End-to-End QoS on Shared Clouds for Highly Dynamic, Large-Scale Sensing Data Streams

Rafael Tolosana-Calasanz, José Ángel Bañares
Aragón Institute of Engineering Research (I3A)
University of Zaragoza, Spain
rafaelt@unizar.es,banares@unizar.es

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

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

***Abstract*—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. This is especially true when data from multiple sensors is pre-processed prior to delivery to users. Where such data is processed in-transit (i.e. from data capture to delivery to a user) over a shared distributed computing infrastructure, it is necessary to provide some Quality of Service (QoS) guarantees to each user. We propose an architecture for supporting QoS for multiple concurrent scientific workflow data streams being processed (prior to delivery to a user) over a shared infrastructure. We consider such an infrastructure to be composed of a number of nodes, each of which has multiple processing units and data buffers. We utilize the "token bucket" model for regulating, on a per workflow stream basis, the data injection rate into such a node. We subsequently demonstrate how a streaming pipeline, with intermediate data size variation (inflation/deflation), can be supported and managed using a dynamic control strategy at each node. Such a strategy supports end-to-end QoS with variations in data size between the various nodes involved in the workflow enactment process.**

I. Introduction

Sensor nodes, which consist of sensing, data processing and communicating components, leverage the idea of (wireless) sensor networks (WSN) based on a collaborative effort of a large number of nodes. These sensors have the unique capability to interact with the physical world making possible the monitoring and control of actions to be performed on the real world. Fig. 1 depicts such a global scenario where various heterogeneous surveillance & monitoring systems could be connected together to form a global sensing infrastructure for collecting, sharing and analyzing a huge amount of data in order to take proper coordinated actions.

Once data from monitoring sensors has been acquired, it must be pre-processed and distributed to a number of users. For instance, in the context of environment monitoring, the same data may need to be: (i) summarized; (ii) fused with other data; (iii) enhanced with geo-location and access rights, prior to delivery to a given user (such as a government laboratory, a researcher in climate studies, etc) – thereby leading to data "inflation" (for (ii) and (iii)) or "deflation" (for (i)). An infrastructure that enables the same data to be processed in various ways, in-transit, prior to delivery to a group of users (with each user having a different data requirement) remains an important challenge. As data are collected continuously

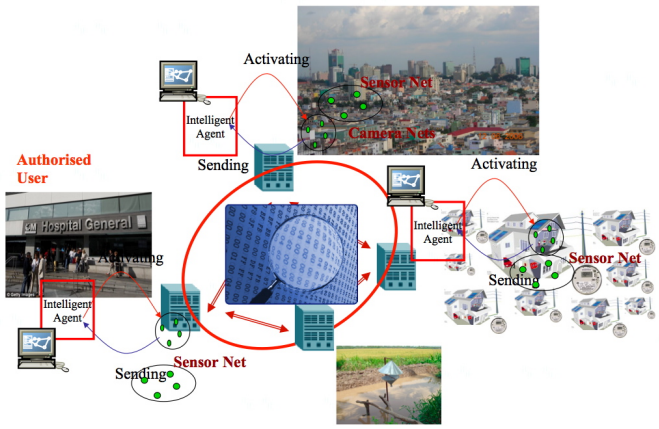

Fig. 1. From pervasive systems to global sensing.

at sensors, intermediate resources must be allocated to store and process these data streams in a scalable manner. In [1], we highlighted the need for supporting Quality of Service (QoS) enforcement mechanisms, so that the requirements of the different applications being executed on the same shared Cloud infrastructure can be met. This is especially true for applications that must stream data through analysis processes [2] and undertake *in-transit* analysis from data source to sink. In such a scenario, each application workflow instance 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 QoS constraints; or (ii) indicate that, given current resources, a particular instance cannot be accepted for execution.

We propose a system architecture which enforces QoS, measured exclusively in terms of throughput, for the simultaneous execution of multiple workflows over a shared, in-transit, data processing infrastructure. We then consider a "data acceptance rate", different from the physical link capacity connecting two processing stages, to support admission control. Fig. 2 depicts this architecture, where it is assumed that the network bandwidth for data transfer is not the bottleneck in the system. 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[3]. Each workflow stage is mapped to a node in the infrastructure, though a node

can enact more than one workflow stage. The token bucket (TB) model [4] is used to regulate on a per-stream basis the input (data injection rate) of each workflow stage (regulation element in Fig. 2). The purpose is to prevent one workflow stream from affecting the QoS properties (mainly throughput) of another by monopolizing computing resources.

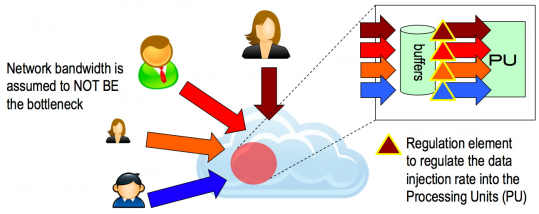

Fig. 2. Shared infrastructure with regulation element per flow

This paper extends [1] to address large-scale sensor-based data streams. In addition to a variable data injection rate from the sensing infrastructure, data inflation/deflation phenomenon may occur at various stages of the workflow system. Data inflation implies that an intermediate node generates a much greater amount of data than it consumes (opposite behavior for data deflation). Therefore, it is mandatory to efficiently manage data inflation/deflation within a streaming workflow to strictly ensure that QoS targets can be met for each stream on an end-to-end basis. An *a priori* knowledge of these data size variations is typically application and dataset dependent and difficult to estimate. It is therefore necessary, in the more general case, to identify how data size impacts buffer sizes, the number of computing resources required and the number of concurrent streams that can co-exist at a node. The latter two of these are tunable parameters that can be adjusted dynamically in our proposed model.

Our key contribution in this work is to add a dynamic control strategy at *each node* to provide dynamic resource utilisation per node by featuring a TB component with autonomic parameter tuning. Using this approach, *each node* is able to *self-regulate* its behaviour as the intermediate data size changes dynamically. The remainder of this paper is structured as follows. Section II reviews enforcement techniques based on the TB model and presents our architecture. Section III introduces data inflation/deflation and the impact this has on our control strategy. We explain the challenges due to data inflation/deflation on the TB configuration. Section IV presents our evaluation scenario and the simulation results. In Section V, related work is briefly discussed. Finally, conclusions and future work are given in Section VI.

II. Enforcing QoS in Superscalar Pipelines

A. Enforcing QoS with Token Bucket model

In our system, the TB model is integrated within an infrastructure node, enacting one or more workflow stages. A TB is characterized by 3 parameters: b, R and C that are respectively the size of the bucket, the token generation rate and the maximum line/processing capacity. The TB model supports a variable data rate and burstiness while enforcing a predefined (negotiated) mean data acceptance rate. In our architecture, 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.

B. Workflow System Architecture for enforcing QoS

The architecture proposed in [1] allows multiple workflows, each having different QoS requirements, to be supported by a workflow enactment engine. 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. In order to keep the abstract 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. As depicted in Fig. 3, the mechanisms for enforcing QoS and avoiding data loss are: i) a TB(one per workflow stream), ii) a processing unit (PU), which is a computing resource container where the number of resources it contains can be modified on-demand, and iii) an Autonomic Data Streaming Service (ADSS), which handles transmission of data to the following node in the infrastructure. It should be observed that there is an input buffer before each component.

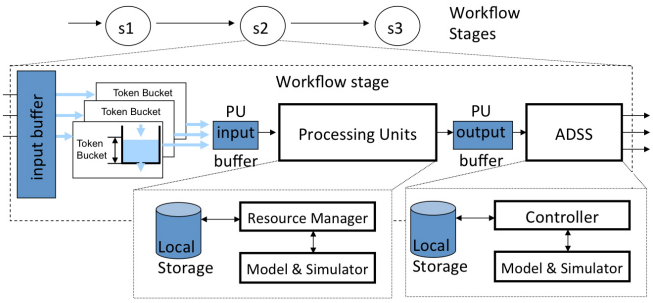

Fig. 3. Workflow System Architecture: the elements of a node

The TB regulates the entrance to the PU, isolates the rates of different data streams, while enforcing QoS and avoiding a data stream starvation. At a PU computation is performed by utilizing multiple resources in parallel. Resources can be added to or removed from the PU at runtime. At the output of each workflow stage, we make use of the ADSS [5] for submission of data to a subsequent node. 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).

III. Data Inflation/Deflation Issues

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. 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 and computing resources 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 is whether for a particular combination of inputs and data sources a large amount of output data may be generated at a workflow stage (leading to data inflation). Conversely, data deflation at a node may lead to an inefficient use of available resources.

A. Data inflation/deflation issues on token bucket

The challenging issues of data inflation/deflation are summarized in Fig. 4 that illustrates the various steps starting from SLA negotiation based on the application's data injection rate (step 1). For simplicity, the figure only shows one flow instance. Step 2 involves instantiating the flow's token bucket parameters at each workflow stage using the initial b and R parameters. Steps 3 and 4 show how incoming data get processed by the first stage leading to data inflation. Subsequently, step 5 identifies the impact on a subsequent workflow stage, where the rate is limited by the token bucket parameters of the stage. This may lead to either data loss or an inability to meet an application's end-to-end QoS.

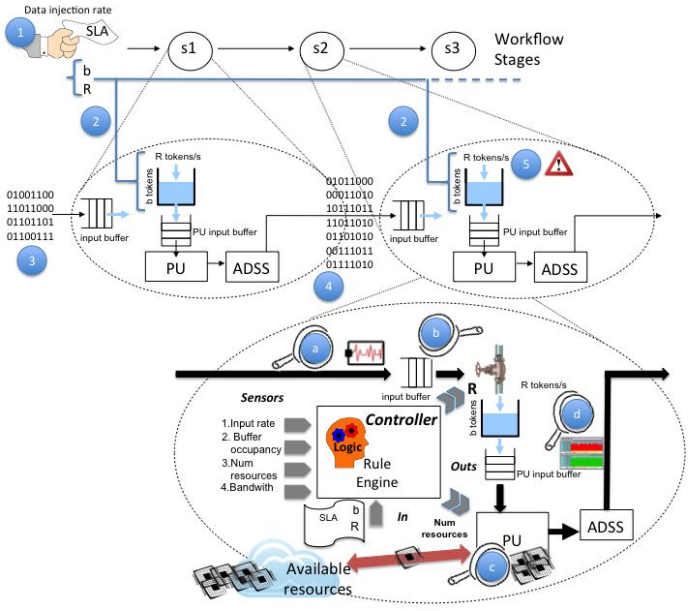

Fig. 4. Example of data inflation at the 2nd stage and detailed self-configuring component.

Pre-defined values of b, R and C cannot overcome effects of data inflation/deflation on resource management or workflow QoS. It is therefore necessary to identify a strategy for altering the processing rate at each node to avoid large buffer overflow or large queue sizes. In consequence, it is mandatory to identify the parameters that can be adjusted at a node based on the computing resources available at the PU, and the data injection rate into the node. The bottom of Fig. 4 shows the control loop to configure the R parameter and the number of resources for each flow instance. For simplicity, the figure shows the regulation of one flow instance.

Each flow instance monitors its input and output rates at each stage at a pre-defined sampling rate (magnifying glass (a) in the figure). Using this initial data, the control strategy is initiated, subsequently recording the input queue buffer occupancy (b), the number of resources in use at the PU (c), and the bandwidth available to transmit data (d).

At each node, the size of each input buffer is chosen in accordance with the agreed requirements of the workflows. For example, the maximum resources by stage limits the maximum processing rate, and therefore the maximum input rate to the next stage. The controller must estimate the buffer size during execution. For example, with an input rate of 100 tokens/s and a buffer size of 2000 tokens, the buffer will be full in 20 seconds. A sampling rate of 10 seconds would be sufficient to control the number of resources and change the output rate if the resources can be deployed in less than 10 seconds.

Depending on the applications' behaviours, i.e. showing signs of burstiness for a period of time, the input buffer capacity can vary. Choosing a threshold for the buffer occupancy can be challenging when heterogeneous applications are being run concurrently. In case the chosen threshold is too high, there is a risk of data loss, when the control loop mechanisms cannot manage to reduce the buffer occupancy. On the other hand, a too low threshold value may lead to inefficient use of resources. When the input buffer size reaches an established threshold, this triggers the controller to initiate one of two possible actions: i) calculate the number of additional resources at the PU needed (based on those available) to process the additional data items generated above rate R. ii) if there are free resources (not being used by other workflow instances), they can be used to increase the R rate of flow associated with this instance. The amount of resources and the rate value will return to their previously agreed values when the input buffer size goes below the threshold. The regulatory actions taken by the controller are implemented by a rule engine. SLAs may be implemented as rules that control the number of resources and the output rate.

B. Integration into a workflow framework

Our approach uses the *Reference net* formalism [6] to specify the workflows and the engine to enact them [1]. Reference nets is a particular class of high-level Petri net that uses Java as an inscription language and extends Petri nets with dynamic net instances, so-called net references, and dynamic transition synchronization through synchronous channels.

Our abstract workflows can have a hierarchical structure consisting of either intermediate nodes or leaf nodes (simple tasks). Fig. 5 shows two abstract workflows, Wf_1 and Wf_2, to be enacted. These workflows represent pipelines as a sequence

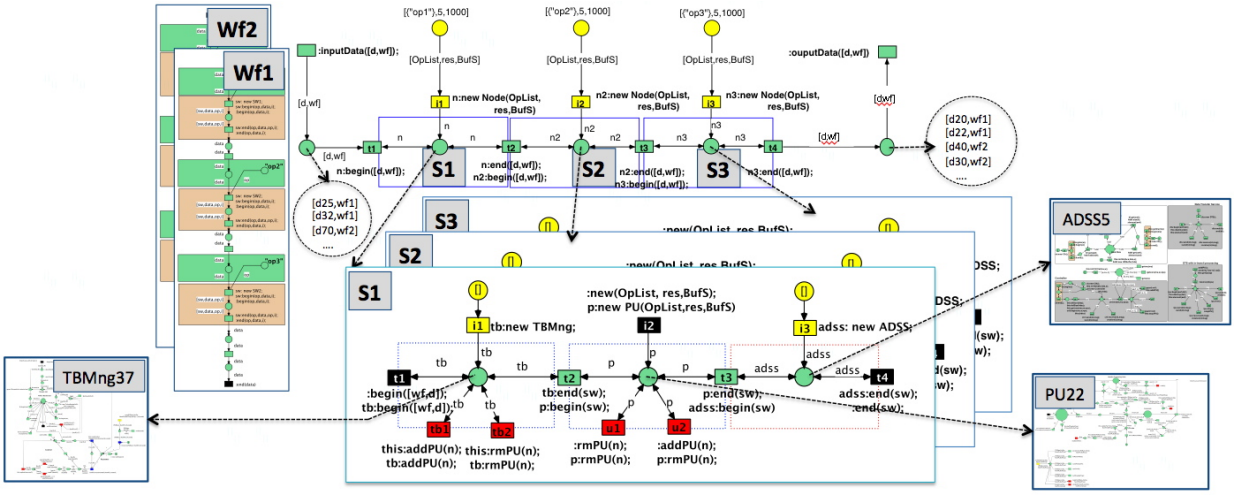

Fig. 5. Autonomic self-configuring component to regulate the stream processing rate at each stage

of operations to process data streams. Elements within a data stream are represented as tokens flowing through the pipeline. Reference nets support different representations [7]: tokens that store remote locations of distributed files, or express structured collections of data. A more detailed description of workflow specification with reference nets can be found in [1].

To enact these workflow specifications we use the super-scalar pipeline [3] model of computation, whereby multiple data elements can be processed in parallel within a workflow stage as long as there are enough available resources. Once multiple workflow instances are created, the workflow engine will use the nets described in Fig. 5 to coordinate their execution. The upper part of Fig. 5 shows three workflow stages $s1, s2, s3$ represented by three nodes. The Reference net for each stage is then explained in the lower part of the figure (we use the terms node and stage interchangeably, as one stage of the workflow maps to one physical node in the enactment process). A node at this level consists of two transitions and a place. Two consecutive nodes, n_i followed by n_{i+1}, share one transition: the final transition of n_i is the initial transition of n_{i+1}. Transitions labelled as i_j are responsible for creating and initialising nodes: the parameters specified in the creational inscription $new\ node(opList, res, bufS)$ are from left to right the list of operations that the resources at the node can perform, the initial number of resources and the buffer size of the PU.

At enactment time, multiple data elements from different workflows are streamed into the PU, introduced one by one at the initial transition via Synchronous Channel $: inputData([d, wf])$. Pairs in the form $[d, wf]$, where d is a data element that belongs to wf's data stream, are introduced in Transition $t1$. d stores either the data itself or a reference to the data element, and wf is a reference to a net instance of Workflow wf. The pair $[d, wf]$ goes through the sequence of nodes and finishes the processing in Transition $t4$.

Each node contains three different components: a token bucket manager, a PU and an ADSS. When a pair $[d, wf]$ enters into the node, it arrives at the token bucket manager component (Transition $t1$). Then, whenever the corresponding token bucket allows the data element to proceed, it enters into the PU. Finally, after the processing, it goes to the final stage which corresponds to the ADSS. Upon completion of the transmission, a data element gets out of the node and enters into the next node (Transition $t4$). Fig. 5 shows how each place of the *node* net references the corresponding `TBMng`, `PU` and `ADSS` net instances. Transitions $tb1$, $tb2$ are Synchronous Channels and allow `TBMng` net to claim or release more resources at the PU for this stage by synchronizing with transitions $u1$, $u2$.

Details about the `PU` net, patterns utilised to map a workflow task to a distributed resource and the `ADSS` net can be found in [5]. This paper focuses on the description of the Token Bucket Manager, represented by the `TBMng` net, and the control loop that regulates the token bucket rate and the number of resources of each workflow instance at each stage.

The Reference net in the upper right part of Fig. 6 implements a token bucket which receives data elements arriving in *input* Transition and leaving at *output* Transition. Once a data element enters into the bucket it is stored in the bf buffer, implemented in this case as a FIFO list, in Place $DataBuffer$. The output Transition is only enabled when there are simultaneously an element in the buffer and a mark in Place $TokenBucket$. A mark in Place $TokenBucket$ will be added by the clock in the bottom right part of the figure at the rate R. Thus, irrespective of the arrival rate of data elements into the token bucket from previous stages, they will only be allowed to proceed to the PU at a constant rate of R. Transition $: update(R)$ modifies the parameter R.

The Reference net on the left part of Fig. 6 implements the token bucket manager component. The upper part of the net forwards incoming data elements to the corresponding token bucket. Each time a data element is injected in a data stream, a reference to the data stream with the agreed values (b, R) arrives in Transition $t1$. If it is the first data element

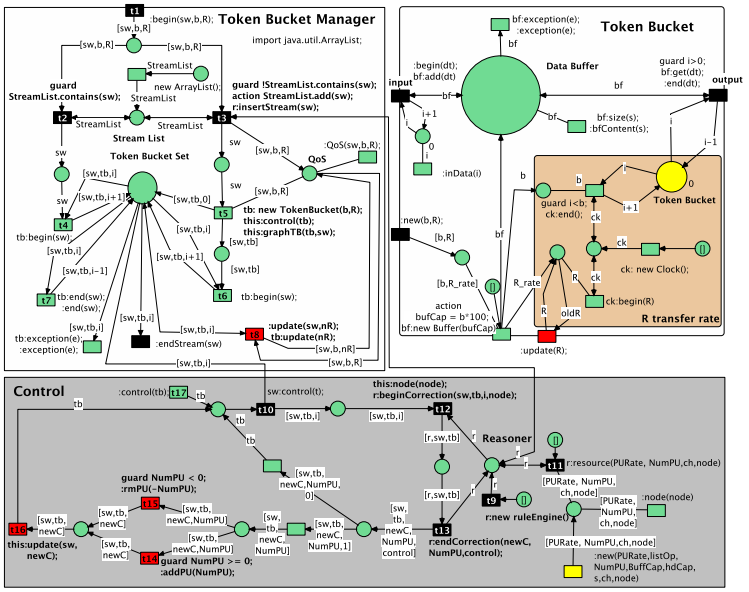

Fig. 6. Token Bucket net and Token Bucket Manager with control loop

of the stream, Transition $t3$ will be enabled and Transition $t2$ disabled. Otherwise, Transition $t2$ will be enabled and Transition $t3$ disabled. In the former case, the new token bucket instance for the data stream will be created in Transition $t5$, and the data element will be added to it when Transition $t6$ fires. In the latter case, the data element will be added to its corresponding $tokenbucket$ instance when Transition $t4$ fires. Finally, once a data element is allowed to proceed, Transition $t7$ is fired and the data element moves to the PU component via Synchronous Channel : $end(sw)$ in Transition $t7$.

The bottom part of the net with grey background is the control loop. A rule engine is instantiated at each stage in transition $t9$. At a sampling rate defined for each workflow instance, a control loop evaluates the actions to be taken and Transition $t10$ initiates this control loop. We implemented the rule engine with JESS (Java Expert System Shell). Configuration data such as SLA information are provided by Transiton $t3$ and Transition $t11$ will insert information about available resources (computing and bandwidth) at this stage. Transition $t12$ executes the rule engine by providing information (input rate, buffer occupancy, etc) to the rule engine as described in Fig. 6. Transition $t13$ will issue a control flag indicating whether it is necessary to modify the token bucket rate or to add/remove resources to/from the PU, or both actions. Transitions $t14$ and $t15$ change the number of resources at the PU. These channels synchronize with $tb1$ and $tb2$ and regulate the number of resources in the PU net. Finally, Transition $t16$ (synchronized with $t10$) changes the rate of the token bucket regulating the throttling rate of the workflow at this stage.

IV. Evaluation scenario

We examine the effectiveness of the token bucket manager control loop, considering use cases found in a number of real world applications, such as stream processing for Smart Grid [8] and LEAD [2] (in environmental data management). To illustrate the key ideas, we abstract these applications to a scenario in which two workflows wf_1 and wf_2 are executed simultaneously over two shared nodes. They have respectively an average arrival rate and a required throughput of 20 and 10 data values/s. They have also negotiated respectively an upper average arrival rate R_1=30 and R_2=15 data values/s. Each PU component in a node (as illustrated in Fig. 3) initially contains 5 identical resources. The average time for an operation depends on the network bandwidth, the number of resources available, the data size, etc. We assume that each resource can process 10 data values/s. Therefore, the overall processing capacity at each node 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. For simplicity, the processing time is also assumed to be the same for each datum. Otherwise, a variable processing rate will only produce throughput variations that are equivalent to data inflation/deflation. 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.

The token bucket size b is set to 20 tokens, that is, during any time period T, the amount of data sent cannot exceed $RT + b$. The data buffer size is set to 100 times the token bucket size to avoid buffer overflow in our evaluation scenario (see Fig. 6). We will conduct two evaluation scenarios. In the first one, we will consider the use of token buckets only at the first processing node, and we will examine the effect of data inflation and adaptation of resources over the next nodes without the token bucket mechanism. In the second scenario we will consider token buckets in all nodes and the use of

different SLA adaptation agreements.

1) Data inflation with rate and resource adaptation only at the first node: The impact on workflow throughput when adapting the number of PUs and the rate of token bucket R is illustrated here. For this propose, the control loop is introduced in the token buckets. The adaptation of resources may be part of a more flexible SLA that may alleviate temporal burstiness, and reduce the size of data buffers. We examine the evolution of throughput when using only adaptive token buckets to regulate the input rate.

In this evaluation scenario data elements are sent to each workflow at the aforementioned rates (20 data values/s for wf_1 and 10 for wf_2), except between time 60s and 140s where wf_1 sends data at a rate of 100 data values/s. wf_1 has an SLA that supports adaptation by adding more resources and by modifying the token bucket rate R at the input node.

Fig. 9 depicts the adaptation of throttling the data rate to the first node. Wf_1 can introduce 10 additional resources. In this case, it increases R, introduces 6 processing units to absorb the additional input rate, and processes data with a throughput of 90 tokens/s. When the input buffer is emptied, the processing units are returned to the pool and the throttling rate is returned to the initial value. A data inflation is also simulated at 360s at the first stage of wf_1 to show its effect on wf_2. Fig. 10 illustrates how the token buckets at the first stage isolate the throughput of wf_2 at the first stage. However, Fig. 11 and 12 show that a single token bucket to throttle data at the input is insufficient. Indeed, a first starvation is produced by the adaptation at the first stage. Fig. 11 shows that the adaptation and the simulated data inflation at the first stage produce data inflations at the second stage, which has not token bucket to shape input rates. Fig. 12 shows that it produces two starvations in wf_2 at the second stage .

2) Self-Adaptative Stage with different SLAs: The adaptive token bucket component is evaluated here. In this scenario, wf_1 has an SLA that seeks adaptation by adding more resources, and modifying the token bucket rate R at each stage. In contrast, wf_2 has an SLA that allows only adapting the token bucket rate R using the free resources. Fig. 13 illustrates the results. The first graph shows the input rate for wf_1. A violation of rate is produced at 60s. The fourth graph shows the input rate for wf_2 with a violation at 600s. The second graph illustrates wf_1's throughput at the first stage, regulated by the token bucket throttling: the first small peak is caused by the allowed burstiness. As the violation persists, the second peak shows how the flexible SLA allows the addition of 6 resources, and the throttling rate is increased until 90 tokens/s. The 3^{rd} peak of 200 tokens/s is produced to simulate a data inflation. The third graph shows how the first violation at the input is solved at the second stage of wf_1 by the token bucket without additional resources, and the data inflation at the second stage introduces additional resources at the second stage of wf_1 until the input buffer of the second stage is emptied.

The fifth and sixth graphs show that all these adaptations of wf_1 do not affect wf_2. The fourth graph depicts an input rate violation of wf_2. The SLA of wf_2 only supports the use of free resources. The fifth graph shows how the first stage increases the rate until 20 tokens/s (5 tokens/s over the initial R of 15 tokens/s). When the buffer is emptied, R returns to the initial value and the throughput returns to 10 tokens/s provided by the input rate. The sixth graph illustrates that it is not necessary to provide adaptations to R for $wf2$ at the second stage.

V. Related Work

Park and Humphrey [9], make use of a token bucket-based data throttling framework for scientific workflows that involve large data transfers between tasks. In contrast to the data parallelism model of computation of Park and Humphrey, we use 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 [10], Kepler [11], [12] and Taverna [13] (amongst many others [14]) – both of which support a data streaming pipeline. In Triana, the streaming model is used by default, and Triana units can be either Web Services or Java executables. Kepler provides a more customisable control management strategy, where a "director" can be used to alter the control flow between components in the workflow. Taverna has recently undergone a radical re-design of the architecture, referred to as Taverna 2 [15]. This new architecture also supports superscalar and streaming pipelining as a model of computation: a producer processor forwards each element as soon as possible to the corresponding consumer processor in the pipeline. In the consumer processor, multiple elements can be processed in parallel as there are multiple threads available (superscalar).

VI. Conclusions and Future Work

Global sensing infrastructures based on large-scale deployment of sensor technology introduce challenging issues for advanced management and processing of continuous data streams. In this paper we propose Cloud technology as a scalable and economically viable solution for heterogeneous surveillance & monitoring systems that present extremely high dynamics in data patterns and processing requirements. We use Token Bucket envelop process to enable running multiple workflow instances over a shared Cloud infrastructure while providing each workflow with a particular QoS requirement. We add a control strategy at each workflow stage to dynamically adjust Token Bucket parameters to adapt the available resources in order to provide QoS on an end-to-end basis, so that variations in data size (data inflation/deflation) between stages can be self-configured by the application. We then use the *Reference net* formalism to both specify the workflows and provide the engine to enact them, therefore proposing a seamless integration of QoS mechanisms for the end-user. Simulations have shown the effectiveness of the approach on test cases inspired by real global sensing applications.

Our future work involves better understanding the relationship between data types, dynamic resource provisioning and admission control of a workflow instance. Another aspect to consider is the adaptive capacity of Token Bucket parameters

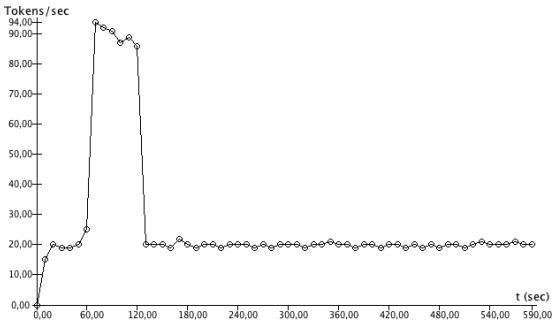

Fig. 7. $Wf1$ input with burstiness

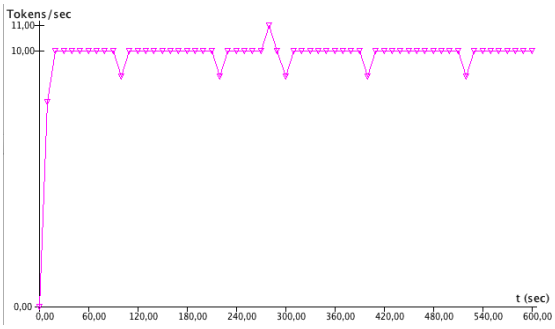

Fig. 8. $Wf2$ input rate

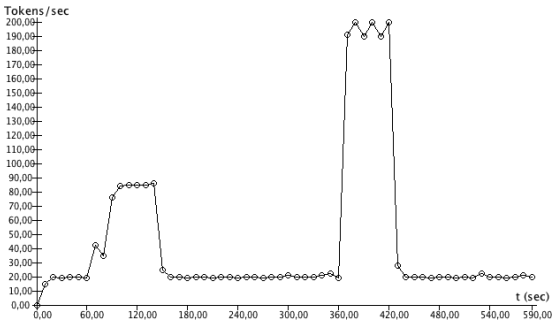

Fig. 9. $Wf1$ throughput at 1st stage with adaptation and inflation

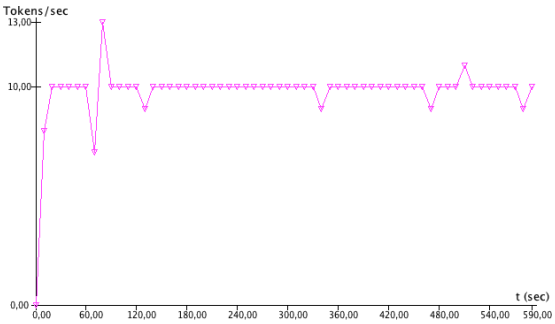

Fig. 10. $Wf2$ throughput at 1st stage

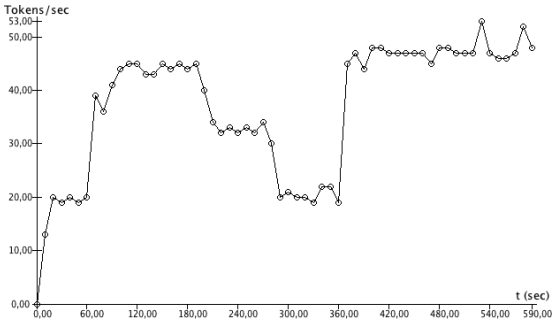

Fig. 11. $Wf1$ throughput at 2nd stage without TB

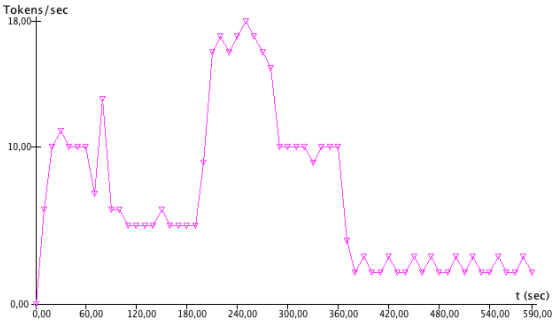

Fig. 12. $Wf2$ throughput at 2nd stage suffering inflation and burstiness

(such as its queue size) [16]. Finally, burstiness introduces a new elastic way to define an SLA for an application to be executed over a shared environment with a mean data injection rate. Data without tokens can be marked or dropped, implying that depending on the application, data can be marked to be lost, to be not processed at this stage, or to be stored in the ADSS node and forwarded to be processed off-line.

REFERENCES

[1] R. Tolosana, J. Banares, C. Pham, and O. Rana, "Enforcing qos in scientific workflow systems enacted over cloud infrastructures," *Journal of Computer and System Sciences*, 2012.

[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] C. Pautasso and G. Alonso, "Parallel computing patterns for Grid workflows," in *Proceedings of the HPDC2006 Workshop on Workflows in Support of Large-Scale Science (WORKS06) June 19-23, Paris, France*, 2006.

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

[5] 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.

[6] O. Kummer, *Referenznetze*. Berlin: Logos Verlag, 2002.

[7] 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.

[8] 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

[9] S.-M. Park and M. Humphrey, "Data throttling for data-intensive workflows," in *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*. IEEE, 2008, pp. 1–11.

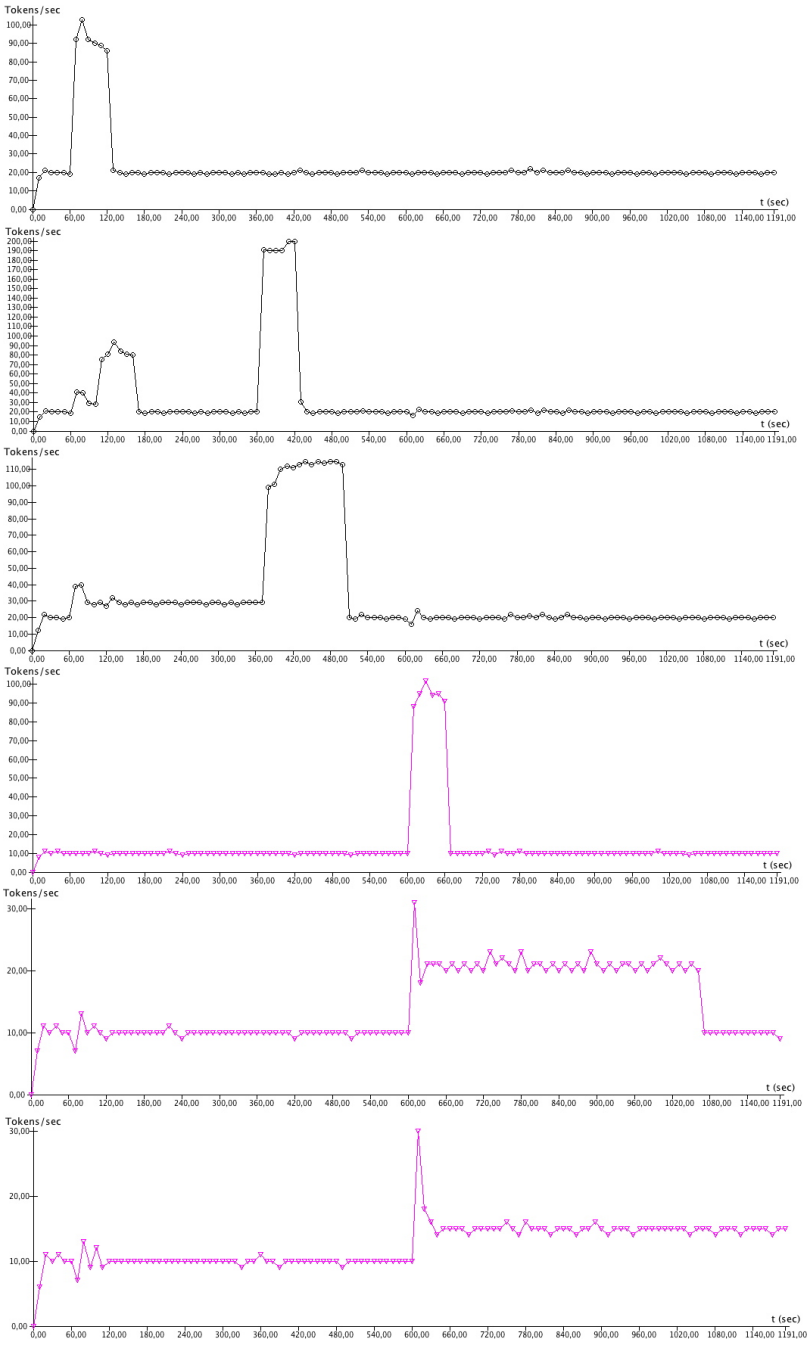

Fig. 13. Throughput rate with self-configuring control and TB at each stage.

[10] 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.

[11] 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.

[12] D. Zinn, S. Bowers, S. Köhler, and B. Ludäscher, "Parallelizing XML data-streaming workflows via MapReduce," *Journal of Computer and System Sciences*, no. To appear, 2010.

[13] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "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.

[14] D. Gannon, E. Deelman, M. Shields, and I. Taylor, *Workflows for eScience*. Springer, 2007, ch. Introduction, pp. 1–9.

[15] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. A. Goble, "Taverna, reloaded," in *Scientific and Statistical Database Management, 22nd International Conference, SSDBM 2010, Heidelberg, Germany, June 30 - July 2, 2010. Proceedings*, ser. Lecture Notes in Computer Science, M. Gertz and B. Ludäscher, Eds., vol. 6187. Springer, 2010, pp. 471–481.

[16] Y.-C. Chen and X. Xu, "An adaptive buffer allocation mechanism for token bucket flow control," in *Vehicular Technology Conference, 2004. VTC2004-Fall. 2004 IEEE 60th*, vol. 4, sept. 2004, pp. 3020 – 3024 Vol. 4.