Basanta-Val, P.; Fernández-García, N.; Wellings, A.J.; Audsley, N.C. Improving the predictability of distributed stream processors. *Future Generation Computer Systems, 2015*, v. 52, pp. 22-36.

DOI: https://doi.org/10.1016/j.future.2015.03.023

Improving the predictability of distributed stream processors

P. Basanta-Val [a,*], N. Fernández-García [b], A.J. Wellings [c], N.C. Audsley [c]

[a] Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid, Avda de la Universidad no 30, Leganes, 28911, Madrid, Spain

[b] Centro Universitario de la Defensa. Escuela Militar Marin, Universidade de Vigo, Spain

[c] Department of Computer Science, University of York, York, YO10 5GH, UK

ABSTRACT

Next generation real-time applications demand big-data infrastructures to process huge and continuous data volumes under complex computational constraints. This type of application raises new issues on current big-data processing infrastructures. The first issue to be considered is that most of current infrastructures for big-data processing were defined for general purpose applications. Thus, they set aside real-time performance, which is in some cases an implicit requirement. A second important limitation is the lack of clear computational models that could be supported by current big-data frameworks. In an effort to reduce this gap, this article contributes along several lines. First, it provides a set of improvements to a computational model called distributed stream processing in order to formalize it as a real-time infrastructure. Second, it proposes some extensions to Storm, one of the most popular stream processors. These extensions are designed to gain an extra control over the resources used by the application in order to improve its predictability. Lastly, the article presents some empirical evidences on the performance that can be expected from this type of infrastructure.

1. Introduction

Current trends in computational infrastructures look at the Internet as a low-cost distributed-computing platform for hosting legacy and next generation applications in the cloud [1–3]. The benefits offered by the use of an infrastructure such as the Internet are diverse: increased computational power, higher availability in 24 × 7 periods, and reduced energy consumption [2,4,5]. One type of application that may potentially benefit from this low-cost execution platform is big-data systems.

A big-data system processes a collection of data that is difficult to process using traditional techniques and, thus, requires specific processing tools. According to [4], the main reasons of this difficulty can be characterized using three V's: The first V is for volume of processed data, the second V is for variety in the data (that is, it can come from different sources and be represented using heterogeneous formats), and the last V refers to velocity (meaning that applications work with data produced at high rates). Typically, big-data applications run analytics on clusters of machines connected to the Internet [6–11]. Each big-data analytic refers to how the data is analyzed to produce a desired output. Typical application domains include meteorology, genome processing, physical simulations, biological research, finance, and Internet search.

Nowadays, to develop these big-data applications, practitioners use different software frameworks, including Hadoop [12], Storm [13,14], and Spark [15] which emphasize different programming aspects of the big-data ecosystem. Hadoop is focused on batch processing of large data sets on commodity machines. The typical deadline for Hadoop applications ranges from minutes to weeks; often processing Petabytes [16] of data. Apache Storm works on a streaming data model and it is targeted to online applications with sub-second deadlines. Lastly, Spark offers optimized

I/O access, reducing computational times in comparison with Hadoop but with heavier computational costs than Storm [16].

Another feature of big-data applications is that they often have real-time requirements that need to be met in order to provide applications with timely response [8,17]. For instance, the Hadron collider outputs a 300 Gb/s stream that has to be filtered to 300 Mb/s for storage and later processing. Some data mining applications, like, for instance, those used for on-line credit card fraud detection, and contextual selection of web advertisements may benefit from real time techniques. In these applications having shorter response-time usually has an economic impact. This is also the case of high frequency trading systems [3,18]. Another simple example of big-data analytic that faces temporal restrictions is the trending topic detection algorithm used in Twitter to show a dynamic, up-to-date, list with the most popular hashtags. The Twitter trending topic detection application is the case of a big-data scenario characterized by the "velocity", as tweets are small pieces of data (140 characters), but are produced at a high rate. In all these cases the response has to meet application deadlines. Furthermore, in many cases big-data applications requirements are complex and may demand the coexistence of several quality-of-service restrictions on a single big-data application.

The common denominator in all these applications is that they may benefit from techniques included in real-time systems to increase the predictability of the infrastructure and also reduce the number of required resources. These types of techniques are well known [19] and may be integrated into the computational infrastructures to have fine-grained control on the number of machines used by an application and in determining an execution timeline for applications with different types of quality.

Most popular frameworks like Hadoop and Storm were designed for the general purpose domain and are inefficient for real-time application development. They provide simple models with minimum parameters that enable simple forms of parallel computation, relying on the map/reduce and stream programming models. They do not explicitly assign different parts of the application to different physical nodes, which is typically required in real-time applications. They also lack the notion of scheduling parameters enforced in different computational nodes. In their current forms, both technologies are inefficient when worst-case analysis is used to determine the worst-case computation times. Fortunately, both technologies may increase their predictability by integrating predictable computational models within their cores. But although some steps have been taken in increasing predictability in big-data infrastructures [20], there are still some important pending issues in the path towards commercial implementations.

In this context, the main goal of this article is to increase the predictability of the stream processing model, in which Storm is inspired, including real-time capabilities. The approach follows the guiding principles that inspired real-time Java [21,22], keeping backward compatibility with plain Storm, allowing plain and predictable coexistence in a single machine. Thus, the real-time Storm is able to run traditional and real-time applications in the same computing cluster (see Fig. 1).

The main improvement in the predictability of Storm in this article is the characterization of the streams of Storm as real-time entities that can be scheduled using existing real-time scheduling theory. This characterization allows reasoning about the characteristics of a real-time application in terms of deadlines, and determining the number of machines required for its implementation.

The rest of the article is organized as follows. Section 2 explores other similar frameworks and their relationships with the real-time stream processing model from the perspective of general purpose and real-time systems. Section 3 defines a simple computational model for real-time stream processing, which is mapped to the Storm framework later in Section 4. Section 5 shows an application developed with this new infrastructure. Section 6 provides practical evidence on the performance one may expect from this type of infrastructure, showing how to deduce the deadlines of the applications from their real-time characterization. Lastly Section 7 highlights the main contributions and our most related ongoing work.

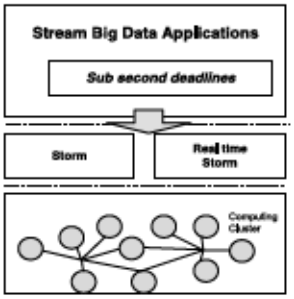

Fig. 1. A framework for real-time big-data stream processing.

2. Related work

2.1. Stream processing technology

Processing high-volume streams with low-latency has traditionally required the development of ad-hoc solutions [23]. Usually, these solutions are expensive to implement, difficult to maintain and are tailored to particular application scenarios, which limit their reusability [24]. To address these limitations and support the development of stream processing applications, several proposals of Stream Processing Engines (like Aurora [25], STREAM [26], Borealis [27], IBM's Stream Processing Core [28], and SEEP [29]) appeared in the state-of-the-art.

However, the emergence of new application scenarios, like high frequency trading, social network content analysis, sensor-based monitoring and control applications, and other low-latency big-data applications, has greatly increased the demands on this kind of stream processing platforms. As indicated in [30–32], there is a demand of general-purpose, highly scalable stream computing solutions that can quickly process vast amounts of data.

Taking this into account, it is not surprising to find in the recent state-of-the art several proposals for low-latency big-data stream processing systems, like S4 (Simple Scalable Streaming System) [13], Storm [33], or Spark Streaming [34,35], some of them backed by important Internet companies like Yahoo (S4) or Twitter (Storm). These systems are designed as general-purpose platforms that can be run on clusters of commodity hardware. Using specific programming interfaces, developers can implement scalable stream processing applications on top of them, taking advantage of the functionalities provided by the platform: information distribution, cluster management, or fault-tolerance.

All these major approaches have been designed with the general purpose performance in mind and do not provide facilities for real-time performance, like increased architecture awareness, low-level access facilities, and deadline characterization. However, most of them may be easily extended via new programming interfaces developed to increase their scalability and fault tolerance. For instance, Storm processor includes the pluggable scheduler concept that may be extended to include different scheduling policies [36] increasing adaptability. Similar architectural benefits have been exploited in this article to pin the execution graphs of the streams to specific machines of a cluster.

2.2. Real-time support for stream processing

Big-data infrastructures like Hadoop and Storm lack efficient implementations to run multiple concurrent real-time applications with different quality-of-service requirements, because they

were developed for general purpose applications [17]. This problem is general and impacts in a number of aspects ranging from high level development models to the low-level programming infrastructures. To address this problem, some authors have explored the computational model of Hadoop and proposed scheduling models for map-reduce applications that run in clusters [20,37,38,14]. Most of them advocate for the inclusion of rate-based and deadline-based scheduling into general computing clusters. This way, these authors have addressed some of the limitations arising from using a general purpose infrastructure for developing of real-time systems because general purpose scheduling policies are not optimal for real-time.

However, the map-reduce programming model is not the main paradigm used in stream processing infrastructures. For instance, the architecture of S4 is inspired by the actor's model. In S4, computation is performed by a set of Processing Elements (PEs), which interact only through messages (events). In the case of Storm, applications are structured in topologies. These topologies are direct acyclic graphs that combine two different types of processing nodes: spouts (stream sources) and bolts (which carry out single-step stream transformations). Spark Streaming is based on the concept of D-Streams (Discretized Streams) [8], a processing model that consists of dividing the total computation to be carried out into a series of stateless, deterministic micro-batch computations on small time intervals. These batches are run in Spark in an in-memory cluster computing framework. The rest of this work is focused in Storm, one of the most popular solutions for sub-second performance, which currently has not integrated real-time performance in its core.

3. Real-time stream model

The proposed model is partially inspired by the transactional model designed for distributed real-time Java [39] that has been adapted to the distributed stream model of Storm [13], as shown later in Section 4. It is also compliant with the definition of distributed stream included in [40].

According to [40] a stream is "*a continuous flux sequence of data or items*" that arrives to a logical node and typically produces an output. This definition resembles the characterization of some real-time applications with input data that produce an output within a maximum deadline [41]. This type of analogy suggests that the models used in real-time computing may be merged with the model of stream processors to produce a more predictable computational model.

In the real-time context, a real-time stream is defined as a continuous sequence of data or items whose processing has some real-time requirements like a deadline from the input to the output.

3.1. Stream model

Let A be an application and let us assume that an application is composed of a set of parallel streams (S_i):

$$A \stackrel{def}{=} (S_1, \ldots, S_n). \tag{1}$$

With each stream (S_i) characterized by its period (T_i), deadline (D_i), and a direct acyclic execution graph (DAG_i) that models the computations that have to be carried out in each activation (in charge of processing data from the flux) of the stream:

$$S_i \stackrel{def}{=} (T_i, D_i, DAG_i). \tag{2}$$

Following the model used for most real-time systems, let us assume that three different activation patterns may be defined for a stream (Fig. 2), namely periodic, sporadic, and aperiodic.

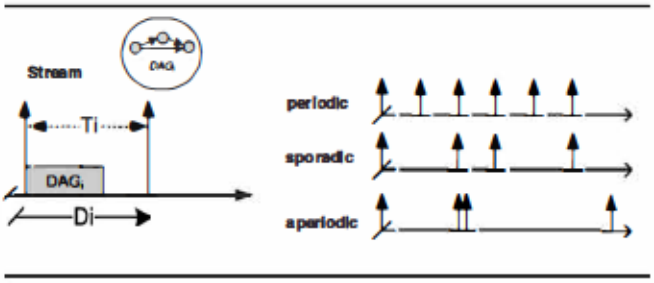

Fig. 2. Activation patterns in a real-time stream.

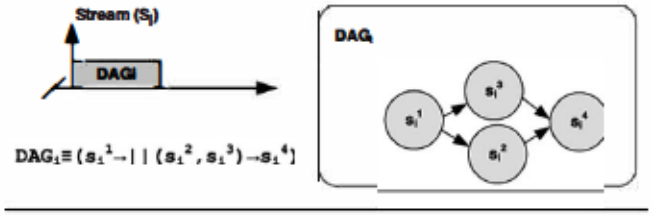

Fig. 3. Sequential and parallel stages in a real-time stream.

Periodic release pattern. This type of pattern refers to different equidistant periodic activations. In such a way that the time among two consecutive activations in the stream (t_i^{res}, t_i^{res+1}) has the following activation pattern:

$$T_i = (t_i^{res+1} - t_i^{res}) \quad \text{for all } res \text{ in } [0, +inf). \tag{3}$$

Sporadic pattern. This type of pattern is characterized by a minimum inter-arrival time $T^{\min}$ (minimum time between two successive activations):

$$T_i^{\min} \leq (t_i^{res+1} - t_i^{res}) \quad \text{for all } res \text{ in } [0, +inf). \tag{4}$$

Aperiodic pattern: In this case, there is not a minimum inter-arrival time for the different activations of a stream, i.e.:

$$\nexists\, T_i^{\min} \neq 0 | T_i^{\min} \leq (t_i^{res+1} - t_i^{res}) \quad \text{for all } res \text{ in } [0, +inf). \tag{5}$$

From the point of view of the real-time systems scheduling theory [41] periodic and sporadic patterns are simpler than aperiodic patterns to analyze.

In the real-time stream model each stream defines a global deadline (D_i). It may be equal ($T_i = D_i$) less or equal to ($T_i \geq D_i$) or larger ($T_i \leq D_i$) than the period. Different techniques of the state-of-the-art are mapped [41] to each different case to compute application response times. $T_i = D_i$ is from the point of view of the different scheduling models the most beneficial situation with efficient online admission control tests [41].

The last part of the model of the stream requires dealing with time taken for the execution of the acyclic graph and its structure. We assumed this time is bounded by C_i^{DAG}. This maximum cost is a typical constraint in real-time sporadic and periodic invocation patterns. The direct acyclic graph is described as a distributed application that consists of a set of sequential ($\rightarrow$) and parallel ($||$) stages (s_i^j). In addition, each stage (s_i^j) of the DAG_i has a maximum computational cost denoted by C_i^j.

Fig. 3 shows details of the execution graph for a simple stream that consists of four different stages ($stages(i) = 4$). It also shows how these four stages are combined to produce an end-to-end execution model. The communication messages between the first and the second and the third stages are point-to-point, multicast, or any application defined policy. In case of Storm the application may decide among different policies for the communication.

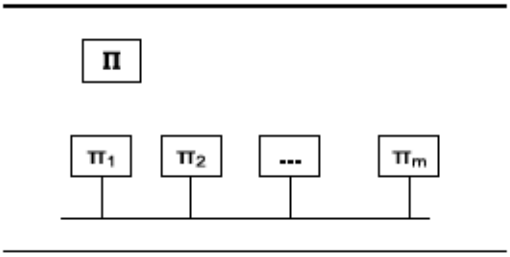

Fig. 4. Simple computational model.

3.2. Computational infrastructure

The model also includes the computational infrastructure. As shown in Fig. 4, the model defines a cluster (Π) as a set of m identical and interconnected computational nodes:

$$\Pi \stackrel{\text{def}}{=} (\pi_1, \ldots, \pi_m). \tag{6}$$

Each node has a direct connection with the other nodes of the network. In addition each node has a priority driven preemptive scheduler that runs different stages of the streams.

3.3. End-to-end response time computations

To be able to perform feasibility analysis [42] as defined by the real-time scheduling theory, all different sequential and parallel stages of all streams (S_i) have to be assigned to a physical machine from the cluster ($\pi_i^{1..stages(i)}$ in Π). In addition, each stage has to define a relationship with the underlying infrastructure reflected in a priority (P_i^j) for all stages of all streams (S_i) in the application (A).

$$S_i = \begin{pmatrix} \left[T_i^1, \ldots, T_i^{stages(i)}\right], \\ D_i, \\ \left[C_i^1, \ldots, C_i^{stages(i)}\right], \\ \left[P_i^1, \ldots, P_i^{stages(i)}\right], \\ \left[\pi_i^1, \ldots, \pi_i^{stages(i)}\right] \end{pmatrix}. \tag{7}$$

Enforcing the activation of all different stages on a node that keeps the periodic activation pattern of the stream [42], one may calculate local worst-case response times ($wcrt_i^j$) in each node using state-of- the art algorithms like response-time analysis (RTA) [42]. Once calculated the worst-case response-time ($wcrt_i^j$) of each stage in isolation, two different cases have to be considered recursively to calculate the end-to-end cost of the execution graph.

First, given a set of *seq(i)* sequential stages their final contribution to the total worst-case response time is calculated by adding the partial contributions given by each element:

$$wcrt_i^{seq(i)} = \sum_{j=1}^{seq(i)} (wcrt_i^j). \tag{8}$$

Second, given a set of *par(i)* parallel stages, their final contribution to the total worst-case response time is calculated as the maximum of all partial worst cases ($wcrt_i^j$) for each element in the set, that is:

$$wcrt_i^{par(i)} = \max\left(wcrt_i^j\right) \quad \text{with } j \text{ in } 1..par(i). \tag{9}$$

Another set of results connects the maximum number of computational machines required to implement a distributed system [43,44] and the utilization of each different task of the system. Among them, one of these sufficient but not necessary equations connects the utilization of the distributed system (U_{app}) with the minimum number of nodes (m) required from the cluster as follows [30]:

$$U_{app} = \sum_{\forall i,j} \left(\frac{C_i^j}{T_i^j}\right) < m \cdot \left(U_{\max} - \max\left(\frac{C_i^j}{T_i^j}\right)\right). \tag{10}$$

In this inequation $U_{\max}$ refers to the maximum utilization given by the scheduler and ranges from 0 to 1. Assuming a rate monotonic system with harmonic tasks and periods equal to deadlines ($T_i^j = D_i^j$) this utilization is 1.0. The last term of the inequation (i.e., $\left(\frac{C_i^j}{T_i^j}\right)$) refers to the maximum wasted utilization due to the fragmentation in the bin-packing algorithm that assigns streams to the nodes of the cluster. Our real-time characterization for streams may use the same computational model because it guarantees periodic activations in all segments of all streams.

The model also assumes that there is a mechanism that is able to guarantee network delays in communications that can be decoupled from the end-to-end costs.

4. Architecture

The previous section has set foundations for a real-time stream model, where each stream is represented by a direct acyclic graph. This general model does not observe any specific technology. This is the goal of this section that analyzes how to extend Storm to make it compatible with the previous computational model.

4.1. Increasing the predictability of Storm

As a layered software stack, Storm may benefit from different optimizations in all its levels (Fig. 5):

- Operating System (OS). At this level a predictable version for Storm would benefit from having a real-time kernel in charge of enforcing the policies typically used in real-time applications, including preemptive scheduling, and priority inheritance protocols.
- Virtual Machine (VM). In addition, the use of virtual machines offers an interesting opportunity. Java's virtual machine model is useful to bridge the gap between the real-time operating system and Storm. In addition, some modern virtual machines support specifications such as the Real-time Specification for Java (RTSJ) [22] to offer enhanced predictability.
- The Storm framework. The computational infrastructure of Storm also includes sources of indeterminism. The current model of Storm does not support the definition or the enforcement of real-time characteristic for different streams. For instance, it does not provide an efficient control to choose in which of the different nodes of the cluster, each application is going be run.

4.2. Integration levels

Applications may identify three different integration levels (Fig. 6):

- Level 0: Access to real-time facilities given by the operating system and underlying virtual machine. This includes mechanism to control the priority given to the execution of an application and their physical location into a cluster of machines.
- Level 1: New interfaces for controlling resource allocation from Storm. At this level, the resources provided by the first level are accessible from the application via an API. The most simplistic API would consist of predictable versions of the two main building blocks of Storm, namely `Spout` and `Bolt`. It should include mechanisms to describe the different real-time characteristics of the streams.
- Level 2: Enhanced services. At this level three different types of facilities would be beneficial for Storm:

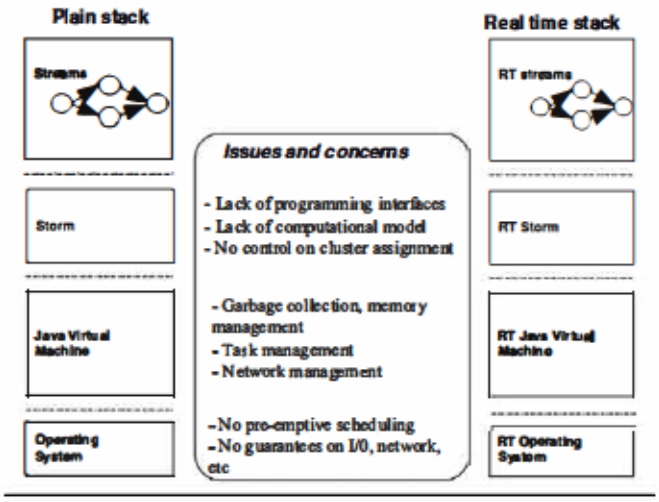

Fig. 5. Transforming the plain software stack into a real-time equivalent.

The first is a scheduler (`Scheduler`) in charge of selecting in which machine of the cluster each stage of an application is going to run. The internal logic of the scheduler is compatible with the algorithms described in Section 3.

The second mechanism is the `Elasticity` facility. A default elasticity model provides a static model in which the number of machines defined does not change. Other models may define minimum and maximum bounds.

The third module (`FaultTolerance`) is in charge of detecting and recovering the application from system failures. Different policies may coexist to express different strategies for detecting a failure in a system of streams and specifying its corresponding recovery mechanism.

4.3. API for real-time Storm

In essence a Storm application has two main classes which are `Spout` and `Bolt` to build a direct acyclic graph with parallel and sequential steps. Typically, spouts are the sources of streamed events sent to other connected bolts. From the infrastructure point of view, they are active objects that are always running. Bolts represent event-driven objects invoked only when different events arrive at the node. Both, the spout and bolt entities may send messages, which are packaged as tuples in a configuration structure called topology.

Mimicking the same structure, the real-time version of Storm incorporates a predictability model shared by the bolts and spouts. This basic model consists of a single interface called `RealtimeEvent` that contains the information required by bolts and spouts. The basic information consists of a priority, a cost, a minimum inter-arrival time valid for periodic and sporadic activations, and an error handler mechanism, that may be configured via *getter* and *setter* operators (Fig. 7). The new classes are `RealtimeBolt` and `RealtimeSpout`. Both classes are under the `es.uc3m.it.rtstorm` package hierarchy.

The lifecycle of a real-time bolt and spout replicates the behavior of their plain counterparts. Both, the plain and real-time spouts are invoked by the infrastructure thread in a user-space via a `nextTuple()` method that generates the next tuple to be processed. In the case of the plain and real-time bolts, there is an `execute(Tuple, BasicOutputCollector)` method which is invoked from an infrastructure thread whenever a new tuple for the object arrives.

The main difference is the time instants and priorities used to invoke the different event processors. In the case of the real-time

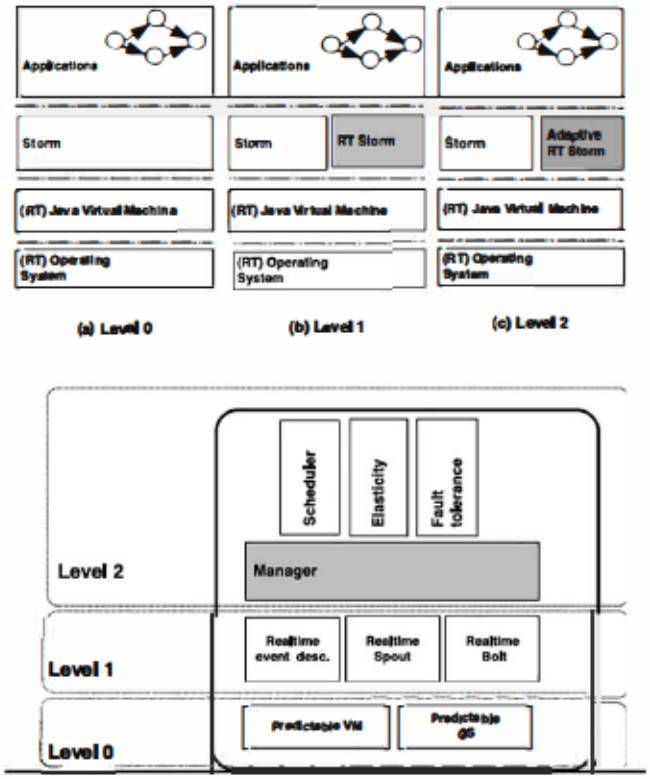

Fig. 6. Real-time services and integration levels.

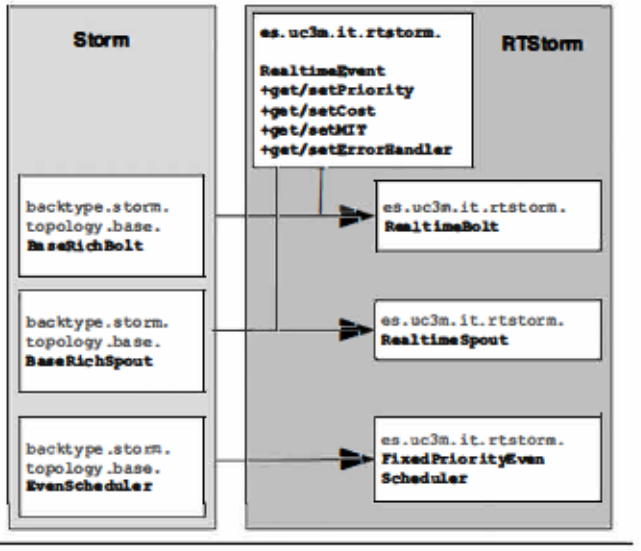

Fig. 7. Relationship between the traditional classes in Storm and its real-time homologous.

spouts, they are invoked in a periodic fashion with a period and priority preconfigured. In the case of a real-time bolt, they are invoked when there is a new event for the bolt ensuring a minimum inter-arrival pattern. Fig. 8 shows the execution timelines for plain and real-time bolts and spouts. One may see how the underlying real-time framework controls the activation of spouts and bolts to ensure proper execution.

The last class of the API refers to the management of the assignment of the different stages of a stream. Storm has a class (`EvenScheduler`) in charge of performing global scheduling ac-

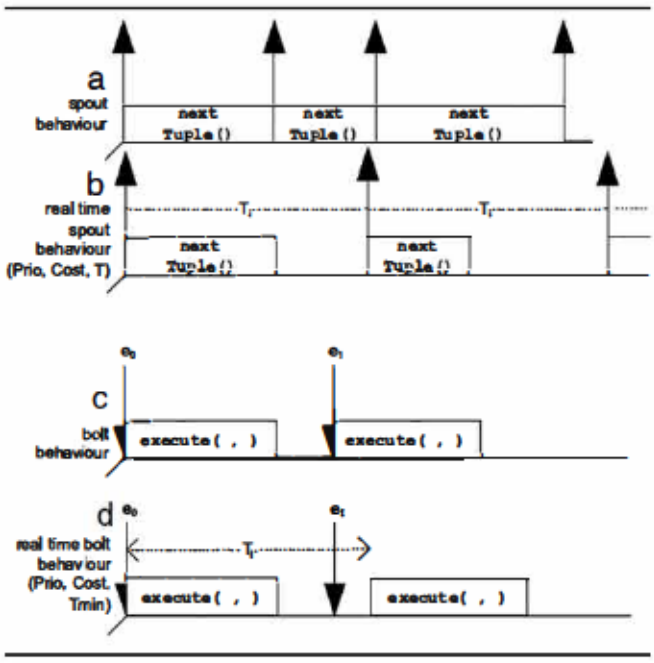

Fig. 8. Timeline execution for bolts, spouts, and their real-time homologous.

tivities that may be used to perform the dynamic adaptability required by the proposed architecture. This template uses the `FixedPriorityEvenScheduler` interface to characterize different scheduling policies.

5. Illustrative example

To show the proposed computational model, a simple application was designed. It is based on a producer–consumer stream application, with one entity running as a producer and two consumers (Fig. 9). The producer outputs tuples which are taken by one of the two consumers in parallel; each stage operating in different data. Both the two consumers and producer are hosted on a Storm cluster. The application also defines a periodic activation pattern for the events of 100 ms at the producer and maximum execution times of 10 ms for producer and the consumers. Each different consumer runs in a different node.

5.1. Computational model

By using the computational model one may define the application as a simple producer–consumer stream:

$$A_{prod-2cons} = (S_{prod-2cons}). \tag{11}$$

The application runs in a cluster with three computational resources, namely *worker_1*, *worker_2 and worker_3*:

$$\Pi_{prod-2cons} = (\pi_{worker\_1}, \pi_{worker\_2}, \pi_{worker\_3}). \tag{12}$$

We may also characterize the direct graph that defines the unique stream as follows:

$$(s^{prod}_{prod-2cons} \rightarrow \| \ (s^{cons1}_{prod-2cons}, s^{cons2}_{prod-2cons})). \tag{13}$$

To define the real-time system, we only need to assign a priority to all stages of the stream: $P_{prod} = P^{max} = P_{cons}$. We also need to assign a fixed machine to each stage of the stream. Combining all these information, the resulting system is characterized as follows:

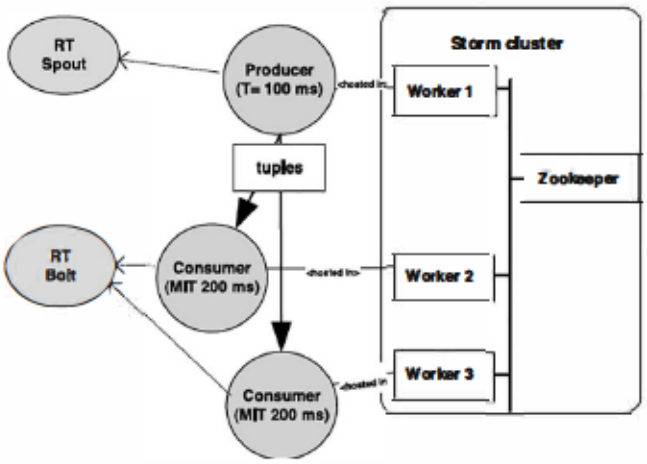

Fig. 9. Single-producer dual-consumer stream application with Storm.

$$S_{prod-2cons} = \begin{pmatrix} [T_{prod} = 100\,\text{ms},\ MIT_{cons1} = 200\,\text{ms},\ MIT_{cons2} = 200\,\text{ms}] \\ [D_{prod-2cons} = 100\,\text{ms}] \\ [C_{prod} = 10\,\text{ms},\ C_{cons1} = 10\,\text{ms},\ C_{cons1} = 10\,\text{ms}] \\ [P_{prod} = P^{max},\ P_{cons} = P^{max},\ P_{cons} = P^{max}] \\ [\pi_{worker_1}, \pi_{worker_2}, \pi_{worker_3}] \end{pmatrix}. \tag{14}$$

Assuming maximum costs and in absence of other applications in the cluster, the output is generated in less than 20 ms. This end-to-end worst-case response time is calculated as follows: 10 ms *for the producer* + max(10 ms, 10 ms) *for the parallel consumers*, taking into account the computational rules defined in Eqs. (8) and (9) and ignoring the communication overheads.

5.2. The application

The listings included Figs. 10, 11, and 12 show the application from the perspective of the programming model. Fig. 10 includes the code in charge of generating new tuples, Fig. 11 the code of the event consumer in charge of processing the tuple. Lastly, Fig. 12 shows how the topology is built and how the new information is attached to this entity during the configuration of the topology.

The application starts by running the producer in charge of generating tuples for the consumer (Fig. 10). To exhibit real-time performance the application inherits from the `RealtimeSpout` class (Fig. 10: line 01). The rest of the behavior of the spout is maintained; the infrastructure initializes the system by invoking `open` and finishes it with `close` methods of class. In addition, failures are notified in `fail` and `violation` methods, while successful notifications for the acknowledgments are sent back to the source by using the `ack` callback method.

From the producer perspective, the platform invokes the `nextTuple` method each time it needs a new tuple. The tuple is set with the `emit` method and sends the information forward (Fig. 10: line 09) to the consumer.

The consumer is implemented by extending the `Realtime Bolt` class (Fig. 11: line 01). The most relevant method is `execute`, in charge of processing the tuple received from an external entity. In the example the processing (Fig. 11: lines 05–09) prints out the sentence sent from the producer. The example includes mechanisms to configure and declare the output types (Fig. 11: lines 02–04). In addition, there is a `violation` method invoked from the infrastructure in a case of failure in the real-time runtime.

In Storm, the links between the producer and the consumer are set in a special class in charge of linking sources of messages and

```
01: public class RTProducer extends RealtimeSpout{
02:  public void ack(Object msgId)
03:   { System.out.println("OK:"+msgId);}
04:  public void close() {}
05:  public void fail(Object msgId)
06:  { System.out.println("FAIL:"+msgId);}
07:  private int counter=0;
08:  public void nextTuple()
09:  { this.collector.emit(new Values("Value"),"Counter "+ (++counter));}
10:  public void open(Map conf, TopologyContext context,
11:                        SpoutOutputCollector collector)
12:             { this.collector=collector; }
13:  public void declareOutputFields(OutputFieldsDeclarer declarer)
14:            { declarer.declare(new Fields("line")); }
15:  public void violation(Exception e)
16:   {  System.out.println("Runtime exception"+e);  }
17: }
```

Fig. 10. Producer in charge of generating the tuple to the consumer.

```
01: public class RTConsumer extends RealtimeBolt{
02:  public void prepare(Map stormConf, TopologyContext context){}
03:  public void declareOutputFields(OutputFieldsDeclarer declarer){}
04:  //Execute method is under realtime constraints
05:  public void execute(Tuple input, BasicOutputCollector collector)
06:  {
07:   String sentence = input.getString(0);
08:   System.out.println("[output]:"+sentence);
09:   }
10:  //Violation is invoked when the application misses deadlines
11:  // or overuns
12:  public void violation(Exception e)
13:  {
14:    System.out.println("Runtime exception"+e);
15:  }
16: }
```

Fig. 11. Consumer in charge of processing the tuple.

destinations. In addition to this type of support, the proposed application (Fig. 12) requires defining the runtime priorities, a minimum inter-arrival among events, and the maximum time of CPU required from the runtime.

The topology starts by allocating the objects for the producer and two consumers in lines 03–05. On each one of them, the application defines priorities (lines 06–08), inter-arrivals (lines 09–11) and maximum costs (lines 12–14) for each object of the application. Then, by using the topology builder, it allocates the producer, the two consumers, and connects them via a `shuffleGrouping` method. In addition, it creates a configuration class in charge of defining specific properties and other configuration parameters of the platform (lines 22–23). Lastly, the configuration information includes the name of the node in which each different object should be running. In this particular application, the producer runs on a node called "`Remote1`", one consumer in "`Remote2`", and another in "`Remote3`".

The last piece of code refers to assignment to the physical cluster of machines. Storm includes the concept of *pluggable schedulers* as a means to carry out this mapping (Fig. 13). There is a `schedule` method that may be overridden with application specific algorithms. In our particular example, it assigns each object to node with a machine of the cluster (see Fig. 13), which is compatible with the configuration parameters given in Fig. 12: lines 22–27.

To assign stages to different clusters, the application may use a bin packing algorithm that assigns different stages of the stream to a particular machine of the cluster. These algorithms are application dependent and are not optimal. Based on a previous algorithm designed in the context of distributed real-time Java applications [45] it is suggested a bin packing algorithm (Alg 1.) that assigns stages to a set of computational nodes. This algorithm calculates the occupation of each node as the sum of all nodes to check if the system is feasible or not and it is compatible with the utilization bound included in Eq. (10).

```
      //Nodes with utilization:  nd[]
      //stages that demand computationation units:  sg[]
01:   assign (nd[], sg[])
02:     for_all segi in seg
03:                 for_ all ndi in nd
04:                   if(nd.inuse()+sgi.required()<nd.limit())
05:                     nd.add(segi)
06:                     jmp cont
07:                   endif
08:                 endfor
09:       error:   error("not feasible");
10:       cont:
11:     endfor
```

Alg. 1. Default assignment algorithm

6. Empirical evaluation

The implementation of Storm has been modified to incorporate the real-time spouts and bolts. Our current testing implementation is rt.0.1-9.2. In addition, a testing infrastructure on a local area network (see Table 1 and Fig. 14) was developed. Following the same strategy of other distributed real-time Java infrastructures (e.g. [22]) the evaluation opts for a 100 Mbps Ethernet network to interconnect their nodes. In addition, each machine has several cores that offer local clustering facilities.

```
01: public class TopologyMain {
02:              public static void main(String[] args){
03:         RTProducer rtprod=new RTProducer();
04:         RTConsumer rtcons=new RTConsumer();
05:         RTConsumer rtcons2=new RTConsumer();
06:         rtprod.setPriority(10);
07:         rtcons.setPriority(10);
08:         rtcons2.setPriority(10);
09:         rtprod.setMit(100,0); //100 ms
10:         rtcons.setMit(200,0); //200 ms
11:         rtcons2.setMit(200,0);//200 ms
12:         rtprod.setCost(10,0); // Maximum consumed CPU: 10 ms
13:         rtcons.setCost(10,0); // Maximum consumed CPU: 10 ms
14:         rtcons2.setCost(10,0); //Maximum consumed CPU: 10 ms
15:         TopologyBuilder builder = new TopologyBuilder();
16:         builder.setSpout("producer", rtprod);
17:         builder.setBolt("consumer1",rtcons)
18:                    .shuffleGrouping("producer");
19:         builder.setBolt("consumer2",rtcons2)
20:                    .shuffleGrouping("producer");
21:         Config conf = new Config();
22:         conf.setNumWorkers(3);
23:         conf.setMaxSpoutPending(5000);
24:         //Information for the plugable scheduler
25:         conf.put("producer","Remote1");
26:         conf.put("consumer1","Remote2");
27:         conf.put("consumer2","Remote3");
28:    }
29: }
```

Fig. 12. Topology description with real-time characterization.

```
01: public class PinnedScheduler extends FixedPriorityEvenScheduler
02:  public class PluggableScheduler implements IScheduler {
03:  @Override
04:  public void prepare(Map list) {
05:  }
06:  @Override
07:  public void schedule(Topologies topologies,
08                         Cluster cluster) {
09:  //Get all supervisors
...
10:  //Get all topologies
…
11:  //Assign topologies to nodes by using the config file
...
13: }
14:}
```

Fig. 13. Real-time scheduler in charge of mapping stages to physical machines.

Several applications have been developed with the following goals:

- To empirically evaluate the overhead introduced by the communications and serialization protocols.
- To empirically evaluate differences between local clustering (i.e., all processing stages running on a single multicore machine) and local area network (LAN) cluster (with several machines connected through IP).
- To evaluate the performance of real-time Storm applications. Our use case study is based on an analytics application that calculates trending topics. This application is a constrained version of the application running on Twitter and it is currently available in Storm. In the trending topics application, the analytic refers to the code that is in charge of analyzing the data (tweets) and its output is the list of the most relevant topics (hashtags).

The goal in all these evaluations is to model the trending topic application as a real-time stream application running on a cluster.

6.1. Computational overheads

For empirical purposes, overhead is defined as the time taken for the transmission of data from one computational node to another. This overhead is due to the data transmission delays of the network and the serialization protocols used to transfer such amount of data.

To compute this overhead on distributed stream processors, the benchmark previously described in [22] for distributed real-time Java was extended with the characteristics of stream processors. The modified benchmark includes operational frequencies from

Table 1
Type of nodes used in the evaluation.

Infrastructure	
CPU	1 GHz, dual core (2 cores)
Memory	8 GB
Network	100 Mb Switched Ethernet 100 Mb
OS	Ubuntu 14 rt-patch
Storm version	rt.0.1-9.2
Benchmark	
Messages types	Small and large (200 bytes to 800 bytes)
Messages	[1,2,4,6,7,12,13,15]
Clusters	Local cluster (single machine with 2 cores) or Networked LAN cluster (with several machines)

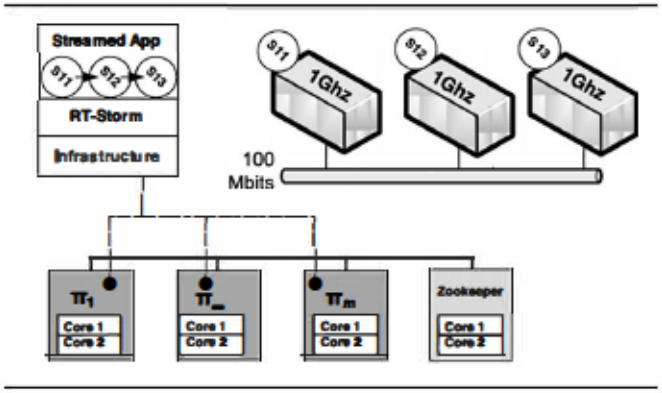

Fig. 14. Benchmark infrastructure.

1 Hz to 1 kHz for environments that emit a variable number of messages (ranging from 1 to 8). It also includes two types of application scenarios: one with small data sets that correspond to short strings, and another which is a large number of strings packed into a tuple. To better characterize the behavior of the network, all experiments have been executed in a local cluster or in an IP networked cluster.

This type of configuration is useful to assess the performance of the distributed application in a networked environment. The results of the experiment for a local cluster are summarized in Fig. 15. Likewise, the networked counterpart results are shown in Fig. 16.

In a local cluster (Fig. 15) the overhead for the communications is moderate, i.e. less than 20%, for applications running at 100 Hz that send a moderate, i.e. 1–2 messages per activation. This overhead reduces drastically as the frequency of the application decreases; for instance with 10 Hz the overhead due to the communications represents 10% of the available time. On the other hand the overhead is high, i.e. more than 80%, in applications with 0.5 kHz activation frequencies. This overhead increases as the number of messages increases.

In general terms, a local area network (LAN) cluster increases the overhead of the applications because exchanged data have to be sent from one node to another. In our particular infrastructure, the cost in communications increased by a fixed factor that is in between five times and seven times the average cost of the communications in a local cluster. In addition, to the extra cost introduced by the serialization protocols, the system has to account that local clusters running in the same machine have mechanisms to avoid the data transmission overhead.

The results on a LAN cluster (see Fig. 16) show how the overhead increases as they are compared to their local cluster equivalent (Fig. 15). In all cases the time available for the application decreases. The lower overhead (<20%) results in a local cluster running at a 100 Hz frequency is now closer to the 10 Hz frequency, reducing the amount of effective time available for the application. Likewise, the previous high overhead ranges (>80%) also moved

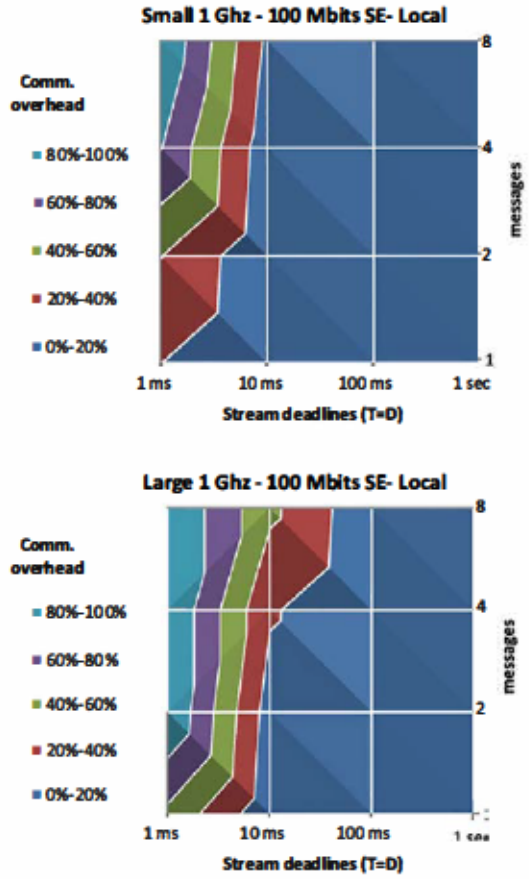

Fig. 15. Overhead performance results in the local cluster.

from the 500 Hz to the 50 Hz range, reducing the time available for the application.

The evaluation results show that local clusters offer an advance ranging from 5 to 7 times the cost of the networked cluster. Local clusters avoid the overhead in communication and serialization protocols. They also exploit the multi-core infrastructure offered by modern CPUs to provide efficient parallel computing platforms, which in the selected infrastructure consisted of two cores.

The main bottleneck in this experiment is in the 100 Mbps network, which may improve its performance with additional gigabit ethernet connections and optical fiber. In the particular case of the performance of an optical connection, it would be in between the local multi-core cluster performance and the 100 Mb networked performance.

6.2. *Trending topics case study*

The first part of the empirical section evaluated the overhead introduced by the Storm infrastructure on a distributed set-up. This overhead is crucial for determining the response time in distributed applications. Now, this information is complemented with the analysis of the results offered by an application.

The selected case study is a reduced version of the trending topic application running in Twitter, which is available for Storm in the following link [46]. The goal was to choose a simple application that could be developed with the proposed real-time stream model. The application calculates the list of the most popular hashtags in Twitter. Typically, the list is updated every two seconds [46] and potentially receives hashtags at an unbounded speed.

The goal was to illustrate the benefits of the real-time stream model scheduling framework proposed in Section 3. In particular, this section shows how they can be used to analyze the relationships among the properties of the cluster (e.g. the number of nodes)

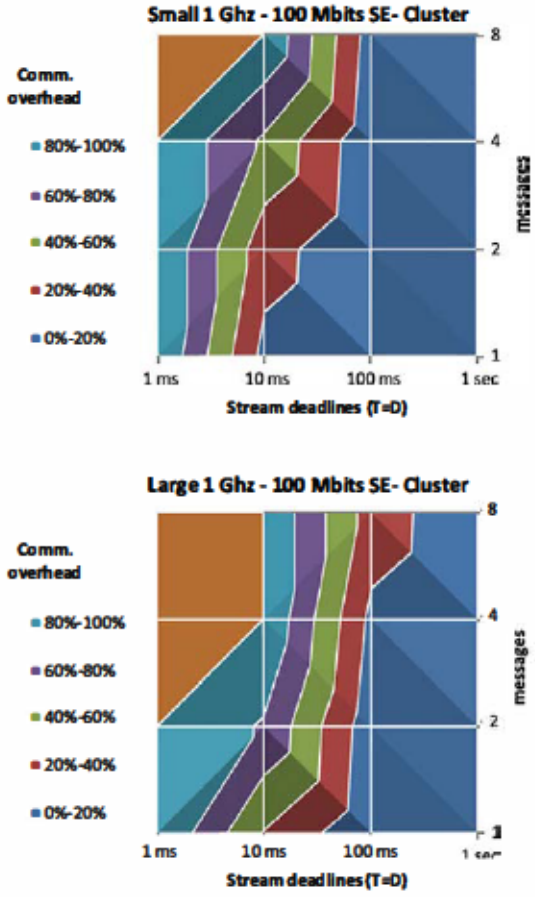

Fig. 16. Overhead performance results in a networked LAN cluster.

and the performance of the system in terms of maximum input frequency and application deadlines. The characterization of an application as a real-time stream enables the possibility of reasoning about its real-time performance.

In this particular set of experiments, the activation of each different bolt and spout has the following constraint: $T = D$, which enables the use of the constraint defined in Eq. (10) to calculate a safer bound for the number of nodes required to implement the system.

6.2.1. Application characterization

The basic case study (Fig. 17) consists of two streams. In the first stream there is a spout which extracts the hashtags from each tweet and outputs tuples, each of them containing a hashtag. Two counter bolts receive the spout tuples, and keep a sliding window counter for each hashtag. This counter indicates the number of times the hashtag has been received at the counter bolt. The tuples provided as output by the counter bolts contain a table which associates hashtags with their respective counters. These tuples are received by an aggregator bolt, which aggregates the tables coming from the different counter bolts and generates a single output table. The output of the aggregator is the input for the second stream, namely, ranker stream, which runs every second and produces an ordered list with the top-ranked hashtags.

In the evaluation of our approach we have used as data source an application that mimics the behavior of the Twitter streaming API [46]. To do so, we captured a trace of actual tweets (1 million) and stored it into a file. Later on, our application reads the file and produces a continuous (and potentially infinite) flow of data simulating Twitter's social network. This input data is then used to feed the real-time application running on top of Storm, with the goal of determining the worst-case computation times of each stage of the application.

With this information, the application may be characterized as two real-time streams. Table 2 contains the characterization of the application costs of each of the stages that compose the application described in Fig. 17. The application consists of two streams: the first is $S_{counter}$ in charge of calculating the number of times each hashtag has been mentioned in tweets. It consists of three stages: source (spout), counter and aggregator (bolts). The second stream is in charge of running the trending topic calculations, which consists of three stages: source processor, counter, and aggregator; and the second stream in charge of producing the final ranking. The ranker has a 1 Hz frequency and the counter stream has a variable frequency, which depends on the rate at which the tweets are received at the input. In this evaluation, the minimum input frequency should be 1 Hz.

With the topology for the trending topic application described in Fig. 17 and Table 2 the maximum application deadlines may be determined. In this case, a safe utilization bound is used for the two streams to ensure that the global utilization is less than 100%. The deadline for the S_{ranker} is always 1 s its period because it consists of a unique stage. In the $S_{counter}$ stream, the end-to-end can be calculated adding the partial deadlines of the three elements; each segment contributes its maximum deadline and the end-to-end deadline is three times the input period (Fig. 18).

Using the real-time scheduling model and the theory associated to the model (see Section 3) one may derive a maximum input frequency (see results in Fig. 19) which is never feasible in a single machine if the system is over 100%. The application has been deployed

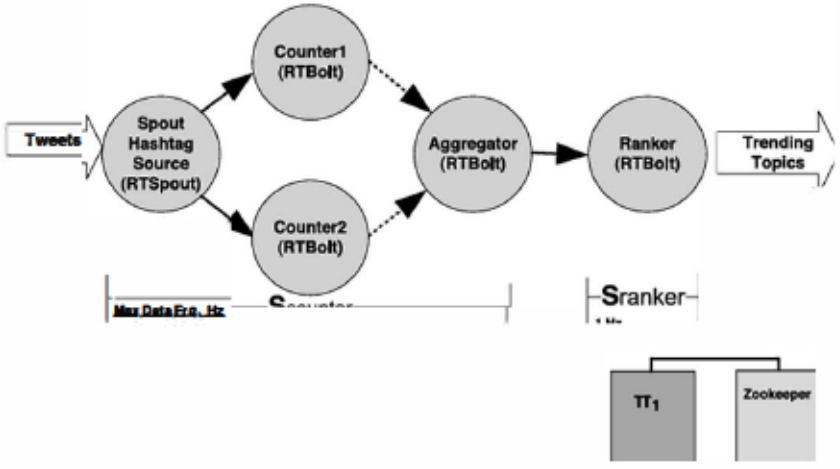

Fig. 17. Trending topics use case: basic configuration. Data entering with a variable maximum frequency (fmax). The characterization of the application is shown in Table 2.

Table 2
Real-time characterization of the trending topic use case application. ($T = D$ to be able to use the utilization bounds described in Eq. (10).)

	Source	Counter	Aggregator	Ranker
Cost (μs)	115	138	150	180
Priority	Inverse to the frequency of the tasks			
Max freq (Fmax).	Data Input freq (>1 Hz)			Data Output freq (1 Hz)
Deadline	$D_i^j = T_i^j = 1/F_i^j$			1 s ($T = D$)

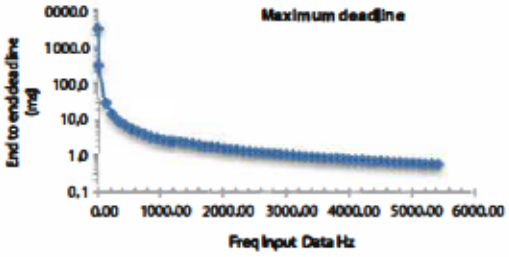

Fig. 18. Maximum deadline of $S_{counter}$ stream for Fig. 17 setup when the frequency at which the data is received changes.

in the local cluster and in the LAN to calculate the maximum input frequency for the system. The most critical section (in terms of performance) is the part in charge of processing the *hashtags* that come at high frequencies. The ranker takes less time because it runs at a very low frequency (1 Hz or less). As in the previous case the local cluster outperforms the LAN cluster; the local cluster may process data up to a maximum frequency empirically set in 1.25 kHz. This frequency reduces to 0.21 kHz in the networked environment due to the overhead of the serialization and communication protocols. According to the relationships expressed in Fig. 18, when the input frequency is 1.25 kHz the output has a maximum bounded delay of 2.4 ms, whereas when the input frequency is 0.21 Hz the maximum delay is 12 ms.

By using the facilities included in Storm, one may add the parallel units to increase the performance of the system, running topologies in parallel, and increasing the maximum frequency of operation of the trending topic application. In these scenarios the utilization of the system may be over 1.0, requiring multiple machines.

6.2.2. *Parallelizing input processing*

This experiment takes the basic trending topic detection application, depicted in Fig. 17, and modifies the layout of the $S_{counter}$ stream by proposing a distributed alternative that divides the traffic of the Internet into n different paths (see Fig. 20) that are processed by independent spouts and bolts. In this scheme, the unique element, which is still receiving information from all nodes, is the aggregator, which runs at a maximum frequency of the input data.

This change in the topology also increases the maximum deadline of the application because the frequency at which each counter receives its input is lower. In the case of a double input flux ($n = 2$), the deadline of the $S_{counter}$ is five times the input period and with four parallel inputs, it is nine times the input frequency. Fig. 21 shows this relationship for two and four parallel inputs.

This configuration increases the performance of the system, measured as the maximum input data frequency to the system, in comparison with the previous configuration (see Fig. 22). The results obtained for the utilization bound show how the maximum frequency of input data may move from 0.2 to 0.47 kHz using a cluster with 16 machines. Likewise, the previous maximum 1.25 kHz bound of the local cluster may be extended to a maximum of 2.7 kHz input if one admits multiple counters in the system. The results also show the main bottleneck of the solution, which is

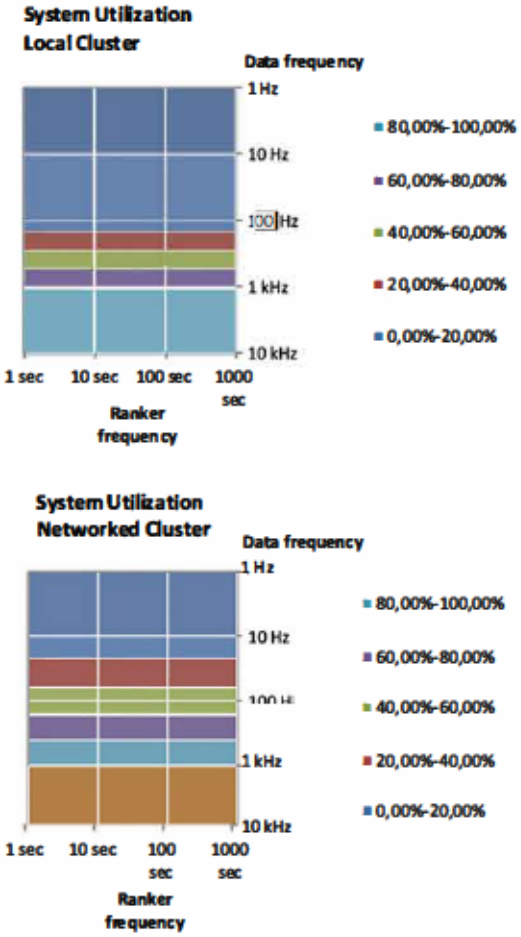

Fig. 19. Basic trending topic application utilization: local cluster vs. LAN cluster.

the aggregator that receives information at a high speed frequency from all fluxes.

Another analysis that may be carried out is the efficiency of the application. In the particular application, the efficiency of the system may be defined as the number of petitions (associated to tweets) that may be processed in a period of time divided by the number of resources (which may be cores of a local cluster or machines of a local area cluster). The higher this number is, the higher the efficiency of the application.

In our particular case, the results (Fig. 23) show how performance is a concave function. The example also shows that the efficiency of the local cluster is higher than the efficiency of the networked cluster.

6.2.3. *Parallelizing the aggregator*

The analysis of the trending topic application showed the main bottleneck of the previous configuration: a single aggregator that receives all trending topic information from all parallel flows. One common solution to this problem is to use a multi-step aggregation output phase where the aggregation of information is carried out in different stages.

Fig. 24 shows how to implement a double step aggregator in the trending topics application. This type of configuration reduces the maximum frequency of the messages that reach any of the aggregators.

This change in the topology increases the ratio among the input frequency and the deadline (Fig. 25). For the new version of the $S_{counter}$ stream two parallel inputs ($n = 2$) the system has a maximum end-to-end deadline which is seven times the input

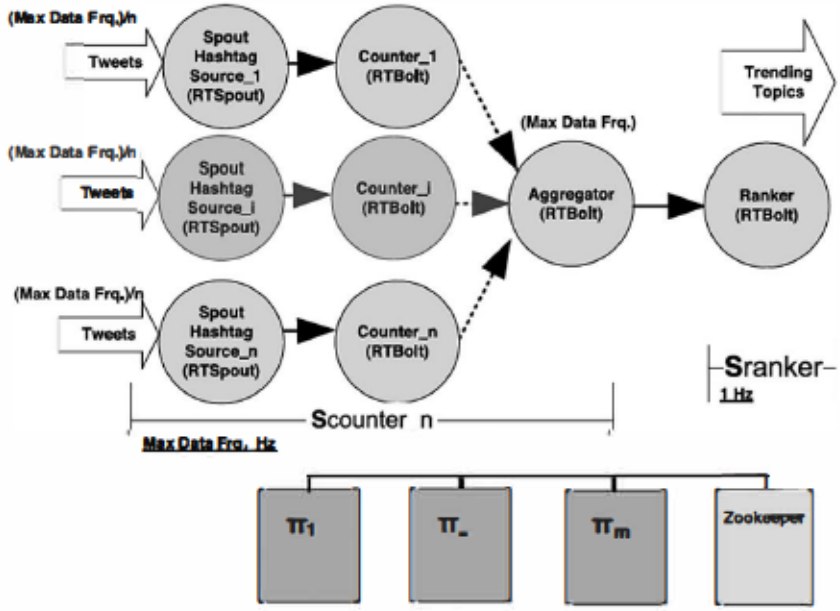

Fig. 20. Trending topic analytic model with n-parallel counters. Multiple spouts are added to offer the possibility of adding new machines to the clusters.

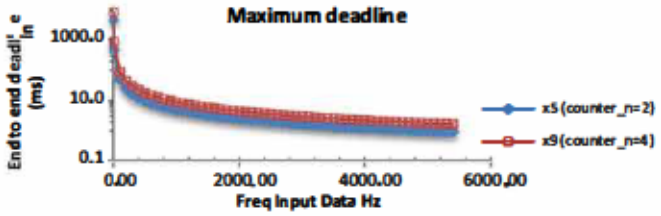

Fig. 21. Maximum deadline of $S_{counter}$ with n-parallel counters.

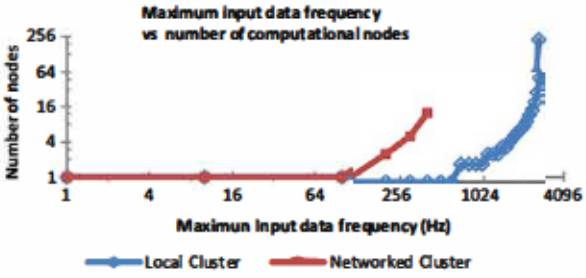

Fig. 22. Number of computational nodes vs. maximum frequency (multiple counters and a single aggregator).

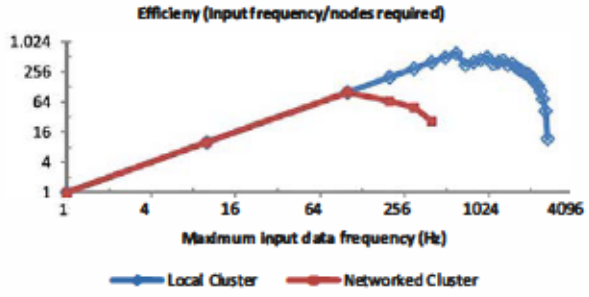

Fig. 23. Efficiency in the cluster in the trending topic application (multiple counters and single aggregator).

Table 3
Response times for all streams with plain Storm and real-time Storm. Cluster with 1 machine and 2 streams with one single segment.

Stream	P	C	T_{min}	D_{max}	Resp.
Plain storm					
Real-time	Default	0.1	0.5	0.5	**0.6**
Background		0.5	1.0	–	0.7
Real time storm					
Real-time	High	0.1	0.5	0.5	**0.1**
Background	Low	0.5	0.10	–	0.7

period. Likewise for four parallel inputs ($n = 4$), the maximum deadline of the output is twelve times the input period.

As a result of the double aggregation, the maximum operational frequency of the input data increases in local and networked clusters (see Fig. 26). In the local cluster, it increases the maximum data input frequency admissible from 2.7 to 5.4 kHz. In the networked cluster, the maximum data input frequency increases from 0.47 to 0.95 kHz.

However, this increase in the maximum frequency also involves a more reduced efficiency because the application has an additional step that consumes additional resources (see Fig. 27). In this particular scenario, the demand of new resources reduces the peak of the curve as the number of stages in the aggregation increases from one to two.

6.2.4. Real-time storm vs. plain storm

The last experiment is focused on a simple use case that illustrates the benefits that can be obtained from the use of the real-time version of Storm. To this end, let us introduce a system with two simple streams (see Fig. 28 and Table 3), one of them with real-time requirements and another heavy stream (with 0.5 utilization) but without real-time constraints. Assuming that two streams are using the same machine and the system does not include the techniques like those described for real-time Storm, then the worst-case response time for the real-time stream is 0.6 ms (because its worst-case response time has to include the heavy node computation). Using the real-time facilities and assigning a lower priority to the stream with no deadline (the heavy stream), then the higher priority stream sees the system in isolation and its response time changes to 0.1 ms.

7. Conclusions and future work

Current big-data applications can improve their predictability by integrating techniques derived from the real-time domain

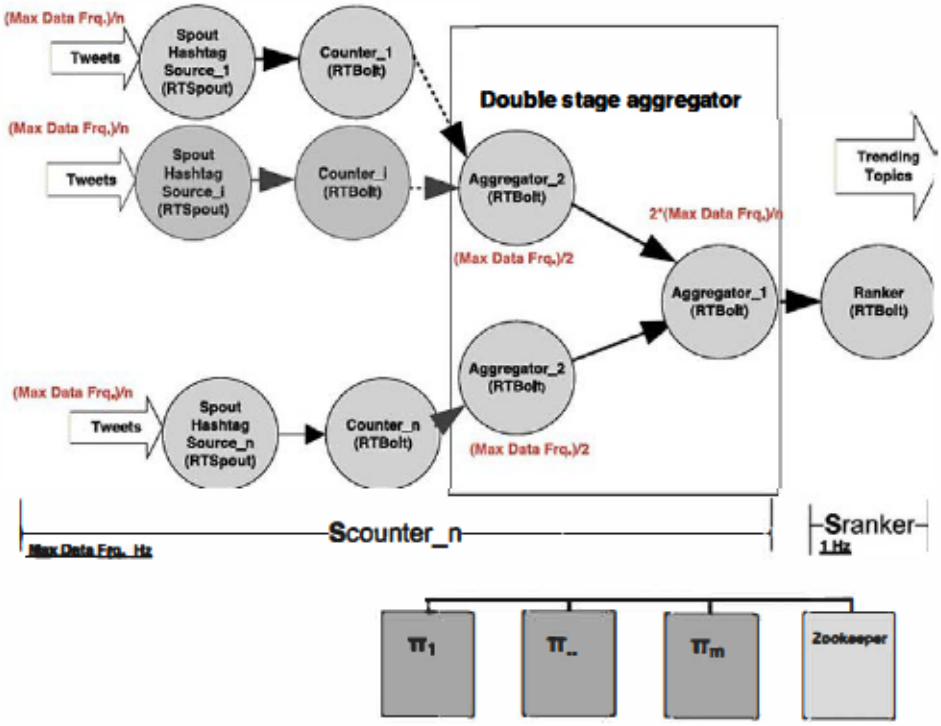

Fig. 24. Trending topic analytic application (multiple counters and a 2-step aggregation).

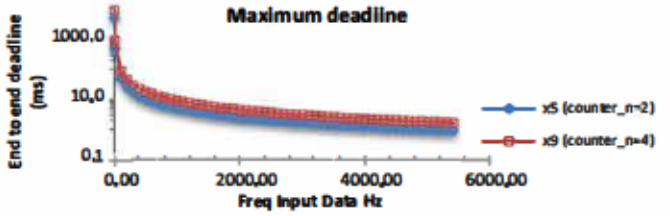

Fig. 25. Trending topic analytic application (multiple counters and 2 aggregators).

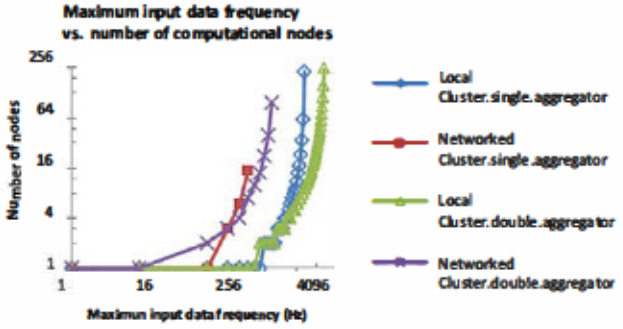

Fig. 26. Number of computational nodes vs. maximum frequency of input data (multiple counters and 2 aggregators).

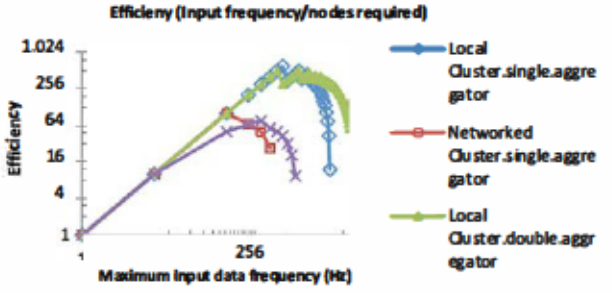

Fig. 27. Efficiency (multiple counters and 2 aggregators).

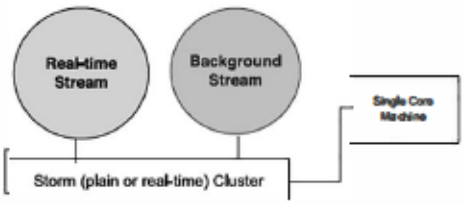

Fig. 28. Single machine cluster running a real-time and background stream in a plain and a real-time cluster.

within their infrastructures. This article has analyzed the integration of traditional scheduling techniques into a popular stream processor named Storm. The integration has addressed changes in the architecture of Storm and new APIs; it also modeled the streams as real-time entities that may be running in a cluster of machines. The identification of stream processors as distributed applications has opened the door to the use of common-off-the-shelf scheduling mechanism to guarantee end-to-end predictability, typically used in other distributed real-time infrastructures. It also provides a backwards compatible infrastructure for real-time Storm where plain and real-time streams may coexist. The empirical evaluation carried out also illustrated the performance one may expect from these infrastructures and illustrated how the scheduling theory can be used to calculate deadlines in Storm applications.

Our ongoing work is focused on expanding the model to other scenarios, including industrial applications, next generation information systems and business intelligence scenarios like those described in [47]. We also plan to address other big-data processing infrastructures, like those using optical-fiber networks in combination with the message passing interface (MPI) [48] technology and map-reduce [49–51].

Acknowledgments

This work has been partially supported by HERMES (Healthy and Efficient Routes in Massive open-data basEd Smart cities). It

has been also partially financed by Distributed Java Infrastructure for Real-Time Big Data (CAS14/00118). It has been also partially funded by eMadrid (S2013/ICE-2715) and by European Union's 7th Framework Programme under Grant Agreement FP7-IC6-318763.

References

[1] R. Buyya, et al., Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility, Future Gener. Comput. Syst. 25 (6) (2009) 599–616.

[2] M. Armbrust, et al., A view of cloud computing, Commun. ACM 53 (4) (2010) 50–58.

[3] M. García-Valls, P. Uriol-Resuela, F. Ibáñez-Vázquez, P. Basanta-Val, Low complexity reconfiguration for real-time data-intensive service-oriented applications, Future Gener. Comput. Syst. 37 (2014) 191–200.

[4] Paul Zikopoulos, Chris Eaton, Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data, first ed., McGraw-Hill Osborne Media, 2011.

[5] A. Jacobs, The pathologies of big data, Commun. ACM 52 (8) (2009) 36–44.

[6] Y. Demchenko, Z. Zhao, P. Grosso, A. Wibisono, C. de Laat, Addressing big-data challenges for scientific data infrastructure, in: CloudCom, 2012, pp. 614–617.

[7] J. Lin, R. Dmitriy, Scaling big-data mining infrastructure: the twitter experience, ACM SIGKDD Explor. Newsl. 14 (2) (2013) 6–19.

[8] K. Kambatla, G. Kollias, V. Kumar, A. Grama, Trends in big-data analytics, J. Parallel Distrib. Comput. 74 (7) (2014) http://dx.doi.org/10.1016/j.jpdc.2014.01.003.

[9] G. Blelloch, Big data on small machines, in: Big Data Analytics 2013, Cambridge, May 23–24, 2013.

[10] H.V. Jagadish, et al., Big data and its technical challenges, Commun. ACM 57 (7) (2014) 86–94.

[11] V.N. Gudivada, R. Baeza-Yates, V.V. Raghavan, Big data: Promises and problems, Computer 48 (3) (2015) 20–23.

[12] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST, pp. 1–10.

[13] Storm, Distributed and fault-tolerant real-time computation. Available (2014) on https://storm.incubator.apache.org/.

[14] G. Lodi, et al., An event-based platform for collaborative threats detection and monitoring, Inf. Syst. 39 (2014) 175–195.

[15] Spark, Lightning-fast cluster computing. Available (2014) on https://spark.apache.org.

[16] M. Rychl, P. Skoda, P. Smrz, Scheduling decisions in stream processing on heterogeneous clusters, in: Eighth International Conference on Complex, Intelligent and Software Intensive Systems, 2014, pp. 614–619.

[17] I. Gray, Y. Chan, N. Audsley, A. Wellings, Architecture-awareness for real-time big-data systems, in: Proceedings of the 21st European MPI Users' Group Meeting, pp. 151–156.

[18] T. Chordia, A. Goyal, B.N. Lehmann, G. Saar, High-frequency trading, J. Financ. Mark. (ISSN: 1386-4181) 16 (4) (2013) 637–645.

[19] C. Grier, K. Thomas, V. Paxson, M. Zhang, @ spam: the underground on 140 characters or less, in: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 27–37.

[20] L.T.X. Phan, Z. Zhang, B.T. Loo, I. Lee, Real-time MapReduce scheduling, in: Technical Report N. MS-CIS-10-32, University of Pennsylvania, 2010.

[21] M.T. Higuera-Toledano, A.J. Wellings, Distributed, Embedded and Real-time Java Systems, Springer, 2012, p. 378, X.

[22] P. Basanta-Val, M. Garcia-Valls, A distributed real-time Java-centric architecture for industrial systems, IEEE Trans. Ind. Inf. 10 (1) (2014) 27–34.

[23] T. Aniello, et al. Cloud-based data stream processing, in: ACM International Conference on Distributed Event-Based Systems, DEBS'14, pp. 238–245.

[24] M. Stonebraker, U. Çetintemel, S. Zdonik, The 8 requirements of real-time stream processing, SIGMOD Rec. 34 (4) (2005) 42–47.

[25] D.J. Abadi, et al., Aurora: a new model and architecture for data stream management, VLDB J. 12 (2) (2003) 120–139.

[26] A. Arasu, et al. STREAM: the stanford stream data manager, in: ACM SIGMOD International Conference on Management of Data, pp. 665–665.

[27] D.J. Abadi, et al. The design of the borealis stream processing engine, in: Conference on Innovative Data Systems Research, CIDR 2005, pp. 277–289.

[28] L. Amini, et al. SPC: a distributed, scalable platform for data mining, in: Data Mining Standards, Services, and Platforms, DMSSP'06, pp. 27–37.

[29] M. Migliavacca, et al. SEEP: scalable and elastic event processing, in: Middleware'10, Article 4, p. 2.

[30] L. Neumeyer, et al. S4: Distributed stream computing platform, in: IEEE International Conference on Data Mining, pp. 170–177.

[31] C. Yixin, L. Tu, Density-based clustering for real-time stream data, in: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

[32] J.A. Silva, et al., Data stream clustering: A survey, ACM Comput. Surv. 46 (1) (2013) 31. http://dx.doi.org/10.1145/2522968.2522981, Article 13.

[33] M. Zaharia, et al. Discretized streams: fault-tolerant streaming computation at scale, in: ACM Symposium on Operating Systems Principles, SOSP'13, pp. 423–438.

[34] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, USA, 1986.

[35] M. Zaharia, et al. Spark: cluster computing with working sets, in: 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pp. 10–10.

[36] L. Aniello, et al. Adaptive online scheduling in Storm, in: ACM International Conference on Distributed Event-Based Systems, DEBS'13, pp. 207–218.

[37] X. Dong, Y. Wang, H. Liao, Scheduling mixed real-time and non-real-time applications in MapReduce environment, in: Parallel and Distributed Systems, ICPADS, 2011.

[38] F. Teng, et al., A novel real-time scheduling algorithm and performance analysis of a MapReduce based cloud, J. Supercomput. 69 (2) (2014) 739–765.

[39] M. Garcia-Valls, P. Basanta-Val, Comparative analysis of two different middleware approaches for reconfiguration of distributed real-time systems, J. Syst. Archit. 60 (2) (2014) 221–233.

[40] L. Golab, M. Tamer Özsu, Issues in data stream management, SIGMOD Rec. 32, 2, 5–14. http://doi.acm.org/10.1145/776985.776986.

[41] P. Basanta Val, M. Garcia Valls, A simple distributed garbage collector for distributed real-time Java, J. Supercomput. (2014) in press. http://dx.doi.org/10.1007/s11227-014-1259-x.

[42] L. Sha, et al., Real-time scheduling theory: A historical perspective, Real-Time Syst. 28 (2–3) (2004) 101–155.

[43] J.M. López, J.L. Díaz, D.F. García, Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling, IEEE Trans. Parallel Distrib. Syst. (2004) 642–653.

[44] R.I. Davis, A. Burns, A survey of hard real-time scheduling for multiprocessor systems, ACM Comput. Surv. (CSUR) 43 (4) (2011) 35.

[45] P. Basanta-Val, M. García-Valls, Towards a reconfiguration service for distributed real-time Java, in: REACTION 2012 Workshops, Puerto Rico, December, 4, 2012.

[46] M. Noll, Implementing real-time trending topics in Storm, Available in 2014 on http://www.michael-noll.com/blog/2013/01/18/implementing-real-time-trending-topics-in-storm/.

[47] V. Chang, The business intelligence as a service in the cloud, Future Gener. Comput. Syst., 37, pp. 512–534.

[48] N. Irizarry, Mixing C and Java™ for high performance computing, MTR130458, MITRE Technical Report, 2013.

[49] J.C.S. dos Anjos, I. Carrera Izurieta, W. Kolberg, A.L. Tibola, L. Bezerra Arantes, C.F.R. Geyer, MRA++: Scheduling and data placement on MapReduce for heterogeneous environments, Future Gener. Comput. Syst. 42 (2015) 22–35.

[50] L. Woo, K. Jin-Soo, M. Seungryoul, Large-scale incremental processing with MapReduce, Future Gener. Comput. Syst. 36 (2014) 66–79.

[51] G. Lianjun, Z. Tang, Gu. Xie, The implementation of MapReduce scheduling algorithm based on priority, in: Parallel Computational Fluid Dynamics, Springer, Berlin, Heidelberg, 2014.