Enforcing Quality of Service on OpenNebula-based Shared Clouds

Rafael Tolosana-Calasanz[1], José Ángel Bañares[1],
Omer Rana[2], Congduc Pham[3], Erotokritos Xydas[4], Charalampos Marmaras[4],
Panagiotis Papadopoulos[5], Liana Cipcigan[4]
[1]Computer Science & Systems Engineering Department, University of Zaragoza, Spain
[2]School of Computer Science & Informatics, Cardiff University, UK
[3]LIUPPA Laboratory, University of Pau, France
[4]Institute of Energy, School of Engineering, Cardiff University, UK
[5]EDF Energy R&D UK Centre, UK

Abstract—**With an increase in the number of monitoring sensors deployed on physical infrastructures, there is a corresponding increase in data volumes that need to be processed. Data measured or collected by sensors is typically processed at destination or "in-transit" (i.e. from data capture to delivery to a user). When such data are processed in-transit over a shared distributed computing infrastructure, it is useful to provide elastic computational capability which can be adapted based on processing requirements and demand. Where Service Level Agreements (SLAs) have been pre-agreed, such available computational capacity needs to be shared in such a way that any Quality of Service related constraints in such SLAs are not violated. This is particularly challenging for time critical applications and with highly variable and unpredictable rates of data generation (e.g. in Smart Grid applications where energy usage patterns may change unpredictably). Previously, we proposed a Reference net based architectural model for supporting QoS for multiple concurrent data streams being processed (prior to delivery to a user) over a shared infrastructure. In this paper, we describe a practical realisation of this architecture using the OpenNebula Cloud platform. We consider our infrastructure to be composed of a number of nodes, each of which has multiple processing units and data buffers. We utilize the "token bucket" model for regulating, on a per stream basis, the data injection rate into each node. We subsequently demonstrate how a streaming pipeline can be supported and managed using a dynamic control strategy at each node.**

I. Introduction & Motivation

Over recent years there has been a growing proliferation of sensors and the embedding of sensor technologies in physical infrastructure (such as in built environments, to support environment monitoring, etc). Work in this area builds on applications of wireless sensor networks (WSN), making use of the collaborative output of a large number of nodes. As a result, there is a need to develop software applications that process the data coming from such sensors "on the fly", i.e. as it is being generated and streamed, and that do not fit into the model of traditional databases and querying paradigm (i.e. where the data needs to be archived prior to processing). Applications that have such characteristics require real time analysis of such data streams, as often the subsequent actions undertaken are dependent on the processing of data over a particular time/sample window. A variety of applications match these characteristics, such as in electricity networks (often termed as "Smart Grids"), for monitoring/managing the demand of energy and its distribution; in surveillance systems for monitoring of behaviours, activities, etc. of people; in biological systems for the observation of a species population (in danger of extinction, protected or of interest to scientists); in traffic management (for monitoring traffic pattern behaviours to manage journey times, manage congestion or support more 'active" management of road networks); in social sciences to determine sentiment of a particular community through real time analysis of a Twitter data stream, etc. In these applications, sensors can range in complexity from a large, environmental instrument that is able to record and transmit data, to the use of specialist wireless sensor networks, transmitting at pre-defined intervals to people-based sensors (where mobile devices are used to generate data at unpredictable times). Another class of applications fall within the area of "Urgent Computing" – which refers to providing prioritized access to computational and data resources to support emergency computations such as severe weather prediction during matters of immediate concern – such as hurricanes, flooding, medical emergencies, etc. Immediate access to computational jobs in critical emergencies must be given, and time cannot be wasted in job queues of high end computational resources. Moreover, as the number of sensors increase, the task of data management (storing and processing) design and implementation within such applications becomes more challenging.

In general terms, when data is generated from a "source", and the result of its analysis is required at a "sink", there are two main ways to analyse and process such data: *in-transit* or *in-situ*. In-situ analysis involves use of computing capacity at the location where data is generated or archived, sometime not possible and difficult to carry out due to cost or time deadline constraints. An alternative approach involves processing the data during transmission to the sink (provided the data is routed through intermediate nodes on its path from source to sink). In this paper, we propose an OpenNebula-based system which enforces Quality of Service (QoS), measured exclusively in terms of throughput, for the simultaneous processing and analysis of multiple data streams over a shared, in-transit, data processing infrastructure. Our work is motivated through scenarios from Smart Grids (electricity networks), where sensors (smart meters) generate data at highly variable and unpredictable bursts. Our system can process and transmit data streams which have such characteristics. It is assumed that the network bandwidth for data transfer is not the bottleneck in the system. We also assume that the processing of a data stream is composed of a sequence of stages and data is routed through these stages. A Petri net-based model of this architecture has been presented in [1], our key contribution is to demonstrate how such an architecture can be realised using the open source OpenNebula Cloud system. A key mechanism for enabling QoS (throughput) in our architecture is the coordinated action of the token bucket-based admission & regulation component with the resource provisioning component (that manages access to virtual machines (VMs) in the OpenNebula system). Therefore, we have to guarantee that the virtualisation overheads do not significantly alter the original incoming rate of packets within each data stream. In particular, we tested that the inter-arrival time between consecutive packets within a data stream being transmitted to a VM is not altered by the hypervisor with significant packet *jitter*.

The remainder of this paper is structured as follows: Section II briefly introduces the system architecture already presented in [1], showing a more detailed description of the admission control models. Section III describes case studies where variable and unexpected rates of data injection may occur, which motivate the need for our proposal. In Section IV, an OpenNebula based implementation of our model is described. Section V presents our evaluation scenario, analysing the overhead due to the use of virtualisation technologies. In Section VII, related work is briefly discussed. Finally, conclusions and future work are given in Section VIII.

II. Autonomic Streaming System Architecture

The system architecture and its corresponding Reference net based models were presented in [1], [2]. The *Reference net* formalism [3] was used to specify the architectural components. Reference nets is a particular class of high-level Petri net that uses Java as an inscription language and extends Petri nets with dynamic net instances, so-called net references, and dynamic transition synchronization through synchronous channels. Our system architecture consists of a pipeline of autonomous computational stages. Each stage consists of a combination of data admission control & regulation, computation, data transfer capability and a rule-based controller component.

The data admission control & regulation component is based on a token bucket model, whose main objective is to isolate data streams, and to regulate access to the subsequent computational stage. At each control period, the access for stream i is accomplished at a rate of R_i data elements, the choice of this parameter is based on the average incoming rate from historical information for i. Additionally, it also allows for a configurable maximum burstiness b_i, above R_i for each data stream i. The data admission control component guarantees that there will not be *starvation*, and that the established QoS per stream will be enforced. The computation stage consists of an elastic pool of computational resources that can be switched on and off at runtime, based on the incoming rate of data streams at each node. This is equivalent, for instance, to the provisioning of a VM (from a pool of available VMs) at runtime. Incoming data are buffered after the admission control stage and before the computational stage. The buffer occupancy is monitored by the computational stage controller, in order to adjust the computational capacity (i.e. computational nodes are added / removed) on demand based on the accumulation of data in the buffer – requiring additional processing capacity to be made available to prevent buffer overflow. We consider: (i) the average throughput per stream, and (ii) the maximum allowed burst per stream, as QoS parameters. Finally, the transmission component monitors the processing capability of the next node in the pipeline as well as the network bandwidth and autonomically adjusts the transfer rate accordingly to avoid data loss.

For simplicity, for each data stream i, initial values of b_i and R_i parameters are selected based on previous runs, and it should be emphasised that the number of allocated computational resources are dependent on b_i and R_i, as the buffer occupancy between both components impacts computational resource allocation. Therefore, it is crucial to select b_i and R_i appropriately; initially they are both based on average values. Nevertheless, a characterisation for b_i and R_i for scenarios in Section III would require an estimate for R_i and a *maximum* allowed burstiness b_i. Consequently, the token bucket-based data access component in the architecture will not succeed in its objective of isolation of flows: in case of significant bursty behaviour of one data stream, there would be performance degradation for the the remaining data streams, as the computational resources would all be consumed for processing data from the burst.

Moreover, even when rates of incoming data streams are predictable/ regular, data transformations at an intermediate stage of processing may lead to data *inflation* / *deflation* [1]. What may not be known apriori, therefore, is whether for a particular combination of inputs and data sources larger volumes of intermediate output data may be generated, thereby leading to *data inflation* in the pipeline. Conversely, data deflation at a node may lead to an inefficient use of available resources.

The challenges of data inflation/deflation are summarized in Figure 1, which illustrates the various steps involved, starting from SLA negotiation based on the application's data injection rate (step 1). For simplicity, the figure only shows one flow instance. Step 2 involves data access and control based on the R parameter. Steps 3 and 4 show how incoming data gets processed by the first stage leading to data inflation. Subsequently, step 5 identifies the impact on a subsequent stage, where the rate is limited by the token bucket parameters of the stage. This may lead to either data loss or an inability to meet an application's end-to-end QoS constraints. Again, pre-defined values of R and buffer sizes cannot overcome effects of data inflation/deflation on resource management or workflow QoS.

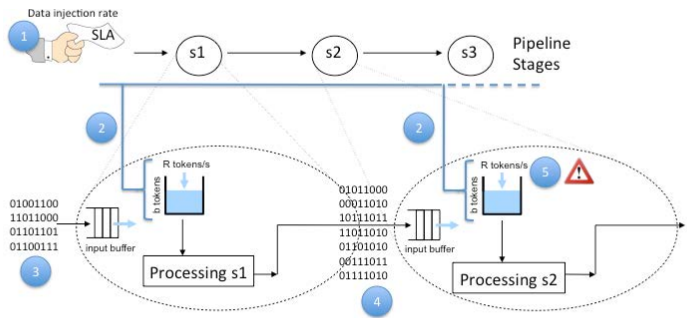

Fig. 1. Example of data inflation at the 2nd stage

A solution to these problems were proposed in [1], and consists of supporting an adaptive R_i token bucket parameter and a fixed b_i for an allowed burst of around 10% to 15%. In the example of Figure 1, there is a data inflation at stage S2 that requires an increase of parameter R_i at this stage. The controller monitors the incoming rate at each node, and when there is a income rate that is following an increasing or decreasing *tendency*, the R_i parameter is updated accordingly.

A. Reference net-based models

Reference net models to support multiple stages in the architecture, as illustrated in figure 1 are provided in [1], [4]. In this subsection, we provide a more detailed description of the traffic shaping model used in our system.

Each node contains three different components: a token bucket manager, a Processing Unit (PU) manager and an Autonomic Data Steaming Service (ADSS). When a data stream enters a node, a tuple with reference to the data stream is generated, leading subsequently to the transfer of data to the token bucket manager component. If constraints in the token bucket manager are met, data elements are passed to the PU manager, followed by the ADSS which forwards these (depending on the network characteristics) to the next node. Figure 5 shows the traffic shaping component modeled by the `TBMng` reference net and the QoS provisioning component that acts in coordination with it. The QoS provisioning component includes a monitor, controller and an adapter to manage computational resources according to the admission control rates. The `PU` and `ADSS` nets description has been removed for simplicity – but are discussed in [4].

Whenever the token bucket manager allows a data element to proceed to the PU component, it enters first into a FIFO buffer, waiting for any available computational resources. The number of computational resources at the PU is regulated by the autonomic controller depending on this FIFO buffer occupancy. A long term (defined in terms of application characteristics) buffer size increase indicates that the number of computational resources must be increased, and vice versa for a decrease in the buffer occupancy.

The Reference net in Fig. 2 shows the token bucket (`TokenBucket`) net which receives data elements arriving

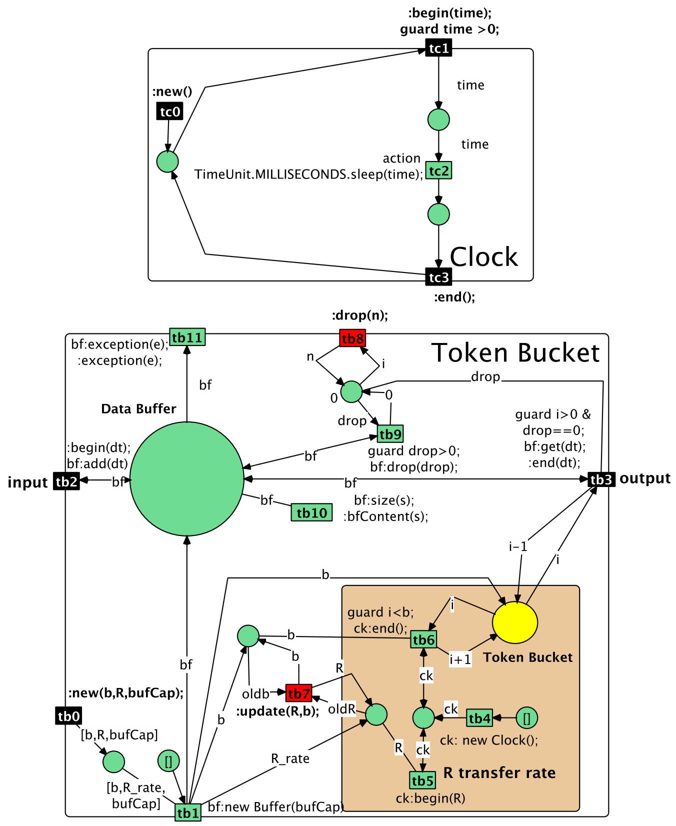

Fig. 2. Token Bucket and Clock nets

in *input* Transition $tb1$ and leaving at *output* Transition $tb3$. Once a data element enters into the token bucket it is stored in the bf buffer, implemented in this case as a FIFO list, in Place $DataBuffer$. The output Transition is only enabled when there is simultaneously an element in the buffer and a mark in Place $TokenBucket$. A mark in Place $TokenBucket$ will be added by the clock shown in the bottom right part of the figure at the rate R. Thus, irrespective of the arrival rate of data elements into the token bucket from previous stages, they will only be allowed to proceed to the PU at a constant rate of R. Transition $:update(R)$ modifies the parameter R. Transition $tb7$ updates the token bucket parameters, and Transition $tb8$ drops n data elements from the FIFO list/buffer.

Fig. 3 depicts the token bucket manager (`TBMng`) component. The upper part of the net forwards incoming data elements to the corresponding token bucket. Each time a data element is injected into a data stream, a reference to the data stream with the agreed values (b, R) arrives in Transition $t1$. If it is the first stream data element, Transition $t3$ will be enabled and Transition $t2$ disabled. Otherwise, the contrary occurs. In the former case, the new token bucket instance for the data stream will be created in Transition $t5$ and the data element will be added to it when Transition $t6$ fires. In the latter case, the data element will be added to its corresponding $tokenbucket$ instance when Transition $t4$ fires.

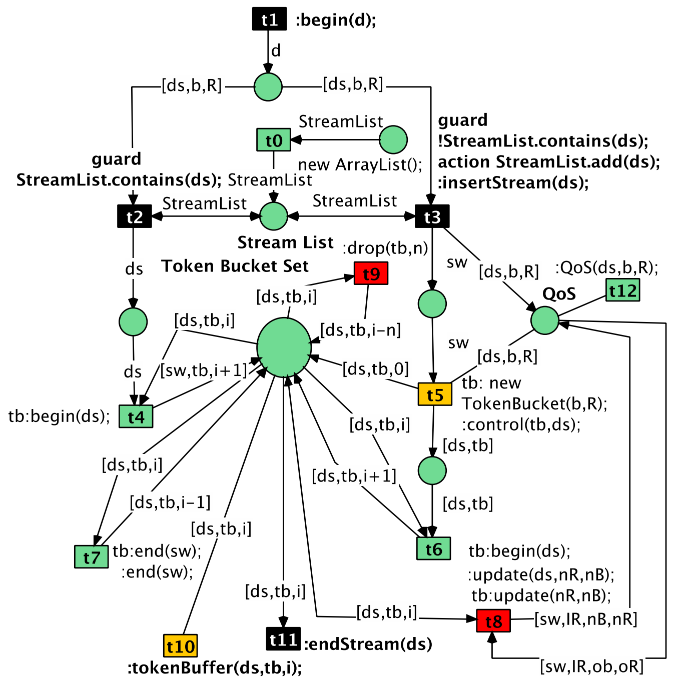

Fig. 3. The Token Bucket Manager (TBMng) net.

Once a data element is allowed to proceed, Transition $t7$ is fired and the data element moves to the PU component via synchronous channel : $end(ds)$ in Transition $t7$. Transitions $t8$ and $t9$ update token bucket parameters and the number of data elements in the associated buffers respectively.

III. Case Studies: Highly, Unexpected Data Generation Rates

We describe two different case studies which involves streaming data being generated at different, highly variable and unexpected rates. The first scenario focuses on data generation from energy meters (measuring energy consumption and forwarding this to utility companies) in Smart Grids. The second scenario corresponds to energy usage in Electric Vehicles (EVs), which will be connected to smart grids in order to charge their batteries, and therefore represent a variable (and unpredictable) load on an energy network. Charging of EVs is dependent on a number of factors, such as road congestion, pricing, driving behaviour of a user etc.

A. Case Study 1: Smart Metering in Smart Grids

Power networks are currently evolving to Smart (energy) Grids, a promising solution for improved energy efficiency, manageability and controllability of available resources in electricity networks. From an architectural point of view [5], smart grids consists of three layers: (i) the physical power layer for transmission and distribution of electricity – the smart physical power layer consists of a wide variety of Distributed Energy Resources (e.g. generation, controllable loads and storage) and power consumers. (ii) The Advanced Metering Infrastructure (AMI), providing a bidirectional communication capability by connecting the electric generators and power consumers. (iii) The application layer, which includes a number of applications for managing power networks such as the automatic gathering of metering information from consumer premises, support for interaction with third-party vendors by the inclusion of customer equipment for monitoring and control, demand response (influencing consumer electricity consumption patterns – i.e. by altering electricity prices), load monitoring & forecasting of energy demand, etc. Energy meters used in AMI are essential for a number of smart grid functionalities such as in monitoring the influence on consumer electricity consumption patterns (demand response).

The data that smart meters transmit can be bursty in nature as these devices are not synchronised and send data independently of each other, a characteristic that may lead to instantaneous peaks in data traffic [6]. As proposed in [6], the transmission of data at a constant rate from the meters may also result in inefficient usage of network bandwidth and computational resources. The rationale behind this is based on the premise that as the total power usage within the utility approaches total available capacity, usage information is required more frequently to detect and forecast a peak load event with low latency to allow a timely response. Therefore, smart meters need to transmit data with a dynamic frequency within each geographic area. energy demand and its .

B. Case Study 2: Electric Vehicle Charging Process

There is an gradual inclusion of Electric Vehicles (EVs) into production facilities and marketing plans of all car manufacturers. These vehicles are anticipated to gain an important market share over conventional Internal Combustion Engine (ICE) powered vehicles. Nevertheless, in order to re-charge their batteries, EVs will have to be connected to Smart Grids [7]. According to recent studies [8], by 2030, in case the charging of EV batteries is left uncontrolled, a significant increase in the electricity demand peaks is to be expected. Moreover, the impact of EVs is anticipated to be at the local level where hotspots will be created that depend on how EVs will cluster within a particular geographical location, creating a possible overload on the low voltage distribution network within that area.

From an electricity infrastructure perspective, EVs can be considered as: (i) mobiles devices expected to be able to connect at various locations at different times, and when they connect, they draw a continuous current from the electricity network; (ii) flexible loads that may allow electricity companies to interrupt or coordinate their charging procedure; (iii) storage devices that may allow electricity companies to request power injections from their batteries back to the energy grid (called Vehicle to Grid *V2G*). Energy companies will need to process consumption data coming from meters at the smart grid AMI layer. Therefore, the challenge that EVs rise is to determine over what period of the day EV drivers are likely to request charging; as this activity first requires a plan for charging the EVs that meets drivers' preferences, battery constraints and electricity (peak) constraints at the minimum economic cost [7]. Any charging request (demand) from a consumer needs to be considered along side other demand profiles, requiring a suitable schedule to be found that does not violate the energy Grid capacity and consumer charging profile constraints. It is therefore essential to understand real usage of charging infrastructures and drivers' behaviour.

This real usage can be anticipated from *The EV Project* [9] [1], which is one of the largest electric vehicle infrastructure demonstration projects. This project focuses on examining the various activities and situations involving EV drivers' behaviour and charging infrastructure usage. For all these reasons, it can be considered as one of the most realistic deployments available. Data gathered by "The EV Project" is reported on a quarterly basis and accessible online. It includes information of interest such as probability distributions for the hourly average usage of charging points, the aggregated hourly charging demand, the average energy requested per charging event (a charging event is defined as the period when a vehicle is connected to a charging unit, during which period some power is transferred), the average recharging time, etc. By the 2nd quarter of 2013, over 2.9 million charging events had been recorded from project participants in the US: approximately 8,300 Nissan Leaf, Chevrolet Volts and Smart ForTwo EVs [9]. These EVs made use of an infrastructure that consisted of nearly 8,200 residential electric vehicle supply equipment (EVSE) charging stations, over 3,750 commercial (publicly available, workplace, and fleet) EVSE charging stations and 87 DC Fast Chargers (DCFC). To support data privacy, each data set provided *aggregates* values for a whole participant electricity network area in the US, namely residential, private non-residential, and public.

Using this data, we are interested in identifying when EV drivers are likely to request charging and the duration of this process. Before "The EV Project" began collecting data, common wisdom had been that 80% of charge events for a typical driver would be at home. Data collected from this project validates this: the percentage of home charging for all regions appeared to stabilize at about 74% of all events for the Leaf and 80% for the Volt. Another important metric is that EVs averaged 1.1 charging events (or requests) per day. For the Volt driver, the average was 1.5 charging events per day. Although Volt drivers charge their vehicles more often, they tend to charge at home.

These metrics are also related to the use of the charging points within the transport/energy infrastructure. Considering only residential areas, Figure 4 illustrates the availability distribution of charging points for weekdays. This graph shows the maximum and minimum percentage of charging units connected across all days for all EVSE for weekdays. A drop in the graph indicates that some EVs disconnected and a rise means that an EV is charging. The behaviour is highly associated with the arrival and departure times of EV drivers at their homes, but it is also influenced by the electricity tariff, in this case, the minimum electricity demand is observed at

[1]http://theevproject.com

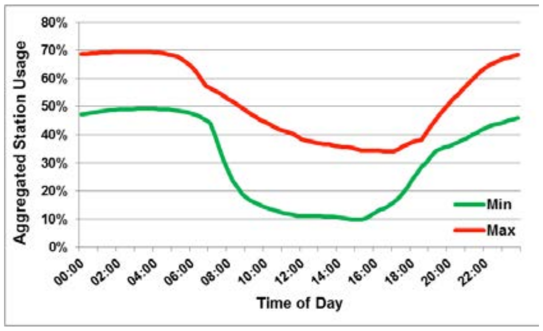

Fig. 4. Aggregated Charging Availability for Weekdays in Residential Areas

6am approximately. This is directly related to the time at which many EV owners go to work and therefore need to stop the charging process. The opposite is happening at night as EV owners prefer to charge at a period of lower energy tariff. In "The EV Project", a charging event is defined as the period when a vehicle is connected to a charging unit, during which period some power is transferred. For Smart Grid management purposes, smart meters submit data about charging patterns at regular intervals (i.e. every 15mins., half-hourly, etc.). Nevertheless, according to Figure 4, it can be concluded that data flows generated for the EV re-charging processes can be highly bursty in reality (a maximum variability of up to 300% in Figure 4 can be observed), which can make any anticipation or prediction of the expected computational demand very difficult.

IV. Open Nebula Based-Implementation

Each node within our system architecture consists of a node with co-located computational resources (processing units), thereby reducing the communication latency and overheads between processing units. We replace the subnet of the PU component at each node by an adapter component that interacts with OpenNebula, so that OpenNebula in turn will allow us to manage a number of virtual machines (VMs). Therefore, as the Reference nets can be interpreted by Renew [10] (a Reference net interpreter), our Reference net-based model becomes executable. This provides a natural way to map a previously developed model into a physically realised system.

A. 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 [11]. 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. Additionally, it also supports two main Cloud interfaces, namely Amazon's EC2-Query API – whereby an OpenNebula Cloud can be accessed by an EC2 query; and OCCI-OGF interface to enable multi-Cloud capability to be realised. The hypervisors supported to run VMs are Xen, KVM and VMware, and in order to enable message-based communication between 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 VMs 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 controlled access. Once a VM is instantiated, OpenNebula also provides different methods to customize the VM and adapt it tby passing contextual information such as network configuration for the VM, user credentials, etc.

B. Integration with OpenNebula

We are utilising OpenNebula for the creation and management of the pool of computational resources for our Processing Unit (PU) components. At every node, for each PU, we assume that there are a number of computational resources that can accomplish the same functionality with similar performance. We, utilise VMs for implementing computational resources, which will be linked with the PU controller by a physical network. We utilise the OpenNebula template repository for storing the OS image to be used for each VM with the required executables already installed. The hypervisor is KVM [12] (Kernel-based Virtual Machine) which 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 VMs running unmodified Linux OS or Windows OS images. Each VM has private virtualized hardware: a network card, disk, graphics adapter, etc. For the purposes 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 PU component: they can be switched on & off depending on the processing requirements. A rule engine controller monitors buffer occupancies and triggers actions such as changing token bucket admission rates and consequently stopping or running new VMs. The OpenNebula adapter component is also responsible for switching on & off the VMs by using of the OpenNebula CLI interface or ssh. Each ready VM establishes a TCP/IP socket connection with the PU component interpreted by the Java-based Reference net interpreter Renew. A token representing this socket connection is used by the PU for managing the VMs. This way, the place that contains tuples, with references to TCP/IP connections and metadata, plays the role of a tuple space that is use for managing resources. VMs can be at the same or different site than PU and traffic shaping components.

Data injection rates to the PU component is regulated by the token bucket manager component for each data stream entering the system. Whenever a data element arrives it is stored in a FIFO buffer until there is a free VM. Once a VM is ready for processing and there are available data in the FIFO buffer, the PU sends the data element to the available VM as depicted in Figure 5. The PU interpreter recovers the TCP connection and data, sends the data element through TCP/IP and waits for the result of the processing.

20 VMs receiving packets			
packet size	MEAN (msecs)	STDV	MADV
8Kb	2,00	0,42	0,30
16Kb	2,00	3,09	0,56
33Kb	3,00	12,98	1,93
65Kb	3,00	21,45	4,15

40 VMs receiving packets			
packet size	MEAN (msecs)	STDV	MADV
8Kb	2,00	4,93	0,54
16Kb	2,00	19,55	2,39
33Kb	3,00	27,22	4,72
65Kb	2,00	36,16	7,72

TABLE I
JITTER MEAN, STANDARD DEVIATION AND MEDIAN ABSOLUTE DEVIATION WITH VMS RECEIVING PACKETS AT 400 PACKETS PER SEC

In case, the local site has not enough resources, the controller can incorporate TCP/IP socket connections with external VMs. In this case, controller must consider the impact of lower data transfer rates on WANs. If throughput of each data stream can be maintained, it is not required to distinguish LAN and WAN socket connections. In the case, some QoS constrains can not be supported by WAN socket connections, the pattern matching mechanism in traditional high level interpreters and tuple spaces provide an expressive way to discriminate connections.

V. EVALUATION

Prior to deploying our Cloud service infrastructure, we tested that QoS amongst the data streams can be enforced with OpenNebula and KVM. A key mechanism for enabling QoS (throughput) in our architecture is the token bucket-based admission & regulation component. Therefore, we have to guarantee that the use of virtualisation technologies does not alter the original incoming rate of packets at each data stream. In particular, we tested that the hypervisor is not introducing significant packet *jitter* altering the packet inter-arrival time defined by the token bucket controller .

Our tests use the *cloudmip* [2] platform of the Université Paul Sabatier of Toulouse, France, which provides an OpenNebula v4.4 platform with 32 physical nodes. Each node contains 32GB RAM and 8 cores. In the first test, 4 VMs sent packets to 20 VMs and we recorded the jitter. Each sender VM runs 5 processes for sending packets at a speed of 400 packets/s. We measured the jitter for various packet sizes: 8Kb, 16K, 33Kb and 65Kb. In the second test, we used 8 VMs sending packets to 40 VMs for recording the jitter as well. In this way, we generated a traffic that will exceed the maximum traffic received from an external network (at a maximum of 100 Mbps). The way we measured jitter is by computing the mean, standard deviation (STDV) of the sample and a robust deviation metric such as median absolute deviation (MADV).

[2]http://cloudmip.univ-tlse3.fr/

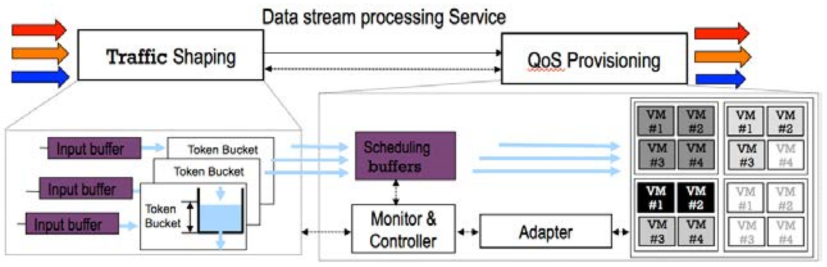

Fig. 5. OpenNebula-based QoS Enforcement for several Data Streams

At a transmission speed of 400 packets per second, the jitter is around 2.5 milliseconds on average. Table I shows that jitter can be maintained in both tests for 8KB and 16KB packet sizes while there is a slight degradation for higher packet sizes. It should be noted that the Linux OS accuracy for timing is at millisecond level. This is the reason why the jitter measured was 2 milliseconds. Moreover, the STDV & MADV are higher for the second test, but this test generates 4Gbps of traffic, which exceeds the maximum allowed traffic from the outside.

Other sources of overhead may occur when trying to schedule more VMs than available resources on a physical machine. Two overhead issues with the hypervisor-VM interaction would be: (i) creating/forking a new VM dynamically; (ii) scheduling between VMs – if there are not enough resources to sustain all VMs concurrently. All these issues will be part of the future work to be considered to deploy our infrastructure in the cloud to enforce QoS in data stream processing applications.

VI. Lessons learned

Data measured or collected by sensors can be typically processed at the sink or "in-transit" over a shared distributed computing infrastructure, where the computational power can be adjusted to the required demand. On one hand, this guarantees an efficient usage of resources. On the other hand, QoS mechanisms must be incorporated into the system in order to guarantee and enforce a previously negotiated Service Level Agreement (SLA) to each user. This is particularly challenging for time critical applications and with highly variable and unpredictable rates of data generation. In previous papers, we proposed a Reference net based architectural model for supporting QoS for multiple concurrent data streams being processed (prior to delivery to a user) over a shared infrastructure, deployed as a sequence of distributed nodes. Each nodes consists of a combination of data access and admission control stage, a processing stage, and a data transmission stage to the following node. In this paper, we describe how to implement the processing stage with OpenNebula, an open source Cloud platform. OpenNebula provides networking & computing virtualization. Therefore, OpenNebula can support elastic VM provisioning within our architecture. Nevertheless, virtualisation technologies impose an overhead that can affect performance for specific applications demanding critical real-time response.

The main advantage of the approach is that the Petri net models we develop can be directly executed. In this way, we have an executable prototype from the beginning. Our previous models simulated all the execution, and assumed that communications will be not a problem. In this paper, we have shown that once deployed the traffic shaping component to regulate data injection, and the PU component to control the data processing and managing the resources, the Cloud hypervisor does not introduce significant overheads in traffic. However, simulations ignore other sources of overhead as previously pointed, and does not consider a lot of administrative work required to have a complete operative system. For example, the deployment of services to process data at each VM. An early deployment of the models in real infrastructures is required to ensure correct assumptions have been made.

The case studies in section III introduce two scenarios of data being generated at an unpredictable and bursty manner – where processing needs to be carried out within a limited time period. Hence, the computational requirement for processing such data streams may not be known apriori. The token bucket-based data access component guarantees that any exceptional behaviour of one particular data stream does not affect another, ensuring that data streams can be isolated. We also observe that data volumes at intermediate stages may also not be known apriori – due to issue of data inflation/deflation. In both cases, the computational resources needed to provide particular processing capability must be dynamically allocated across the data streams entering a system, achieved in our system by adding / removing VMs in the OpenNebula system.

VII. Related Work

Research in Data Stream Management Systems (DSMS) and Complex Event Processing (CEP) has evolved separately, despite the fact that both communities share a number of important similarities and challenges such as scalability, fault tolerance and performance. Data Stream Management Systems (DSMS) shifted the paradigm of Database Management Systems as the need for efficiently processing streamed data sets in real-time arose. In general, DSMSs focus on performance by restricting the language in which they can be programmed to graphs of operators with well-defined semantics [13]. This allows these systems to automatically rewrite or compile the specified stream pipelines to a more efficient version. Scalability and query distribution were considered in Aurora [14], Borealis [15] and Stream Cloud [16]. Additionally, QoS support in DSMS has been a critical requirement [13] and various scheduling strategies and heuristics have been developed. When data streams arrives at an expected rate with low variability, near optimal scheduling strategies have shown to enforce QoS for multiple data streams successfully. However, if the data streams arrive with unpredictable and variable bursts, the scheduling heuristics comprise a combination of strategies that may not always show satisfactory QoS enforcement [13]. These systems therefore employ load shedding strategies – i.e. discarding of data elements from a stream when the loss of some data elements is acceptable. In our approach, we rely on estimations of average input rates for data streams, a token bucket model for regulating data access to the computational resources, on the elastic Processing Unit Component (increase / decrease of the number of computational resources) and on the autonomous behavior of each node. Our approach focuses on the need to isolate data streams, so that any exceptional behaviour in one stream does not affect another. However, one limitation of our approach is the inability to synchronise across streams – as the token bucket will alter the order in which streams are subsequently transmitted to the processing units.

DSMSs have little or no support to express events as the outcome of continuous queries nor further support to form complex events. CEP has seen a resurgence in the last few years, though support for handling events, rules, and triggers was accomplished more than two decades ago. Examples of CEP are SPADE/IBM InfoSphere Streams, Esper and DROOLS Fusion [17]. The main difference between existing CEP and stream processing systems is that in the former each event is processed at arrival time, while stream processing systems involve accumulating a data set over a time (or count – i.e. when a certain number of events have arrived) window and processing it at once. Event processing systems assume that the incoming events are not bursty and do not generally consider the presence of queues/buffers between event operators. Additionally, most CEP have little support for QoS requirements, except for the MavEStream system [13] that integrates CEP into a QoS-driven DSMS system. Again, MavEStream system enforces QoS by complex scheduling heuristics that may not always perform suitably under bursty conditions. Additionally, this lack of QoS support in CEP has also been recently considered in the literature: in [18] several micro-benchmarks were proposed to compare different CEP engines and assess their scalability with respect to a number of queries. Various ways of evaluating their ability to respond to changes in load conditions were also discussed. Scalability has not received much attention in CEP, some event processing languages extend production rules to support event processing and provide run-time optimizations by extending the Rete [19] or Treat [20] algorithms to scale with the number of queries. In [21] the importance of scalability for CEP is recognised and event processing is partitioned into a number of stages. At each stage resources can process all of the incoming events in parallel under peak input event traffic conditions. However this approach does not provide a run-time adaptation, does not involve queues and buffers and assumes the best-effort strategy (as is usual in CEP).

Other related literature includes a survey on integration of sensing data with Cloud infrastructure [22], where various application scenarios and sensor platforms are outlined. The survey does not specifically focus on streaming data or particular requirements that such applications introduce, however it provides a good overview of approaches that are currently being used in the community. Other application specific approaches include the Storm system [23], which is used to process large volumes of data from the Twitter streaming API. This is an open source system for supporting real time computation and integrated with the Hadoop platform (for batch mode computation). The "Bolt" concept in Storm can be mapped to our processing stages and a "Spout" as a source for an incoming data stream (although there is no equivalent in Storm for managing flow rates of incoming data streams).

VIII. Conclusions

The increasing number of sensors and their software applications leads to greater challenges for processing, storing and transmitting the large amounts of data generated. Moreover, some application domains generate data in a highly bursty and unpredictable rate, such as in Smart (energy) Grids. In this paper, we propose an OpenNebula-based system architecture as a scalable and economically viable solution for heterogeneous surveillance & monitoring systems that involve changing data generation patterns and processing requirements. We use token bucket envelop process to enable running multiple data streams over a shared Cloud infrastructure while providing each data stream with a particular QoS requirement. We add a control strategy at each computational stage to dynamically adjust token bucket parameters to adapt the available resources in order to provide QoS on an end-to-end basis, so that variations in incoming rates can be self-configured by the application. Experiments have shown that the overhead from virtualisation technologies measured in terms of packet jitter is below 3ms, therefore not altering significantly the token-bucket control on packet inter-arrival time.

Acknowledgment: This work was partially supported by the Aquitaine-Aragon OMNIDATA project. Information for the study case is based upon work provided by The EV Project and supported by the U.S. Department of Energy under Award Number DE EE0002194 to ECOtality North America. This research was supported by the Engineering and Physical Sciences Research Council (EPSRC) which funded the project Smart Management of Electric Vehicle

References

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

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

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

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

[5] D. J. Leeds, "The smart grid in 2010: Market segments, applications and industry players," GTM Research, Tech. Rep., 2009.

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

[7] P. Papadopoulos, "Integration of electric vehicles intro distribution networks," Ph.D. dissertation, Cardiff University, 2012.

[8] P. Papadopoulos, O. Akizu, L. Cipcigan, N. Jenkins, and E. Zabala, "Electricity demand with electric cars in 2030: comparing Great Britain and Spain," *Proceedings of the Institution of Mechanical Engineers, Part A: Journal of Power and Energy*, vol. 225, no. 5, pp. 551–566, 2011.

[9] S. Schey, "Q2 2013 report – the ev project," ECOTality North America, Tech. Rep., 2013.

[10] O. Kummer and F. Wienberg, "Renew - the Reference net workshop," in *Tool Demonstrations, Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 26-30, 2000*, Computer Science Department, Aarhus University, Aarhus, Denmark, 2000, pp. 87–89.

[11] D. Milojičić, I. Llorente, and R. S. Montero, "OpenNebula: A Cloud Management Tool," *Internet Computing, IEEE*, vol. 15, no. 2, pp. 11–14, March 2011.

[12] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

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

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

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

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

[17] O. Etzion and P. Niblett, *Event Processing in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.

[18] M. R. N. Mendes, P. Bizarro, and P. Marques, "A performance study of event processing systems," in *TPCTC*, ser. Lecture Notes in Computer Science, R. O. Nambiar and M. Poess, Eds., vol. 5895. Springer, 2009, pp. 221–236.

[19] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17–37, 1982.

[20] D. P. Miranker, "Treat: A better match algorithm for ai production system matching," in *AAAI*, K. D. Forbus and H. E. Shrobe, Eds. Morgan Kaufmann, 1987, pp. 42–47.

[21] G. T. Lakshmanan, Y. G. Rabinovich, and O. Etzion, "A stratified approach for supporting high throughput event processing applications," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '09. New York, NY, USA: ACM, 2009, pp. 5:1–5:12. [Online]. Available: http://doi.acm.org/10.1145/1619258.1619265

[22] A. Cuzzocrea, G.Fortino, and O. Rana, "Managing Data and Processes in Cloud-Enabled Large-Scale Sensor Networks: State-of-the-Art and Future Research Directions," in *Proceedings of IEEE/ACM CCGrid conference, Delft, The Netherlands*. IEEE Computer Society Press, pp. 583–588.

[23] A. I. Project, "Storm: Distributed and fault-tolerant realtime computation – http://storm.incubator.apache.org/," [Online; accessed March 2014].