Enforcing QoS in scientific workflow systems enacted over Cloud infrastructures

Rafael Tolosana-Calasanz, José Ángel Bañares, Pham Congduc, Omer F. Rana

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

ABSTRACT

The ability to support Quality of Service (QoS) constraints is an important requirement in some scientific applications. With the increasing use of Cloud computing infrastructures, where access to resources is shared, dynamic and provisioned on-demand, identifying how QoS constraints can be supported becomes an important challenge. However, access to dedicated resources is often not possible in existing Cloud deployments and limited QoS guarantees are provided by many commercial providers (often restricted to error rate and availability, rather than particular QoS metrics such as latency or access time). We propose a workflow system architecture which enforces QoS for the simultaneous execution of multiple scientific workflows over a shared infrastructure (such as a Cloud environment). Our approach involves multiple pipeline workflow instances, with each instance having its own QoS requirements. These workflows are composed of a number of stages, with each stage being mapped to one or more physical resources. A stage involves a combination of data access, computation and data transfer capability. A token bucket-based data throttling framework is embedded into the workflow system architecture. Each workflow instance stage regulates the amount of data that is injected into the shared resources, allowing for bursts of data to be injected while at the same time providing isolation of workflow streams. We demonstrate our approach by using the Montage workflow, and develop a Reference net model of the workflow.

1. Introduction

Quality of Service (QoS) remains an important concern for many distributed applications with stringent time constraints on data transfer and job execution. In the past, QoS constraints have often been associated with multimedia data streaming and video/audio analysis. A more recent example within the scientific computing domain is within the area of "Urgent Computing" – which refers to providing prioritised 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. Such applications are driven by the need to give immediate access to computational jobs in critical emergencies, which cannot waste time waiting in job queues of high end computational resources. It is also necessary for data needed by such applications to be transferred within bounded times between various components that make up the application. Some example applications in this area are outlined under the SPRUCE project (Special PRiority and Urgent Computing

This work is partially supported by the research project TIN2010-17905, granted by the Spanish Ministry of Education and Science, and by the OMNIDATA project funded by the Aquitaine–Aragon collaboration research program between Aquitaine region (France) and Aragon region (Spain).

E-mail address: rafael@unizar.es (R. Tolosana-Calasanz).

© 2011 Elsevier Inc. All rights reserved.

such as disaster relief during crisis or specific weather predictions (storm tracking) where the need for obtaining results in a bounded time is crucial [1], coastal ocean observing and prediction which rely on global data acquisition systems (global sensing infrastructures) that would continuously stream large amounts of data and expect a bounded processing time [2], and neurosurgical imaging using simulation. It is useful to note however that the QoS constraints associated with “urgent computing” applications can be generalised, for instance where multiple instances of the same application need to be executed over shared infrastructure (and each instance must adhere to its own QoS constraints). In such a scenario, it is necessary for each application instance to be isolated from another, and for the underlying coordination mechanism to adapt the infrastructure to either: (i) run all instances without violating their particular QoS constraints; or (ii) indicate that given current resources a particular instance cannot be accepted for execution.

There are already examples of projects that can deploy workflows over a Cloud environment, such as myGrid/myexperiment.org in BioSciences and Microsoft’s Windows Workflow Foundation and Azure. In the myGrid project [3], a user is able to select the underlying infrastructure for enacting a particular workflow; this infrastructure can be a private Cloud or involve access to a commercial system such as EC2 from Amazon.com. A user is required to select the number of resources involved and the workflow enactment engine is responsible for subsequent execution. Nevertheless, access to dedicated resources is often not possible in existing Cloud deployment, and limited QoS guarantees are provided by many commercial providers – often restricted to error rate and availability, rather than particular QoS metrics such as latency or access time. As the use of Cloud computing becomes more dominant in both scientific and industrial user communities, it will be more necessary for Cloud providers to support QoS enforcement mechanisms so that applications cannot utilise resources exceeding an established Service Level Agreement (SLA). Because in such a case, the QoS of other applications could be compromised.

We propose a workflow system architecture which enforces QoS for the simultaneous execution of multiple workflows in a shared infrastructure (such as a Cloud computing environment). We assume a workflow is composed of a number of stages, each being mapped to one or more physical resources. A stage involves a combination of data access, computation and data transfer capability. Our focus is on applications where multiple instances of the same workflow need to be executed over shared resources, with each workflow instance containing its own QoS requirement. Our observations are similar to those of Park and Humphrey [4], who identified that although obstacles to workflow efficiency are often found within the application, such as inherently limited parallelism because of the workflow definition, often reduced performance also arises due to the workflow engine that maps the abstract workflow to underlying resources in an inefficient manner. This is particularly relevant for applications that involve large data transfers between workflow tasks, where data location and link bandwidth is used to determine how to move large files (utilising the highest capacity links).

An effective solution based on ad hoc manual tuning and heuristics is not just tedious but error prone and infeasible. Our key contribution in this work is the definition of an adaptive workflow stage which: (i) enables QoS properties of each workflow instance to be adhered to by providing an envelope process control mechanism; (ii) enables dynamic adaptation of data transfer and resource use within each workflow stage. We consider a workflow to consist of a number of such adaptive stages and consider QoS constraints that can be associated with each stage. We present a model of each workflow stage using Reference nets (a Petri-based representation), and demonstrate how this can be used to model the Montage workflow. The model can be used to tune parameters of the control mechanisms and thereby enable multiple workflow instances to co-exist over a shared, distributed infrastructure. We also identify how dynamically provisioned Cloud computing resources can be used to execute such a workflow stage, in order to meet QoS constraints identified for each stage.

The remainder of this paper is structured as follows. In Section 2, we describe the QoS enforcement issues and the envelope process techniques for rate regulation to enable simultaneous execution of multiple workflows in a shared infrastructure. Section 3 presents the Reference net formalism and in Section 4, the Montage toolkit and the Montage workflow are introduced. Our complete QoS enforcement framework is described in Section 5. An evaluation scenario is given in Section 6 and related work is discussed in Section 7. Finally, in Section 8, the conclusions and future work are provided.

2. QoS enforcement

2.1. The token bucket envelope process

Within a workflow a “data acceptance rate” parameter could be used to identify the rate at which a workflow stage is able to receive and process data. This is often different from the physical link capacity that connects two workflow stages. Our focus in subsequently analysis is on data acceptance rate, with the need to provide isolation between data streams associated with different workflow instances.

To support QoS, the traffic associated with each workflow instance must be managed, so that as long as the data stream for a workflow stays within its predefined limits, the system will be able to provide QoS guarantees. To achieve this, a rate envelope process will be used to regulate the “data acceptance rate”. Envelope processes [5] have been mainly used in communication networks to bound user’s traffic under a given rate. Among envelope processes, the token bucket [6] model provides a simple, yet effective, way to allow variable data rates and burstiness while enforcing a predefined (negotiated) mean rate.

As illustrated in Fig. 1, a token bucket envelope process is characterised 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 initialisation 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 on the right in Fig. 1. 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 by Fig. 2. The left part of Fig. 2 shows the flow’s instantaneous sending rate, while the right part of the figure shows the cumulated number of bits sent at time $t$. As illustrated, user traffic is enforced to fit under the token bucket envelop. These basic concepts of a token bucket may be enhanced in a number of ways:

- an extra bucket space can be negotiated to allow the user to temporarily save tokens beyond the $b$ threshold. These excess tokens may have a limited lifetime,
- user’s traffic can be marked as non-priority traffic instead of being delayed when the user does not have enough tokens. In this way, if the system is not congested, the user’s excess traffic can get through,
- random delays could be inserted to avoid flows from becoming synchronised,
- the token bucket provides an open-loop control, it is however possible to add closed-loop control mechanisms to dynamically modify token bucket parameters at runtime. Such modifications would also take into account the current queue size and the number of resources available to process the data at a given node.

2.2. Enforcing QoS in multiple superscalar pipeline workflows

In our proposal, such a token bucket model for data traffic characterisation is integrated with each workflow stage. Our approach involves multiple superscalar pipeline workflows being enacted over distributed nodes connected by a public network. Each workflow instance operates on different data elements, which are streamed over the distributed nodes. In this pipeline model of computation, a vector of input data elements is streamed into a sequence of tasks (or stages): parallelism is achieved as input data elements are processed simultaneously by the pipeline of tasks. However, due to the heterogeneity of tasks and the uncertainty of scientific workflow environments, it cannot be assumed that all tasks in the pipeline take the same time to process their inputs. In a superscalar pipeline [7], this restriction is overcome by executing multiple task instances per pipeline stage, so that data elements do not have to wait to be processed as long as there are multiple resources at a stage available.

We utilise one token bucket per workflow at the input of each workflow stage, in order to i) throttle data elements to the computation phase at a predefined rate and ii) to prevent one stream from affecting the QoS properties of another. This use of token bucket enables the provision of traffic characterisation and enforcement while at the same time providing an isolation of workflow streams. At each workflow stage, the number of resource instances available for execution can be dynamically tuned during the execution by utilising the elastic properties of a Cloud infrastructure, so that (i) resources
can be used more efficiently and unused resources released for other streams, and (ii) additional resources (if available) can be added if the QoS requirements of a particular stream cannot be met. At the output of each workflow stage we make use of the Autonomic Data Streaming Service (ADSS) [8] for submission of data to a subsequent workflow stage. The ADSS makes use of performance information (transfer time, storage capacity, etc.) to adapt data transfer between workflow stages, for instance the ADSS can detect a network congestion between two workflow stages and react to it by reducing the data transmission rate over the network and temporarily store data onto disk (thereby avoiding data loss).

3. Reference nets

The Reference net formalism [9] is a special class of high-level Petri net (adhered to the Nets-within-Nets [10] paradigm) that uses Java as an inscription language, and extends Petri nets with dynamic net instances, net references, and dynamic transition synchronisation through synchronous channels. Reference nets consist of places, transitions and arcs. The input and output arcs have a behaviour similar to ordinary Petri nets [11]. Every net element can have associated semantic inscriptions: places can have initialisation expressions, which are evaluated and serve as their initial markings. Arcs can have optional arc inscriptions: when a transition fires, its arc expressions are evaluated and tokens are moved according to the result. Transitions can be equipped with a variety of inscriptions, including Java inscriptions, in which the equality operator “=” can be used to influence the binding of variables that are elsewhere. The binding is similar to the way variables are used in logic programming languages such as Prolog. Additionally, the inscription language of Reference nets has been extended to include tuples. A tuple is denoted by a comma-separated list of expressions that are enclosed in square brackets. Tuples are useful for storing a whole group of related values inside a token and hence in a single place. The nets hold two kinds of tokens: valued tokens and tokens which correspond to a reference. By default, an arc will transport a black token, denoted by [1]. In case an inscription is added to an arc, that inscription will be evaluated and the result will determine which kind of token is moved.

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. They synchronise two transitions which both fire atomically at the same time. Both transitions must agree on the name of the channel and on a set of parameters before they can synchronise. The initiating transition must have a special inscription – called downlink – which makes a request to a designated subordinated net. A downlink consists of an expression that must evaluate to a net reference (usually a variable), a colon (:), the name of the channel, and an optional list of arguments. On the other side, the transition must be inscribed with an uplink, which serve requests for everyone. Generally, transitions with an uplink cannot fire without being requested explicitly by another transition with a matching downlink. A transition has both uplink and downlinks, and may have multiple downlinks. Channels can also take a list of parameters. Although there is a direction of invocation, this direction needs not coincide with the direction of information transfer. Indeed, it is possible that a single synchronisation transfers information in both directions.

In order to illustrate the main concepts of Reference nets, Fig. 3 depicts a Reference net model that represents a Resource Consumer and two Resource providers, each of them with allocating capacity of two resources. Services are required with a QoS and data to be processed when Consumer invokes the match channel. Communication happens when unification of variables is possible. In the state represented in the figure, transition labelled with the downlink this:match channel in the Consumer may synchronise with transitions labelled with uplinks :match in the Resource Provider 1 and Resource Provider 2.

The unification between the channel variables gives the following possible firing modes:

- Synchronised firing modes of Consumer with Resource Provider 1:
  - request=serv1, QoS=2, inputData=data1, resource=resource[14] and
  - request=serv2, QoS=2, inputData=data2, resource=resource[14],
  - service=serv1 or service=serv2, QoS=2, res=resource[14].

- Synchronised firing modes of Consumer with Resource Provider 2:
  - request=serv2, QoS=2, inputData=data2, resource=resource[10] or resource[12] and
  - request=serv3, QoS=2, inputData=data3, resource=resource[10] or resource[12],
  - service=serv2, QoS=2, res=resource[10] or res=resource[12] and
  - service=serv3, QoS=2, res=resource[10] or res=resource[12].

The firing of these synchronised transitions provides the Consumer with a reference to the resource instance that will execute the service. The Consumer may execute the service and recover the result synchronising with the :begin and :end channels of the resource, once the Resource Provider has allocated the resource. The resource Provider allocates and liberates the resource synchronisation with the :allocate and free channels, respectively.
Renew \(^2\) [12] is a Java-based Reference net interpreter and a Reference net graphical modelling tool used in this work. Petri nets and Reference nets have been used for specifying scientific workflows [13,14]. Reference nets along with Java inscriptions have been used for implementing a service-oriented workflow engine, D Vega [15].

4. Montage

To illustrate our approach, we use an example workflow that has been widely used within the scientific computing community. The Montage workflow provides a representative space science application, typically utilising a large number of images that are stored in distributed archives and that are, in most cases, remote with respect to the available computational resources. At the end of workflow execution, the outcome is the integration of these astronomical images into a single image mosaic. The processing of these images involve computational and data management challenges, some already addressed by the Pegasus project [16] along with the Montage toolkit.\(^3\) The process for obtaining an image mosaic within Montage can be summarised in the following steps:

- re-projection of input images to a common spatial scale, coordinate system, and a World Coordinate System projection (WCS) (WCS specifies image coordinate to sky coordinate transformations for a number of different coordinate systems and projections useful in astronomy);
- modelling of background radiation in images to achieve common flux scales and background levels by minimising the inter-image differences;
- rectification of images to a common flux scale and background level;
- co-addition of re-projected, background-matched images into a final mosaic.

\(^2\) http://www.renew.de.
\(^3\) http://montage.ipac.caltech.edu/.
Fig. 4. Montage abstract workflow for three input files: (left) pictorial representation, including data and metadata and (right) its equivalent data parallelism abstract workflow.

A Montage mosaic job can be described in terms of an abstract workflow, as illustrated in Fig. 4. On the left, the rectangles in white background represent a parallel task that runs on distributed resources, whereas the rectangles in black background illustrate synchronisation tasks that gather output from tasks at the previous level, and provide input for parallel tasks at the next level. The solid line represents the movement of large image files and the dotted line represents the movement of metadata (typically a few kilobytes). Execution is based on data parallelism, hence tasks can be executed in parallel as long as there are no data dependencies among them. With this model of computation and as a consequence of the dependencies that appear among the input files forming the mosaic, the abstract workflow structure varies depending on the number of input files. More information on the description of the tasks can be found in [17]. Pegasus has been used to run the parallelised version of Montage, on a number of different cluster and Grid environments. The execution of the workflow is performed by the workflow manager DAGMan and the associated Condor-G.

5. The proposed QoS enforcement framework

5.1. Workflow system architecture for enforcing QoS

We demonstrate how multiple workflows, each having different QoS requirements, can be supported by a workflow engine. We assume that i) data transmissions required for meeting QoS, on average, do not exceed the network bandwidth available and ii) the required processing capability on average does not exceed the computational power of the resources available. Our intention is to keep the abstract workflow independent of the resources used to subsequently enact it. This is achieved by means of the synchronous channels between the simple task patterns and the workflow engine in the Reference net model. The next step involves mapping workflow tasks to dynamically provisioned distributed resources. The low degree of coupling between workflow tasks and the workflow engine (in our approach) allows a number of alternative task mapping strategies to be used (such as the use of a meta-scheduler or a resource broker).

Each workflow stage needs to be mapped to one or more nodes of the computational infrastructure. Nodes can offer different services and can execute multiple workflow tasks. As shown in Fig. 5, a node contains a token bucket (one per workflow stream), a processing unit, and an Autonomic Data Streaming Service (ADSS). Each token bucket regulates the
flow of input to the processing unit, isolates the workflow streams and guarantees that the QoS of each stream is within the pre-defined interval. The processing unit performs the computation by utilising multiple resources in parallel. Resources can be added to or removed from the processing unit at runtime, but for the sake of simplicity, we assume that all resources are identical (i.e. perform the same functionality and have a similar execution performance). The ADSS is responsible for forwarding data to the following node in the sequence, and guarantees that there is no data loss, as outlined in [8]. In summary, the mechanisms for enforcing QoS and avoiding data loss are: (i) a token bucket and a buffer at the processing unit input, to regulate data ingest and isolate data streams for each workflow instance; (ii) processing units with dynamic capacity: the number of resources can be modified on-demand, and (iii) an Autonomic Data Streaming Service (ADSS) at the output which provides a temporary data buffer, thereby regulating the transfer rate (based on available bandwidth and input buffer capacity of the following node). Once elements go through the token bucket, they are stored at the input buffer, waiting for free resources to carry out required processing.

The adaptation parameters in the system include: (i) the number of resources required for execution at each processing element; (ii) the size of the input and output buffers; (iii) \( \mu \) (transfer rate), \( \omega \) (transfer rate to local disk to avoid data loss) and the transfer path (between the sending and receiving nodes) in the ADSS.

5.2. Specifying superscalar pipeline workflows: the Montage case

To demonstrate our approach, we will use multiple instances of the Montage workflow, with each instance having a different QoS requirement. We use the superscalar pipeline [7] model of computation, whereby multiple data elements can be processed in parallel within a workflow stage as long as there are enough resources available. In our approach, users specify an abstract workflow (i.e. without being constrained to specific resources) with their QoS requirements, specified in terms of throughput (number of data elements in the stream processed per unit of time).

Our abstract workflows can have a hierarchical structure in which nodes can be either intermediate nodes or leaf nodes (simple tasks). Additionally, nodes at the same hierarchical level are connected sequentially (in compliance with the pipeline paradigm). The Reference net patterns that can be used for the workflow specification are depicted in Fig. 6. Elements within a data stream are represented as tokens flowing through the pipeline: tokens can either be data elements, or be references to remote data elements (in which case, a token represents a URL to the location of the data). Fig. 6a) shows the workflow pattern for an intermediate node. It can be seen that a data element (or its reference) \( idata \) token is received in Transition \( t_1 \). After the firing of Transition \( t_1 \), an instance of its descendant nodes (\( SWf \)) is created (\( sw: new \ SWf \)) and the enactment of the execution of the descendants is started (\( sw: begin(idata) \)). When the descendants finish their execution, Transition \( t_2 \) is fired and the output data is obtained (\( odata \)). Fig. 6b) depicts the leaf workflow node. Similarly, a data element \( idata \) token is received in Transition \( t_3 \). After the firing of Transition \( t_3 \), a new instance of a simple task (\( Task \)) is created (\( w: new \ Task \)) and its enactment is started (\( t: begin(idata) \)). When the task finishes its execution, Transition \( t_4 \) is fired and the output data is obtained (\( odata \)). Fig. 6c) shows how Reference net patterns a) and b) can be combined in sequence to form a pipeline. In c), an intermediate pattern is sequentially connected to a simple task pattern by a place and the corresponding input and output arcs. Using the superscalar pipeline model of computation, multiple data streams may be sent to a pipeline stage. This happens when the initial transition of either an intermediate node (Transition \( t_1 \)) or a simple task (Transition \( t_3 \)) are fired several times and the corresponding data tokens co-exist in the same enactment place – impacting the ordering between the streams. In this paper, we allow a data stream to be executed out of order, and order is only enforced at the
Fig. 6. Abstract part of the workflow specification expressed in terms of Reference net-based patterns: a) intermediate workflow node pattern, b) leaf workflow node pattern and c) a sequential composition of two tasks: an intermediate workflow node and a leaf workflow node.

beginning of the pipeline and at the end. The workflow engine can therefore re-order streams for intermediate stages within a workflow.

The Montage workflow is generally represented as a Directed Acyclic Graph using the data parallelism paradigm. Fig. 7(a) shows an alternative representation using the superscalar pipeline paradigm. Multiple data tokens (URLs) representing the (remote) input image files to generate the mosaic are streamed into the pipeline. The main advantage of this representation is that the workflow structure remains invariable and is independent of the input. This differs from the Montage DAG approach, in Fig. 7(b), a DAG Montage workflow for three input files.

One of the challenges of a superscalar pipeline version of Montage, however, is that the data elements of the stream are not independent of each other: some operations require all the input data elements of the stream to be synchronised, some others have to be applied only to a combination of data elements: for instance the mDiff operation has to be applied only between those input image files that overlap. Montage provides the mOverlaps operation in order to determine which pair of images overlap on a region of the sky, and upon invocation it creates an image metadata table that can be used by mDiff. In the workflow of Fig. 7(a), the tasks with a white background represent a workflow stage where multiple data elements are processed in parallel. In contrast, the task with a black background represent the synchronising tasks that gather output from previous tasks and generate input for subsequent parallel tasks. The Reference net workflow engine is responsible for supporting such synchronisation, as explained in [18].

5.3. Reference nets for the workflow engine architecture

Once multiple workflow instances are created in the workflow system, the workflow engine utilises the nets described in this section in order to coordinate the execution. The Reference net in Fig. 8 shows an example sequence of three nodes. A node at this level consists of two transitions and a place. Two consecutive nodes, n_i followed by n_{i+1}, share one transition: the final transition of n_i is the initial transition of n_{i+1}. Transitions labelled as c_i are responsible for creating and initialising nodes: the parameters specified in the creational inscription new node(opList, res, bufS), indicate opList the list of operations that resources at the node can perform, res the initial number of resources, and bufS the buffer size of the processing unit.

At enactment time, multiple data elements from different workflows are streamed into processing units, introduced one by one at the initial transition via Synchronous Channel :inputData([d, wf]), where d is a data element that belongs to the stream of data of wf, are introduced in Transition t1. Variable d stores either the data itself or a reference to the data element, and wf is a reference to a net instance of workflow wf. The pair [d, wf] goes through the sequence of nodes and finishes the processing in Transition t4, the result is retrieved in Transition t5.

The Reference net in Fig. 9 implements a node. As discussed in Section 5.1, a workflow stage contains three different components: a token bucket manager, a processing unit and an ADSS. When a pair [d, wf] enters into the node, it arrives at the token bucket manager component (Transition t1). Then, whenever the corresponding token bucket allows the data element to proceed, it enters into the processing unit component. Finally, after the processing, it goes to the final stage which corresponds to the ADSS, upon completion of the transmission, a data element gets out of the node and enters into the following one (Transition t4).
The Reference net in Fig. 10 implements the token bucket manager component. It serves to forward incoming data elements to the token bucket of the stream they belong to. Additionally, in case of the initial data element of a stream, a new token bucket instance is also created and initialised. Each time a data element is injected in a data stream, a reference to the data stream with the agreed values \((R, b, C)\) arrives in Transition \(t1\). In case of the initial data element of the stream, Transition \(t3\) will be enabled and Transition \(t2\) disabled. In other case, Transition \(t2\) will be enabled and Transition \(t3\) disabled. In the former case, the new token bucket instance for the flow will be created in Transition \(t5\), and the data element will be added to it when Transition \(t6\) fires. In the latter, the data element will be added to its corresponding token bucket instance when Transition \(t4\) fires. Finally, once a data element is allowed to proceed, Transition \(t7\) is fired and the data element moved to the processing unit component via Synchronous Channel :\(end(d)\) in Transition \(t7\).
Fig. 9. Reference net that implements a node.

Fig. 10. Reference net that implements the token bucket manager.

The Reference net in Fig. 11 implements a token bucket. A data element arrives in a token bucket in the input Transition and leaves at output Transition. Once a data element enters into the bucket, it is stored in a buffer, implemented in this case as a first-in, first-out list, called bf. It should be noticed that the output Transition is only enabled when there is an element in the buffer and simultaneously there is a mark in Place P1. A mark in Place P1 will be added by the clock in the
Fig. 11. Reference net that implements the token bucket.

bottom right part of the figure every CRate units of time. This corresponds to the pre-defined throughput. 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 processing unit at a constant rate of CRate. Details about the Reference nets for the Processing Unit, patterns utilised to map a workflow task to a distributed resource and the ADSS can be found in [8].

5.4. Component implementation and deployment

Reference nets can be interpreted by the Renew software tool, so that the models described above also represent a prototype implementation of the approach. Both implementation and deployment details for the processing unit and the ADSS components can be found in [8].

We make use of a token bucket, as illustrated in Fig. 11, for each flow entering a processing node – with each token regulating the data ingest into a processing node. Hence, data elements are temporarily stored in a buffer and not allowed to progress to the processing units if there are no tokens available. Tokens are produced by a timer at an agreed frequency – a parameter that must be defined by the node administrator. In Fig. 11, this timer is referred to as Clock. Therefore, the two main sources of overhead are due to the buffer and the timer implementation, which also limit the overall scalability of the proposed approach.

6. Evaluation scenario

Unlike in the Montage portal [17] where users only indicate the parameters that describe the mosaic to be constructed, our approach also enables a user to identify the files to initiate the enactment of a workflow and to execute multiple Montage workflow instances with their own QoS requirements. In this section we examine the effectiveness of the token bucket approach for enforcing QoS for each workflow instance. We propose a simulation scenario in which two Montage workflows wf1 and wf2 are executed over two nodes: Montage wf1 has a required throughput of 20 data elements/s, whereas Montage wf2 has a required throughput of 10 data elements/s.

Based on the workflow description in Section 5.2, we divide the workflow so that the first node executes the initial tasks until the mProject task and the second node executes from the mDiff task to the end. Each processing unit at a workflow node (as illustrated in Fig. 5) contains 5 identical resources. The average time for a Montage operation depends on: network bandwidth, the number of resources within a processing element, the size of files, etc. We assume that each resource can perform the mProject task and mAdd task at a constant rate of 10 data elements/s. Therefore, the overall processing capacity at each node is 50 data elements/s. In the nodes, we choose buffer sizes so that buffer overflows cannot happen and also assume that the network bandwidth between nodes is enough for meeting the QoS requirements at an average transmission rate of both workflows. For the experiments, we make use of the same Reference net model for simulating job
executions and data transmissions. Job executions are accomplished by sub-nets that wait for the execution time of a task and then proceed. Data transmissions are simulated by tokens that are provided at a stipulated rate.

To evaluate we conducted two different workflow executions: (i) with the token bucket mechanism enabled, and (ii) without any token bucket mechanism. Data elements are sent to each workflow with a rate corresponding to their required throughput, with one exception: between 60 and 140 s after the start of the experiment, workflow $wf_1$ is sent data at a rate of 100 data elements/s. We assume that each token represents a 1 Mb data size (corresponding to the size of an image in this context). In the token bucket version, the $wf_1$ is limited at the rate $C_1$ of 30 tokens/s, and the $wf_2$ is limited at the rate $C_2$ of 15 tokens/s.

Figs. 12 and 13 show data input rates for workflows $wf_1$ and $wf_2$, respectively. Figs. 14 and 15 show the overall throughput of workflow $wf_1$ and workflow $wf_2$ without any token bucket mechanism. Fig. 15 shows how the introduction of a large number of data elements in workflow $wf_1$ affects workflow $wf_2$’s throughput. When the input data rate for $wf_1$ recovers
to its normal value, workflow \( w_{f_2} \)'s throughput increases to \( C_2 \) until the token bucket buffer is emptied and recovers to its normal rate, as there were a large number of data elements in the queue of the processing unit buffer.

Figs. 16 and 17 show how the use of the token bucket mechanism isolates workflow \( w_{f_1} \) from workflow \( w_{f_2} \). The effects lead to both workflows completing on time, with the mean throughput rate of \( w_{f_2} \) being preserved.

The evaluation of the ADSS, and a simple strategy to incorporate and release processing units according to the incoming rate was accomplished in [8].

7. Related work

Park and Humphrey [4] had also applied the token bucket mechanism in scientific workflows, but with a different purpose. The main differences compared to our proposal is that i) they utilise token bucket for data throttling over a network, whereas we use it for data throttling over the shared resources, ii) they consider data parallelism as the model of compu-
Gomes et al. [19] also represent scientific workflows as a Petri net and demonstrate how patterns and operators can be used to adapt, dynamically, the structure of a workflow. They discuss the use of behavioural patterns and operators which can be applied to a pipeline structure (encoded in their work as a 'structural pattern') for changing data flow between com-
ponents in the workflow. The approach they advocate provides more of an overall framework within which the techniques discussed in this work can also be applied. A key difference is the use of Reference nets in our work, which has a dynamic modelling capability that is not captured in their object oriented Petri net representation.

Various workflow systems are currently used for scientific applications – such as Triana [20], Kepler [21,22] and Taverna [3] (among many others [23]) – both of which support a data streaming pipeline. In Triana, the streaming model is used by default, and Triana units can be either Web Services or Java executables. Kepler provides a more customisable control management strategy, where a “director” can be used to alter the control flow between components in the workflow. Taverna provides a data flow model of computation, whereby a workflow consists of processors (representing software components) that are connected through data dependency links. However, Taverna has recently undergone a radical re-design of the architecture, referred to as Taverna 2 [24]. This new architecture also supports superscalar and streaming pipelining as a model of computation: a producer processor forwards each element as soon as possible to the corresponding consumer processor in the pipeline. In the consumer processor, multiple elements can be processed in parallel as there are multiple threads available (superscalar).

In addition to enforcing QoS on multiple workflows over a shared infrastructure, a key difference between such workflow systems and our approach is the availability of a model that can be used to infer properties of the streaming behaviour. Our approach provides both a Reference net model – enabling subsequent analysis of the model to be undertaken to better understand how resources (such as network bandwidth, memory buffer or hard disk) are used, and an implementation that can be directly executed via DVega. In previous work, we have also demonstrated how a Reference net-based workflow representation can be mapped into a Taverna workflow [25].

8. Conclusions

A workflow system for enforcement of QoS of multiple scientific workflow instances, over a shared infrastructure such as a Cloud computing environment is presented. We make use of Reference nets (a type of Petri nets) and the superscalar model of computation for the specification of our workflows. In this model of computation, a vector of input data elements is streamed into a sequence of tasks (or stages), and each stage can execute multiple task instances in parallel, as long as there are resources available. Each workflow stage consists of a token bucket, multiple processing units and an autonomic data streaming service.

We make use of a token bucket per stream at the input of each node, to avoid one stream from affecting the QoS properties of another. A token bucket stores data elements from a stream and forwards them to the processing unit (computational phase) via a buffer. The processing unit component incorporates a number of resources for the computation that can be dynamically tuned during execution, so that (i) resources can be used more efficiently and unused resources released for other streams, and (ii) additional resources (if available) can be added if the QoS requirements of a particular stream cannot be met. At the output of each workflow stage, we make use of the Autonomic Data Streaming Service (ADSS) [8] for submission of data to a subsequent workflow stage. The ADSS makes use of performance information (transfer time, storage capacity, etc.) to adapt data transfer between workflow stages. For instance, the ADSS can detect a network congestion between two workflow stages and react to it by reducing the data transmission rate over the network and temporarily store data onto disk (thereby avoiding data loss). We demonstrate our approach by using the Montage workflow, showing that this proposal can have a potential benefit for similar applications from Earth and Space sciences. We validate our workflow system by conducting a simulation that gives evidence that the QoS properties of simultaneous execution of workflows are enforced.

Our key contribution in this work is the definition of an adaptive workflow stage, which: (i) enables QoS properties for each workflow instance to be adhered to; (ii) enables dynamic adaptation of data transfer and resource usage within each workflow stage. As future work, we aim to investigate token bucket variants for better exploiting QoS enforcing and incorporating ADSS variants, so that the ADSS can also play an important role in QoS enforcement.

References


