Extending CometCloud to Process Dynamic Data Streams on Heterogeneous Infrastructures

Rafael Tolosana-Calasanz[1], Javier Diaz-Montes[2], Omer Rana[3] and Manish Parashar[2]
[1] Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain
[2] Rutgers Discovery Informatics Institute, Rutgers University, USA
[3] School of Computer Science & Informatics, Cardiff University, UK
contact author: rafaelt@unizar.es

***Abstract*—Coordination of multiple concurrent data stream processing, carried out through a distributed Cloud infrastructure, is described. The coordination (control) is carried out through the use of a Reference net (a particular type of Petri net) based interpreter, implemented alongside the CometCloud system. One of the benefits of this approach is that the model can also be executed directly to support the coordination action. The proposed approach supports the simultaneous processing of data streams and enables dynamic scale-up of heterogeneous computational resources on demand, while meeting the particular quality of service requirements (throughput) for each data stream. We assume that the processing to be applied to each data stream is known a priori. The workflow interpreter monitors the arrival rate and throughput of each data stream, as a consequence of carrying out the execution using CometCloud. We demonstrate the use of the control strategy using two key actions – allocating and deallocating resources dynamically based on the number of tasks waiting to be executed (using a predefined threshold). However, a variety of other control actions can also be supported and are described in this work. Evaluation is carried out using a distributed CometCloud deployment – where the allocation of new resources can be based on a number of different criteria, such as: (i) differences between sites, i.e. based on the types of resources supported (e.g. GPU vs. CPU only, FPGAs, etc), (ii) cost of execution; (iii) failure rate and likely resilience, etc.**

I. Introduction

Over recent years, the significant proliferation of geographically distributed sensors has led to a number of applications in areas such as surveillance and monitoring, *smart-* traffic management, cities, energy management in built environments, etc. Examples of these applications include: weather forecasting and ocean observation [1], "Urgent Computing" [2], and more recently data analysis from electricity meters to support "Smart (Power) Grids" [3]. In all these scenarios, data source (sensor) nodes can vary in complexity from smart phones to specialist instruments. Sensors transmit raw data continuously, that needs to be processed over long periods of time with the purpose of monitoring physical or environmental conditions, such as energy consumption, temperature, traffic congestion, noise-levels, humidity at a particular geographical location, or sentiment analysis of a particular user community (using Twitter data, for instance), etc. Often the raw data elements captured from different sources can also be aggregated into complex events – which are subsequently further analysed.

Data capture sources (sensors) often have limited computational resources and battery power, requiring data elements to be processed at destination or en-route (also referred to as *in-transit* processing) [4]. These applications also have Quality of Service (QoS) constraints, requiring processing to be completed within a particular time interval (deadline). In some cases any missed deadline can lead to results that are unusable, to scenarios where infrequent deadline misses can be tolerated. Moreover, data streams in such applications are generally large-scale and generated continuously at a rate that cannot always be estimated in advance. Determining the optimal computational resource configuration for the system becomes important – therefore, dynamic computational resource provisioning remains a major challenge in order to handle variable event loads efficiently, and in order to meet the particular QoS targets (often specified within an Service Level Agreement (SLA)) per stream.

When processing is accomplished at destination involving a pool of heterogeneous computing resources, such as data-centres, and private and public Clouds, the scaling mechanisms also need to deal with variable communication requirements. They are imposed by the diversity of different resources as well as different *overheads*. These requirements can be achieved by introducing communication *orthogonality* in the interactions, whereby interacting peers do not have any prior knowledge about each other. Linda [5] communication & coordination paradigm is based on this principle, leading to two important consequences: space-uncoupling (also referred to as distributed naming) and time-uncoupling. Since the interacting processes do not have to know each other in advance, the scalability of the system can be achieved in a more flexible way.

This communication orthogonality is also one of the foundations of the CometCloud autonomic framework [6]. CometCloud supports real-world applications on dynamically federated, hybrid infrastructures integrating public & private Clouds, data-centers and Grids. At the infrastructure-level, CometCloud provides a range of services for dynamic federation and coordination in order to enable on-demand scale-up (to scale vertically: adding more resources to a single site), scale-out (to scale horizontally: adding more resources from external sites), and scale-down (to reduce either nodes or resources). CometCloud also supports a number of programming models and services for autonomic management and monitoring of an application, as well as of the underlying infrastructure. Among the supported programming models is master/ worker, whereby one process (the master) submits the computational tasks to the workers via a (virtually) shared "tuplespace", which typically performs computations for the master in parallel.

In this paper, we extend CometCloud with an autonomic streaming workflow interpreter, which is specified using Reference nets (a particular type of Petri net), and which is directly executable. It supports the simultaneous processing of data streams and the dynamic scale-up of heterogeneous computational resources on demand, while meeting the particular QoS requirements (throughput) for each data stream. Our focus is to handle data analytics for data streams whose processing deadlines are of the order of minutes or hours, rather than data streams which have strict real time processing requirements. Two main contributions are provided in this paper: the first one is conceptual and provided by our workflow streaming interpreter based on Reference nets. The interpret demonstrates how simultaneous processing of multiple data streams can be achieved using a tuplespace abstraction using CometCloud. In this paper, we assume mutual independence of streams, having each their own functional and non-functional (SLA) requirements. The non-functional requirement being stream throughput. We also assume that for each data stream its functional requirements can be *explicitly* specified as a sequence of operations to be applied to each data element. Additionally, each workflow task (operation), in our proposal, is specified as a sequence of two Linda operations: i) an *out* operation writes a request into CometCloud's tuplespace, specifying the operation name and its arguments; and ii) an *in* operation retrieves a result back upon completion. Therefore, the effect is that the functional requirements for each data stream are defined in an orthogonal way, and they are uncoupled in time and space from the interacting peers that actually perform the computations.

This characteristic leads to the second contribution of this paper. The uncoupling between workflow task submission and peers which execute tasks, allows CometCloud to dynamically modify the capacities and capabilities allocated for each data stream. The relevance and novelty of the contribution lies in the fact that most existing systems [4], [7], [8] consider homogeneous nodes to support scale-up. Our workflow interpreter can dynamically allocate/ deallocate resources based on the number of pending tasks (using a pre-defined threshold to trigger such allocation/ deallocation). The rest of the paper is structured as follows. In Section II, we provide a brief overview of Reference nets. In Section III, CometCloud's architecture and functionality is briefly described. Our workflow streaming interpreter is presented in Section IV. An evaluation scenario is given in Section VI. Related work is discussed in Section VII, and finally conclusions and future work are outlined in Section VIII.

II. Background: Reference nets

Reference nets are a particular type of Petri net that supports Java code inscriptions, which are typically embedded and also Reference nets can be interpreted by Renew [9], a Java-based editor and simulator. The most important consequence of this fact is that Petri net based models become executable, converting the formal specification models into rapid prototyped systems. Although it is out of the scope of this paper, further formal analysis can be conducted on the Petri nets models, providing insights on the system's behaviour and performance.

Petri nets [10] have been recognised by their ability to represent parallel or concurrent processes. A Petri net is a graph with two kind of nodes, places and transitions, which represent conditions and actions respectively. Places can also contain elements called tokens, which evolve through the places to complete the state representation. The execution of actions require the satisfaction of preconditions represented by input arcs going from places to transitions, whereas postconditions are specified by output arcs. In high-level Petri nets nodes are typed representing the type of state for each place, the type of event for each transition, and the type of objects associated with the tokens that flow through the net.

The *Reference net* formalism is a special class of high-level Petri net (adhering to the Nets-within-Nets [11] paradigm) that uses Java as an inscription language, and extends Petri nets with dynamic net instances, net references, and dynamic transition synchronisation through synchronous channels. Reference nets consist of places, transitions and arcs. The input and output arcs have a behaviour similar to ordinary Petri nets. Every net element can have associated semantic inscriptions: places can have initialisation expressions, which are evaluated and serve as their initial markings. Arcs can have optional arc inscriptions: when a transition fires, its arc expressions are evaluated and tokens are moved according to the result. Transitions can be equipped with a variety of inscriptions, including Java inscriptions, in which the equality operator "=" can be used to influence the binding of variables that are elsewhere. The binding is similar to the way variables are used in logic programming languages such as Prolog. Additionally, the inscription language of Reference nets has been extended to include tuples. A tuple is denoted by a comma-separated list of expressions that are enclosed in square brackets. Tuples are useful for storing a whole group of related values inside a token and hence in a single place. The nets hold two kinds of tokens: valued tokens and tokens which correspond to a reference. By default, an arc will transport a black token, denoted by `[]`. In case an inscription is added to an arc, that inscription will be evaluated and the result will determine which kind of token is moved. Additionally, there are creation inscriptions that deal with the creation of net instances. Net instances can communicate with each other by means of synchronous channels. They synchronise two transitions which both fire atomically at the same time. Both transitions must agree on the name of the channel and on a set of parameters before they can synchronise. The initiating transition must have a special inscription – called *downlink* – which makes a request to a designated subordinated net. A downlink consists of an expression that must evaluate to a net reference (usually a variable), a colon (:), the name of the channel, and an optional list of arguments. On the other side, the transition must be inscribed with an *uplink*, which serve requests for everyone. Channels can also take a list of parameters. Although there is a direction of invocation, this direction need not coincide with the direction of information transfer. Indeed, it is possible that a single synchronisation transfers information in both directions.

III. Background: The CometCloud Autonomic Computing Framework

CometCloud is an autonomic framework for enabling real-world applications on software-defined federated cyberinfrastructure, including hybrid infrastructures integrating public & private Clouds, data-centers and Grids. The overarching goal of CometCloud is to realize a software-defined federation with

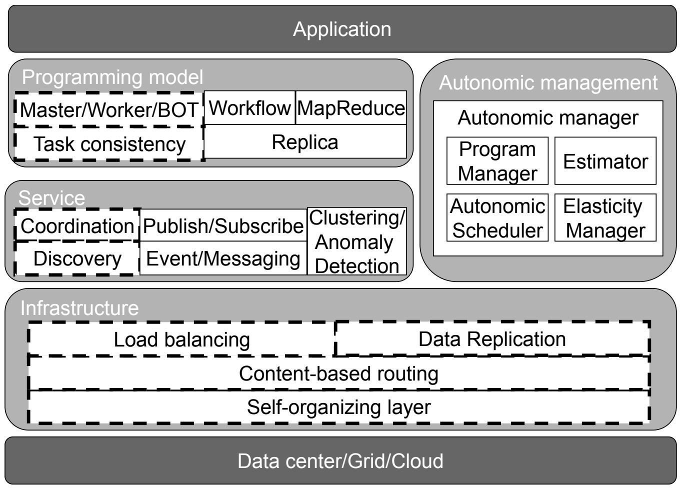


Fig. 1: CometCloud system architecture

cloud abstractions that offer resources in an elastic and on-demand way. It also provides abstractions and mechanisms to support a range of programming paradigms and applications requirements on top of the federation. In this section, essential aspects of CometCloud's architecture are briefly discussed, including its autonomic support and behaviour, and the provided master/ worker programming model. CometCloud uses a Linda-like tuple space referred to as "CometSpace" – which is implemented using a Peer-2-Peer overlay network. In this way, a virtual shared space for storing data can be implemented by aggregating the capability of a number of distributed storage and compute resources.

CometCloud's system architecture is depicted in Fig. 1. It consists of three architectural layers, namely the programming layer for giving support to different programming models (e.g. master / worker, MapReduce or data-driven workflows), the middleware layer for supporting inter-process communication, and the infrastructure layer for exploiting different computational infrastructures transparently.

Specifically, CometCloud enables policy-based autonomic cloudbridging and cloudbursting [6]. Autonomic cloudbridging enables on-the-fly integration of local computational environments (datacenters, grids) and public cloud services (such as Amazon EC2 and Eucalyptus), and autonomic cloudbursting enables dynamic application scale-out to address dynamic workloads, spikes in demands, and other extreme requirements. Key motivations for autonomic cloudbursts include load dynamics for applications whose workloads can vary significantly. This includes the number of application tasks as well the computational requirements of a task. The computational environment must dynamically grow (or shrink) in response to these dynamics while still maintaining strict deadlines. The computational environment must be able to dynamically adapt to satisfy the functional requirements while still maintaining the non-functional ones.

Since application tasks can have very heterogeneous and dynamic priorities, and must be assigned resources and scheduled accordingly, budgets and economic models can be used to dynamically provision computational resources based on the priority and criticality of the application task. For example, application tasks can be assigned budgets and can be assigned resources based on this budget. The computational environment must be able to handle heterogeneous and dynamic provisioning and scheduling requirements. On the other hand, failures can have negative consequences for meeting non-functional constraints. CometCloud integrates different fault tolerance & recovery mechanisms, so that in the presence of faults, where possible computations can finalise correctly without impacting application QoS, including deadlines and accuracies.

A. *Autonomics in CometCloud*

Two main approaches are available in CometCloud for autonomic behaviour, namely master-based and worker-based autonomics. In the worker-based autonomics, each worker acts as agents in a marketplace [12], where a worker can select the type of tasks it is interested in, in other words, a worker has the autonomy to decide when to select a tuple based on its utility function (in this instance, each tuple contains properties of the task required to be executed by the master). In such a context, based on the incurred cost and on the revenue when charging for their services, workers aspire to maximise their profit. A worker therefore can opportunistically select tuples that enable it to maximise its revenue. Over time, a worker is able to identify tasks which are most suitable based on resources it has, and can build reputation in the marketplace for executing such tasks. Alternatively, workers select tuples based on a blind auction process, i.e. the master puts a task into the tuplespace, and then each worker places an offer. The master decides which worker has the best offer, and considering the reputation, it decides which worker takes the task.

In master-based autonomics, the master uses monitoring data to decide a control action, such as: load balancing, resource provisioning and resource federation management available within CometCloud at the infrastructure and services layers. While in this approach, the workers' sole responsibility is to execute tasks, the master can dynamically allocate new workers from preferred sites, or add additional sites to the federation [13]. In this paper, we are focused on master-based autonomics, by adding a Reference net interpreter at the master for support work load management subject to specific QoS constraints.

IV. Autonomic Workflow Streaming in CometCloud

In order to support workflow streaming programming model, we designed and implemented a Reference net-based workflow interpreter that can process multiple data streams simultaneously coming from different distributed sources for processing. The workflow interpreter coordinates the processing of the data elements (based on task dependency constraints identified in the workflow specifications), and it also dynamically scales the required resources on demand, while enforcing the particular QoS requirements for each data stream, measured in terms of throughput. For both purposes, the interpreter interacts with CometCloud in two different ways: i) to dispatch processing tasks to distributed computational resources in CometCloud's federation (by inserting task tuples into CometSpace and by retrieving the results back), and ii)

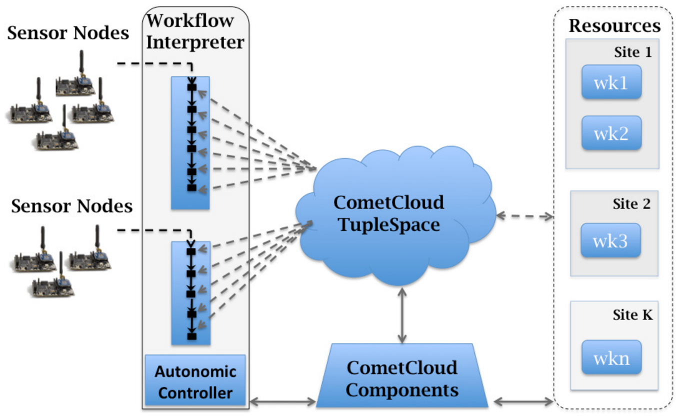


Fig. 2: Autonomic High-level System Architecture for Large-Scale Stream Processing in CometCloud

to monitor and control the computational resources around the tuple space for dynamically scaling them on demand.

Figure 2 depicts the workflow interpreter, CometCloud, and the federation of computational resources. In particular, the mechanism is as follows. The interpreter is responsible for creating and enacting the workflow instances and for receiving the data elements of the streams. When a data element arrives in the interpreter, it is injected into the corresponding streaming workflow pipeline instance for processing. Then, as a data element advances through its corresponding workflow pipeline, task tuples will be written to and retrieved from CometSpace. Initially, the interpreter allocates a number of worker nodes to each data stream, based on historical executions. These nodes will withdraw workflow task tuples, perform the required computations, and finally they will write the result tuple back into CometSpace. The result tuple will be taken by the streaming interpreter, which will re-direct it to the corresponding workflow instance.

Hence, CometSpace acts as a buffer of task requests which are waiting to be retrieved by worker nodes. CometCloud's middleware components can be configured so that workflow tasks from the same data stream are stored within the same location.

A. Streaming Workflow Specification

We assume that the operations to be applied to each data stream have to be specified a priori, and the workflow streaming model of computation will be applied for this purpose. This model of computation consists of a sequence of one or more tasks applied sequentially to a vector of input data elements as they are received from sensors. We assume that a data element can pass through a workflow pipeline task as it is produced by its predecessor (avoiding blocking semantics). In consequence, unlike other pipeline models of computation, multiple data elements could be executing the same task at a time, or even data elements can finalise their execution in an out-of-order manner.

Each of our workflow tasks are specified in an abstract way – without binding to any computational resource, and exclusively in terms of two CometCloud's Linda-like operations:

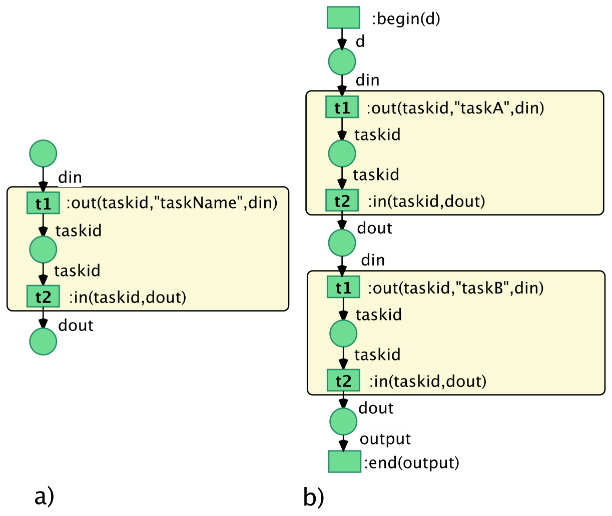


Fig. 3: a) Linda-based Streaming Workflow Task Pattern; b) Streaming Workflow Example

an *out* operation writing the operation name and its arguments into CometSpace, and an *in* operation retrieving the results from CometSpace upon completion. The main advantage of having explicit and abstract streaming workflow specifications is that by doing so, each task is uncoupled in time and space from the worker node that will execute it, and this provides greater degrees of flexibility in the execution.

In our proposal, a streaming workflow is specified as depicted by Fig. 3. As the Reference nets are actually interpreted, these specifications are also utilised for the actual execution of the coordination mechanism. A simple workflow task is specified in Fig. 3 a). It consists of two transitions, Transitions *t1* and *t2*. Both of them require a data element as a prerequisite in order to get fired once, and only one data element at a time is consumed at each firing. After the firing of a transition, the results get synchronised with the streaming workflow master Reference net by means of synchronous channels. Transition *t1* has Synchronous Channel *out* associated with it, whereas Transition *t2* has Synchronous Channel *in*. The arguments taken at Channel *out* are the operation name and the data element, the variable *taskid* is received so that it can be subsequently used by retrieving the value at Channel *in*. Then, once the task is performed, Channel *in* obtains corresponding result.

Fig. 3 b) shows a streaming workflow example. Data elements are introduced by means of Channel *begin* at the initial transition, whereas a data element finalises its execution and sends the result at Channel *end*. The illustrated workflow is a sequential composition of two tasks, named "task A" and "task B", respectively. In particular, tasks are expected to be connected by means of a data dependency, the output *dout* of a tasks becomes the input *din* of the following task in the sequence. It is important to note that the model is following streaming semantics, whereby multiple data elements can be executing a task simultaneously, and even the processing for a data element can finalise out of order for previously arrived data elements.

B. Streaming Workflow Interpreter

The streaming workflow interpreter presented in Figure 2 is now specified in terms of a Reference net in Fig. 4. Its transitions are labelled functionally in 3 main groups: the ones starting with letter *i* are responsible for initialisation purposes, the ones starting with letter *t* are responsible for coordination and execution of data elements within the workflows, and finally the ones starting with letter *c* are responsible for the autonomic behaviour of the interpreter.

A workflow instance as specified in Fig. 3 b) is created by each firing of Transition *i2* of the net of Fig. 4. This Transition *i2*, when fired, retrieves a unique *streamid* from Transition *t6*, creates a workflow instance and introduces it in the Place *p2* where all the workflow instances are stored in the net. Thus, any workflow instance in that place can be identified by its *streamid*. It should be noted that the interpreter can trigger four different operations to any workflow instance stored in that place, namely *out* (write a tuple), *in* (retrieve a tuple), *begin* (introduce a data element in the workflow pipeline), and *end* (get a data element processed from the workflow pipeline). The *begin* operation is accomplished when firing Transition *t3*, this transition uses three synchronous channels when firing. An external net invokes Channel *receive*, providing both a data element to be processed, and the corresponding *streamid*, identifying the workflow instance that will perform the computation. In turn, that transition invokes Channel *Rin* to register the entrance of a data element for subsequent monitoring purposes, and also invokes Channel *begin* of the workflow instance associated to *streamid*, introducing the data element into it. Analogously, Transition *t4* has a similar mechanism of operation: it consists of three channels, Channel *end* of the workflow instance is invoked to extract the final data element from the workflow, Channel *Rout* is invoked for monitoring purposes (to compute the throughput), and Channel *receive* is invoked by an external net to store the output.

Transitions *t1, t2* are utilised for writing (out operation) and retrieving tuples (in operation) from CometCloud respectively. In addition to the workflow, they also utilise a subnet called CometCloudConn (stored in Place *p1*) which directly interacts with CometCloud's tuplespace, and with CometCloud's middleware, by means of Java calls. Transition *t2* invokes a Synchronous Channel *out* of a workflow instance (*w:out(taskid,op,d)*, see both Figs. 3 & 4), receiving the task name (*op*), and the input (*d*), then it gets a unique identifier for a task from Channel *getId*, and finally invokes Synchronous Channel *out* of the CometCloudConn (*m:out(streamid,taskid,op,d)*), providing the task identifier, the task name, and the input. Once, the CometCloudConn gets the results from CometCloud, Transition *t2* can be fired. In such a case, the Channel *in* from CommetCloudConn (*m:in(streamid,taskid,d)*) provides the output data element *d*, which is subsequently provided to the workflow by means of its Channel *in* (*w:in(taskid,d)*).

C. Autonomic Behaviour of our Streaming Workflow Interpreter

Our assumption in this paper is that the workload coming from the data streams cannot be predicted in advance. Bursty behaviours from sensor generation rates often require elastic scaling of computational resources so that the QoS of the data streams are enforced, and resources are not under utilised. The goal of the workflow interpreter is therefore to process multiple data streams in accordance with their respective workflow plan, and enforce QoS for each data stream. In addition to bursty behaviours of any particular data stream, deviations from the QoS goal may also be due to the computational infrastructure: unexpected failures, performance fluctuation, queue wait time variation, etc. For all these cases, the workflow interpreter has to guarantee that data stream computations are isolated and not affecting one another.

Our focus in this paper is to develop a reactive autonomic behaviour for our streaming workflow interpreter, in combination with some limited near-future prediction. For such a purpose, the workflow interpreter is monitoring the incoming data rate for each stream, and also the obtained departure rate (throughput), as a consequence of the processing carried out on the data. In Fig. 4, the arrival rate for each data element is recorded by Transition *c1*, which is fired every time a data element is introduced into a workflow (synchronisation between Transitions *c1* and *t3*). The departure rate is recorded by the firing of Transition *c2* (synchronisation between Transitions *c2* and *t4*). It also makes use of CometCloud capabilities to query the number of workflow tasks per data stream that have not been taken from the tuplespace. This value can act as an indicator for predict changes likely to take place in the near future. The number of pending tasks above or below established thresholds indicates that the workload of a data stream has varied and, in consequence, the number of workers must be updated accordingly without manual intervention. The thresholds constitute a dead-zone to prevent inefficient and ineffective control brevity when a system is sufficiently close to its target value [14]. No action is therefore taken when the number of tuples is between the low and upper values.

The autonomic behaviour for the interpreter is built upon a closed feedback-loop, which is triggered periodically, it is configured now to be performed every second (however, this parameter will be considered to be self-modified in the future). The mechanism makes use of Transition *c3* that generates a token every 1000ms, which will subsequently fire Transition *c4*. On firing Transition *c4*, Channel *getTuples* from CometCloudConn is invoked, providing the number of tuples per data stream that have been written into CometSpace, but have not been retrieved yet. CometCloudConn net queries CometCloud middleware system for such a purpose. Furthermore, Channel *getCurrentRates* is also invoked, retrieving for each stream, the arrival rate and the output, which have been previously recorded. Once Transition *c4* is fired, it enables Transition *c5*, where the workflow controller is invoked for accomplishing autonomic actions. The controller receives the monitored information previously gathered (input rate, throughput and number of pending tuples per stream) and provides a list of actions, one per stream. By means of the double arc getting out of Transition *c5*, a token (representing an action) is generated per each element of the list. This means that Transition *c6* will be enabled as many times as tokens (representing actions) generated. Each token specifies the number of workers to be allocated / deallocated to a data stream. Transition *c6*, when fired, invokes Synchronous Channel *workers* of CometCloudConn net, which in turn re-directs the action to CometCloud's middleware.

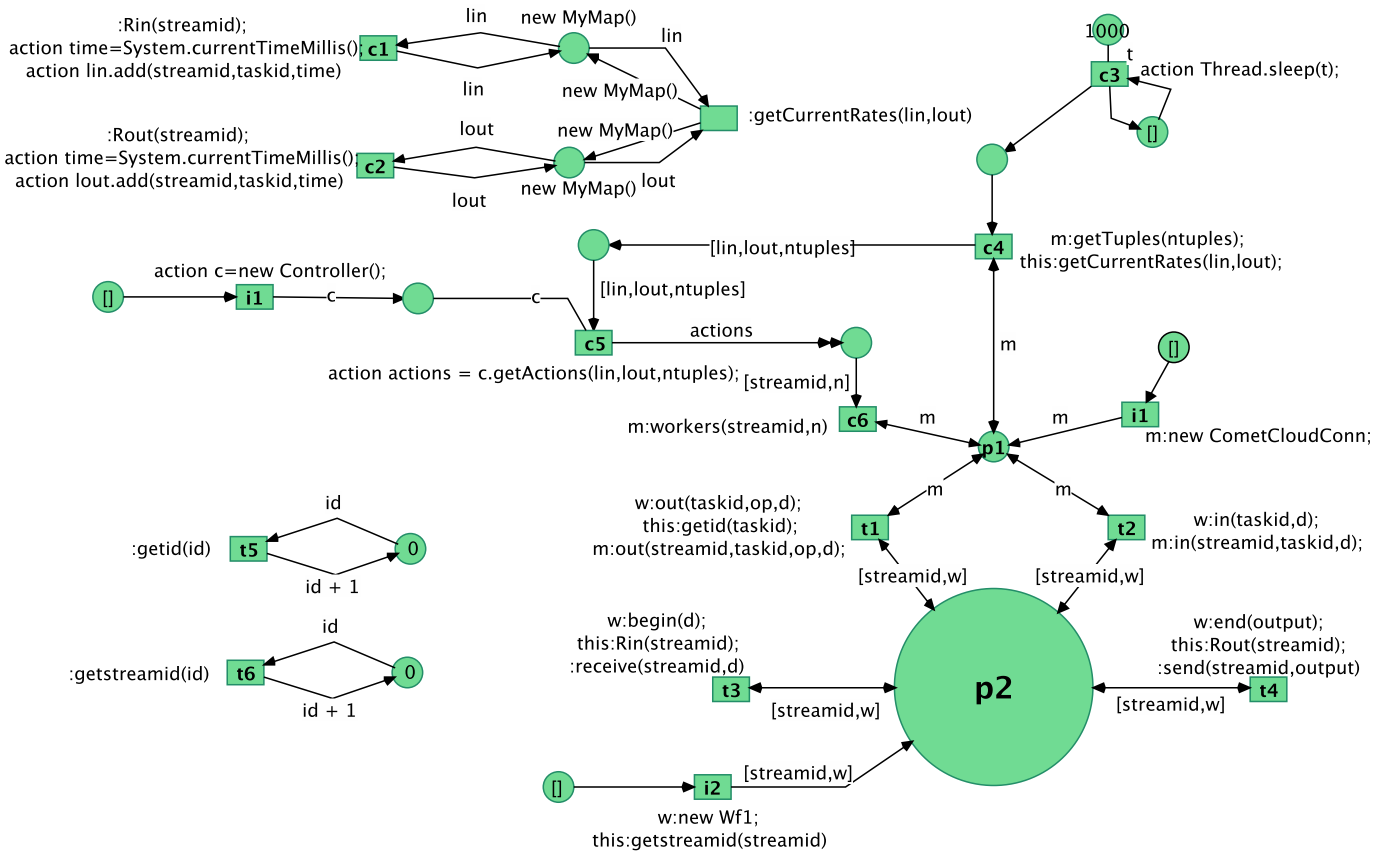


Fig. 4: Autonomic Workflow Streaming Master Reference net

Essentially, we provide the controller with two main actions: i) to dynamically allocate a computational resource to a data stream; which is triggered when the number of tasks pending to be executed is above the higher threshold, ii) to dynamically deallocate a computational resource to a data stream; which is triggered when the number of tasks pending to be executed is below a threshold and the output rate is below the goal throughput. For the configuration of the dead zone logic-based threshold interval, if it is too wide, the system will have a slow response, incurring into QoS penalties, whereas if it is too short, there may be ineffective resource allocations / deallocation leading to resource underutilisation. Therefore, the interval must consider the income rate (it indicates the accumulation of tuples), the goal throughput, and how the workers selected perform, also including related overheads, such as network transmissions, new worker allocations, queueing waiting times, etc. We currently based it on historical executions, and we expect to dynamically adjust it in the future to suit changes in environment volatility.

Nevertheless, the action of allocating a new resource can be enriched by the possibility of choosing the resources among different sites. The following criteria may be used to support this action:

- Data burst acceleration: whenever there is a considerable burst of data, tuples will begin to accumulate in CometSpace. In such a situation, resources need to be allocated over a short time span in order to avoid QoS penalties. A key action in this instance would be to utilise resources from distributed sites involved in a Cloud federation (as demonstrated in here). Cloud infrastructures provide access to a potentially large pool of resources which can be used execute tasks.
- High Performance Computing (HPC) resource conservation: tightly coupled HPC resources are often essential for executing scientific applications, and access to such resources is very limited. It is therefore necessary to decide when to use HPC resources over more generally available Cloud resources. For example, we could use a Cloud to process high throughput tasks and HPC resources to process tasks which are computationally intensive. This could be done considering runtime and budget constraints.
- Resilience: resource selection may also be influenced by failure rates, average time between failures or fault management mechanisms in place. This choice may lead to the consumption of a greater number of resources (replicas). Once a fault has been detected, additional Cloud resources can be requested dynamically.
- Cost: when a budget is enforced on the workflow interpreter for the overall processing of the data streams, the number of allocatable resources is restricted by the

budget.

V. Experiment Methodology

We consider two independent streaming workflows that require data processing at two different rates. In this experiments, we are going to show how our autonomic manager is able to independently adapt the number of resources to the input data to meet our QoS objective, which consists of maintaining throughput equal to input rate. The streaming workflow 0 starts inserting tasks at a rate of one and a half seconds during a period of 30 seconds, then its rate increases and inserts a task every 700 milliseconds during another 30 seconds, finally its rate slows down to a rate of one task every 3 seconds. The streaming workflow 1 inserts tasks at a constant rate of one and a half seconds. Every task takes around 4 seconds.

We are going to study the behaviour of our autonomic framework by the simultaneous execution of the two workflows in two different computational environments: (a) only resources from our local datacenter, and (b) a federation context where we have local resources and external cloud resources. Table I show the number of resources we use from each site. Next, we provide the description of each site:

- Rutgers site (local resources): cluster based infrastructure with 32 dedicated cluster machines. Each node has 8 cores, 6 GB memory, 146 GB storage and Gigabit Ethernet connection. The measured latency on the network is 0.227ms in average.
- Hotel cloud site: cloud infrastructure based on the Nimbus [15] IaaS, which is located at the University of Chicago. We have used instances of type small, where each instance has 1 cores and 2 GB of memory. The networking infrastructure is DDR Infiniband and the measured latency of the cloud virtual network is 0.096ms in average.
- India cloud site: cloud infrastructure based on the OpenStack [16] IaaS, which is located at Indiana University. We have used instances of type small, where each instance has 1 cores and 2 GB of memory. The networking infrastructure is DDR Infiniband and the measured latency of the cloud virtual network is 0.706ms in average.

TABLE I: Resources available at each site.

Site	Number of Workers
Rutgers	6
India	3
Hotel	7

The interconnection network overhead between sites is as follows: 36 ms between Rutgers and India; 4 ms between Rutgers and India; and 6.7 ms between India and Hotel.

VI. Experimental Validation

Figure 5 shows the result for both scenarios, on the left column, we only use resources local to our data center. On the right column, we federate cloud resources to complement local ones. We show two types of graphs for both scenarios: i) one type that displays instant input and output rates, as well as the resources involved in the computation; and ii) another type that displays average input and output rates. The instant graphs can reflect how sudden changes on input rate trigger an action, the allocation / deallocation of resources, and how that action is affecting the throughput. In contrast, average graphs can show the behaviour in average terms, eliminating the effect of sudden changes.

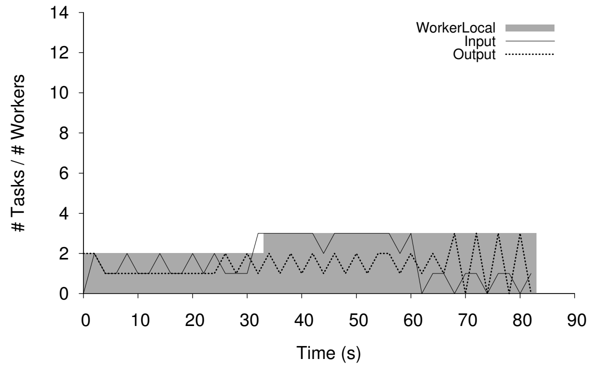
It should also be noted that for the execution of the workflows, it is not enough with the local resources, and remote ones are required. In all the cases, both workflows have allocated 2 resources (see Figures 5a, 5c, 5b, 5d). Therefore, in the scenario where only local resources are involved, when the rate of workflow 1 increases at second 30 (see Figure 5a), the controller of the workflow interpreter cannot find more local resources to enforce its QoS. Figure 5c. Three resources were allocated at that time for workflow 0, whereas workflow 1 has the three remaining ones. Therefore, for workflow 0, the QoS is missed for the period 30 - 60. At second 60, the input rate of workflow 0 significantly decreases, allowing the pending tasks to be processed. That circumstance is also reflected by the average input and output rates for workflow 0, displayed on top of Figure 5e.

For the scenario combining local and remote resources, it can be seen that at second 30, the workflow interpreter can allocate remote resources for maintaining throughput for workflow 0 (see Figure 5c). The income rate of workflow 0 is decreased by second 60, which eventually provokes that the number of allocated resources is decreased by around second 70. Average throughputs for workflow 0 can be compared in Figures 5e and 5f, top. With remote resources (Figure 5f, top), average throughput is close to the average input rate. Besides, having some additional remote resources for workflow 1 also improve the enforcement of throughput, as it can be seen by comparing Figures 5e and 5f, bottom.

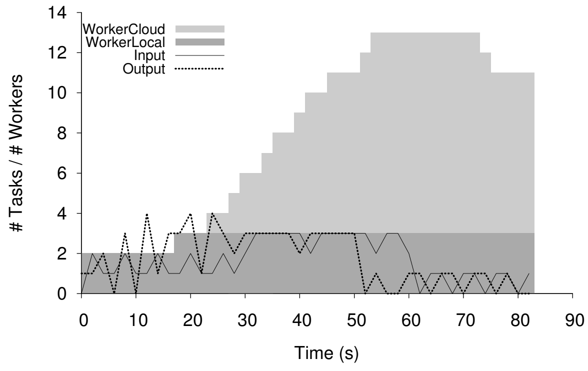
VII. Related Work

In the past few years, due to the proliferation of sensors in a number of domains (road traffic, social network analysis, etc.), there has been an emergence of applications that do not fit into that model of traditional databases and querying paradigm. Indeed, the increasing deployment of sensor network infrastructures has led to large volumes of data becoming available, leading to new challenges in storing, processing and transmitting such data [17]. Distributed systems represent an essential paradigm for addressing the challenges of the large amounts of generated data.

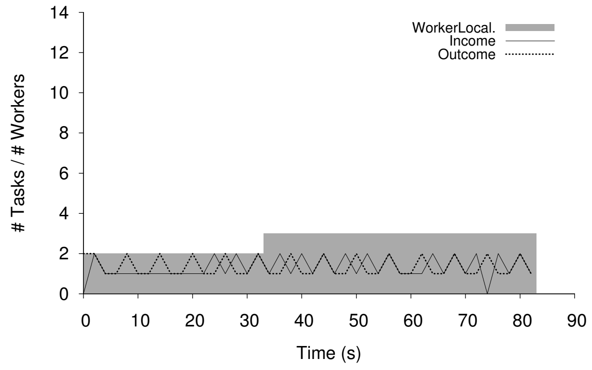
For that reason, stream processing frameworks such as Yahoo's S4 [18], or IBM InfoSphere Streams [19] provide streaming programming abstractions to build and deploy tasks as distributed applications at scale for commodity clusters and clouds. Nevertheless, even that these systems support high input data rates, they do not consider variable input rates, which is our focus in this paper. In some other approaches, the parallelism is extracted from the data stream query operators they provide, Aurora [20], Borealis [21] and Stream Cloud [22], which differs that in our case, we explicitly exploit the parallelism by having multiple data elements in multiple workflow pipelines.



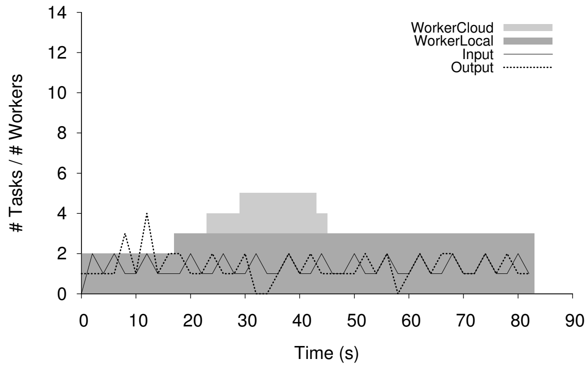
(a) Instant input-output rates for workflow 0



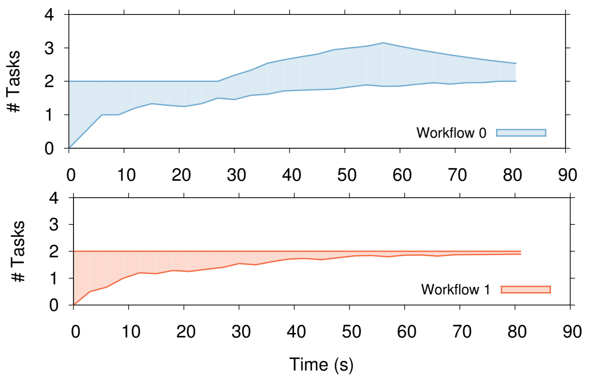
(b) Instant input-output rates for workflow 0



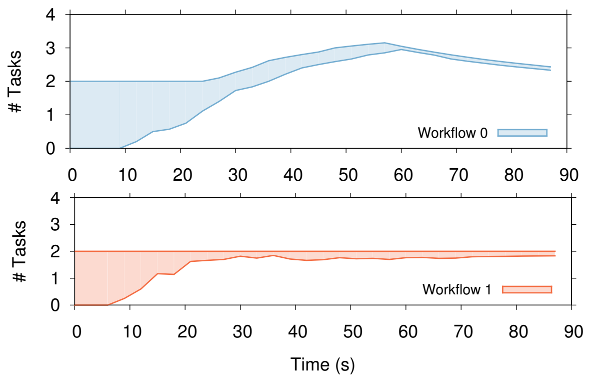
(c) Instant input-output rates for workflow 1



(d) Instant input-output rates for workflow 1



(e) Difference between average input-ouput rates



(f) Difference between average input-ouput rates

Fig. 5: Summary of experimental results. On the left column, we only use resources local to our data center. On the right column, we federate cloud resources to complement local ones

On the other hand, there is a number of scientific workflow engines that incorporate the streaming workflow model of computation and elastic infrastructures, such as Kepler [23], [24] and Triana [25]. But to the best of our knowledge they are not considering dynamism in streaming income rate or heterogeneity in the infrastructure.

Our work is closely related to three approaches. In [26], the goal is to allocate resources dynamically from a Cloud, so that the processing rate can match the rate of data arrival. They also consider variable transient input rates. In our approach we make use of a federation of heterogeneous resources and for the selection of resources, we propose autonomic based mechanisms and policies. In [7], the authors propose a workflow specification where each task consists of one or more alternate implementations with different non-functional properties, so that the system can choose any of them dynamically at runtime. In this paper, we have not considered dynamism at workflow-level, but our dynamic provisioning of resources is accomplished in a federation of heterogeneous resources. Finally, the work in [4], [27], [28], [29] consists of a sequence of nodes, where each node has multiple data buffers and computational resources – whose numbers can be adjusted in an elastic way. They utilize the token bucket model for regulating, on a per stream basis, he data injection rate into such nodes. The main difference to our approach in here is that instead of utilising multiple nodes, we assume CometCloud system as a coordination mechanism that can outsource the computation when required. Besides, instead of the token bucket mechanism we exploit autonomic behaviours, and CometSpace for buffering.

VIII. Conclusion

In this paper, we describe how autonomic capability could be integrated with a federated Cloud environment. Our approach is demonstrated by extending the capabilities of a master in the CometCloud system with an autonomic streaming workflow interpreter, which is specified in terms of an executable Reference net (a type of Petri net) model. The benefit of this approach is that the model can be analysed directly but can then also be executed. Using Reference nets, the model is therefore also the actual executable system. Our workflow interpreter supports the simultaneous processing of data streams and enables elastic scale-up of heterogeneous computational resources, while meeting the the QoS requirement of throughput for each data stream. We assume that the processing to be applied to each data stream must be specified *a priori*, and the workflow streaming model of computation and Petri nets will be applied for this purpose. The workflow interpreter monitors the arrival rate of each data stream, and the obtained departure rate, as a consequence of the execution. It also makes use of CometCloud capabilities to query the number of workflow tasks per data stream that have not been taken from CometSpace. This value can act as an indicator to adapt the behaviour of the interpreter over the near future. Essentially, we provide the controller with two main actions: i) to dynamically allocate a computational resource to a data stream; which is triggered when the number of tasks pending to be executed is above the higher threshold, ii) to dynamically deallocate a computational resource to a data stream; which is triggered when the number of tasks pending to be executed is below a threshold and the input rate is below the throughput. The action of allocating a new resource can be enriched by the possibility of choosing the resources among different sites, involving a number of policies that consider either cost, resilience, reduced use of more costly (high performance computing) resources, or the occurrence of severe data bursts.

Acknowledgment

Rafael Tolosana's work was supported in part by the Spanish Ministry of Education, under the Framework Program CEI Iberus (Universidad de Zaragoza) for Faculty Staff, Mobility Call 2014. This work was supported in part by the NSF under grants OCI 1339036, OCI 1310283, DMS 1228203, IIP 0758566 and by an IBM Faculty Award. This project used resources from FutureGrid supported in part by NSF OCI-0910812. This work was partially supported by the Spanish Ministry of Economy under the program "Programa de I+D+i Estatal de Investigación, Desarrollo e innovación Orientada a los Retos de la Sociedad", project identifier TIN2013-40809-R.

References

[1] G. Allen, B. P., T. Kosar, A. Kulshrestha, G. Namala, S. Tummala, and E. Seidel, "Cyberinfrastructure for coastal hazard prediction," *CTWatch Quarterly*, vol. 4(1), 2008.

[2] S. Marru, D. Gannon, S. Nadella, P. Beckman, D. B. Weber, K. A. Brewster, and K. K. Droegemeier, "Lead cyberinfrastructure to track real-time storms using spruce urgent computing," *CTWatch Quarterly*, vol. 4(1), 2008.

[3] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna, "Adaptive rate stream processing for smart grid applications on clouds," in *Proceedings of the 2nd international workshop on Scientific cloud computing*, ser. ScienceCloud '11. New York, NY, USA: ACM, 2011, pp. 33–38. [Online]. Available: http://doi.acm.org/10.1145/1996109.1996116

[4] R. Tolosana-Calasanz, J. A. Bañares, and O. F. Rana, "Autonomic streaming pipeline for scientific workflows," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 16, pp. 1868–1892, 2011.

[5] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[6] H. Kim and M. Parashar, *CometCloud: An Autonomic Cloud Engine*. John Wiley & Sons, Inc., 2011, pp. 275–297. [Online]. Available: http://dx.doi.org/10.1002/9780470940105.ch10

[7] A. G. Kumbhare, Y. Simmhan, and V. K. Prasanna, "Exploiting application dynamism and cloud elasticity for continuous dataflows," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, W. Gropp and S. Matsuoka, Eds. ACM, 2013, p. 57.

[8] R. Tolosana-Calasanz, J. A. Bañares, C. Pham, and O. F. Rana, "End-to-end QoS on shared clouds for highly dynamic, large-scale sensing data streams," in *Proc. of 1st International Workshop on Data-intensive Process Management in Large-Scale Sensor Systems (DPMSS 2012): From Sensor Networks to Sensor Clouds*, 2012, pp. 904–911.

[9] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, and R. Valk, "An extensible editor and simulation engine for Petri nets: Renew," in *Applications and Theory of Petri Nets 2004, 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings*, ser. Lecture Notes in Computer Science, J. Cortadella and W. Reisig, Eds., vol. 3099. Springer, 2004, pp. 484–493.

[10] T. Murata, "Petri nets: Properties, analysis and applications," in *Proceedings of IEEE*, vol. 77, Apr. 1989, pp. 541–580.

[11] R. Valk, "Petri nets as token objects: An introduction to elementary object nets," in *Application and Theory of Petri Nets 1998, 19th International Conference, ICATPN '98, Lisbon, Portugal, June 22-26, 1998, Proceedings*, ser. Lecture Notes in Computer Science, J. Desel and M. Silva, Eds., vol. 1420. Springer, 1998, pp. 1–25.

[12] I. Petri, T. Beach, M. Zou, J. Diaz-Montes, O. Rana, and M. Parashar, "Exploring models and mechanisms for exchanging resources in a federated cloud," in *IEEE international conference on cloud engineering (IC2E 2014), Boston, Massachusetts, March 2014*, IEEE, Ed., 2014.

[13] J. Diaz-Montes, M. Zou, R. Singh, S. Tao, and M. Parashar, "Data-driven workflows in multi-cloud marketplaces," in *IEEE International Conference on Cloud Computing (Cloud), Alaska, June 2014*, Accepted.

[14] T. Eze, R. Anthony, C. Walshaw, and A. Soper, "A new architecture for trustworthy autonomic systems," in *EMERGING 2012, The Fourth International Conference on Emerging Network Intelligence*, 2012, pp. 62–68.

[15] "Nimbus Project. http://www.nimbusproject.org/."

[16] "OpenStack Project. http://www.openstack.org/."

[17] L. Golab and M. T. Özsu, "Issues in data stream management," *SIGMOD Rec.*, vol. 32, no. 2, pp. 5–14, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/776985.776986

[18] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ser. ICDMW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 170–177. [Online]. Available: http://dx.doi.org/10.1109/ICDMW.2010.172

[19] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in

Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 1093–1104. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807291

[20] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," in *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[21] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The Design of the Borealis Stream Processing Engine," in *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[22] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez, "Streamcloud: A large scale data streaming system," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, june 2010, pp. 126 –137.

[23] L. Dou, D. Zinn, T. McPhillips, S. Kohler, S. Riddle, S. Bowers, and B. Ludascher, "Scientific workflow design 2.0: Demonstrating streaming data collections in kepler," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, April 2011, pp. 1296–1299.

[24] D. Zinn, Q. Hart, T. McPhillips, B. Ludaescher, Y. Simmhan, M. Giakkoupis, and V. K. Prasanna, "Towards reliable, performant workflows for streaming-applications on cloud platforms," in *11st International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011), May 2011, Newport Beach, USA*, 2011.

[25] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, "Programming scientific and distributed workflow with Triana services: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1021–1037, 2006.

[26] S. Vijayakumar, Q. Zhu, and G. Agrawal, "Dynamic resource provisioning for data streaming applications in a cloud environment," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, Nov 2010, pp. 441–448.

[27] R. Tolosana-Calasanz, J. Á. Bañares, C. Pham, and O. F. Rana, "Enforcing qos in scientific workflow systems enacted over cloud infrastructures," *Journal of Computer and System Sciences*, vol. 78, no. 5, pp. 1300–1315, 2012.

[28] R. Tolosana-Calasanz, J. Á. Bañares, C. Pham, and O. Rana, "Revenue models for streaming applications over shared clouds," in *1st International Workshop on Clouds for Business and Business for Clouds (C4BB4C), held at the 10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA2012*, 2012.

[29] J. Á. Bañares, O. F. Rana, R. Tolosana-Calasanz, and C. Pham, "Revenue creation for rate adaptive stream management in multi-tenancy environments," in *Economics of Grids, Clouds, Systems, and Services - 10th International Conference, GECON 2013, Zaragoza, Spain, September 18-20, 2013. Proceedings*, ser. Lecture Notes in Computer Science, J. Altmann, K. Vanmechelen, and O. F. Rana, Eds., vol. 8193. Springer, 2013, pp. 122–137.