that achieved by cRR. We can see that after 600 s, with dQoS_la there is a transient period of about 400 s (clearly visible in Figure 3c), where some distributed schedulers are executed to improve the application availability and reduce the network usage (Figure 3a). We also observe that during this experiment, the application scheduled with dQoS_la resent 2.53% of the tuples, while such percentage increases to 20.23% when scheduled with cRR.

Exploiting Utilization Dimension. The second experiment investigates dQoS_lu, which uses network latency and node utilization as QoS attributes. To make the placement decision resilient to minor fluctuations in the nodes' utilization, we weigh twice the latency with respect to the utilization ($w_l = 1; w_u = 0.5$). We start the application and, 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 three application executors and two free worker nodes. Figure 4 shows the minimum, average, and maximum utilization of the subset of worker nodes that run the application, while the beginning of the stress event is represented with a vertical dotted line. Since dQoS_lu is aware of the execution environment, after a transient period which ends at 1500 s, it moves all the application operators on lightly loaded nodes (i.e., load balancing). Each new placement decision of a dQoS_lu scheduler is represented with a vertical dot-dash line in Figure 4b. On the contrary, as shown in Figure 4a, the default Storm scheduler cannot react and change its scheduling decision.



(a) Node utilization for cRR

(b) Node utilization for dQoS_lu

Figure 4: Comparison of dQoS_lu with the default Storm scheduler (cRR) when the the nodes' utilization changes

5. CONCLUSIONS

We designed and implemented a distributed QoS-aware scheduler for DSP systems based on Storm, which is able of operating in a geographically distributed and dynamic environment. The experimental results show that our scheduler outperforms the centralized default one and enhances the system with adaptation capabilities to react to changes in a distributed fashion. We envision a fully integrated cross-layer QoS-aware management system for DSP. As future work, we plain to integrate the capability of controlling and managing network resources, taking advantage of recent advances in the networking research area, e.g., Software Defined Networking (SDN).

6. REFERENCES

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

[2] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Distributed QoS-aware scheduling in Storm. Technical Report DICII RR-15.7, Univ. Roma Tor Vergata, 2015. `www.ce.uniroma2.it/publications/RR-15.7.pdf`.

[3] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4), 2004.

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

[5] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.

[6] P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *Proc. of IEEE ICDE '06*, 2006.

[7] J. Xu, Z. Chen, J. Tang, and S. Su. T-Storm: Traffic-aware online scheduling in Storm. In *Proc. of IEEE ICDCS '14*, pages 535–544, June 2014.