Revenue creation for rate adaptive stream management in multi-tenancy environments

José Ángel Bañares[2], Omer F. Rana[1], Rafael Tolosana-Calasanz[2], and Congduc Pham[3]

[1] School of Computer Science & Informatics
Cardiff University, United Kingdom o.f.rana@cs.cardiff.ac.uk
[2] Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, Spain rafaelt, banares@unizar.es
[3] LIUPPA Laboratory
University of Pau, France congduc.pham@univ-pau.fr

Abstract. With the increasing availability of streaming applications from mobile devices to dedicated sensors, understanding how such streaming content can be processed within some time threshold remains an important requirement. We investigate how a computational infrastructure responds to such streaming content based on the revenue per stream – taking account of the price paid to process each stream, the penalty per stream if the pre-agreed throughput rate is not met, and the cost of resource provisioning within the infrastructure. We use a token-bucket based rate adaptation strategy to limit the data injection rate of each data stream, along with the use of a shared token-bucket to enable better allocation of computational resource to each stream. We demonstrate how the shared token-bucket based approach can enhance the performance of a particular class of applications, whilst still maintaining a minimal quality of service for all streams entering the system.

1 Introduction

The increasing deployment of sensor network infrastructures has led to large volumes of data becoming available, which are often required to be processed in real-time. In addition, data from these sensors may be streamed in an unpredictable manner (i.e. the availability of data may not be known apriori) with potential bursty behaviour in data generation. Data source (sensor) can vary in complexity from smart phones to specialist instruments, and can consist of sensing, data processing and communication components. Data streams in such applications are generally large-scale and distributed, and generated continuously at a rate that cannot be estimated in advance. Scalability remains a major requirement for such applications, to handle variable event loads efficiently [1].

Multi-tenancy Cloud environments enable such concurrent data streams (with data becoming available at unpredictable times) to be processed using a shared, distributed computing infastructure. This leads to challenges in offering Quality of Service (QoS) guarantees for each data stream, specified in Service Level

Agreements (SLAs). SLAs identify the cost that a user must pay to achieve the required QoS, and the penalty in case the QoS cannot be met. Stream descriptions may, in some cases, provide placeholders in the SLA for data that will be generated at some time in the future. Assuming the maximisation of the revenue as the provider's objective, then it must decide which streams to accept for storage and analysis; and how many (computational / storage) resources to allocate to each stream in order to improve overall revenue. When the real-time requirements demand a rapid reaction, the dynamic provisioning of resource (i.e. from an elastic resource provider) may not be useful, since the delay incurred might be too high. Alternatively, idle resources that were initially allocated for other streams could be re-allocated, avoiding the penalisation.

This paper extends our previous contributions in this area; papers [2–4] describe a revenue-based resource management strategy for bursty data streams on shared Clouds. This contribution extends the token bucket model used previously to enable: (i) the re-distribution of unused resources amongst data streams; and (ii) a dynamic re-allocation of resources to streams likely to generate greater revenue for the provider. These extensions are provided by a direct addition of business rules in the token bucket behavior – as an alternative to using a rule engine alongside a token bucket model, which has a significant performance overhead. The remainder of this paper is structured as follows. Section 2 describes the revenue model and the resource requirements for QoS in data stream processing applications. Section 3 shows the system architecture based on the token bucket model and actions the provider can take to maximize revenue: using a rule-based approach with token bucket model extensions. Section 4 shows our evaluation and simulation results. In Section 5, related work is briefly discussed. Finally, conclusions and future work are outlined in Section 6.

2 Revenue based resource management

We consider a provider centric view of costs incurred to provide data stream processing services over a number of available computational resources (e.g. a pool of virtual machines in an elastic infrastructure). A provider may use a (pre-agreed and reserved) posted price, a spot price (to gain revenue from currently unused capacity), or on an on-demand use (the most costly for the user) for resources, on a per-unit-time basis – as currently undertaken by `Amazon.com` in their EC2 and S3 services. In the case of data stream processing services, this cost may also be negotiated between the user and the provider using QoS criteria. How such a price is set is not the focus of this work, our primary interest is in identifying what are the performance objectives that can be established in a SLA, and what actions the provider can perform to guarantee the agreed QoS and maximize the revenue. A key distinction between batch-based execution on a Cloud infrastructure is that the query/computation and data are generally available before the execution commences. In a streamed application, a query is often executed continuously on dynamically available data. An SLA is therefore essential to identify what a user must pay the provider, often based on a previous

estimation of resources required/used. Conversely, the provider can also utilize previously similar stream processing capability to identify resources required and any penalties paid in the past (for service degradation that violated the SLA). Due to the greater potential variation likely to be seen in stream processing applications, an SLA therefore protects both the user and the provider.

Defining QoS properties in an SLA is very application dependent. In applications such a commercial Web hosting, QoS levels specify parameters such as request rate, for example expressed as served URLs per period; and data bandwidth, that specifies the aggregate bandwidth in bytes per second to be allocated in the contract [5]. In other applications such as video-on-demand, QoS levels may represent frame rates and average frame sizes. In the context of data stream, the analysis can include min/max/avg calculations on a data or sample time window, an event analysis, a summarisation of data over a time window, etc. [6] provides a useful summary of the performance objectives of event processing and their associated metrics (see table 2).

Performance Objectives and their metrics	
Objective Name	Objective metrics
Max input throughput	Max. number of input events processed within an interval
Max output throughput	Max. number of derived events produced within an interval
Min average latency	Min. average time to process an event
Min Maximal latency	Min. the maximal time to process an event
Jitter	Min. value of the variance in processing times
Real-time	Min. of the deviation in latency from a given value

Table 1. Performance objectives and their associated metrics for Event Processing [6]

When a shared Cloud infrastructure is being used, a provider may serve multiple users using a common resource pool through a "multi-tenancy" architecture. This architecture is used to offer multiple functions over a shared infrastructure to one or more users. The revenue for the provider in this case is the total of all prices charged to users minus the cost of all required resources and the penalties incurred for degraded services.

We assume that the provider (client) monitor their offered (provided) QoS properties over fixed time intervals. The revenue obtained by the provider over a particular time interval is assumed to be constant, and determined by the price clients pay for allocated resources to process their data streams, minus the cost incurred by the provision of these resources (generally identified as operational expenditure (OPEX)). A sudden peak in data, due to sudden data injection or traffic burstiness can produce shortage of resources to process such bursts, over some time slots/intervals. The provider can either accept the penalty due to the unavailability of resources, or can provide additional resources in an elastic way. We define the benefit function for a provider over a particular a time interval for n clients (represented as Instant Revenue) as:

$$\text{(Eq. 1)}\ \ Instant\ Revenue = \sum_{i=1}^{n}(CostPU_{client} - CostPU_{provider}) * \#PU \\ - \sum_{i=1}^{n} \#penalties_i * CostPenalties_i \\ - \Delta\#PU * CostPU_{provider}$$

where $CostPU_{client}$ and $CostPU_{provider}$ are respectively the price of each processing unit (PU) for the client and the provider, $\#PU$ represents the number of resources (in PUs) provisioned by the provider for supporting the aggregated requests of n clients, and $\Delta\#PU$ the number of resources allocated to avoid penalties over bursty periods. The global revenue is the accumulated $InstantRevenue$ over time.

Eq. 1 can be extended to account for additional capabilities, for instance the cost of provisioning additional PUs ($\Delta\#PU$) – which can include the number of virtual machines executed on a single physical machine. Alternatively, the number of processing units can be a function of an estimated workload as a function of data size defined by a data window ($CostPU_{clienti} = f(operation, datasize)$), etc. We will consider Eq. 1 in this paper for sake of simplicity and we will assume for the same reason that data streams can be classified according to the benefit and penalty values of their respective QoS levels as: "Gold" – for high penalty and revenue; "Silver" – for medium penalty and revenue, and "Bronze" – for low revenue and no penalty [7]. This class approach for provisioning resource is commonly found in many commercial data centres and network providers today.

3 Dynamic control of resources under revenue-based management

The revenue model can be used internally by a provider to decide what actions are the most "financially" suitable to dynamically manage resources on a near real-time basis. For instance, when a failure to meet the minimum QoS level for a given user is predicted or detected, a provider may perform the following actions:

- action (1): allocate new local resources or *buy* remote resources,
- action (2): redistribute unused resources by users,
- action (3): redistribute pre-allocated resources from less prioritized users to more prioritized users ("Bronze" to "Silver" to "Gold", or "Bronze" to "Silver").

Each of these actions could have a different cost or penalty for the provider. For instance, allocating new local resources is usually less costly than buying remote resources (using other providers' resources for instance), but may be more costly than redistributing pre-allocated resources from Silver users to Gold users. This could occur because the penalty for not satisfying these Silver users may be less than the cost of allocating new local resources, especially for a short period of time, or because this redistribution of resources may not impact the chosen Silver users due to statistical multiplexing of user needs. When redistributing unused resources, a typical SLA would indicate a negotiated mean data injection rate to

be supported by the provider of the computational resource(s). Therefore, when the amount of injected data over a given time period is smaller than the predicted value, some pre-allocated resources are unutilized. In this case, the redistribution of these unused resources can be done at a low cost by the provider. Hence, we assume that due to the inherent variation in stream processing, it is often difficult to predict accurately the resource demand across multiple time frames. Consequently, this introduces a slack in the system, whereby unused resources may be reallocated to reduce penalties for other data streams in the system. We proposed in our previous work [3] an architecture that uses the token bucket model to perform traffic shaping on user data flows. We also defined how token bucket parameters can be controlled by a rule engine to prioritize data streams. In this paper, we will explain how self-controlled actions could also be directly implemented with different extensions of the token bucket model, introducing for instance an intermediate, shared bucket that will collect unused resources that can be later on be re-distributed across different user classes.

3.1 Dynamic management of resources

QoS requirements are often defined using the worst case scenario – i.e. the maximum number of resources required to achieve a particular QoS objective. However, some data streams may not use the resources that they have reserved and these unused resources could be used to process other streams to increase revenue. Hence, spare capacity in the system could be reallocated. This is particularly useful to handle periods of bursty behavior on some streams. The provider's objective is to maximize its revenue by the management of available computational resources (e.g. a pool of virtual machines in an elastic infrastructure) to process each data stream in accordance with its SLA, taking into account various costs and penalties. It is therefore necessary to regulate end-user's data injection rate according to an agreed SLA, to monitor whether enough resources have been provisioned, and to perform actions to redistribute resources when needed. We described in [3] a modular architecture (illustrated in Fig. 1) consisting of a *traffic shapping* component and a *QoS provisioning* component that provides a dynamic management of resources. We will quickly review the main features of this architecture.

The *traffic shapping* component provides a token bucket per data stream. Within a data stream, it is often useful to identify a "data acceptance rate", which is often different from the physical link capacity connecting nodes and which identifies the rate at which a client can send data to be processed by the server. The data stream processing service tries to maintain this acceptance rate as the output rate. We characterise it for each flow by means of three QoS parameters: (i) average throughput (average number of data elements processed per second), (ii) maximum allowed burst, and (iii) an optional load shedding (data dropping) rate. We make the first two parameters match R and b of the token bucket respectively. For each data stream, its associated token bucket will allow data elements to enter into the processing stage according to the R parameter. The token bucket can also accept a burst of b data elements.

Subsequently, a data element is forwarded to a First Come First Serve (FCFS) queue buffer at a processing unit (PU). In addition to regulating access to the PU and enforcing QoS per data stream, the token bucket also achieves stream isolation, i.e. a data burst in one stream does not interfere with another. The load shedding mechanism acts at input buffers by discarding older data elements of a flow at a specified rate. It is only active, however, when triggered by the controller component.

The *QoS provisioning* component takes decisions about the allocation and redistribution of resources based on the monitoring of buffers and token buckets. For example, availability of data in buffers of a token bucket implies data injection over the agreed mean rate, which can trigger different actions based on occupancy thresholds: 1) dropping data from the buffers, 2) allocating additional resources to consume the burst of data, 3) reallocation of resources from other streams. The number of allocated resources for providing service to the aggregate demand may not be enough for a bursty period. In this case, the controller must detect data streams that require more resources. Data in the computational phase are stored in buffers associated with each data stream (we denote these as PU buffers to differentiate them from TB buffers). The PU buffer size can be used to detect when data are been buffered because there are not enough allocated resources. For instance, during each control interval T the maximum amount of data that can appear is $RT + b$. If the PU buffer size is greater than b, this suggests that not enough resources have been provisioned to sustain the QoS of this data stream. Note that during a time interval b data can be transferred to the processing phase if there are enough tokens in the TB.

The bottom part of Fig. 1 shows the control loop configuring 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 glasses (a) in the figure). Using these initial parameter values, the control strategy is initiated, subsequently recording the TB (b) and PU (c) input queue buffer occupancies, and the number of resources in use at the PU (d). The size of each input buffer is chosen in accordance with the agreed requirements of the data flow. The controller must estimate the buffer size during execution. When the input buffer size reaches an established threshold, it triggers the controller to initiate one of two possible actions: (i) calculate the number of additional resources (PU) needed (based on those available) to process the additional data items generated above rate R; (ii) if there are free local resources (not being used by other data flows), they can be used to increase the rate R 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. A detailed description of this control loop and validation scenarios can be found in [3].

It appeared that allocating new resources, action (1), may not be suitable for handling short periods of resource shortage. The time required to get statistics and the inference process of the rule engine does not allow introduction of

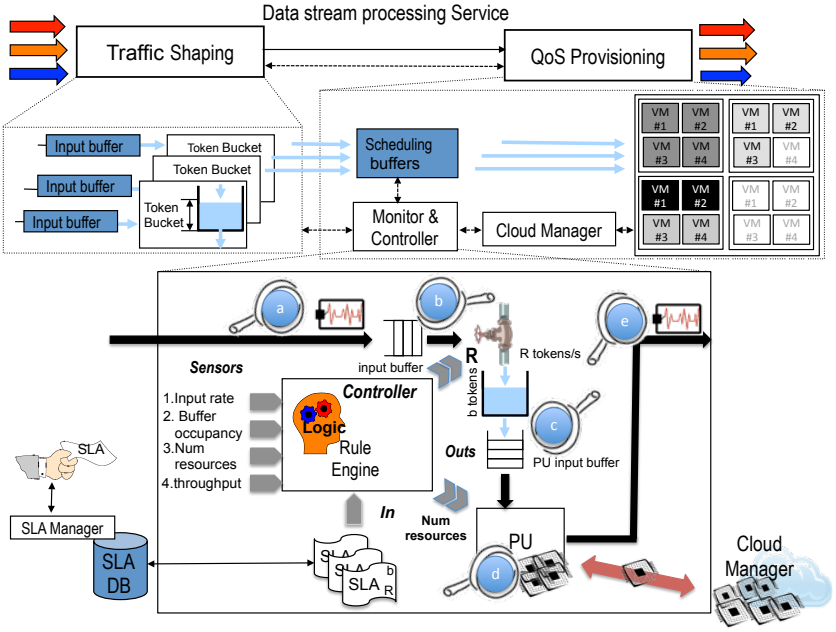

Fig. 1. Control loop for decision making.

new resources in near real-time. In this paper, we propose to additionally investigate action (2), redistribute unused resources, and action (3), redistribute pre-allocated resources from less prioritized to more prioritized users. The choice of the final action will be determined by the revenue model using a cost, or penalty, associated with each action. We will describe how these actions can be easily implemented by the provider by extending the previous token bucket model.

3.2 Redistribute unused resources by users

When the real amount of injected data over a given time period is lower than the predicted amount, tokens can be saved by a user and they accumulate in its associated TB up to a maximum of b tokens (which is the bucket size). Normally, these excess tokens are dropped by the TB to avoid very large bursts of data in the future. However, it is possible for a provider to save these tokens in an additional shared bucket (of maximum size B_{max}) and to redistribute them at a low cost – as these tokens typical represent unused resources that have already been allocated. Figure 2 illustrates this behavior. These tokens in excess could also have a limited lifetime as symbolically represented by the clock in Fig. 2 in order to limit their usage within a few control intervals only.

Collecting tokens in excess and redistribution of tokens can be performed globally over all user classes. However, limiting token movement within the same

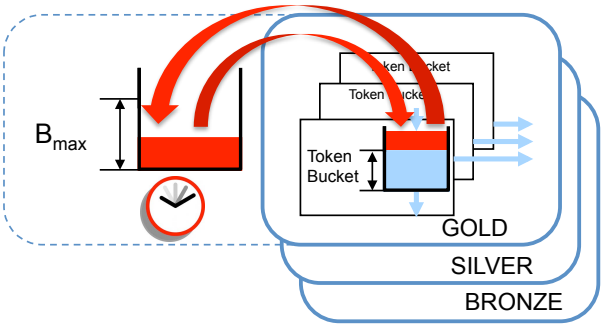

Fig. 2. Redistributing unused resources over a control period.

class may be easier to support, e.g. excess tokens from Gold users can only be redistributed to other Gold users. Fig. 2 with the dashed box illustrates this solution where each user class should have their own additional bucket space. The B_{max} parameter can be different for each user class. For instance, $B_{max}^{gold} > B_{max}^{silver} > B_{max}^{bronze}$. The rationale behind different values for B_{max} is that unused resources from Bronze users could be considered more *volatile* than unused resources from Silver or Gold users for instance, as Bronze user resources may have been statistically allocated. It is possible to generalize this architecture for a higher number of classes where $B_{max}^{C_n} > B_{max}^{C_{n-1}} > ... > B_{max}^{C_2} > B_{max}^{C_1}$

3.3 Redistribute pre-allocated resources from less prioritized users to more prioritized users

The case of redistributing pre-allocated resources is quite different from the unused resources case: tokens from a chosen user's bucket will be moved directly to another user's bucket. Figure 3 illustrates this redistribution process from a Bronze user to a Silver user. Redistribution from less prioritized users

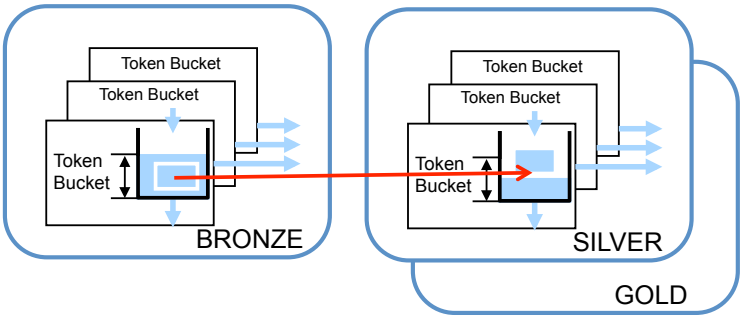

Fig. 3. Redistribution from low priority users to high priority users.

to more prioritized users is typically the most financially efficient solution for the provider. Moving tokens directly from one bucket to another may generate temporary resource shortages for the data flow from which tokens are taken. As a result, at time of shortage, the revenue model will decide again between the 3 possible actions it can perform.

3.4 An example of dynamic management of resources

Let us explain with an example how redistribution of unused resources and redistribution of pre-allocated resources from less prioritized users could be used consistently and conjointly by a provider. Let us denote by $TB_u^{C_n}$ the token bucket of a user stream u in class C_n and by $TB_{unused}^{C_n}$ the shared bucket space in class C_n to keep unused resources (tokens) up to $B_{max}^{C_n}$.

Consider that a provider has under-estimated the resources that should be allocated to a user of class C_n (one reason could be a bursty injection period). When the system detects that this user does not have a sufficient processing rate according to its negotiated token bucket data injection rate, the provider can take unused tokens from $TB_{unused}^{C_n}$ that have been collected within class C_n, if any. If there are no unused tokens in class C_n, the provider will take tokens directly from $TB_u^{C_i}$ of a user u in a lower class C_i, with $1 \leq i \leq n-1$, and not from the shared bucket space $TB_{unused}^{C_i}$ of these lower classes C_i. The reason is that resources collected in $TB_{unused}^{C_i}$ represent more "volatile" resources than resources kept in $TB_u^{C_i}$ that normally could be somehow mapped to real resources in the current control interval.

By doing so, the class C_n user demand can be satisfied at minimum cost, therefore limiting the penalty for the provider. If the C_i class users, $1 \leq i \leq n-1$, from whom tokens have been taken away by users in class C_n have token/resource shortage, the system will first try to take tokens from the shared unused resource bucket space of the corresponding class, i.e. $TB_{unused}^{C_i}$, if any, and only then will try to take token directly from a token bucket of a lower class C_j, i.e. from $TB_u^{C_j}$, $1 \leq j \leq i-1$. This process could be repeated at each class C_i. We can therefore see how this 2-level token movement system can be used to optimally move resources (unused or pre-allocated) based on a maximum revenue strategy.

4 Evaluation Scenarios

The redistribution of pre-allocated resources from less prioritized users to more prioritized was illustrated in the evaluation scenarios of [3] by means of the rule engine controlling TB parameters. In this paper, we will present the results of different simulation scenarios to show the redistribution of unused resources by an additional bucket that collects tokens in excess and redistribute them over the same class, as proposed in section 3.2. We consider two scenarios: (i) using the rule-engine controller, which validates the shared bucket in an elastic provisioning approach with tokens representing reliable allocated resources; and

(ii) without use of controller actions, which validates the shared bucket with more *volatile* tokens.

The scenarios has been modeled using the *Reference net* formalism [8] to specify the decision making component. The models have been simulated in Renew (see http://www.renew.de), a Java-based editor and simulator based on reference nets that integrates the Petri net formalism, and the Java programming language. The Java Expert System Shell (Jess), is used to trigger actions based on threshold monitored values, such as token bucket and PU buffers, and input/output rates. For this work, the token bucket manager model that provides a token bucket for each new data stream presented in [2] has been extended with the common shared bucket. The modeled behavior moves excess tokens to the common bucket, and all data streams can make use of these tokens if their buckets are empty and there are no pending data items to be processed in the PU buffers. At the end of each control period, the common bucket is emptied: therefore the lifetime of collected unused resources is limited to a control interval.

4.1 Redistribution of tokens in an elastic scenario

The first scenario considers data streams at the same priority level. We assume 4 Gold (i.e. high priority) customer streams with a period of control of T=1 second and all data streams have same requirements: R=20 and b=10. The maximum number of data to be processed is 120 data chunk/second and a token is required to process a data chunk. We assume that each resource can process 10 data chunk/s (therefore requiring in the worst case a maximum of 12 processing units). Input streams follow on ON-OFF process where ON and OFF periods follow a uniform distribution between 2 to 5 seconds and alternate each other. Data injection rates within the ON period follows an exponential law (Poisson distribution) therefore varying the data injection rate over time. On average about 4 resources are required for the 4 data streams (each stream sends on average 20 data/second half of the time). For the first set of simulations we compare the behavior of the system with and without the common bucket (of capacity $B_{max}^{gold} = 80$ tokens). These simulations are developed in combination with the use of the rule engine to provide enough resources throughout the simulation period. The rule engine triggers actions for dropping data when the TB buffer occupancy is over an established threshold, adding/removing resources in a elastic way (borrowing resources from low priority data streams) and tuning TB parameters to use the new added resources or available resources when PU buffers have accumulated data (which is an indication that not enough resources are available). All simulations reproduce the same input data injection rates for comparison purpose.

To calculate the revenue with Eq. 1 we assume a cost of 20 units/second per PU for clients and 15 units/second per PU for provider. We assume the client pays for having the processing rate R all the time. Taking into account that the data stream rates are irregular and the client send data at a rate $R/2$ on average, the provider will suffer from a high penalization, for example 30 times the price paid by the client, i.e. 600 units, if it does not provide enough resources.

A penalty occurs when the output rate is under the agreed rate R if there are data in the TB buffer. In this way, it is easy for the client to monitor whether the provider is allocating enough resources or not. If the buffer is full, the output rate should be at least equal to R. If the throughput is under this value, data in the buffer are being accumulated and will be delayed to be processed in the next control intervals due to the lack of resources.

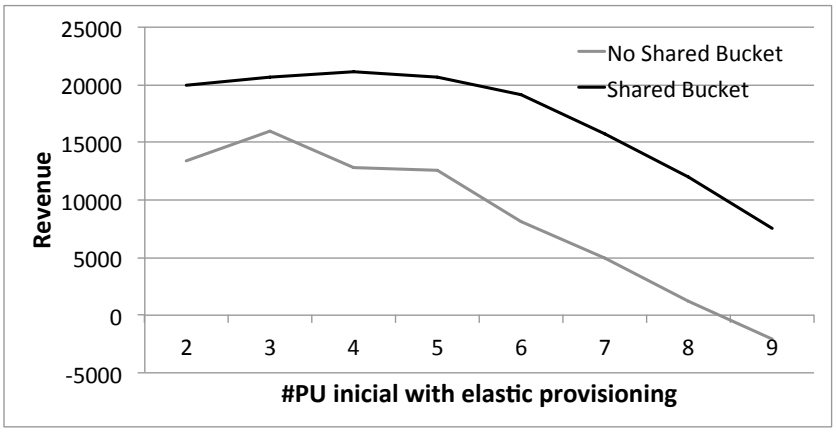

Fig. 4. Scenario I: Redistribution of tokens in an elastic scenario with different baseline PUs (horizontal axis) and using control loop to avoid penalties. Revenue is measured in an abstract unit, but can be mapped to a particular economic currency.

Figure 4 shows the provider's revenue for different number of initial PUs and an elastic provisioning of resources scenario. The x-axis represents the initial baseline number of resources and the y-axis the aggregated revenue over 300 seconds of simulation. These results show the maximum revenue when enough resources are available to satisfy the demand. Providing less resources than this baseline increases the number of penalties, and providing more resources as baseline increases the cost. The common bucket however does not improve significantly the aggregated throughput as shown in Figure 6, but the throughput of each individual data stream is improved as shown in the sample data stream output of figure 5. If we look at time interval 20s-40s, 70s-80s and 140s-150s we can see that the shared bucket allows the output throughput to closely follow the input data injection rate. This behavior can be more clearly seen with 9 PUs than with 4 PUs, i.e. when there are globally enough resources. Without the shared bucket the output throughput is clearly limited by the b parameter (maximum amount of tokens in the bucket) and a shortage of tokens limits the output throughput to R until the TB buffer is emptied.

The bottom of Figure 7 shows that the average number of PUs provisioned (their cost being represented by the last term in Eq. 1) in an elastic scenario is not affected by the use of the shared bucket. However the number of penalizations

(second term in Eq 1.) is clearly reduced with the use of the shared bucket as illustrated in Figure 7(top).

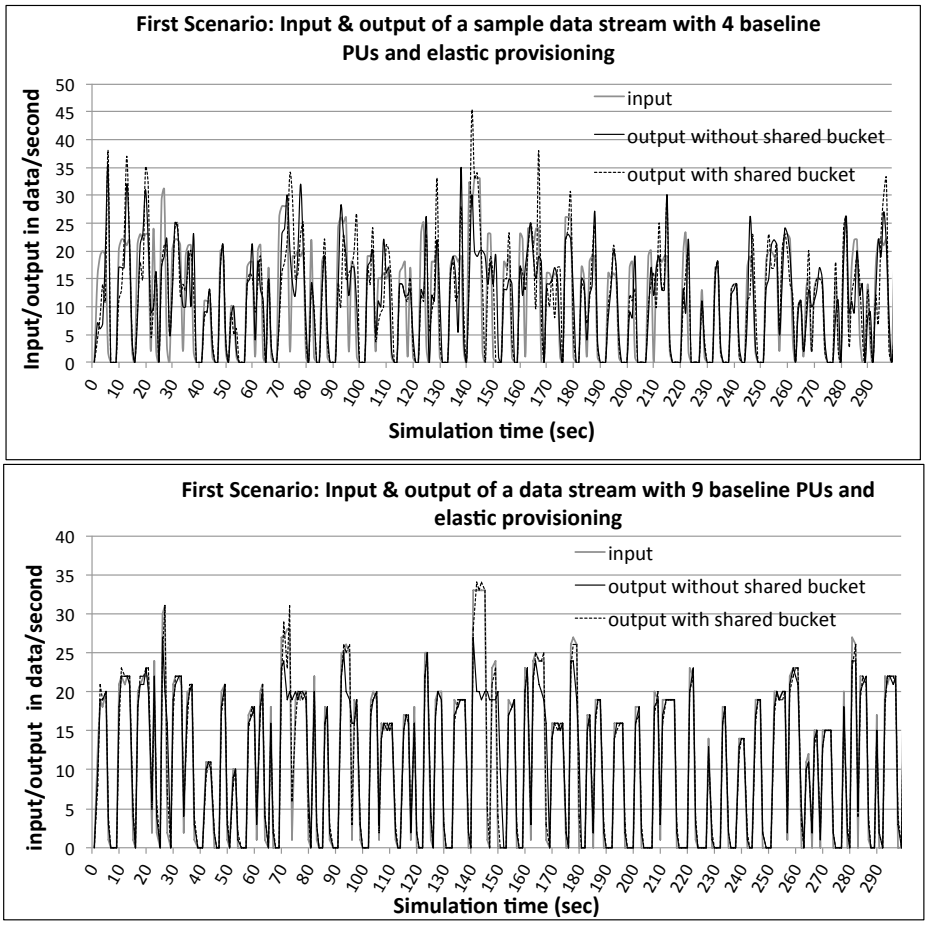

Fig. 5. Data stream input & output in an elastic provisioning scenario.

4.2 Redistribution of tokens in a non-elastic scenario

The second scenario uses the same number of data streams than previously but without rules to provide additional resources in an elastic way. Therefore, when there are shortage of resources, the benefit of the redistribution feature can be better seen. In this scenario data streams have a more sporadic behavior to enable greater usage of the shared bucket: ON and OFF period durations follow a uniform distribution between 1 to 3 seconds, but now an ON period have a

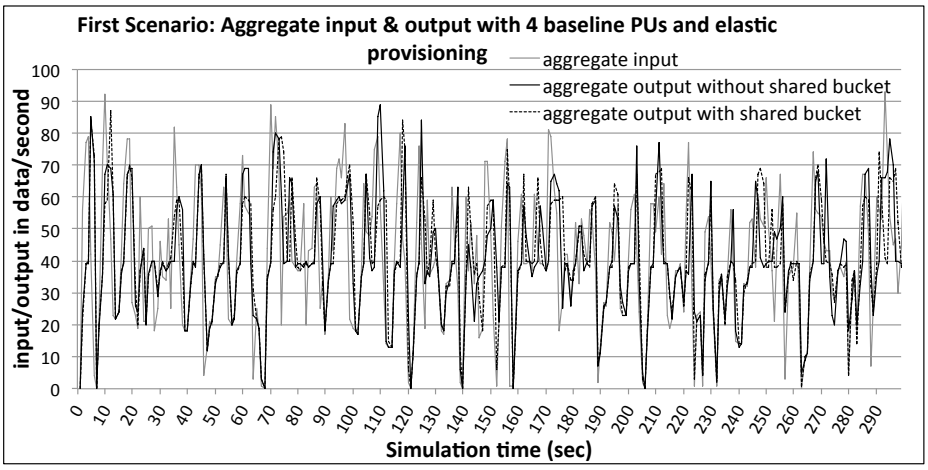

Fig. 6. Aggregated input and output in an elastic provisioning scenario.

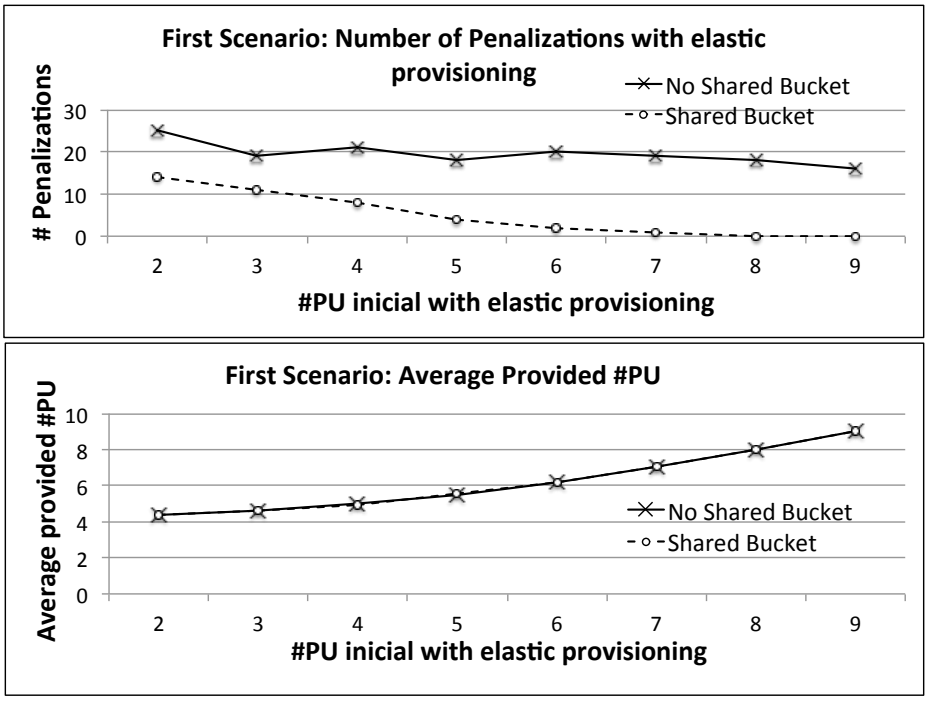

Fig. 7. Average number of PU in an elastic scenario and number of penalties.

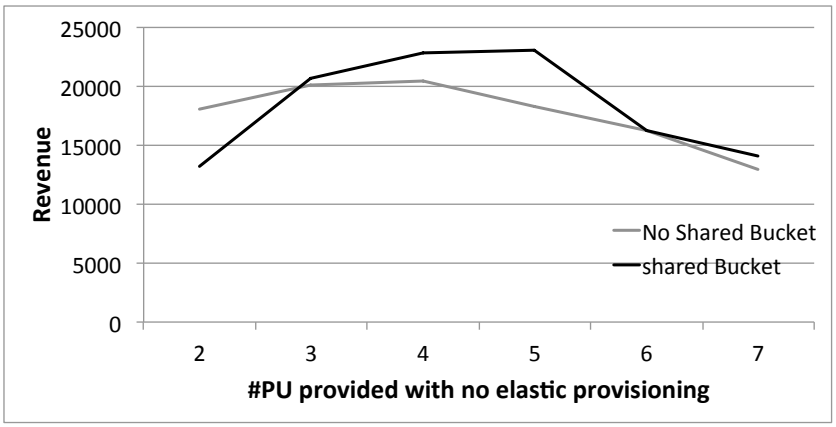

Fig. 8. Scenario II: Redistribution of tokens in a non-elastic scenario with different baseline PUs (horizontal axis).

probability of 1/3 to occur. Again, data injection rates follow a Poisson distribution. Therefore, for 4 data streams sending on average 20 data chunk/second the number of required resources is around 3. Figure 8 shows the provider's revenue with different number of initial provisioned PUs. With less than 3 PUs, the number of penalizations makes the revenue to decrease and the shared bucket gives a lower revenue when there is shortage of resources. Provisioning between 3 and 5 PUs makes the shared bucket very useful as a low cost solution to balance the usage of resources between classes. Figure 9 shows how throughput closely follows the input data injection rate at time interval 110s-120s and 210s-220s.

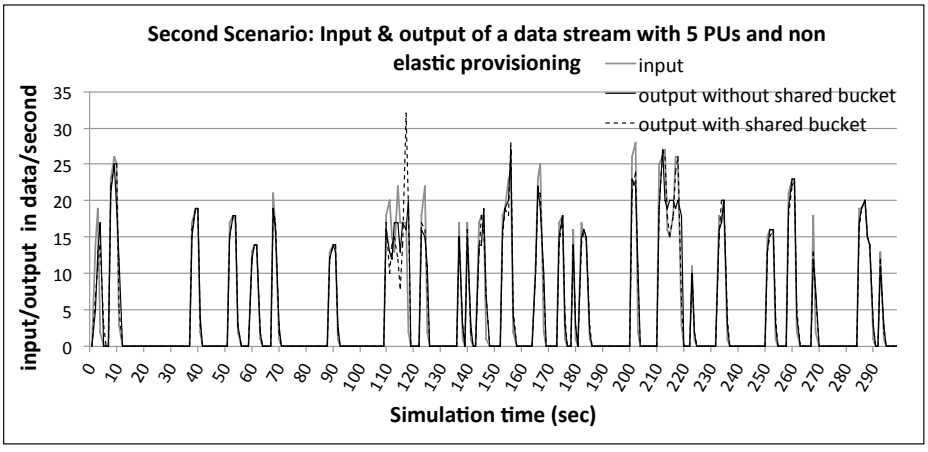

Fig. 9. Data stream input & output in a non elastic scenario.

5 Related Work

Auto-scaling of resources has been identified as one of the main challenges for Cloud Computing. The main concern to optimize the use of resources is to automatically scale quickly up and down in response to load in order to save money, but without violating SLAs [1]. There is emerging interest in processing automated elastic resource provisioning over shared Cloud. Three main approaches have been pointed out to quickly scale resources [9]. First, reactive mechanisms, mainly use elasticity rules or threshold-based rules pre-defined by service providers [10, 7, 11]. Second, predictive mechanisms try to learn from previous data history and resource usage to construct mathematical models to forecast resource demands. These approaches are useful when regular behavior pattern can be identified, but can not forecast unpredictable burstiness [12, 13]. This problem has been considered in [14] to propose pattern matching scaling based algorithms as an alternative to mathematical models that do not consider arbitrarily-repetitive self-similarities. And third, hybrid approaches [15] that integrate the 2 previous approaches or, more recently, use theory of control [16]. A brief reference to related work on elastic resources provisioning of workflow, streaming and event processing have been presented in [3].

6 Conclusion and Future work

We propose (i) an architecture that features a token bucket process envelop to support data throttling, (ii) a rule-based control loop to enable corrective actions to be triggered when QoS is violated: the control loop monitors QoS for each application and chooses an action that maximises revenue over a pre-defined control interval, and dynamic corrective actions embedded in token bucket extensions to (iii) re-distribute unused resources among users, and (iv) to re-distribute pre-allocated resources from less prioritized users to more prioritized users – in the context of stream processing applications. The validation scenarios have shown that the token bucket extension based on a shared bucket to redistribute resources increases data stream throughput when there are enough resources on average to serve the aggregated demand. We showed that from a revenue-based perspective, optimization of resources with low-cost solutions using local unused resources is very effective compared to buying remote resources. Future work will consider additional aspects to calculate the instant revenue considering the cost of additional processing units, taking into account the number of virtual machines that can be executed on a single machine, or the estimation of the workload as a function of historical data (using previous service executions) with different parameters (operation, data size, window size, etc.).

References

1. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A view of cloud computing,”

Commun. ACM, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672
2. R. Tolosana-Calasanz, J. Á. Bañares, C. Pham, and O. F. Rana, "Enforcing QoS in scientific workflow systems enacted over Cloud infrastructures," *J. Comput. Syst. Sci.*, vol. 78, no. 5, pp. 1300–1315, 2012.
3. R. Tolosana-Calasanz, J. Á. Bañares, C. Pham, and O. F. Rana, "Revenue-based resource management on shared clouds for heterogenous bursty data streams," in *GECON*, ser. Lecture Notes in Computer Science, K. Vanmechelen, J. Altmann, and O. F. Rana, Eds., vol. 7714. Springer, 2012, pp. 61–75.
4. R. Tolosana-Calasanz, J. Á. Bañares, O. Rana, L. Cipcigan, P. Papadopoulus, and C. Pham, "A Distributed In-Transit Processing Infrastructure for Forecasting Electric Vehicle Charging Demand," in *DPMSS workshop alongside 13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16*, 2013.
5. T. Abdelzaher and N. Bhatti, "Web server QoS management by adaptive content delivery," in *IWQoS '99. 7th Int. Workshop on*, 1999, pp. 216 –225.
6. O. Etzion and P. Niblett, *Event Processing in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.
7. M. Macías, J. O. Fitó, and J. Guitart, "Rule-based sla management for revenue maximisation in cloud computing markets," in *CNSM*. IEEE, 2010, pp. 354–357.
8. O. Kummer, *Referenznetze*. Berlin: Logos Verlag, 2002.
9. J. Jiang, J. Lu, G. Zhang, and G. Long, "Optimal Cloud Resource Auto-Scaling for Web Applications," in *13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16*, 2013, pp. 58–65.
10. "Amazon Auto Scaling." [Online]. Available: http://aws.amazon.com/autoscaling/ .Lastaccessed:July2013.
11. G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar, "SYBL: an Extensible Language for Controlling Elasticity in Cloud Applications," in *13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16*, 2013, pp. 112–119.
12. D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA," in *13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16*, 2013, pp. 34–41.
13. F. Morais, F. Brasileiro, R. Lopes, R. Araújo, W. Satterfield, and L. Rosa, "Autoflex: Service Agnostic Auto-scaling Framework for IaaS Deployment Models," in *13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16*, 2013, pp. 112–119.
14. Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: elastic resource scaling for multi-tenant Cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 5:1–5:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038921
15. W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the Cloud," *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 871–879, Jun. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.future.2010.10.016
16. P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 13–26. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519068