Revenue Models for Streaming Applications over Shared Clouds

Rafael Tolosana-Calasanz, José Ángel Bañares
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, Spain
rafaelt@unizar.es,banares@unizar.es

Congduc Pham
LIUPPA Laboratory
University of Pau, France
congduc.pham@univ-pau.fr

Omer F. Rana
School of Computer Science & Informatics
Cardiff University, United Kingdom
o.f.rana@cs.cardiff.ac.uk

***Abstract*—When multiple users execute their streaming applications over a shared Cloud infrastructure, the provider typically captures the Quality of Service (QoS) for each application at a Service Level Agreement (SLA). Such an SLA identifies the cost that a user must pay to achieve the required QoS, and a penalty that must be paid to the user in case the QoS cannot be met. Assuming the maximisation of the revenue as the provider's objective, then it must decide: (i) which user streams to accept for storage and analysis; (ii) how many (computational / storage) resources to allocate to each stream in order to improve overall revenue and minimise cost. In this paper, we analyse revenue models for in-transit streaming applications, executed over a shared Cloud infrastructure under the presence of faulty computational resources. We propose an architecture that features a token bucket process envelop to accept user streams; and a control loop to enable resource allocation, while minimising operational cost.**

***Index Terms*—Streaming workflows; Cloud computing**

I. Introduction

The number of applications that process data in a stream basis has increased significantly over recent years. Such applications include weather forecasting and ocean observation from sensors, text analysis, and more recently data analysis from electricity meters to support "Smart (Power) Grids". Sensor nodes can vary in complexity from smart phones to specialist instruments, and can consist of sensing, data processing and communicating components. Similarly, a Smart Grid comprises a number of power generation companies that gather and use data about electricity consumption, in order to accurately adapt energy generation and tariffs to real-time demand. Smart Grids record electrical energy consumption with digital meters, which periodically stream this information in real time to utility companies.

Data streams in such applications are generally large-scale and distributed, and generated continuously at a rate that cannot be estimated in advance. Data elements are streamed from their source to their sink, and may typically be processed at distributed nodes *in transit*, rather than accomplishing it entirely at source / destination. The benefit of such an approach is many fold: (i) to reduce power consumption at source (which may have limited battery capacity) and sink (which may have limited data storage space); (ii) enable the outcome of data analysis to be shared between multiple users; (iii) alter the processing rate at intermediate (in transit) nodes to achieve a particular QoS requirement; (iv) combine data streams with archived data at intermediate nodes; (v) enable fault tolerance to be supported at intermediate nodes – thereby providing an overall resilient infrastructure that masks faults generated due to the generation of large data volumes (referred to as data inflation) or failure of resources involved in data processing. We describe revenue models for such in-transit analysis and demonstrate how fault tolerance can be used in this context.

Various existing works [1], [2], [3], [4] identify how Cloud infrastructures can be used to support data stream analysis, where each stream must be isolated from another and for the underlying coordination mechanism to adapt the infrastructure to either: (i) run all instances without violating their particular Quality of Service (QoS) constraints; or (ii) indicate that, given current resources, a particular instance cannot be accepted for execution. The QoS demand of each stream is captured in a Service Level Agreement (SLA) – which must be pre-agreed with each in-transit node prior to analysis. Such an SLA identifies the cost that a user must pay to achieve the required QoS and a penalty that must be paid to the user if the QoS cannot be met. In [3], we proposed a system architecture that enforces QoS, measured exclusively in terms of throughput, for the simultaneous execution of multiple streaming applications, expressed as workflows over a shared in-transit, data processing infrastructure. We then considered a "data acceptance rate", different from the physical link capacity connecting two processing stages, to support admission control. The objective was to prevent one workflow stream from affecting the QoS properties of another by monopolizing computing resources. The token bucket (TB) model [5] was used to regulate on a per-stream basis the input (data injection rate) of each workflow stage.

We investigate two aspects in this work – understanding revenue models for in-transit analysis and the impact on faults on such revenue models. We extend the architecture in [3] so that multiple workflow streams can be executed simultaneously on a shared Cloud infrastructure, and their QoS (throughput) can be enforced under the presence of *faulty* computational resources. A faulty resource can distort an application's performance –due to the overhead of the fault tolerance mechanism and therefore an SLA violation (resulting in a penalty the must be paid by the provider).

We assume a workflow is composed of a sequence of stages and datasets are transmitted through the stages following the pipeline streaming model of computation [6]. Each workflow stage is mapped to a node in the infrastructure, though a node can enact more than one workflow stage. Using this approach, *each node* is able to *self-regulate* its behaviour dynamically. The remainder of this paper is structured as follows. Section II describes revenue models for in-transit analysis and Section III the system architecture based on the token bucket model. The extension of the architecture with a fault-aware dynamic resource provisioning model is presented in Section IV. Section V shows our evaluation scenario and simulation results. In Section VI, related work is briefly discussed. Finally, conclusions are given in Section VII.

II. Revenue Models for in-transit Analysis

In-transit analysis provides a useful abstraction for separating data capture/use and analysis, enabling different actors (i.e. service providers) to be involved in each of these processes. Hence data capture may be carried out by a different actor compared to subsequent analysis – enabling multiple capabilities from different actors to be combined at different costs. Each actor may differ in their ability to undertake particular types of analysis that meet varying QoS constraints – leading to different payments that must be made to them by a user to achieve the overall operation. In our formulation of this problem, we first consider a user centric view, identifying the cost that a user must pay one or more providers to perform analysis on a data stream.

For instance, consider sensor S providing a data stream at cost $c(S)$ – this cost may be for a subscription paid by a user to access a data stream, a one off payment to stream data for a particular time period, or a licence cost incurred as part of an analysis operation. Once data from the sensor is available, it must subsequently be stored and analysed using various analysis functions, identified as O_i (for the i^{th} function), made available by a provider P. The cost for carrying out operation O_i by provider $P1$ or $P2$ may be represented as $c(O_i)_{P1}$ and $c(O_i)_{P2}$ respectively. The function $c()$ is used to calculate the price paid by a user for carrying out an operation and may vary over time. A provider may also use a posted price for an operation they can provide a user per unit time of resource use (as currently undertaken by Amazon.com in their EC2 and S3 services), or the price may be based on demand for a particular operation. This cost may also be negotiated between the user and the provider. How such a price is set is not the focus of this work, our primary interest is in identifying that such a cost exists and must be made known to the user. The operator O_i in the context of data stream analysis can include min/max calculations on the data stream, an event analysis, a summarisation of data over a time window, etc. The total cost incurred by a user is therefore: $c(S) + min(\forall_j \sum_{i=1}^{k} c(O_i)_{Pj})$ for k in-transit operations carried out on a single data stream by j possible providers.

We can also consider a provider centric view of costs incurred to provide function O. Where a shared Cloud infrastructure is being used, a provider may serve multiple users using a

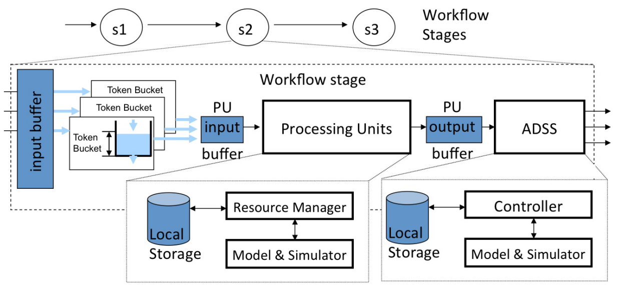

Fig. 1. Workflow System Architecture: the elements of a node

common resource pool through a "multi-tenancy" architecture, or offer multiple functions over their shared infrastructure to one or more users. In both cases, the revenue for the provider is the sum of all the prices charged to n users for accomplishing m operations O, $\sum_{a=1}^{n}\sum_{b=1}^{m} Pr(O_{ab})$. The provider in turn incurs in a cost for performing such operations, $c(O_{ab})$, but can also incur in a financial penalty $PSLA_a$ for user a when the QoS targets, identified in the SLA of user a, are not met. If we assume the objective of the provider as to maximise revenue, then it must decide: (i) which user streams to accept for storage and analysis; (ii) how many resources (including storage space and computational capacity) to allocate to each stream in order to improve overall revenue (generally over a time horizon). Both of these considerations are based on the SLA that a user and provider have agreed to. However, from a pure financial perspective, a provider will incur in SLA penalisation for a user a, when the cost of allocating the required resources is strictly higher than the penalisation. Therefore, by minimising the cost either due to allocation of resources or to SLA penalisation, we get the following benefit function for the provider: $\sum_{a=1}^{n}\sum_{b=1}^{m} Pr(O_{ab}) - min(\sum_{a=1}^{n}\sum_{b=1}^{m} c(O_{ab}), \sum_{a=1}^{n} PSLA_a)$. Our discussion in subsequent sections takes a provider centric view to achieve a particular revenue target. This is undertaken by supporting admission control via a token bucket to achieve (i), which is explained in section III; and a control loop based architecture for minimising cost and enabling resource allocation to achieve (ii) that is shown in section IV.

III. System Architecture

The system supports the enactment of multiple stream workflows simultaneously, each having different QoS requirements (measured in terms of throughput). We assume that i) data transmissions required for meeting QoS, on average, do not exceed the network bandwidth available, and ii) the required computations on average do not exceed the computational power of the resources available.

A stream workflow is composed of a sequence of stages and datasets are transmitted through the stages following the pipeline streaming model of computation [7]. Within a streaming based workflow, it is often useful to identify a "data acceptance rate", which identifies the rate at which a workflow stage can receive and process data. This is often different from the physical link capacity connecting two workflow

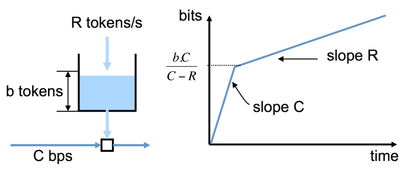

Fig. 2. Token bucket: principle (left), rate enforcement (right)

stages. In order to keep the workflow independent of the resources used to subsequently enact it, a workflow stage needs to be mapped to one or more nodes, and for the sake of simplicity, this is arranged by the user, rather than by a scheduler. Accordingly, nodes can offer different services and are allowed to perform more than one workflow task and may have multiple computing resources available.

A node, as depicted in Fig. 1, involves a combination of data access, computation, and data transfer capability. Data access is responsible for regulating the entrance of multiple streams to the computational stage, isolating the rates of different data streams, while enforcing QoS and avoiding a data stream starvation. The data access component is based on the token bucket (TB) model [5] for traffic characterisation. The TB model supports a variable data rate and burstiness while enforcing a predefined (negotiated) mean data acceptance rate.

It is characterized by 3 parameters shown in Fig. 2: b, R and C that are, respectively, the size of the bucket, the token generation rate and the maximum line/processing capacity. The token bucket can contain b tokens and may be full at initialization time. In practice, in the discrete model, a data packet of S bits can only be sent when there are at least S tokens in the bucket. Tokens are generated and introduced in the bucket at the rate of R tokens/s. R typically represents the mean rate that will be negotiated between the customer and the provider at their SLA. When there are enough tokens in the bucket, a user can send at the rate $C > R$, otherwise the data rate is R. When the user sends at a rate $r < R$ then generated tokens will build up in the bucket for future usage. In this way, a token bucket supports bursts of traffic up to a regulated maximum, enforcing on a long term basis the negotiated rate R. In our architecture, the TB regulates the access to the computational resources, a TB stores data elements from a stream and forwards them to the computational phase of a workflow stage at a predefined rate. This technique represents a flexible mechanism for traffic characterization and enforcement, and enables isolation of workflow streams. A more detailed description of the architecture and the traffic enforcement can be found in [3]

The computational phase, called Processing Unit (PU), is a computing resource container, whose resources can vary in granularity from being a single machine, a dynamically modifiable cluster to multiple clusters –accessible through a resource management system. At a PU computation is performed by utilizing multiple resources in parallel. On the other hand, the transfer component, the Autonomic Data Streaming Service (ADSS) [8], handles transmission of data to the following node in the infrastructure. The ADSS can detect a network congestion between two nodes and react to it by reducing the data transmission rate over the network and temporarily storing data onto disk (thereby avoiding data loss). Finally, it should be observed that there is an input buffer before each component.

IV. Dynamic Resource Provisioning Model

We assume that the proposed architecture is attempting to enforce the end-to-end throughput (the primary QoS criteria we consider here) for each workflow stream: each application i is streaming data elements into the system at an average rate of R_i (as specified in the SLA by the token bucket parameter R_i). Hence, for each workflow stage, it is necessary to identify the data storage and processing requirements – derived from the overall throughput requirement of the workflow. These requirements are subsequently used to identify the size of buffers needed per node. We assume that these requirements are either known by the user enacting the workflow or derived from prior runs of the workflow (and refer to these as the Service Level Agreement (SLA) established with each node). However, what may not be known in advance is whether the occurrence of a fault in a resource may distort the previous estimations.

A. Faults, Scientific Workflows, and QoS

Faults may arise due to hardware and network failures, and software or application errors. We consider the fault handling activity from an event-condition-action perspective. Consider f_i being a single fault (hardware/software), and $\{f\}$ a set of faults, leading to a *known* event/error e_i. The event causes a single action a_i or a set of actions (executed in some sequence) $\{a\}$ to be invoked to overcome the effect of the fault (undertaken using an automated system or by a human user). This can be expressed as:

$$(f_i|\{f\}) \rightarrow^{d_1} e_i \rightarrow^{d_2} \{m\} \rightarrow^{d_3} (f_i|\{a\}) \rightarrow^{d_4}$$

where d_i represents a time duration, with (d_1) representing the time after which a fault leads to an error message, (d_2) the time to *detect* the error message using one or more monitoring tools m, (d_3) the time to select, invoke, and accomplish a *recovery* (corrective) action, and (d_4) the time over which the action must execute for the system to recover from the fault. Therefore, considering a task i at a workflow stage j with an average execution time in stage j without failures of t_{ij}, that is failing up to k times for completion, the overall execution time for i is: $tf_{ij} = k * (t'_{ij} + d_1 + d_2 + d_3 + d_4) + t_{ij}$, with $t'_{ij} < t_{ij}$, and t'_{ij} represents normal execution time of task i until a fault happens, and stops normal execution.

A workflow environment can generally consist of multiple tiers –such as third-party resource management, third-party middleware-supported data distribution and workflow enactors, and user front ends (portals). Each of these tiers are subject to faults, and each may support their own fault tolerance capabilities. For instance, a resource manager may detect a local fault and invoke an action without exposing the fault to the user portal (for instance). Some faults can be masked to the workflow-application level by different fault tolerant mechanisms incorporated at the underlying architectural tiers, namely middleware and resource. However, as this is not always possible, to improve the fault tolerance of the overall system, the workflow-level also needs develop its own fault tolerant mechanisms [9], [10]. These mechanisms [11] mainly consists of: a fault detection component, which incorporates fault detection algorithms with different degrees of sophistication, leading to different cost; a fault identification component, which according to information gathered from monitoring, once a fault is detected, can suggest the best recovery action; and a fault correction component, which supports a variety of mechanisms for the recovery from a fault.

Either a fault masked by underlying third-party tiers or masked by our workflow system, a faulty resource may distort the estimated performance, due to the overhead of the fault tolerance mechanism. As a consequence, this may lead to unexpected delays of a task, and in turn to a QoS degradation for the workflow the task belongs to. Additionally, this performance degradation may also affect the QoS of other workflows that have data elements in queue for processing, as their elapsed time in queue may be increased. On the other hand, in case a recovery of a resource is not possible or in case the resource is not available, the number of resources at the PU of a node also needs to be increased.

Therefore, the workflow system must incorporate a dynamic resource provisioning mechanism that in case of fault occurrence in a resource, allows the system to re-act, incorporating more resources, and subsequently after a period of time, in case of overprovisioning is achieved, releasing them.

B. Dynamic Resource Provisioning Mechanism

The processing time for a data stream element depends on the nature of each data element (i.e. as a consequence of processing, data can vary in size), the computation involved (i.e. some computations may behave differently under different scenarios), and the behaviour of the computational resources utilised (i.e. failures occur). Although the resources that a PU can handle can vary in granularity from being a single machine, a dynamically modifiable cluster to multiple clusters –accessible through a resource management system, for the sake of simplicity of our provisioning model, we assume here i) homogeneity at the resources contained in a PU, and ii) no data size variations within the elements of a stream while processing them. We also assume no prior knowledge of the fault occurrence at our system.

An estimation of the minimum (initial) number of resources $Num\hat{R}es_j$ required at stage j can be derived from the agreed SLA and historical past executions without failures. As discussed earlier in this paper, the agreed SLA establishes that each application i is going to feed data elements into the system at an average rate of R_i (specified as the token bucket parameter R_i). In order to enforce end-to-end QoS, each node j tries to maintain an output rate of R_i, and to stream data to the following node. On the other hand, t_{ij} is the time required for executing task i on stage j. The inverse of t_{ij} is actually the maximum output rate possible (without considering failures) for each node. We denote $\hat{\delta_{ij}}$ as an estimation of this output rate node, estimated from historical past executions without considering failures nor their overheads: $\hat{\delta_{ij}} = AVG(1/t_{ij})$ In consequence, in order to maintain R_i as the output rate, the minimum number of resources required at stage j can be obtained from:

$$Num\hat{R}es_j = \sum_{i=1}^{n} R_i/\hat{\delta_{ij}}$$

In the event of fault occurrence while processing a data element of i at stage j, the time required for processing will be greater, then the processing output slower than estimated, and this may provoke that the effective output rate is slower than R_i. In order to avoid such a circumstance, a rule-based *control loop* has been introduced at each node. In case of a monitored QoS degradation for stream i at j, the control either i) allocates an extra number of computational resources, or ii) assumes the SLA penalisation for i. The action that minimises the cost will be taken. The first action seeks to enforce QoS for stream i, and the rest of applications. However, when the penalisation is assumed for i, there is a QoS degradation for i, but the delay incurred by the faulty computational resource may also affect other applications. In such a case, the control at this step periodically measures the risk of QoS degradation in other applications. Then, for each application near to be degraded (below a threshold), the system estimates the minimum cost from assuming SLA penalisations, or adding the extra computational resources. The required number of resources similarly as before:

$$Num\hat{R}es_j = \sum_{i=1}^{n} R_i/\delta_{ij}$$

where δ_{ij} now represents the inverse of the monitored execution time (i.e it may include fault occurrence, and its derived overhead). Again, the control will only allocate extra resources when the sum of all the SLA penalisations incurred is strictly higher than the cost of allocating more resources. Additionally, we also propose to monitor the input data buffer, in order to release resources. When the input buffer occupancy is below a threshold, which could be due to a number of applications streaming data below the agreed R_i, the number of resources at the PU can be diminished.

Finally, in order to avoid over reactions to any variation difference in the output rate, the rule-based controller guarantees that, before effectively modifying the number of allocated resources, there is an effective deviation tendency through a previously established, and customisable period of time.

V. Evaluation scenario

In this section, we evaluate the influence of faulty computational resources in the QoS of applications as described in Section IV-B. To illustrate the key ideas, we consider a scenario in which two workflows wf_1 and wf_2 are executed simultaneously over two shared nodes. They have respectively negotiated an average arrival rate and throughput of R_1=30 and R_2=15 data values/s. Each PU component in a node (as

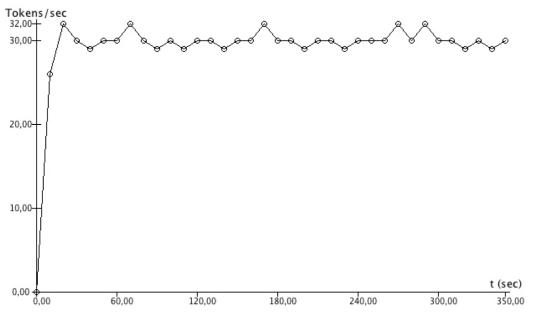

Fig. 3. $Wf1$ input rate

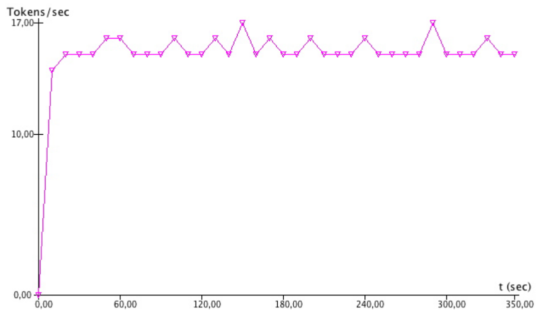

Fig. 4. $Wf2$ input rate

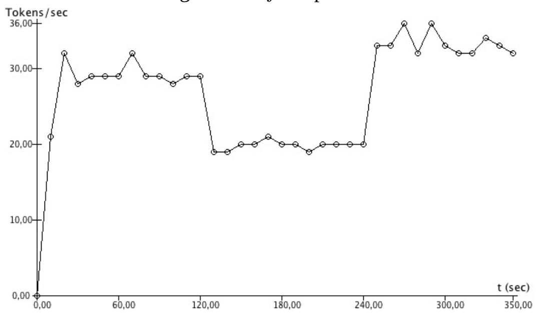

Fig. 5. $Wf1$ throughput without adaptation to processing rates

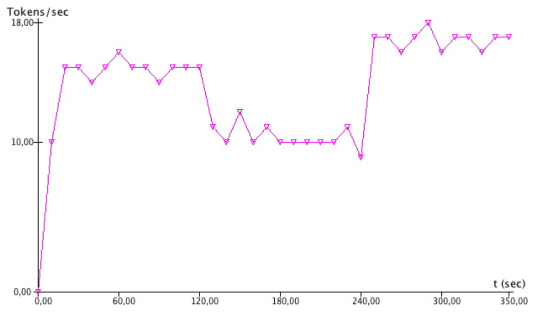

Fig. 6. $Wf2$ throughput affected by processing rates of $Wf1$

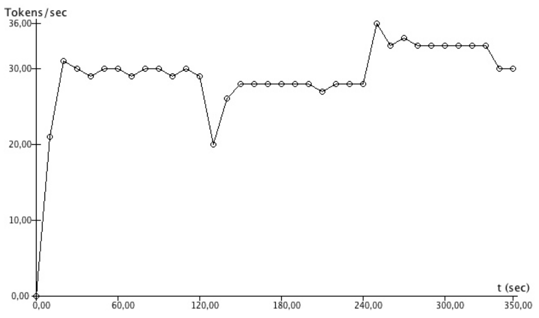

Fig. 7. $Wf1$ throughput with adaptation of resources to processing rate variation

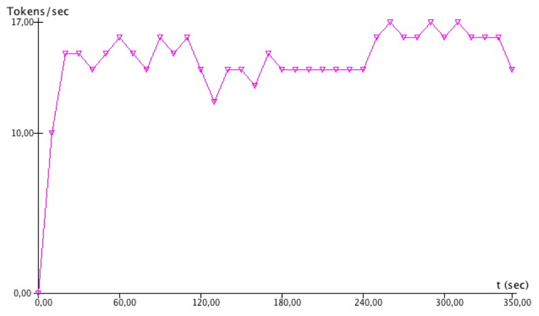

Fig. 8. $Wf2$ throughput with not affection by processing rate variation of $Wf1$

illustrated in Fig. 1) initially contains 5 identical resources. Users assumed that each resource would process 10 token/s (a token represents a data value). The overall processing capacity provided at each node by the provider is 50 data values/s, which is 5 data values/s more than the total sum of the upper average processing rate of both workflow instances. We assume that the network bandwidth between nodes is enough for meeting the QoS requirements. The ADSS can transmit at the rate of 300 tokens/s in our evaluation scenario.

Figs. 3 and 4 show the data injection rate of the applications according to the agreed rates, and shaped by the token bucket. Figs. 5 and 6 show the output rates of wf_1 and wf_2 with no addition of resources in case the fault tolerance mechanism introduces extra processing overheads. In this simulation, we can observe, as a consequence of the fault tolerance mechanism, that the processing rates for wf_1 at the first stage is reduced to 5 tokens/s between time 120s and 240s. Resources are provided to injected tokens assuming all tokens require the same processing time. However, in this new scenario, tokens coming from wf_1 requires more processing time. Fig. 5 shows how throughput of wf_1 falls to 20 t/s at 120s, and Fig. 6 shows how this variation affects the other applications sharing the resources.

Figs. 7 and 8 show the output rates of wf_1 and wf_2 with the provisioning of resources mechanism presented in Section IV-B. The control loop triggers the addition of new resources when the difference of input and output rates of wf_1 in the first stage is over a given threshold. Without loss of generality, we assume here that the penalization cost is larger than the additional resource cost. In this case, the control loop adds two additional resources to the first stage. At time 240s, wf_1 recovers its initial data processing rate of 10 tokens/s. The control loop returns resources when the number of resources show more capacity than the input rate. We can also see that though the fault tolerance mechanism affects the throughput of the other workflows, the impact here is smaller and the average throughput is maintained. The figures also show that the processing rate after this variation is a little over the input rate because of the tokens that were built up in the buffer of the PU.

VI. Related Work

Although workflows, stream and event processing were considered as three separate threads of research in intensive data applications, they share a number of important similarities and challenges such as scalability, fault tolerance and performance that promotes their consideration synergistically [12]: **Data Stream Management System (DSMS)** shifts the paradigm of DBMS processing directly incoming streams instead of storing them first. These works focus on performance by restricting the language in which they can be programmed to graphs of operators with well-defined semantics. This allows the systems to automatically rewrite or compile the specified stream pipelines to a more efficient version. Scalability and query distribution are considered in Aurora [13], Borealis [14] and Stream Cloud [15]. **Complex event processing (CEP)** has seen a resurgence in the last few years, though the need for events, rules, and triggers was accomplished more than two decades ago. Examples of CEP are SPADE/IBM InfoSphere Streams, Esper and DROOLS Fusion [16]. **Scientific Workflows**: they have emerged as a paradigm for representing and managing complex distributed computations. Recently, how to extend traditional state-based workflow management techniques and pipelines with the necessary features to integrate streaming data services has taken relevance[2], [6].

The main difference of workflow based stream applications with DMS and SPE is the focus on the composition of heterogeneous black box services. Park and Humphrey [17] make use of a token bucket-based data throttling framework for scientific workflows that involve large data transfers between tasks. In [8], we propose a superscalar pipeline to enforce the QoS of multiple workflow instances in a shared infrastructure. Various workflow systems are currently used for scientific applications – such as Triana [18], Kepler [19] and Taverna [20] – with support a data streaming pipeline.

VII. Conclusions

There is an emerging interest in processing data streams in shared Cloud infrastructures. Data elements are streamed from their source to their sink, and may typically be processed at distributed nodes in transit. In this paper, we consider the execution of simultaneous data stream applications on a shared Cloud, under the presence of faulty computational resources. Our system captures the QoS (expressed in terms of throughput) for each application at a SLA. The SLA identifies the cost that a user must pay to achieve the required QoS, and a penalty that must be paid to the user in case the QoS cannot be met. Our aim is: i) to enforce QoS for each application, ii) analyse revenue models for in-transit streaming applications, under the presence of faulty computational resources, so that the operational cost is minimised, and the revenue maximised. To meet such objectives, we propose an architecture that features a token bucket process envelop to shape data trhottling, and a rule-based control loop to enable resource allocation, while minimising cost. The control loop monitors QoS for each application and, when there is a QoS degradation (i.e. due to a fault overhead), chooses the action with the minimum cost: either assume the SLA penalisation or to allocate more computational resources.

Acknowledgment

This work was partially supported by projects TIN2010-17905 (Spanish Ministry of Education and Science), FEDER/POCTEFA 35/08 PIREGRID and Aquitaine-Aragon OMNIDATA.

References

[1] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna, “Adaptive rate stream processing for smart grid applications on clouds,” in *2nd Intl workshop on Scientific cloud computing*, ScienceCloud ’11. New York, NY, USA: ACM, 2011, pp. 33–38.

[2] 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 Intl Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011), May 2011, Newport Beach, USA*, 2011.

[3] R. Tolosana, J. A. Bañare, C. Pham, and O. Rana, “Enforcing QoS in scientific workflow systems enacted over cloud infrastructures,” *JCSS*, 2012.

[4] 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 *1st Intl Workshop on Data-intensive Process Management in Large-Scale Sensor Systems (DPMSS12): From Sensor Networks to Sensor Clouds*, 2012.

[5] C. Partridge, *Gigabit Networking*. Addison-Wesley, 1994.

[6] B. Biörnstad, *A workflow approach to stream processing*, 2008. [Online]. Available: http://books.google.es/books?id=_NLRSAAACAAJ

[7] C. Pautasso and G. Alonso, “Parallel computing patterns for Grid workflows,” in *HPDC06 Workshop on Workflows in Support of Large-Scale Science (WORKS06) June 19-23, Paris, France*, 2006.

[8] 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.

[9] R. Tolosana-Calasanz, J. A. Bañares, O. F. Rana, P. Álvarez, J. Ezpeleta, and A. Hoheisel, “Adaptive exception handling for scientific workflows,” *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 5, pp. 617–642, 2010.

[10] R. Tolosana-Calasanz, J. A. Bañares, P. Álvarez, J. Ezpeleta, and O. F. Rana, “An Uncoordinated Asynchronous Checkpointing Model for Hierarchical Scientific Workflows,” *JCSS*, 76(6), pp. 403–415, 2010.

[11] R. Tolosana-Calasanz, M. Lackovic, O. F. Rana, J. A. Bañares, and D. Talia, “Characterizing quality of resilience in scientific workflows,” in *6th workshop on Workflows in support of large-scale science*, WORKS ’11. NY, USA: ACM, 2011, pp. 117–126.

[12] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*, 1st ed. Springer, 2009.

[13] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik, “Scalable distributed stream processing,” in *CIDR*, 2003.

[14] D. J. Abadi et al., “The Design of the Borealis Stream Processing Engine,” in *2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[15] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez, “Streamcloud: A large scale data streaming system,” in *IEEE ICDCS*, june 2010, pp. 126 –137.

[16] O. Etzion and P. Niblett, *Event Processing in Action*. Manning, 2010.

[17] S.-M. Park and M. Humphrey, “Data throttling for data-intensive workflows,” in *22nd IEEE IPDPS, Miami, Florida USA, April 14-18, 2008*. IEEE, 2008, pp. 1–11.

[18] I. Taylor, M. Shields, I. Wang, and A. Harrison, *Workflows for eScience*. Springer, 2007, ch. The Triana Workflow Environment: Architecture and Applications, pp. 320–339.

[19] T. M. McPhillips and S. Bowers, “An approach for pipelining nested collections in scientific workflows,” *SIGMOD Record*, vol. 34, no. 3, pp. 12–17, 2005.

[20] T. Oinn et al., “Taverna: lessons in creating a workflow environment for the life sciences: Research articles,” *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1067–1100, 2006.