Resource management for bursty streams on multi-tenancy cloud environments

Rafael Tolosana-Calasanz\textsuperscript{a,}\textsuperscript{*}, José Ángel Bañares\textsuperscript{a}, Congduc Pham\textsuperscript{b}, Omer F. Rana\textsuperscript{c}

\textsuperscript{a} Computer Science and Systems Engineering Department, Aragón Institute of Engineering Research (I3A), University of Zaragoza, Spain
\textsuperscript{b} LIUPPA Laboratory, University of Pau, France
\textsuperscript{c} School of Computer Science & Informatics, Cardiff University, United Kingdom

\textbf{HIGHLIGHTS}

- We provide a system for simultaneous bursty data streams on shared Clouds.
- We enforce QoS based on a profit-based resource management model.
- We provide real experiments within an OpenNebula based data centre.

\textbf{ABSTRACT}

The number of applications that need to process data continuously over long periods of time has increased significantly over recent years. The emerging Internet of Things and Smart Cities scenarios also confirm the requirement for real time, large scale data processing. When data from multiple sources are processed over a shared distributed computing infrastructure, it is necessary to provide some Quality of Service (QoS) guarantees for each data stream, specified in a Service Level Agreement (SLA). SLAs identify the price that a user must pay to achieve the required QoS, and the penalty that the provider will pay the user in case of QoS violation. Assuming maximization of revenue as a Cloud provider’s objective, then it must decide which streams to accept for storage and analysis; and how many resources to allocate for each stream. When the real-time requirements demand a rapid reaction, dynamic resource provisioning policies and mechanisms may not be useful, since the delays and overheads incurred might be too high. Alternatively, idle resources that were initially allocated for other streams could be re-allocated, avoiding subsequent penalties. In this paper, we propose a system architecture for supporting QoS for concurrent data streams to be composed of self-regulating nodes. Each node features an envelope process for regulating and controlling data access and a resource manager to enable resource allocation, and selective SLA violations, while maximizing revenue. Our resource manager, based on a shared token bucket, enables: (i) the redistribution of unused resources amongst data streams; and (ii) a dynamic re-allocation of resources to streams likely to generate greater profit for the provider. We extend previous work by providing a Petri-net based model of system components, and we evaluate our approach on an OpenNebula-based Cloud infrastructure.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

The number of applications that need to process data continuously over long periods of time has increased significantly over recent years. Often the raw data captured from the source is converted into complex events—which are subsequently further analysed. Such applications include weather forecasting and ocean observation [1], text analysis (especially with the growing requirement to analyse social media data, for instance), “Urgent Computing” [2], and more recently data analysis from electricity meters to
support "Smart (Power) Grids" [3]. The emerging Internet of Things and Smart Cities scenarios also strongly confirm that increasing deployment of sensor network infrastructures generate large volumes of data that are often required to be processed in real-time. Data streams in such applications can be large-scale, 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 [4].

Multi-tenancy Cloud environments enable such concurrent data streams (with data becoming available at unpredictable times) to be processed using a shared, distributed computing infrastructure. When multiple applications are executed over the same shared elastic infrastructure, each stream must be isolated from the other in order to either: (i) run all instances without violating their particular Quality of Service (QoS) constraints; or (ii) indicate that, given current resources, a particular stream 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 between the stream owner/generator and the service provider (hosting the analysis capability) a priori. 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 [5].

Assuming the maximization of profit as the main Cloud provider's objective, then it must be decided which streams to accept for storage and analysis; and how many resources to allocate to each stream in order to improve its overall profit. This task is highly challenging with aggregated, unpredictable and bursty data flows that usually make both predictive and simple reactive approaches unsuitable. Even dynamic provisioning of resources may not be useful to provide a profit to the Cloud provider since the delay incurred might be too high—it may take several seconds to add new resources (e.g. instantiate new Virtual Machines (VMs)), and a scaling-up action might generate substantial penalties and overheads.

Our main contributions consist of data admission and control policies to regulate data access and manage the impact of data bursts, and a policy for resource redistribution that tries to minimize the cost of QoS penalty violation, maximizing the overall profit. The rationale behind this latter policy is that current mechanisms for scaling resources in Cloud infrastructures have severe associated delays which may provoke large financial penalties. Overall, our main contributions can be summarized as follows: (i) an improved profit model that takes into account both profit and penalties, (ii) a set of dynamic control actions to manage resources with maximization of a provider’s profit, (iii) a unified token-based resource management model for realizing the profit-oriented actions. This model aims at optimizing the utilization of unused resources and allowing dynamic and consistent re-allocation of resources. Section 4 describes the Reference net model of the control logic used. Section 5 shows our evaluation scenarios and simulation results. Section 6 presents our deployment and experiments on an OpenNebula-based Cloud infrastructure. In Section 7, most closely related work is discussed. Finally, the conclusions and future work are given in Section 8.

2. Profit-based resource management

2.1. Profit-based model

We consider a provider centric view of costs incurred to provide data stream processing services over a number of available computational resources. If we assume the objective of the provider is to maximize 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 profit revenue (generally over a time horizon); and (iii) what actions could be performed to dynamically modify and adjust the usage of resources. The first two considerations can generally be based on the SLA that a user and a provider have agreed to while the last point could be considered internal to the provider as a way to optimize resource utilization. A provider may use a (pre-agreed and reserved) posted price, a spot price (to gain profit from currently unused capacity), or an on-demand use (the most costly for the user) of 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 an SLA, and what actions the provider can perform to guarantee the agreed QoS and maximize the profit. 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 as commercial Web hosting, QoS levels specify parameters such as request rate, for example expressed as served URLs per period or the number of concurrent users served; and data bandwidth, that specifies the aggregate bandwidth in bytes per second to be allocated in the contract [12]. In other applications such as video-on-demand, QoS levels may represent frame rates and average frame sizes. In the context of a data stream, the
analysis can include min/max/avg calculations on a data or sample time window, an event analysis, a summarization of data over a time window, etc. Etzion [13] provides a useful summary of the performance objectives of event processing and their associated metrics (see Table 1).

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 and subsequent profit 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) monitors their offered (provided) QoS properties over fixed time intervals. The profit 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. 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 time interval for n clients (represented as Instant Profit) as:

\[
\text{Instant Profit} = \sum_{i=1}^{n} \left( \text{Price}_{PU_{client}} - \text{Cost}_{PU_{provider}} \right) \times \#PU
\]

\[- \sum_{i=1}^{n} \#\text{penalties}_i \times \text{CostPenalties}_i
\]

\[- \Delta \#PU \times \text{Cost}_{PU_{provider}} \]

where \(\text{Price}_{PU_{client}}\) represents the unitary price per virtual machine (VM) (referred to as processing unit (PU) in our paper) that we charge to client \(i\). \(\text{Cost}_{PU_{provider}}\), is the cost incurred while requesting the VM (PU) in a Cloud provider for client \(i\). \#PU represents the overall number of VMs (in PUs) provisioned by the system for supporting the aggregated requests of \(n\) clients, and \(\Delta \#PU\) the number of resources allocated to avoid penalties over bursty periods. The aggregate profit is the accumulated Instant Profit over time. \(\text{Cost}_{PU_{provider}}\) consists of both the: (i) capital expenditure incurred by the provider in acquiring hardware resources and software licenses for running the Cloud infrastructure; (ii) operational expenditure incurred based on the likely demand seen from external customers, which includes energy/cooling costs, system management and administration costs, any recurring licensing costs in addition to those part of the capital expenditure. Increasingly, (i) is becoming less significant (as hardware costs come down) and (ii) becomes increasingly more important for the provider to optimize on over longer time frames. The capital expenditure is often considered over longer time frames, compared to operational expenditure. A provider must therefore factor in both capital and operational cost and add an additional “profit” to this figure when quoting a price to a client—in this work we do not focus on how such cost is actually determined by the provider, but there is some work already addressing these aspects [14,15].

Eq. (1) can be extended to account for additional capabilities by refining the cost of provisioning additional PUs (\(\Delta \#PU\)). These PUs can be seen in the equation as generic resources (not only physical CPU) and at deployment time on a given Cloud infrastructure be differentiated according to the infrastructure’s possibilities. For instance, if the Cloud infrastructure is capable of allocating VMs, then Eq. (1) can reflect this feature by having an associated cost for launching new VMs on an existing single physical CPU. The number of PUs can also be a function of an estimated workload, defined by a data window \(\text{Cost}_{PU_{client}} = f(\text{operation, data size, i})\), 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 profit; “Silver”—for medium penalty and profit, and “Bronze”—for low profit and no penalty [16]. This class based approach for provisioning of resources is commonly found in many commercial data centres and network providers today.

2.2. Dynamic control of resources under profit-based management

We assume that there exists a provider with a pool of resources that can be allocated/deallocated, depending on the workload from a number of clients. The profit 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. 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 profit.

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 profit by the management of available computational resources (e.g. a pool of VMs 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 an 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. For instance, when a failure to meet the minimum QoS level for a given user is predicted or detected, a provider may redistribute pre-allocated resources from less prioritized users to more prioritized users (“Bronze” to “Silver” to “Gold”, or “Bronze” to “Silver”).

In particular, what we propose is that such clients can negotiate an SLA based on the service classes, \(C_i = 1 \leq i \leq n\), ordered by increasing penalty to the provider in case of an SLA violation. Consider that a provider has under-estimated the number of computational resources that should have been allocated to a user of class \(C_i\) (i.e. one reason might be a bursty injection period). When the system detects the violation, the provider has a number of actions: (i) to allocate new computational resources, our focus in this paper, however due to the associated delays in dynamically allocating a VM (seconds to minutes at present), this may lead to immediate SLA violations (and this has a cost for the provider); (ii) to
take unused computational resources associated to class $C_i$ if any; (iii) to take unused resource from less prioritized classes, if any; (iv) in case there are no unused resources, the provider has to assume a penalization, but as penalization will have a lower cost in lower prioritized classes, the provider borrows resources from less prioritized users. By doing so, the user demand in class $C_i$ can be satisfied at minimum cost, therefore limiting the penalty for the provider. If the $C_i$ class users, $1 \leq i \leq i - 1$, from whom resources have been taken away by the user in class $C_i$ have resource shortage, the controller will repeat the process. 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 profit strategy.

Each of these actions could have a different cost or penalty for the provider and once again, the term “resource” should be taken here in the most general context, and a distinction may be made at deployment time according to the Cloud infrastructure/middleware capabilities. For instance, when allocating new local resources, the cost of launching a new VM on a physical CPU may be considered lower than the cost of provisioning a new CPU/server, assuming that the infrastructure/middleware is capable of allocating a resource at that granularity. Then, allocating new local resources is less costly than buying new remote resources (using other providers’ resources for instance), but may be more costly than redistributing pre-allocated resources from less prioritized users to more prioritized users, e.g. from Silver users to Gold users. One reason being that the penalty for not satisfying these Silver users may be less than the cost of allocating new local resources (VMs or CPUs), 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 size of injected data over a given time period is smaller than the predicted value, some pre-allocated resources are unused. In this case, these unused resources can be redistributed at a very 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.

3. System architecture for dynamic management of resources

Our system architecture can process a number of data streams simultaneously, with the main objective of maintaining a negotiated SLA (throughput), while minimizing the number of computational resources involved. The underlying resource management policy is based on a business model: each SLA violation leads to an associated penalization, whereas scaling-up the computational resources involved has an associated cost. Therefore, the system controller main policy is to trigger the action that maximizes profit, by maximizing revenue (the amount of money that customers must pay for their data stream processing) and by minimizing cost (the cost of either assuming a penalization or launching additional computational resources).

In the following subsections, we describe the mechanisms that implement this policy. Our system architecture consists of an Event Processing Network (EPN) composed of a sequence of geographically distributed nodes. Each node features an envelope process for regulating and controlling data access and a resource manager to enable resource allocation, and selective SLA violations, while maximizing revenue. Our resource manager, which has a number of computational resources available (i.e. virtual machines), is based on rules and a shared token bucket. It enables: (i) the re-distribution of unused resources amongst data streams; and (ii) a dynamic re-allocation of resources to streams likely to generate greater profit for the provider.

3.1. Traffic shaping component

The token bucket envelope process has been already utilized in the networking context, for the provisioning of deterministic service guarantees [17,18]. As illustrated in Fig. 2, a token bucket envelope process 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 capacity. The token bucket can contain $b$ tokens and may be full at initialization time. In the fluid model, a customer is allowed to send one bit of data if there is one token in the bucket, in which case one token is consumed. Practically, 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. When there are enough tokens in the bucket, a user can send at the rate $C > R$, otherwise the data rate is $R$—as illustrated in the middle in Fig. 2. 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 allows bursts of traffic up to a regulated maximum, enforcing on a long term basis the negotiated rate $R$, as illustrated on the right of Fig. 2.

We make use of a TB per data stream in our traffic shaping component that regulates data access for processing, rather than the typical use of TB that regulates bandwidth traffic. 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 also the output rate. We characterize it for each flow by means of three QoS parameters: (i) average throughput (average number of data elements processed per second), (ii) maximum allowed burstiness, and (iii) an optional load shedding (data dropping) rate. We make the first two parameters match $R$ and $b$ of the token bucket parameters, 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) buffer queue 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.

Additionally, TB as an envelop process can also be used to estimate cost depending on the resources required during each control period $T$ to process the worst case traffic $RT + b$, for each data stream [18]. As a provider is aware of the maximum output rate and buffer capacity associated with the resources they manage, the provider must allocate enough resources to provide the total processing rate capacity $R$ and buffer space $B$ to guarantee a maximum delay $d$ for each data stream as shown on the right of Fig. 2. $E(t)$ represents the envelop curve, that is, the upper bound on data arriving up to time $t$ and $S(t)$ the service curve, the upper bound on data departing up to time $t$ with the provided resources. A deterministic service ensures that no data are dropped or delayed beyond their guaranteed delay bound. Additionally, in [18] admission control tests are described considering the token buckets of all involved data streams. Therefore, the TB model provides a simple mechanism for traffic shaping, and a model to estimate the worst case demand.

However, determining the effective number of computational resources (i.e. a pool of VMs as part of an elastic infrastructure), and data storage and processing requirements can be a challenge in the
event processing context. The processing rate will depend on the operation, the event processing engine, the use of heterogeneous machines [19], and other specific event processing operations [20]. The impact of these parameters on performance (and therefore SLA compliance) can be evaluated through profiling from previous executions.

3.2. QoS provisioning 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, presence of data in the buffer of a token bucket implies data injection exceeding the agreed mean rate, which can trigger different actions based on occupancy thresholds: (1) dropping data from the buffer, (2) allocating additional resources to consume this additional data, (3) reallocation of resources from other streams. The number of allocated resources for providing service to the aggregate demand may not be enough over 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 number of tokens in the PU buffer can be used to detect when data have 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 arrows from the top of Fig. 1 represent data streams flowing through the system. 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, after a pre-defined sampling period (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 number of resources allocated will return to their previously agreed values when the data size in the input buffer goes below the threshold. A detailed description of this control loop and validation scenarios can be found in [10].

3.3. Rule-based SLA management

To improve data processing throughput, a common strategy is to allocate resources which do not always cater for the worst case scenario—to prevent over allocation of resources for each stream (leading to starvation of resources for other streams). It is instead useful to identify how to react, quickly, when a likely case scenario—to prevent over allocation of resources for each data stream (we denote these as PU buffers to differentiate them from TB buffers). The number of tokens in the PU buffer can be used to detect when data have 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 arrows from the top of Fig. 1 represent data streams flowing through the system. 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, after a pre-defined sampling period (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 number of resources allocated will return to their previously agreed values when the data size in the input buffer goes below the threshold. A detailed description of this control loop and validation scenarios can be found in [10].

3.4. Extending the token bucket model for resource redistribution

3.4.1. Collecting unused resources

When the actual size of injected data over a given time period is lower than the predicted, tokens accumulate in its 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 Bmax) and to redistribute them at a low cost—as these tokens typically represent unused resources that have already been allocated. Fig. 3 illustrates this behavior. The arrow from the TB to the shared TB on the left represents the movement of these tokens that are to be dropped and can be sent to the shared bucket instead. In contrast, the arrow from the shared bucket to the right part of the figure represents the utilization of these unused tokens by any of the data stream clients with shortage of resources. These tokens in excess could also have a limited lifetime as symbolically represented by the clock in Fig. 3 in order to limit their usage within a few control intervals.
Collecting tokens in excess and redistribution of tokens can be performed globally over all user classes. However, limiting token movement within the same class may be easier to support, e.g. excess tokens from Gold users can only be redistributed to other Gold users. Fig. 3 with the dashed box (left) illustrates this solution: each SLA class has its own additional bucket space. Hence, for each token bucket, each token that is not consumed by a data stream (i.e. there is no data element from the corresponding data stream arriving at such particular time) instead of being discarded, the token is sent to the additional bucket space associated to the class. Other data streams of the same SLA class can, therefore, make use of the unused resources. By following this strategy, we are assuming a relationship between tokens and computational resources, which is discussed in detail in Section 3.4.3. The capacity of the additional bucket is $B_\text{max}$ and may be different for each SLA class. For instance, $B_\text{gold} > B_\text{silver} > B_\text{bronze}$. The rationale behind different values for $B_\text{max}$ 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 resource may have been statistically allocated. It is possible to generalize this architecture for a higher number of classes where $B_\text{Gold} > B_\text{Silver} > \ldots > B_\text{Bronze}$.  

3.4.2. Distribute 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. Fig. 4 illustrates this redistribution process from a Bronze user to a Silver one. Redistribution from low to high priority streams is a more 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 the time of shortage, the profit model will decide again between the 3 possible actions it can perform.

3.4.3. From tokens to resources

The dynamic redistribution of resources is realized through a token management mechanisms, requiring tokens to be associated with VMs or physical processors. We assume that computational resources are initially allocated based on a negotiated SLA between the client and the Cloud provider. This enables the underlying resources to be abstracted, as the number of actual VMs (for instance), allocated for each token can be a decision made by the Cloud provider. During execution, shortage of resources are handled in an abstract manner using tokens, i.e. either by collecting unused tokens or by redistributing pre-allocated tokens. In both cases, the mapping of tokens to the available resource pool can change over time.

4. System architecture Petri Net specification

The specification of architectural components has been modeled using a Petri Net that uses Java as inscription language. We
use Petri Nets as an executable architectural component description language that provides precise and concise specifications of complex concurrent behaviors. The use of Java complements Petri Nets with the modeling of complex data structures and the integration with different Java libraries such as the rule engine or the integration with Cloud infrastructures. In this way, a Petri Net-based model can be simulated and finally executed and tested with integration of external (non-Petri Net based) components.

4.1. Reference nets models

Petri nets [21] have been recognized for their ability to represent parallel or concurrent processes. A Petri net is a graph with two kinds of nodes, places and transitions, which represent conditions and actions. Places can also contain elements called tokens, which evolve through the places to complete the state representation. The execution of actions require the satisfaction of preconditions represented by input arcs going from places to transitions, whereas postconditions are specified by output arcs. In high-level Petri nets, nodes are typed representing the type of state for each place, the type of event for each transition, and the type of objects associated with the tokens that flow through the net.

Specifically, the Reference net formalism is a special class of high-level Petri net (adhered to the Nets-within-Nets [22] paradigm) that uses Java as an inscription language, and extends Petri nets with dynamic net instances, net references, and dynamic transition synchronization through synchronous channels. The input and output arcs have a behavior similar to ordinary Petri nets. Petri nets are synchronized by means of a Synchronous Channel (SC). The input arc of a synchronization channel is a binding that is used by the synchronous channel to synchronize with the output arc of another net.

4.2. Reference nets model for architectural components

Additionally, there are creation inscriptions that deal with the creation of net instances and synchronous channels. New net instances can be created by transitions that carry creation inscriptions, which consist of a variable name, a colon (:), the reserved word new and the name of the net. Net instances can communicate with each other by means of synchronous channels. Finally, Reference nets can be enacted by the Renew interpreter (a Java-based editor and Reference nets [23] simulator).

Fig. 5 displays the Reference net for a node in our system. A node consists of a sequence of three different parts: the Traffic Shaping component implemented by a token bucket manager (TBMng, which stores the set of TBs), a Processing Unit manager (PU) that corresponds to the QoS Provisioning component and the Autonomic Data Streaming Service (ADSS). The Autonomic Data Streaming Service (ADSS) handles transmission of data to the following node. It 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 losses). Data elements are streamed from their source to their sink, and may be processed en-route at intermediary nodes (referred to “in transit” processing), rather than entirely at source/destination—details of the ADSS net can be found in [24].

From left to right in Fig. 5, each place of the node net stores the corresponding TBMng, PU and ADSS net instances. Transitions t1–t3 instantiate these nets with their corresponding initial values. When data ds enters into the node, it arrives at the TBMng component (Transition t1). The tuple ds contains the datum and the reference to the data stream. When there are computational resources available (VMs) in the node, it enters the PU component for the execution in Transition t2 and when it finalizes the execution it goes through Transition t31 (entering the ADSS transmission component in t32). Finally, the data element will be transmitted to the following node after firing Transition t4.

It should be noted that this node net is connected to Net Control depicted in Fig. 6, which actually controls the execution flow. The specific mechanism for triggering the flow of the node net and its underlying nets is based on Synchronous Channels. In particular, Transitions t5 from node net and t6 from control net are synchronized by means of a Synchronous Channel control, which adds a new arriving data stream for control. Analogously, Transitions t7 (from the node) and t8 (from the control) are synchronized for introducing a new arriving data stream into the rule based control (Synchronous Channel insertDataStream). Finally, Transitions t9, t10 (from node) and t11 from Net Control...
are synchronized for monitoring the execution. From $t_9$, information of state from the corresponding TB is obtained, from $t_{10}$, information of occupancy of the PU buffer is retrieved, and from $t_{11}$ an action is being triggered. Transition $t_{11}$ is the responsible for firing $t_9$ and $t_{10}$ in accordance with the control period that each data stream specifies. On the other hand, Transitions $t_{b2}$ (node) and $t_{17}$ (control) support the dropping of data temporarily buffered in the TB, whereas Transition $t_{b1}$ permits the control in its Transition $t_{20}$ to update TB parameters associated to a data stream.

Fig. 6 displays an abstraction of Net control—the control loop of a node. The rule engine described above is instantiated at Transition $t_{i4}$ and it is periodically fed with information about available resources (computing and network) at the node. At a sampling rate defined for each data stream, control loops evaluate the actions to be taken for each data stream. As discussed above, Transition $t_{11}$ initiates this control loop collecting information of TB and PU buffer occupancy. Transition $t_{12}$ executes the rule engine by collecting the specified aggregate input and output rates. The rule engine takes two kinds of actions, globally at the node level taking into account the aggregated traffic, and actions for the data stream that is in control in this moment.

Actions at the node level can incorporate or release resources and prioritize or violate the SLA of data streams. Internally, the rule engine will mark data streams whose SLA are violated, and the number of resources to be added or released. Actions related to the addition of resources taken from less prioritized data streams correspond to Transitions $t_{14}$ and $t_{15}$ using the parameter $numPU$ to add/remove the required number of resources. These channels synchronize with Channels $u_1$ and $u_2$ of Fig. 5 and regulate the number of resources in the PU net.

After these transitions, the next place will enable a flag to control parameters related to the data stream under control. When the flag takes the value 2, Transition $t_{20}$ synchronizes with Transition $t_{b1}$ of Fig. 5 to update and coordinate the token bucket rate parameters with the number of resources. The data drop action mentioned previously is performed when the flag takes the value 1. If the flag takes the value 0, no action correction is performed.

Rules are implemented using JESS (Java Expert System Shell) [25], which can support event processing as event–condition–action (ECA) rules [13].

### 4.2.1. Reference nets model for TBM

Fig. 7 depicts the token bucket manager (TBM) 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 $t_1$. If it is the first stream data element, Transition $t_3$ will be enabled and Transition $t_2$ disabled. Otherwise, the contrary occurs. In the former case, the new token bucket instance for the data stream will be created in Transition $t_{b1}$ and the data element will be added to its corresponding token bucket instance when Transition $t_{b2}$ fires. In the latter case, the data element will be added to its corresponding token bucket instance when Transition $t_{b2}$ fires. Once a data element is allowed to proceed, Transition $t_{b3}$ is fired and the data element moves to the PU component via Synchronous Channel: $end(ds)$ in Transition $t_7$. Transitions $t_8$ and $t_9$ update PU parameters and the size of the data in TB buffers respectively.

The model has been extended with a common bucket place to collect surplus tokens from all token buckets. 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. Transition \( t_{14} \) collects these tokens. A data element is allowed to proceed from buffers in any TokenBucket net with empty buckets (when Transition \( t_{7},t_2 \) fires) if the total number of data accumulated in buffers is under a threshold (2 data in the figure). This value is updated by transition \( t_{16} \) with channel: \( bufPU(n) \). In this way, tokens in the common bucket are used when there are enough processing resources to support the current aggregated rate. At the end of each control period, the common bucket is emptied by Transition \( t_{15} \) with channel: \( emptyB \); therefore the lifetime of collected unused resources is limited to one control interval. The \( TB \) net details can be depicted in the Appendix.

### 4.2.2. Reference nets for the resource manager

Fig. 8 shows an abstraction of the PU net that specifies the Resource Manager behavior. Each time a data element is sent to the resource manager to be processed, a reference to the data stream arrives in Transition \( t_1 \), and once it is processed it is sent to the ADSS component by \( t_2 \). Transition \( t_{18} \) retrieves data from the buffer and assigns idle resources from the Processing Units Place, \( t_{19} \) begins the data processing and \( t_{20} \) receives the result. Subsequently, \( t_{12} \) releases the resource back into the idle processing units pool. The main place of the net is Place Buffer, which contains NetBufferMng net. This net is analogous to the upper net in Fig. 7, but instead of TBs, it contains FIFO Buffers. Transition \( t_{35} \) allows the controller to monitor data buffered waiting for entering into processing for each data stream. Transitions \( t_{14} \) to \( t_{6} \) specify a list of operations that can be carried out on this node.

In this reference net, there are two places managing resources. On one hand, Place PU proxies and its associated transitions trigger processes that interact with an IaaS middleware to allocate and de-allocate VMs. Transition \( t_{23} \) allows the controller to trigger a process that communicates with a Cloud middleware a create and deploy new VMs, whereas Transition \( t_{22} \) allows the controller to trigger the opposite operation, deleting a VM instance from the node. On the other hand, Place PU Proxies is used to provide the available resources for processing. Place PU Proxies stores surrogate references to computational resources (i.e. VMs) and its main purpose here is for the creation and destruction of computational resources: once a computational resource is created, then it is transferred in to Place Processing Units. In contrast, when a computational resource needs to be removed, it is transferred from Place Processing Units to Place PU Proxies. The actual mechanism for implementing this has also been synchronous channels. In particular, the controller communicates its actions via Transitions \( t_{10} \) and \( t_{15} \), which add and remove, respectively, computational resources. Specifically, Transition \( t_{10} \) will end up firing Transition \( t_{13} \) which synchronizes with \( t_{22} \), while \( t_{15} \) will enable the firing of \( t_{8} \), which synchronizes with \( t_{21} \). In the simulation scenarios, computational resources have been modeled by sub-nets with temporal delays simulating processing actions. However, in a real implementation, they are Java pieces of code containing socket objects connected to the VMs via TCP/IP communication channels. Transitions \( t_{27} \) to \( t_{29} \) support the retrieval and update of variables for the node (i.e. number of processing units, list of operations that the node is able to perform, etc.).

For the sake of simplicity, the following simplifications in the model were made: (i) Data from the buffers are taken in a round-robin manner. Clients in this case expect a similar performance as an unloaded best-effort node. In case of hard real-time constraints, the admission control test depends on the scheduling strategies such as First-Come–First-Served, Static Priority, or Earliest-Deadline-First [17]. (ii) We assumed constant processing rate and not data inflation/deflation although they can be simulated as shown in [9]. In the simulation, only one operation (simple or a set of operations specified by an EPN subnet) by node is processed. The Resource Manager might be processed as operations as possible in the EPN in the case enough resources are available, supporting a dynamic in transit processing of the EPN [24].

### 5. Evaluation scenarios by simulation

We propose three evaluation scenarios to show the behavior of our controller. Scenarios: (i) the addition/removal of resources to the queue that provisions “Gold” streams taking resources from “Bronze” streams; (ii) the selective violation of “Silver” data stream SLAs to avoid violations of Gold data streams; and (iii) a final scenario 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 3.4. These scenarios were chosen to demonstrate how profit generation is affected by the choice of a resource allocation strategy within a node, using components discussed in Section 3 and using rules and Token Bucket extensions presented in Section 3.3, and Section 3.4, respectively. Due to space limitations, the first and second scenarios can be found in Appendices 2 and 3, while we are presenting the last scenario in this section.

We assume that a penalty occurs when the PU buffer occupancy of a data stream is greater than a pre-defined threshold, which
means that not enough resources are provided. We consider that each
token allows a data chunk to access the processing phase,
representing a predefined number of events (e.g. 1 data = 10³
events) and that a unit of cost is incurred for each unit of processing
rate (data/s). The penalty for “Gold” streams will be two times the
cost of the required resources to provide the service and one time
for “Silver” streams. For simplicity, we assume that: (i) resources
are homogeneous, and (ii) there is no data size variation within
the elements of a stream during processing. We note that a number of
other scenarios can also be defined, based on the context of use of
the proposed system. All scenarios have been simulated interpreting
the reference net models with Renew.

5.1. Scenario: Redistribution of resources

In this scenario, we validate the TB extension with a common
shared bucket for redistributing allocated resources. Tokens in the
shared bucket represent non used resources, but they may not have the
same representative value in all cases when the provider fol-

ows the under provisioning strategy. For this reason, we consider two sub-scenarios: (i) using the rule-engine controller, which val-
idates the shared bucket in an elastic provisioning approach with
tokens representing reliable allocated resources; and (ii) without
using of controller actions, which validates the shared bucket with
more volatile tokens.

5.1.1. Redistribution of tokens in an elastic scenario

The third scenario considers data streams at the same priority
level. Table 3 identifies and summarizes simulation parameters. We
assume 4 Gold (i.e. high priority) customer streams with a
period of control of \( T = 1 \) s and all data streams have the same
requirements: \( R = 20 \) data chunks/s on average and an allowed
burstiness of \( b = 10 \) data chunks. The maximum number of data
to be processed is 120 data chunk/s 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 and 5 s 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 chunks/s half of the time). For the first
set of simulations we compare the behavior of the system with and
without the shared bucket (of capacity \( B_{\text{gold}} = 80 \) 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 an elastic way (borrowing resources
from low priority data streams) and tuning TB parameters to use
the newly 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 profit with Eq. (1) we assume a price of 20 units
per PU for clients and a cost of 15 units per PU for provider. We
assume that 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 sends 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
time intervals due to the lack of resources.

Fig. 9 shows the provider’s profit for different number of ini-
tial PUs and in an elastic provisioning of resources scenario. The
x-axis represents the initial baseline number of resources and the
y-axis the aggregated profit over 300 s of simulation. These results
show the maximum profit 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 the aggregated throughput significantly as shown in
Fig. 11, but the throughput of each individual data stream is im-
proved as shown in the sample data stream output of Fig. 10. If we
look at time interval 70–80 s and 140–150 s, we can see that the
shared bucket allows the output throughput to follow the input
data injection rate more closely. This behavior can be more clearly
seen with 9 PUs than with 4 PUs, i.e. when there are enough re-
sources globally. Without the shared bucket, the output through-
put 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.

Fig. 12 left shows that the average number of PUs provided
(their cost being represented by the last term in Eq. (1)) in an elastic
scenario is not affected by the use of a shared bucket. However,
Fig. 12 right illustrates that the number of penalizations decreases
with higher number of baseline PUs. Besides, it is important to
highlight that for the same number of baseline PUs, the number of
penalizations is less with the use of a shared bucket.

5.1.2. Redistribution of tokens in a non-elastic scenario

This scenario uses the same number of data streams as pre-
viously, 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 sce-
nario 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 and 3 s, but now an ON period have
a probability of 1/3 to occur. Again, data injection rates follow a
Poisson distribution. Therefore, for 4 data streams sending on aver-
age 20 data chunk/s the number of required resources is around 3.

![Fig. 9](image_url)

**Fig. 9.** Scenario redistribution of tokens in an elastic scenario with different baseline PUs (horizontal axis) and using control loop to avoid penalties. Profit is measured in an abstract unit, but can be mapped to a particular economic currency.
Fig. 10. Data stream samples input and output in elastic provisioning scenarios.

Fig. 11. Aggregated input and output in an elastic provisioning scenario.

Fig. 12. Average number of PU in an elastic scenario and number of penalties.

Fig. 13 shows the provider’s profit with different number of initial provisioned PUs. With less than 3 PUs, the number of penalizations makes the profit to decrease and the shared bucket gives a lower profit 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.
6. OpenNebula-based implementation

In previous sections, we validated our models in terms of simulation. In this section, we want to test the feasibility of our proposal in a real Cloud infrastructure. For such a purpose, we exploit an OpenNebula data centre for a real implementation of our model. The fact that our Reference net models are executable, as they can be interpreted by Renew, allows us to interface directly with OpenNebula from the nets: create and switch on and off Virtual Machines (VMs), transmit data to the data centre and collect back the results.

6.1. OpenNebula functionality and architecture

OpenNebula exposes user and administrator functionality for creating, and managing private or hybrid, heterogeneous Clouds. In particular, OpenNebula provides virtualization networking, image, and physical resource configuration, management, monitoring, and accounting [26]. Services can be hosted in VMs and then submitted, monitored and controlled in the Cloud by using Sunstone or any of the OpenNebula system interfaces, namely Command Line Interface (CLI), XML-RPC API, OpenNebula Ruby, and Java Cloud APIs.

The hypervisors supported to run VMs are Xen, KVM and VMware, and in order to enable message communication among them, physical and virtual network links can be used. OpenNebula supports the creation of Virtual Networks by mapping them on top of the physical ones. In order to facilitate the creation of virtual machines and to manage and share data, the storage system of OpenNebula is provided to create disk images. These images can be shared among OpenNebula cloud users and used by several VMs. The images are stored at a template repository system with an identifier. Renew uses two concurrent threads for reaching the required traffic rates sending and receiving data to VMs.

6.2. Integration with OpenNebula

Fig. 14 illustrates the integration of our system architecture with OpenNebula. We assume that there is an OpenNebula-based independent and associated pool of VMs to each of our nodes. Then, the nets at each node interface with OpenNebula in two different ways: (i) the Resource Manager net from Fig. 8 makes use of the Command Line Interface through a Java implementation of ssh, connecting to the front-end node, with the purpose of managing VMs, and switching them on and off; (ii) the Resource Manager net from Fig. 8 establishes TCP/IP channels with the VMs available for sending computations on data elements and subsequently gathering the results. The key difference with our models presented earlier is that the Clock nets simulating computational resources in the Resource Manager net have been replaced here by real VMs.

We utilize the OpenNebula template repository for storing the OS image to be used for each VM with the required executables already installed. As a hypervisor, we are utilizing KVM [27] (Kernel-based Virtual Machine). KVM is an open-source hypervisor for Linux OS on x86 hardware. It is fully integrated into the Linux kernel, and supports the execution of multiple virtual machines running unmodified Linux OS or Windows OS images. Each VM has private virtualized hardware: a network card, disk, graphics adapter, etc. For the purpose of this paper, we run 64-bit Scientific Linux OS VM on an x86 virtualized hardware. Once the VMs are all set up, they are ready for the operational purpose of the Resource Manager component: they can be switched on and off depending on the processing requirements. Finally, we assume that all the computational resources within a data centre can communicate with a high-speed LAN, reducing the communication latency and overheads.

6.3. Evaluation scenarios

In this section, we want to highlight the feasibility of the approach in a real scenario. Before deploying our cloud service infrastructure, we tested that we can enforce QoS among the data streams with OpenNebula and KVM. In particular, we verified and tested that the hypervisor of VMs is not introducing significant packet jitter, altering the packet inter-arrival time. The results of such an evaluation can be found in [28]. Due to space constraints, we will just reproduce the third scenario from Section 5.1. In such a scenario, we validated the TB extension with a common shared bucket for redistributing allocated resources. In Section 5.1, we considered two sub-scenarios: (i) redistribution of tokens in an elastic scenario, and (ii) redistribution of tokens in a non-elastic scenario. We are focused here on the elastic scenario.

The Traffic Shaping and QoS Provisioning components are interpreted by Renew 2.4 on a Mac mini with 2.4 GHz Intel Core 2 Duo and 8 GB at the University of Zaragoza (Spain) and the VMs are deployed in the OpenNebula cloudmip platform at the University of Toulouse (France). Each token represents a 1000-byte packet in this scenario and they are sent to the VMs with an operation name and an identifier. Renew uses two concurrent threads for reaching the required traffic rates sending and receiving data to VMs.

Fig. 15 illustrates that the aggregated real traffic (input and output) and the traffic of a sample data stream have a similar behavior than the one obtained by simulation in Figs. 10 and 11. Besides, Fig. 16 demonstrates that the use of a shared bucket and the rest of control mechanisms proposed in Section 3 achieve in a real system implementation a profit graph with a slope similar to the ones obtained by simulation. Finally, Fig. 17 illustrates the occupancy of the buffers involved, as well as how the elasticity mechanisms of the system are triggered when required (i.e. the occupancy of the buffer is above a given threshold, more VMs are switched on and added to the pool).

7. Related work

Resource provisioning, resource allocation, resource mapping, and resource adaptation in Cloud-based infrastructures have received significant attention over the last years [29]. Three main approaches have been pointed out to scale resources. First, reactive mechanisms mainly use monitored values and apply elasticity rules or threshold-based rules pre-defined by service providers [30,16,31]. Second, predictive mechanisms try to learn from previous data history and resource usage to construct mathematical models to forecast resource demands. These approaches

Fig. 13. Scenario III: Redistribution of tokens in a non-elastic scenario with different baseline PUs (horizontal axis).

Fig. 14. Integration with OpenNebula.

Fig. 15. Resource management for bursty streams on multi-tenancy cloud environments, Future Generation Computer Systems (2015), http://dx.doi.org/10.1016/j.future.2015.03.012
are useful when regular behavior pattern can be identified and predicted [32–34]. And third, hybrid approaches [35,36] that integrate the two previous ones.

In this paper, we focused on dynamic provisioning in the Cloud, but in the scope of data streams. Indeed, data streams have also been gaining interest, as the significant proliferation of geographically distributed sensors has led to a number of applications in areas such as surveillance and monitoring, smart-traffic management, cities, etc. These applications need to process large volumes of data in a stream basis as they become available, generating new challenges in scalability, storage, processing, and storage. Dynamic resource provisioning in that context has been considered in [37], where the main goal is to allocate resources for one particular data stream dynamically from a Cloud, so that the processing rate can match the rate of data arrival. Besides, they also consider variable transient input rates. Another similar approach
that provides autonomic auto-scaling of resources is given in [38].

There, the challenge is to dynamically provide with resources under the presence of large bursts of data. Our proposal differs from both of them in that we consider multiple data streams being processed over a shared Cloud. Moreover, in addition to provide technological support to enforce Quality of Service per data stream, we also build a business model that triggers different technological actions to provide resources dynamically and maximize Cloud provider’s profit. In particular, we make use of TB, the scheduling buffers, and the autonomic computing control loop in order to schedule application data elements onto processing resources. The main advantage of such a combination is that (i) we utilize TB parameters for specifying application QoS; (ii) the TB along with the buffers and the autonomic computing loop is a simple and QoS-driven scheduling heuristic, supporting variable bursts; (iii) besides, in this paper, we integrate the profit model with our scheduling heuristic.

On the other hand, Data Stream Management Systems (DSMS), workflow and event processing technologies have also been dealing with data stream processing from different perspectives. Indeed, they share a number of important similarities and challenges such as scalability, fault tolerance and performance that enable them to be considered synergistically [39]. DSMS typically partition their operations onto distributed processing resources and they incorporate different scheduling heuristics and QoS depending on the application characteristics. In DSMS, the parallelism is therefore extracted from the data stream query operators they provide, such as in Aurora [40], Borealis [41] and Stream Cloud [42]. They differ from our proposal in which we do not extract parallelism, but we explicitly require the user to express it for each data stream and we also exploit it by processing multiple data streams simultaneously.

A number of Complex Event Processing (CEP) systems have seen a resurgence in the last few years exploiting the distributed computing paradigm for tackling large-scale data stream processing, such as Yahoo’s S4 [43], IBM InfoSphere Streams [44], or DROOLS Fusion [45]. They provide programming abstractions to build and deploy streaming tasks as distributed applications at scale for commodity clusters and clouds. Nevertheless, they do not deal with variability in input rates, which is our focus in this paper. In general terms, unlike Data Stream Management Systems (DSMS), the notion of QoS is not present in event processing literature [39].

Finally, there is a number of scientific workflow engines that incorporate the streaming workflow model of computation and elastic infrastructures, such as Kepler [46] and Triana [47]. But to the best of our knowledge they are not considering dynamism in streaming income rate nor they are providing QoS guarantees. In [48], the authors propose a workflow specification where each task consists of one or more alternate implementations with different non-functional properties, so that the system can choose any of them dynamically at runtime. In this paper, we have not considered dynamism at specification-level, but the focus is at dynamic provisioning of resources driven by a profit-based business model.

8. Conclusion and future work

There is an emerging interest in processing data streams over shared Cloud infrastructures, with data elements being processed at distributed nodes in transit from source to sink. We consider the execution of simultaneous data stream over such infrastructure, with each stream having particular QoS objectives (throughput or latency, for instance), expressed within an SLA. We established three different classes of customers submitting data streams (Gold, Silver and Bronze), with each class providing a different revenue and penalty to the provider. In this paper, our aim is to enforce QoS for each application and use a profit model that combines cost of provisioning and penalties incurred due to SLA violations. With a unified token-based resource management model, we proposed corrective profit-oriented actions. As dynamic corrective actions, we considered (i) to re-distribute unused resources among users, and (ii) to re-distribute pre-allocated resources from less prioritized users to more prioritized users. All the control logic have been implemented with a Reference net model that can be used for both simulations and real deployment. We presented extensive simulations of various scenario demonstrating the effectiveness of our proposed profit-oriented control mechanism and the unified token-based resource management. We also presented promising preliminary results in deploying our control architecture on an OpenNebula Cloud infrastructure, extending the Reference net model to an executable environment in a very simple manner. However, we want to investigate in the future how the integration of our profit-based model in other Cloud middleware can be realized in a generic way, offering the possibility of specifically taking into account the different functionalities of existing Cloud middleware such as varying the number of VMs, varying the amount of memory or the amount of CPU cores.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.future.2015.03.012.

References


Rafael Tolosana-Calasanz is currently an Associate Professor at the Computer Science and Systems Engineering Department of the University of Zaragoza, Spain. He holds a Ph.D. from the University of Zaragoza and his research interests lie in the intersection of Distributed and Concurrent Systems, Scientific Workflows and Petri nets.

José Ángel Bañares was born in Zaragoza, Spain in 1966. He received the M.S. degree in Industrial-Electrical engineering and the Ph.D. degree in Computer Science from the University of Zaragoza, in 1991 and 1996, respectively. He joined the faculty of the University of Zaragoza as an Assistant Professor in 1994, and since 1999 he has been an Associate Professor. His research interests include Petri nets, Artificial Intelligence and Distributed Computing.

Congduc Pham obtained his Ph.D. in Computer Science in July 1997 at the LIP6 Laboratory (Laboratoire d'Informatique de Paris 6), University Pierre and Marie Curie. He also received his Habilitation in 2003 from University Claude Bernard Lyon 1. He received a Diplôme d'Études Approfondies (DEA) in computer systems from the University Pierre et Marie Curie in 1993. He also obtained the Magistère d'Informatique Appliquée de l’Ile de France in 1993. He spent one year at the University of California, Los Angeles as a Post-Doctoral Fellow with Professor R. L. Bagrodia. From 1998 to 2005, he was an Associate Professor at the University of Lyon, member of the INRIA RESO project in the LIP laboratory at the ENS Lyon. He is now at the University of Pau and the LIUPPA laboratory. From September 2006 to July 2009, he was Director of the LIUPPA laboratory.

Omer F. Rana is a Professor of Performance Engineering and member of the Distributed Collaborative Computing Group in the School of Computer Science at Cardiff University. He holds a Ph.D. in “Neural Computing and Parallel Architectures” from Imperial College, London University. He has worked as a Software Developer with Marshall Biotechnology Limited (London) and as an Adviser to the US-based “Grid Technology Partners”. He has been Deputy Director of the Welsh eScience Centre. He has participated as a theme leader within a number of European and UK-funded networks. He co-led the “Jini” working group at the Open Grid Forum, and has contributed to the Service Level Agreements efforts (and integration of these with workflow techniques) within the GRAAP working group.