The QUASIT Model and Framework for Scalable Data Stream Processing with Quality of Service

Paolo Bellavista, Antonio Corradi, and Andrea Reale

DISI - University of Bologna, Italy
{paolo.bellavista,antonio.corradi,andrea.reale}@unibo.it

Abstract. Many academic and industrial research activities have recently recognized the relevance of expressive models and effective frameworks for highly scalable data processing, such as MapReduce. This paper presents the novel Quasit programming model and runtime framework for stream processing in datacenters, with its original capabilities of i) allowing developers to choose among a large set of quality policies to associate with their processing tasks in a fine-grained way, and ii) effectively managing processing execution depending on the associated quality indications. The paper describes the Quasit programming model, via the primary design/implementation choices made in the Quasit runtime framework (available for download from the project Web site) to achieve maximum scalability, flexibility, and reusability. The first experiences with our prototype and the reported experimental results show the feasibility of our approach and its good performance in terms of both limited overhead and horizontal scalability.

Keywords: Stream Processing, Scalability, Quality of Service, Support Frameworks.

1 Introduction

In the last years we have experienced an unprecedented growth in the amount of digital information created everywhere and accumulated day by day. New data are continuously generated by very heterogeneous sources and for very different purposes: for instance, people periodically update their status on social networks and post multimedia data on the Web; industrial sensors monitor critical operational/safety parameters of production plants; most importantly, the recent mass market success of always-connected mobile and portable devices featuring rich sensing capabilities, such as smartphones or tablets, has created an unprecedented scenario where users continuously sense and share data about the physical environment in which they move and act.

A common trend to face the challenge of processing this huge amount of data is to leverage the computing power of commodity computers inside datacenters [1]: by using highly-parallel and fault-tolerant software architectures, extremely complex processing tasks can be performed while keeping costs reasonably limited. In this perspective, frameworks that help handling the complexities of parallel

C. Borcea et al. (Eds.): MobilWare 2012, LNICST 65, pp. 92–107, 2013.

processing on large clusters, e.g., Google MapReduce [2] and Microsoft Dryad [3], have received enormous attention and are currently widely used in production scenarios. However, while most of these frameworks make static assumptions about the input of their jobs, there is a large class of application domains that need to deal with dynamically changing datasets in form of large *data streams*.

In *data stream processing*, a possibly very large number of streams, coming from multiple and heterogeneous sources, need to be constantly monitored and processed effectively, often in (near) real-time. A very challenging and still open aspect deals with how the computational resources available for stream processing are allocated and used: differently from batch scenarios, where input-data characteristics are usually known a priori, in stream processing it is often hard to predict how the input load will dynamically change. Nonetheless, stream processing solutions are normally required to handle unexpected load peaks, especially when producing mission-critical output, e.g., when monitoring safety conditions and triggering alarms in response to constraint violations.

To properly manage the specific dynamic characteristics of load conditions in *stream processing* scenarios, we claim that there is the need for novel expressive models and effective frameworks that allow developers to describe, with the most appropriate abstraction level and detail, the application-specific requirements of their stream processing case; at the same time, there is the need of frameworks that efficiently support these models and exploit requirement descriptions to achieve the most suitable *Quality of Service* (QoS) in spite of dynamically changing runtime conditions.

The paper presents Quasit, a novel QoS-enabled stream processing model, and the framework supporting this model at runtime that is currently under implementation. Quasit is designed to run effectively on large clusters of commodity hardware and to automatically handle various types of failures. As common in many Stream Processing Engines (SPEs) (e.g., [4,5,6,7,8]), Quasit models stream processing problems as directed acyclic graphs, where nodes represent data transformation stages and edges represent information flows between them. Originally, Quasit allows every element of the *streaming information graph* to be annotated with QoS specifications, used by the runtime framework to adapt to both dynamic load conditions and user-defined quality requirements. In addition, Quasit lets developers define and reuse their custom stream processing *operators*, by supporting their easy dynamic arrangement in graphs to be automatically deployed on the infrastructure of available computational resources. The design of Quasit operators supports a functional-like programming style that clearly separates operator behavior and state, thus making it easier for our runtime framework to support different and sophisticated strategies for QoS provisioning. The source code of our Quasit prototype is freely available for download from the Quasit project Web site[1].

The paper remainder is organized as follows. Section 2 overviews the frameworks in the literature that share some common characteristics with Quasit, by clearly pointing out which are the original aspects of our proposal. In Section 3

[1] `http://lia.deis.unibo.it/research/quasit`

we present the Quasit stream processing model and its QoS support. A description of the Quasit framework architecture and of some central implementation insights is given in Section 4, followed by some preliminary evaluation results that show the feasibility of the approach and the effectiveness of our prototype implementation.

2 Related Work

The most popular model for processing large datasets inside datacenters is certainly MapReduce [2], which has recently received a lot of attention thanks to its ease of use and the diffusion of open source implementations, such as Apache Hadoop[2]. In MapReduce, developers have to model their processing problems only in terms of *map* and *reduce* functions. Leveraging this constraint, the MapReduce runtime takes care of efficiently running the defined functions against input data while providing fault-tolerance and horizontal scalability. This programming model makes the simplifying assumption that input consists of static datasets stored in a distributed file system such as GFS [9], and, thus, is not appropriate for dynamic streaming processing scenarios where input data cannot be statically known.

Given the industrial success of MapReduce, several authors have tried to enhance it with more dynamic and advanced stream processing capabilities. For example, [10,11,12] leverage a *map-reduce-merge* strategy (originally proposed by [13]) to run MapReduce jobs on datasets that are dynamically created as the result of *windowing* operations on data streams; partial output from these jobs is then joined through the additional *merge* step. DEDUCE [14] permits to define MapReduce operators through an extension of the SPADE language [15], and to use these operators within an IBM System S[3] stream processing graph; DEDUCE jobs can run on either static datasets or, as in the previously cited approaches, sliding windows over streaming data. In [16], instead, the authors propose HOP, a modified version of Hadoop that, by supporting intra- and inter-job pipelined communication between map and reduce tasks, permits to run continuous MapReduce jobs. All these examples show the interest in extending MapReduce to solve stream processing problems that can be modeled as a sequence of batch jobs working on "slices" of input streams. However, we claim that, by using a model that is inherently designed to work with static input, these solutions cannot offer the flexibility of a native stream-oriented programming model and are often inadequate to effectively deal with the dynamic characteristics of streaming data, such as highly variable sample rate.

Some existing solutions, similarly to Quasit, use directed graphs to model stream processing problems and to distribute processing responsibilities on available nodes. The Borealis Stream Processing Engine [4,17], for instance, allows users to create *query diagrams* to answer *continuous queries* about input data

[2] `http://hadoop.apache.org`, last accessed in June 2012.
[3] Currently commercialized under the IBM InfoSphere Streams brand.

streams. Users can choose among a set of available operators (defined in a specific query algebra [18]) to build directed graphs that model their stream processing cases. Very interestingly, Borealis allows developers to define QoS specifications for the output of their query diagrams: it is possible to estimate the output quality as a function of *response times*, *event drops*, or specific (and user-defined) *event values*. Quasit adopts these solution guidelines by improving and extending them: Quasit users can additionally define their own operators by directly programming them, and acquire a more direct control of quality-related parameters of every part of the processing graph.

Dryad [3] by Microsoft Research also models computations as directed acyclic graphs. In Dryad graphs, vertices are mapped to native programs that are executed — each in its own process — by the Dryad framework: mainly because of the overhead associated to spawning and managing full processes, the grain of Dryad computational components is coarser than Quasit operators, which, instead, are very lightweight objects confined in the Java Runtime Environment. In addition, while Quasit specifically targets continuous stream processing, Dryad, like MapReduce, seems more oriented to the execution of batch-like jobs where input datasets are fixed and known a priori.

Also SPC [5], the core of IBM System S, and S4, a recent project by Yahoo! [8], share some similarities with Quasit in terms of goals and solution guidelines. Both let developers model their continuous stream processing problems as graphs of *Processing Elements* (PEs), which, similarly to Quasit simple operators, may be user-defined. The main difference between Quasit and these two projects is that our proposal is primarily focused on the support of a rich set of QoS-related parameters to customize stream processing behavior, while SPC and S4 do not allow rich QoS specifications.

3 The Quasit Stream Processing Model

Quasit is used to process multiple input data streams concurrently, to perform arbitrary transformations on them, and to produce other data streams as output, which can be fed to other systems for storage or further processing. A Quasit data stream is modeled as a temporal sequence of data samples, whose content is a set of key-value attributes. Any stream is associated with one data type that defines the keys and types of the attributes of its samples.

The basic modeling unit in Quasit is the *Streaming Information Graph* (SIG), a weakly connected acyclic and directed graph that represents the information flow and the transformations that, applied to one or more input streams, produce an output data stream. The nodes of a SIG represent data transformation stages, while its edges model communication dependencies. Figure 1 depicts a simple example of SIG.

Three different kinds of SIG nodes are possible: *data source*, *data sink*, or *operator*. A *data source* node identifies a data stream that is conceptually out of the SIG and its role is to abstract from the actual nature of the stream

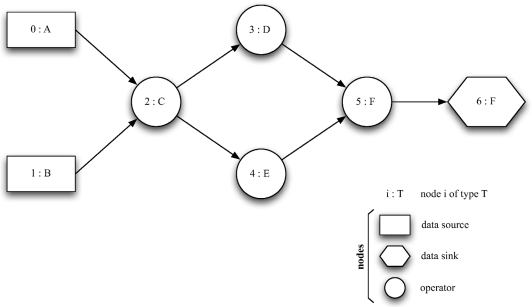

Fig. 1. Simple SIG example, with two data source nodes, one sink node, and four operator nodes. source0 and source1 respectively produce a data stream of typeA and typeB; operator2 receives them as input and produces a typeC data stream, received by operators 3 and 4, producing respectively typeD and typeE data streams. Finally, the typeF data stream generated by operator5 goes into data sink6, of the same type.

producer; it can represent either an external stream source or the output of another Quasit SIG. A *data sink* node, conversely, represents the destination of the data stream that is the output of the SIG; data sinks can be used either to redirect output streams to other systems for additional processing steps or storage, or to connect the output of a SIG with the input of another SIG. An *operator* node associates with one or more input data streams and *generates* exactly one output stream. SIG edges model communication *channels* between nodes.

Every element of a SIG (either node or edge) may be labeled with a QoS specification: QoS specifications allow users to enrich their processing graphs with additional information about non-functional quality requirements. Given the centrality of QoS specifications and their runtime support in Quasit, we will devote a specific section (Section 3.2) to them; but, before that, let us first present the basic building block of our SIG, i.e., the *operator* component, based on which developers can model their stream processing issues in terms of composition of simple transformation stages.

3.1 Operators

An *operator* performs arbitrary operations on the data samples it receives as input, and produces samples for its output stream. We designed Quasit operators having in mind three main goals. First, an operator should be *"concurrency friendly"*: whenever the application semantics allow it, the execution of different operators should be parallelized across all the available processing resources; this should require few or no effort at all for the developer defining the operator. Second, operators should be *easily manageable* in order to allow the Quasit framework to effectively control their execution at runtime, e.g., by moving them from a processing node to another, saving and restoring their processing state,

or transparently recovering them from failures. Third, the operator abstraction should favor *maximum reusability* in order to let developers model their problems in terms of SIGs by writing as less new code as possible.

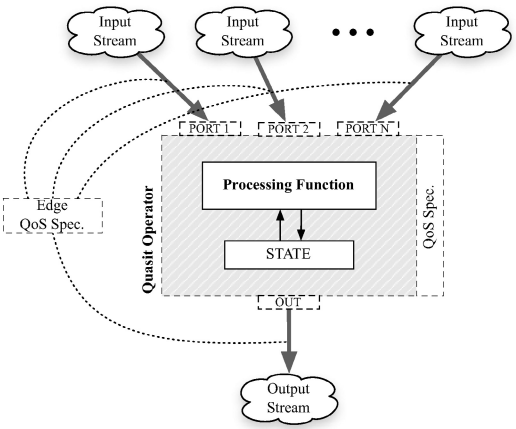

Fig. 2. Structure of a Quasit *simple operator*

Quasit operators can be *simple* or *composite*, and both types can be either *stateful* or *stateless*, depending on whether they need a processing state to be kept or not. A *simple operator* logically consists of several sub-components, as shown schematically in Figure 2. It always has one or more *input ports* and exactly one output port: input ports model the input requirements of the operator, while the output port represents its output contract. The behavior of the operator depends on the combination of its *state* and *processing function*, or solely on the *processing function* in the case of stateless operator.

The processing function is a user-defined function that the Quasit framework invokes asynchronously as data samples are available at input ports. If the operator is stateless, the function takes one parameter, which is bound at runtime to the incoming data samples; if it is stateful, a further parameter is present and is bound to the current state of the operator. The output of the processing function is a tuple made of two optional components: if present, the first is the data sample to send to the output port; the second, always absent for stateless operators, represents the new state the operator will assume. In other words, by defining an operator's processing function, developers specify the set of transformations that, applied to the input, produce its output and state transitions.

Quasit adopts an asynchronous and event-based processing approach, according to which an operator produces output and/or changes its state only in response to incoming data; this permits a large number of operators to share processing resources very efficiently, by enabling high execution *concurrency* in multi-processor and multi-core environments. Furthermore, the sharp separation

between the behavior of the operator, expressed through its (stateless) processing function, and its processing/communication state gives Quasit great *flexibility* in taking transparent management decisions at runtime, in order to effectively support the execution of operator components. For instance, Quasit can offer complex and differentiated state persistence/reliability policies, which would have been much more difficult to realize if state was kept mixed with processing logic.

To achieve *maximum reusability*, Quasit introduces a mechanism that permits to use already defined operators as building blocks for creating more complex and powerful ones, i.e., *composite operators*. Developers can define composite operators by arranging existing operators (either simple or composite) into a special type of SIG that completely defines the execution characteristics of the composite operator, called *Operator Definition SIG* (OD-SIG). Operator composability permits to easily encapsulate complex behavior into composite operators, and leverage them to model many problems, with evident reusability advantages.

3.2 QoS Support in Quasit

One of the most original aspects of Quasit is its ability to let developers augment their stream processing models with very rich and differentiated QoS specifications, to be used at runtime to guide the Quasit framework in the management of system behavior and resource allocation according to the desired quality requirements. Related to the design of Quasit QoS-related features, our main goal is to support a wide spectrum of QoS policies, ranging from simple and high-level quality indications (allowing developers to express their requirements quickly and with as few effort as possible) to richer and lower-level parameters, to be used for finer performance tuning when a deeper and more QoS-aware control over processing is needed.

In particular, any SIG element can be augmented with an optional *QoS Specification*, defining a set of non-functional configuration parameters or constraints. Depending on its target, a QoS specification can consist of several *QoS Policies*, each policy influencing a different quality aspect. In this paper, because of the limited space available, we will not provide a detailed and exhaustive description of all the QoS Policies supported by the Quasit framework (some of them are currently under implementation). However, in order to provide readers at least with a high-level view of the practical aspects that can be regulated through QoS augmentation of SIGs, we report, in Table 1, a concise list of the Quasit QoS policies, also showing their applicability scope and their possible values.

As far as we know, the rich variety of QoS modeling options available in Quasit is unique in the literature about data stream processing solutions. Let us remark again that a proper tuning of the various QoS Specifications attached to SIG elements permits to flexibly adapt the Quasit runtime to different application scenarios, by deeply influencing its strategies for effectively allocating and scheduling the dynamically available processing resources; some details about how the Quasit framework effectively puts into execution the Quasit SIG elements and manages them at runtime are presented in the following part of the paper about Quasit framework design and implementation.

Table 1. Concise list of Quasit *QoS Policies*

Element	QoS Policy	Possible values
Data Sink	Output Priority	*Priority value*
Operator	Processing cap	*Time threshold*
Operator	State fault tolerance	*Replication factor*
Operator	State consistency	*Lazy*, *Snapshot*, *Strong*
Operator	Queuing Spec.	*Input queues size*, *Scheduling policies*
Operator	Input Ordering	*No order*, *Causal*
Channel	Delivery Semantics	*Best Effort*, *At most once*, *At least once*, *Exactly once*, *Probabilistic*
Channel	Deadline	*Time threshold*

4 The Quasit Framework Prototype

In the following, we present the results of our research work of design, implementation, experimental validation, and quantitative evaluation of a first prototype of the Quasit framework, which implements the Quasit model previously described; let us remark once again that the source code of our framework is freely available for download, evaluation, and extension at our project Web site[1].

This section is structured in three parts: in the first (Section 4.1) we present the Quasit architecture; in Section 4.2 we overview how QoS is achieved and controlled at runtime, while in Section 4.3 we provide some implementation insights about the current Quasit prototype.

4.1 Distributed Architecture

Like other systems for data management and processing in datacenters [2,3,8,9], the Quasit distributed architecture follows a simple *master-workers* model, where a logically centralized node (the *master*) implements management and coordination tasks, while a possibly large number of *worker* nodes perform data processing tasks. In particular, Quasit user-defined SIGs are deployed and executed by a set of computing nodes called *Quasit Runtime Nodes* (QRNs), which are monitored and managed by one *Quasit Domain Manager* (QDM), as shown in Figure 3. The set of QRN nodes and the QDM that manages them are collectively called *domain*. A domain runs one or more SIGs, providing advanced runtime services, such as tolerance to operator/QRN failures, and — most importantly — QoS-based management of SIG execution. New SIGs can be added to the domain dynamically at runtime. We assume that QRNs are connected through a high-speed local area network (LAN), as typically occurs in datacenter scenarios.

In order to distribute the workload and leverage all the dynamically available resources, Quasit decomposes arbitrarily complex user SIGs in smaller units, which are then assigned to individual worker nodes and executed in parallel. The granularity of work decomposition and distribution is determined by the defined *simple operators*.

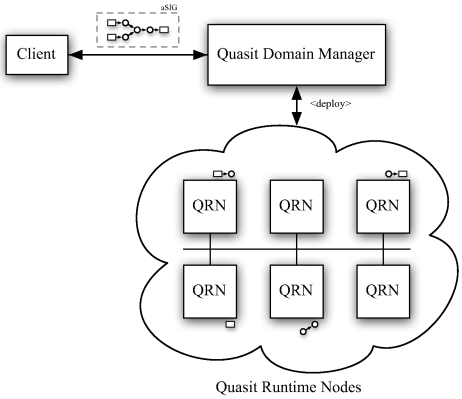

Fig. 3. A Quasit domain includes one QDM (conceptually centralized entity with monitoring and management responsibilities) and several QRNs as middleware instances performing the actual stream processing

Clients submit SIGs to the QDM, which is responsible of planning and continuously monitoring their distributed execution. As soon as a new SIG is received, the QDM must decide an initial partitioning, in order to determine its distributed execution among the available QRNs. The QDM takes this decision by running an *operator placement algorithm* that exploits information about the current status of the QRNs in the domain (e.g., the list of operators already running and their resource availability) to *optimize the execution cost* of the SIG according to the enforced QoS-aware cost function. The development of a proper cost function and placement algorithm is one of our main research challenges: in the current prototype we are exploring a greedy algorithm, called *affinity placement*, which sequentially assigns every operator to the QRN that minimizes its local execution cost, and two additional more trivial algorithms, primarily used as comparison references, i.e., *uniform* and *random* placement, which respectively distribute the operators uniformly (according to a topological ordering of graph vertices) and randomly on the QRNs. An accurate description of the algorithms is out of the scope of this paper, which aims at providing the first high-level presentation of the Quasit model and framework. Although conceptually centralized (and currently implemented in a centralized way), let us point out that the QDM does not represent a bottleneck for the Quasit architecture, because it is not directly involved either in data processing or in any data transfer; moreover, we plan to implement resilience to QDM failures through traditional replication techniques applied to the only QDM entity [21].

A QRN implements a QoS-aware execution container for Quasit operators and is responsible for offering them scheduling and communication support. Reflecting the operator model, the QRN execution model is *asynchronous and event-based*. Communication between operators is managed by the set of distributed QRNs according to a PUB/SUB interaction model: every output port of operators (or data sinks) running on a QRN associates with a named endpoint; QRNs

subscribe to all the endpoints associated with the input ports of operators (and data sinks) that they are running, and store the samples from these subscriptions in event queues associated with the input ports. A pool of executor threads is used to pick samples from the queues, dispatch them to their destination operators, and execute the associated processing function.

4.2 QoS Management

QoS policies defined at model-level on Quasit SIGs are enforced at runtime thanks to a two level QoS-management architecture, realized through the interaction of one *domain QoS manager*, running within the QDM, and several *node QoS managers*, one for each QRN. The domain QoS manager performs global admission control and QoS-based system configuration, while node QoS managers leverage the computational resources of the QRNs on which they execute to implement and enforce the requested QoS policies on locally running operators and I/O ports.

In order to provide a better insight about this QoS management scheme, let us briefly examine its role in the process of deployment and execution of a SIG. At *deployment time*, the domain QoS manager, after having checked whether the QoS policies applied to the SIG are self-consistent, performs a translation phase, during which user-level QoS policies are transformed to implementation specific configuration parameters, which are sent to QRNs inside operator deployment commands. For example, QoS policies on channels, such as the *delivery semantics* policy, are translated into configuration parameters for the PUB/SUB protocol and for the network queues used by the ports corresponding to the channel endpoints. Node QoS managers use these data to provide an initial configuration for the instances of operator and ports they are responsible of. At *execution time*, QoS monitoring tasks are cooperatively performed by domain and node QoS managers: node managers continuously collect data about the behavior of their locally running components, and try to autonomously adjust their configuration to avoid possible QoS violations; for example, they can reallocate their local resources by giving a greater share to operators with higher priority (thus, penalizing the less important ones). At the same time, they also forward monitoring data to the domain QoS manager, which will use them to take authoritative decisions in case adaption actions of single local managers are not sufficient to avoid QoS violations; for example, it can decide to move an operator from a QRN to another in case the latter has a greater amount of resources to allocate to its execution.

4.3 Implementation Insights

Our QDM and QRN components are realized using the Scala[4] programming language. Scala has been preferred to other possible alternatives for three main reasons: first, the language runtime comes with a rich library that offers an

[4] `http://www.scala-lang.org/`, last accessed in June 2012.

excellent support for writing concurrent and multi-threaded applications; second, its elegant and concise syntax allows us to simplify the design of the user API through which developers model their stream processing problems; third, Scala code, once compiled, is executed on the solid and widely supported Java Runtime Environment.

Quasit PUB/SUB interactions are instead realized on top of the OMG Data Distribution Service (DDS) [22] middleware, which is used as the basis for both reliable group membership management and inter-QRN SIG channels. The choice of using a DDS-based communication middleware grants several benefits. First, DDS message dissemination uses an IP-multicast-based protocol that well fits the typical one-to-many communication patterns of Quasit operators and perfectly adapts to network characteristics of datacenters where nodes are commonly arranged in a hierarchy of Ethernet segments, connected by layer2 switches. Second, the DDS standard defines a rich set of QoS parameters, that can be used to configure and personalize many low-level details of the communication middleware: using DDS to implement our PUB/SUB communication layer has provided us with a solid ground on which we build our ad-hoc QoS enforcement mechanisms, especially those relative to channels. Whenever possible, in fact, we exploit mappings between high-level Quasit QoS policies and possible configurations of the various DDS QoS parameters, and set up the QRN networking layers according to them.

Finally, the scheduling of actors and the management of their queues is currently implemented using the Scala Actors framework [23]: every operator is represented by an actor instance, which perfectly suits our event-based processing model. Currently, the scheduling of these actors is taken care by a *work-stealing* pool of threads based on the Java Fork/Join framework [24]. This scheduler, in the currently available version of the Quasit prototype, does not permit any QoS-based configuration: we plan to add this feature as a future implementation step.

5 Preliminary Evaluation

In this section we present some first preliminary results collected while testing our Quasit framework prototype in a relatively small-scale deployment environment. The reported results demonstrate anyway the feasibility and the effectiveness of our approach.

The selected and simple test scenario consists of an external source producing a periodic stream of image frames. For instance, this stream could correspond to the sequence of key frames of a video produced by a security camera. These image samples are transformed through a series of manipulation steps, and then streamed again to an external destination. The samples generated from the source correspond to the repetition of a 192x128 24bpp PNG image, which is a scaled version of one of the photos from a public test set by Kodak[5]. The size of each sample is approximately 43 KB.

[5] `kodim23.png`, publicly available at `http://r0k.us/graphics/kodak/`, last accessed in June 2012.

We have modeled the image manipulation process as a pipeline of Quasit operators, whose processing function is implemented as stateless OpenCV[6]-based transformations. The combination of these operators forms a 30 steps pipeline-shaped SIG (as shown in Figure 4) deployed and run on top of the Quasit framework prototype. All the stages of this pipeline have approximatively the same computational complexity. Let us note that this simple scenario is anyway highly representative because i) pipeline-shaped patterns are very common in more complex SIGs and ii) the number of involved operators (30) is relatively high and close to the real size of many SIGs of practical application interest.

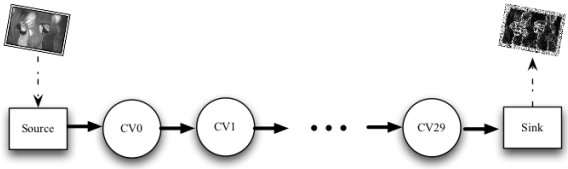

Fig. 4. The simple and pipeline-shaped SIG used in this experimental evaluation

The testbed Quasit domain consists of one machine running the QDM component, plus from one up to four different physical nodes having the role of QRNs. The QRNs are interconnected through one Ethernet segment, while the QDM, although in the same IP subnet, is separated from the QRNs by two switches. The machine hosting the QDM is also used as the external source and sink of the image frames. The hardware and software configuration of the machines is shown in Table 2.

In each experiment run, we feed the deployed SIG with 500 image samples, not counting "warm-up" and "cool-down" sets of samples processed when the SIG pipeline is not full. For each configuration, we have collected the results of 15 to 50 runs of the same experiment (depending on the variability of results).

The experimental results reported in the following aim at discussing two main performance aspects that we have measured on our testbed:

- The management overhead with respect to an ideal parallel processing scenario.
- The ability to scale horizontally, by dynamically adding QRNs to one Quasit domain.

In order to quantitatively evaluate the overhead imposed by the Quasit middleware (if compared with the maximum possible improvement of stream processing performance thanks to parallelization), we have also designed a very simple simulator that models our scenario but omits all the overhead associated with middleware-level management of operators (including operator scheduling) and inter-QRN network communication. The simulator models a group of parallel

[6] OpenCV, `http://opencv.willowgarage.com/wiki/`, last accessed in June 2012.

Table 2. Hardware and software configuration of QRN nodes

Host: Intel Pentium Dual-Core E2160 @ 1.80GHz w/ 2 GB RAM
RAM: 2 GB
Network Interface: Gigabit Ethernet
OS: Ubuntu 11.04 (Linux kernel 3.0.0)
DDS: OpenSplice DDS 5.4.1 Community Edition
Scala: 2.9.1-final
JVM: OpenJDK 64-bit Server VM (IcedTea7-2.0 build 147)
JVM Flags: -Xms128M -Xmx512M -Xss4M

workers arranged in a pipeline; their number reflects the number of available CPUs across all the QRNs. OpenCV transformations of the original SIG are distributed evenly among workers, and each of them executes sequentially, for each incoming sample, the transformations it is responsible for, before forwarding it to the next worker. In the simulations, we measure the average time needed to perform a complete processing of an image sample by varying the rate at which new samples are produced, and we compare the results with the performance data obtained on a real deployment environment with 4 QRNs in a Quasit domain (operators deployed according to the uniform placement strategy). In the real deployment environment, image processing time is measured as the sample *round trip time* (RTT), i.e., the time interval between the generation of a new frame and the reception of the processed version of that frame (recall that the external source/sink of the input/output streams coincide in our simple pipeline-shaped test SIG). Figure 5a shows the distribution of the measured RTTs while increasing generation rates in the real deployment and the average processing time in the "ideal" simulated scenario.

Clearly, in both cases, the processing time increases abruptly as soon as our Quasit framework is no longer able to keep up with image production rate and the input queue of the first operator (worker) starts filling up. For low sample rates, Quasit performance is very close to the ideal one, thus demonstrating a limited overhead in unloaded conditions; the difference tends to grow as the input rate increases; we experienced that this is mainly due to the overhead introduced by operator scheduling, which is completely neglected in the simplified simulated scenario.

About our second evaluation goal of verifying the ability of Quasit to scale as additional QRNs are added to a domain, we have deployed the same test pipeline-shaped SIG on four different execution environments, with respectively one, two, three, or four QRNs. In all cases we have deployed the graph using the uniform placement strategy. Figure 5b shows the results. The trend of the curves is the same in all the examined domains: as long as the production rate does not exceed the maximum processing rate in unloaded conditions, the average sample RTT is constant and low (around 450 milliseconds); as soon as Quasit is no longer able to keep up with the sample arrival rate, the average processing time starts to grow. However, the results show that by adding processing resources to one Quasit domain, it is seamlessly possible to increase the Quasit ability to

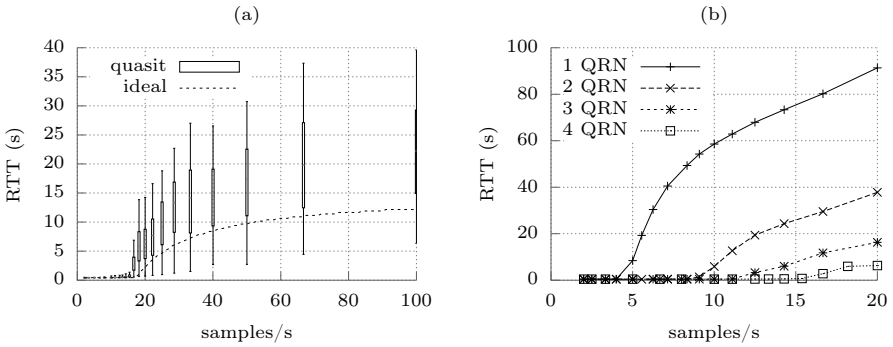

Fig. 5. (a) Distribution of sample processing with 4 QRNs and uniform operator placement. The dashed line represents the performance upper bound in ideal conditions. (b) Comparison of average processing times using 1, 2, 3, or 4 QRNs and uniform placement.

serve more aggressive input rates, with reasonably limited overhead. In fact, it can be seen that by using two, three, or four QRNs Quasit case serve an input rate respectively 1.82, 2.5, and 3.34 times faster if compared to the one QRN configuration[7], thus showing a limited degradation. Of course, the possible speed-up does not grow linearly with the number of QRNs because of the overhead due to management and network communication. However, the system ability to scale horizontally also depends strongly on the characteristics of the SIGs being executed: for this reason, Quasit fosters a SIG design made of many fine grained components sharing no state, giving the framework many parallelization opportunities to be exploited according to the required QoS level and resource availability.

6 Conclusive Remarks and Future Work

In this paper we have introduced Quasit, both a programming model and a framework prototype for stream processing in datacenters. Compared to existing literature and available industrial solutions, Quasit is original in its ability to offer a large set of QoS policies to customize its processing behavior according to user-defined application requirements. The model of data stream processing is simple and easy to use: it is based on easy-to-define operators and events, and it permits to model, design, and realize stream processing operations in a simple but flexible way. Our first prototype of the Quasit runtime, although still early and partial, represents a concrete proof-of-concept of a possible implementation of the proposed model (available for extension and refinement to the community of researchers/practitioners in the field), and encourages further development.

[7] For this comparison, we have considered the sample rate at which the system starts to become overloaded and to accumulate data at the operator queues.

We are concentrating our future work along two main directions. On the one hand, we will extend our prototype toward the implementation of a richer set of QoS policies for SIG operators and channels, and we will experiment alternative operator placement and management strategies. On the other hand, we are performing a more significant set of experiments to verify the ability of our Quasit model and prototype to sustain challenging large-scale deployment environments, with a special focus on dynamic differentiation of stream processing services depending on QoS requirements specified at the SIG level.

References

1. Barroso, L., Dean, J., Holzle, U.: Web search for a planet: the Google cluster architecture. IEEE Micro 23(2), 22–28 (2003)
2. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51(1), 107–113 (2008)
3. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, vol. 41(3), pp. 59–72. ACM, New York (2007)
4. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: 2nd Biennial Conference on Innovative Data Systems Research (CIDR), pp. 277–289. VLDB Endowment (2005)
5. Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Park, Y., Venkatramani, C.: SPC: A distributed, scalable platform for data mining. In: Grossman, R., Connelly, S. (eds.) 4th International workshop on Data Mining Standards, Services and Platforms (DM-SS), pp. 27–37. ACM, New York (2006)
6. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: The Stanford Data Stream Management System, Technical report, Stanford InfoLab (2004)
7. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring streams: a new class of data management applications. In: 28th International Conference on Very Large Data Bases (VLDB 2002), pp. 215–226. VLDB Endowment (2002)
8. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed Stream Computing Platform. In: 2010 IEEE International Conference on Data Mining Workshops (ICDMW 2010), pp. 170–177. IEEE, Los Alamitos (2010)
9. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google File System. ACM SIGOPS Operating Systems Rev. 37(5), 29–43 (2003)
10. Alves, D., Bizarro, P., Marques, P.: Flood: elastic streaming Map-Reduce. In: 4th ACM International Conference on Distributed Event-Based Systems (DEBS 2010), pp. 113–114. ACM, New York (2010)
11. Horey, J.: A programming framework for integrating web-based spatiotemporal sensor data with MapReduce capabilities. In: ACM SIGSPATIAL International Workshop on GeoStreaming, pp. 51–58. ACM, New York (2010)
12. Logothetis, D., Yocum, K.: Ad-hoc data processing in the cloud. Proceedings of the VLDB Endowment 1(2), 1472–1475 (2008)

13. Yang, H.-C., Dasdan, A., Hsiao, R., Parker, D.: Map-reduce-merge: simplified relational data processing on large clusters. In: 2007 ACM SIGMOD International Conference on Management of Data, pp. 1029–1040. ACM, New York (2007)
14. Kumar, V., Andrade, H., Gedik, B., Wu, K.-L.: DEDUCE: at the intersection of Map-Reduce and stream processing. In: Manolescu, I., Spaccapietra, S., Teubner, J., Kitsuregawa, M., Leger, A., Naumann, F., Ailamaki, A., Ozcan, F. (eds.) 13th International Conference on Extending Database Technology (EDBT 2010), pp. 657–662. ACM, New York (2010)
15. Gedik, B., Andrade, H., Wu, K.-L., Yu, P.S., Doo, M.: SPADE: the System S declarative stream processing engine. In: 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008), pp. 1123–1134. ACM, New York (2008)
16. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce Online. In: 7th USENIX Conference on Networked Systems Design and Implementation (NSDI 2010). USENIX Association, Berkeley (2010)
17. Ahmad, Y., Tatbul, N., Xing, W., Xing, Y., Zdonik, S., Berg, B., Cetintemel, U., Humphrey, M., Hwang, J.-H., Jhingran, A., Maskey, A., Papaemmanouil, O., Rasin, A.: Distributed operation in the Borealis stream processing engine. In: 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), pp. 882–884. ACM, New York (2005)
18. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB Journal The International Journal on Very Large Data Bases 12(2), 120–139 (2003)
19. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An Overview of the Scala Programming Language. Technical Report, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland (2004)
20. Emir, B., Odersky, M., Williams, J.: Matching Objects with Patterns. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 273–298. Springer, Heidelberg (2007)
21. Guerraoui, R., Schiper, A.: Software-based replication for fault tolerance. Computer 30(4), 68–74 (1997)
22. Object Management Group: Data Distribution Service for Real-time Systems, version 1.2. Technical report, Object Management Group (2007)
23. Haller, P., Odersky, M.: Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
24. Lea, D.: A Java fork/join framework. In: ACM 2000 Conference on Java Grande (JAVA 2000), pp. 36–43. ACM, New York (2000)