ICST Transactions on Mobile Communications and Applications Research Article ICST.ORG

Scalable Stream Processing with Quality of Service for Smart City Crowdsensing Applications

Paolo Bellavista, Antonio Corradi, and Andrea Reale

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

Abstract

Crowdsensing is emerging as a powerful paradigm capable of leveraging the collective, though imprecise, monitoring capabilities of common people carrying smartphones or other personal devices, which can effectively become real-time mobile sensors, collecting information about the physical places they live in. This unprecedented amount of information, considered collectively, offers new valuable opportunities to understand more thoroughly the environment in which we live and, more importantly, gives the chance to use this deeper knowledge to act and improve, in a virtuous loop, the environment itself. However, managing this process is a hard technical challenge, spanning several socio-technical issues: here, we focus on the related quality, reliability, and scalability trade-offs by proposing an architecture for crowdsensing platforms that dynamically self-configure and self-adapt depending on application-specific quality requirements. In the context of this general architecture, the paper will specifically focus on the Quasit distributed stream processing middleware, and show how Quasit can be used to process and analyze crowdsensing-generated data flows with differentiated quality requirements in a highly scalable and reliable way.

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

Received on 20 April 2013; accepted on 26 June 2013; published on 16 December 2013

doi:10.4108/mca.1.3.e6

1. Introduction

The vision of smart cities as urban areas where people, places, environment, and administrations become closer and get connected through novel ICT services and networks, is becoming reality at an increasingly faster pace. Thanks to disruptive technologies such as location-based services and ubiquitous connectivity via multiple interfaces, cities are promising candidates to become, in the near future, a central development and deployment platform for a novel and increasingly important set of services. Confirming this trend, in the last years there have been several initiatives led by governments and industries directed toward the study and development of smart urban areas. Examples of these initiatives are the many European Digital Agenda funded projects, such as European Digital Cities, InfoCities, IntelCity roadmap, Intelligent Cities, and EUROCITIES [15], or industry-led activities, such as the IBM Smarter Cities project[1], or the Intel Collaborative Research Institute for Sustainable Connected Cities[2]. At the same time, the widespread diffusion of smartphones and tablets with heterogeneous connectivity and rich sensing capabilities creates novel opportunities for extending the reach of smart city services. Among them, one of the most interesting and still open directions is the exploitation of the new sensing paradigm that has been usually defined as *crowdsensing*.

Crowdsensing shifts the principles of the more traditional crowdsourcing processes — mostly diffused in static Internet scenarios (see, for example, Wikipedia[3]

[1] http://www.ibm.com/smarterplanet/us/en/smarter_cities/, last visited in August 2013.
[2] https://www.intel-university-collaboration.net/?page_id=1420, last visited in August 2013.
[3] http://www.wikipedia.org/, last visited in August 2013.

or Amazon Mechanical Turk[4]) — to cyber-physical urban spaces, by leveraging the unprecedented monitoring capabilities of mobile citizens and coupling them with distributed actuation tasks to be completed collaboratively, for instance by moving data or physical items (e.g., bikes of a smart city bike sharing service) to maximize the targeted objective functions (e.g., uniform coverage of traffic monitoring or uniform bike availability at bike pickup points). This way, relatively complex distributed goals can be achieved thanks to the weakly-organized, and massive-scale cooperation of the mobile *crowd*: in a typical crowdsensing application, people are asked to perform simple and often very fine-grained geo-based tasks that usually involve tracing some physical real-world measure through their mobile devices, such as registering noise or environmental pollution in some area, or taking photos of specific locations.

Connecting participatory sensing with crowd-based actions in what we call the *crowdsensing loop* is a fundamental, continuous step of real-time and large-scale data monitoring and analysis: by aggregating and processing the flowing streams of data coming from collaborative citizens, it is possible to obtain significant information about the current status of the city and its inhabitants, and to build models that can accurately forecast city dynamics, which, in turn, can help decision-makers to determine the appropriate actions and policies that improve the overall urban quality-of-life. The very peculiar characteristics of these new processing scenario pose several fundamental challenges to existing data processing platforms, and call for novel models/architectures that can satisfy strong requirements of scalability, quality, reliability, and cost-effectiveness.

This paper introduces a general and novel crowdsensing architecture, whose ultimate goal is to guide the realization of scalable and reliable crowdsensing platforms that exploits the concept of *quality* at two complementary levels, i.e., i) at *task generation/assignment* level and ii) at *data processing* level. After briefly introducing our ongoing work to implement this architecture in a working middleware-level solution, we concentrate the focus on the main subject of this paper by describing in detail Quasit, a novel data stream processing model and middleware implementation specifically designed for scenarios with strong scalability and quality requirements. Quasit is built to run effectively on large clusters of commodity hardware and to automatically handle various types of failures. Originally, Quasit allows to annotate its processing elements with QoS specifications, which are leveraged at runtime to adapt their behavior to both dynamic load conditions and user-defined quality requirements. We present preliminary results obtained with our under-development prototype (open-source and freely available for download[5]), showing that out system combines the easy definition of processing functionalities and QoS requirements, with automatic scalability to the available processing resources.

Let us note that the Quasit stream processing model and framework can also be used to application domains and scenarios other than smart city crowdsensing applications described here. We claim that any big-data stream-oriented application benefiting from QoS-aware task assignment and data processing could fruitfully exploit the QoS-aware scalability of Quasit, with significant performance improvements if compared with traditional, non-QoS-aware stream processing frameworks. However, in this paper, for the sake of description focus, we will describe only how Quasit may be usefully adopted in the wide domain of crowdsensing applications.

The remainder of the paper is organized as follows. In Section 1 we present our novel crowdsensing architecture, by introducing the challenges that it tries to solve and explaining the solution approach that it proposes. In the context of this architecture, we introduce Quasit, whose processing model is presented in Section 3. A description of the design of the Quasit prototype and some central implementation insights is given in Section 4, followed, in Section 5, by a set of preliminary evaluation results that show the feasibility of the approach and the effectiveness of our prototype implementation, also compared to a solid, state-of-the-art alternative as Apache S4 [26]. Section 6 overviews the work in the literature sharing common characteristics with Quasit, and clearly points out the original aspects and technical elements of our proposal. Conclusive remarks and directions of ongoing research work end the paper.

This paper is an extended version of [9]. It originally introduces Quasit within our broader vision of crowdsensing in Smart Cities (Section 2), and includes an extended discussion of the supported QoS policies (Section 3) and additional experimental results in the performance evaluation of our prototype (Section 5).

2. The Crowdsensing Loop

A crowdsensing platform supports the execution of several sensing applications, each typically having one high-level goal, which is decomposed in several small geo-located sensing *tasks* to be executed by the crowd by opportunistically exploiting people movements through the city. The types of applications and tasks

[4] https://www.mturk.com/, last visited in August 2013.

[5] http://lia.deis.unibo.it/research/quasit, last visited in August 2013.

managed by the platform can be highly heterogeneous, and can involve capturing, harvesting, and processing very different *physical features* of the real world, as well as performing actuation tasks that change the status of the cyber-physical space.

A critical aspect to consider, in order to develop a scalable crowdsensing platform, is the important trade-off between the obtained sensing/actuation accuracy and the crowdsensing cost. The execution of a task instance by a citizen is intrinsically *unreliable*: a person may simply refuse to execute a task, or be unable to complete it; more importantly, the quality of the crowd-sensed data can have a high variability due to the poor accuracy/precision of the sensors embedded into personal devices. Task *replication*, i.e., assigning copies of the same task to several people in order to have higher confidence on the collected data or to achieve monitoring reliability through multiple source participation, is the normal solution to this series of challenging issues. However, the efficient determination of how much to replicate a task, or to whom to assign different tasks in a real crowdsensing urban scenario is still an open problem. Moreover, as the result of the execution of thousands, possibly replicated, sensing tasks from different applications, large volumes of sensing data are continuously produced and need to be processed effectively and in almost real time. Notwithstanding the recent advances in data-center/cloud data elaboration technologies, managing this unprecedented volume can still present unacceptable costs and scalability limitations if the peculiar characteristics of crowd-sensed data are neglected by the processing framework.

Our crowdsensing architecture, shown in Figure 1, aims at tackling these fundamental quality/cost trade-offs by putting the task generation/assignment and the data processing phases in a closed loop, and by leveraging at the same time user-provided and autonomously-inferred quality requirements to optimize the platform runtime execution. By processing the data received from the crowd, the data processing component builds two important artifacts. On the one hand, it learns models that incorporate the history of the sensed features and that can be queried by the user to monitor or predict the status of the real world aspects of interest for her sensing applications. On the other hand, it builds and constantly updates *profiles* of users, regions, and sensing features. These profiles are the original core of our architecture, since they represent the elements that close the crowdsensing loop.

This way, cross knowledge of user histories, of the characteristics of different geographical regions, and of the importance of the contribution that different sensing features have on the output models, can be leveraged by the platform to self-regulate the allocation of the available human and computational resources, also by taking into careful consideration application-level quality/reliability requirements. For example, task assignment strategies can be specifically tailored to the citizens' habits or can use personalized incentive types [29] in order to maximize the chance of obtaining the desired sensing accuracy while minimizing costs. Similarly, data-processing components should use the combination of the learned profiles to prioritize the analysis of data streams that can potentially give a more important contribution to the output models, while delaying or discarding less critical information. For instance, considering environmental pollution monitoring, the number of replicas for the monitoring task of a given area can be minimized by assigning them to people who habitually or recently frequented that area. Similarly, previous knowledge about the fact that collecting traffic information in a green area has a minor influence on the air pollution model than the humidity information, can be used to configure the data analysis phase to give priority (and hence more resources) to the processing of humidity crowd-sensed data, by possibly discarding less important monitored features.

2.1. A Scalable Platform for Quality-aware Crowdsensing

By following the ideas, model, and architecture presented above, we are developing a middleware-level crowdsensing platform, based on the convergence of the McSense[6] middleware for crowdsensing task management and the Quasit[5] [10] stream processing engine for quality-aware data analysis.

McSense, developed in collaboration with the New Jersey Institute of Technology, is a middleware for the management of crowdsensing flows; it consists of a central control component, responsible of quality-aware task generation and assignment, and a Software Development Kit (SDK) (only available for the Android platform at the time of writing) that can be used to develop mobile sensing applications that support users in accomplishing all the phases of their crowdsensing tasks. The control component offers ad-hoc tools to describe geo-based sensing tasks and to specify their specific quality requirements, such as the desired reliability (as the expected probability of completing a task) or task completion time. These quality parameters, in conjunction with the profiles knowledge base, are leveraged by a pluggable task assignment algorithm that allocate the appropriate amount of human resources by concretely assigning and dispatching task instances to citizens. Currently, two algorithms can be alternatively chosen: one based on the *recency* of people's last visit to given target locations,

[6] http://lia.deis.unibo.it/research/McSense/, last visited in August 2013.

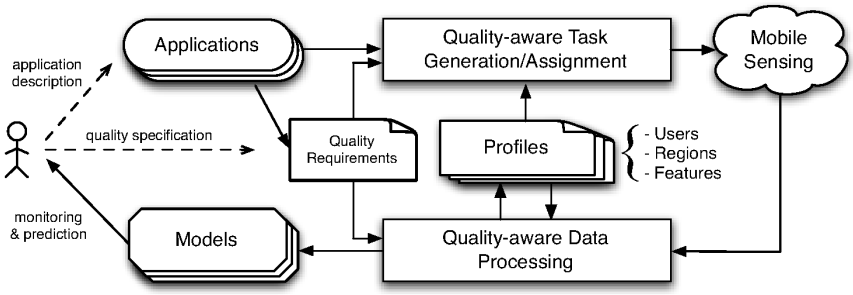

Figure 1. A general high-level architecture for quality-based crowdsensing platforms.

the other based on people habits of *attendance* to certain places. Further details on the McSense platform can be found in [11] or on the project Web site[6].

The other core component of the architecture, i.e., the data back-end processing platform, is being developed by integrating the Quasit framework into the crowdsensing flow. Quasit is a quality-aware stream processing framework for data-center environments designed to handle large volumes of streaming data by seamlessly scaling to the available computing/memory/network resources. It gives developers a simplified processing model that describes streaming problems as the composition of processing graphs (called SIGs, see the following), each made of an interconnected set of reusable *operators*, which define the transformations that, applied to the input data, produce the desired output. Originally, Quasit offers advanced configuration/customization features, by permitting to associate QoS specifications to all the elements of its application models (operators, communication channels, or entire graphs). The set of supported QoS specifications permits to configure the system through several QoS parameters, ranging from high-level indications (e.g., output priorities) to low-level ones (e.g., detailed set-up of network buffers). 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 the runtime framework to support different and sophisticated strategies for QoS provisioning.

The original features of Quasit to auto-configure its behavior depending on application-dependent QoS requirements fit the needs of our crowdsensing vision. Crowdsensing quality requirements, either explicitly specified at the application level or autonomously inferred through users, regions, and features profiles, are automatically mapped to Quasit QoS specifications in order to set up the most appropriate data processing quality. Let us note that, for instance, the possibility to consider only a subset of all available data sources under critical congestion situations is particularly useful in crowdsensing applications due to their intrinsic nature with redundant data streams, typically with similar values, concurrently originated by different smart city "observers". In these cases, proper QoS management of stream processing can allow to achieve the most suitable trade-off between latency and completeness, only to mention a simple example. Developers of crowdsensing applications define their own data aggregation/processing algorithms by arranging prebuilt or custom operators into Quasit SIGs, and associate specific QoS parameters directly to the graphs elements, this way expressing their application-specific requirements (for example, to indicate fine grained low-level resource needs of the various parts of their data analysis steps). In the following sections, we describe the Quasit processing model and the related prototype, and we thoroughly analyze the unique design characteristics that make the platform scalable, easy to use, and capable to effectively support the specification of quality requirements with arbitrary level of detail.

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

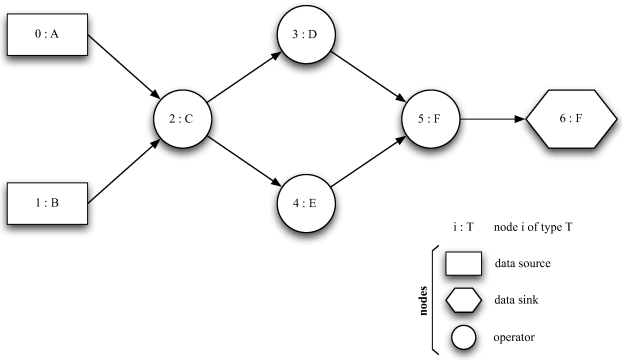

Figure 2. Simple SIG example, with two data source nodes, one sink node, and four operator nodes. source0 and source1 respectively produce a data stream of typeA and typeB; operator2 receives them as input and produces a typeC data stream, received by operators 3 and 4, producing respectively typeD and typeE data streams. Finally, the typeF data stream generated by operator5 goes into data sink6, of the same type.

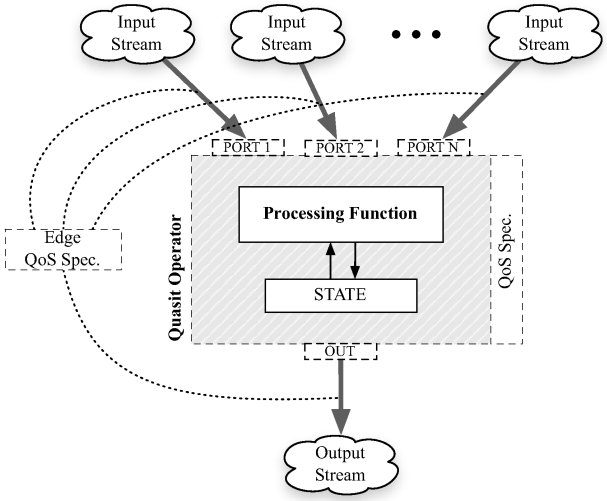

Figure 3. Structure of a Quasit *simple operator*.

its edges model communication dependencies. Figure 2 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 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 Section 3.2 to their discussion; 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.

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 3. 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. Table 1 reports the list of Quasit QoS policies, by concisely showing their applicability scope and their possible values.

Table 1. 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	Queueing 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*

In order to provide readers with a high-level overview of the practical aspects that can be regulated through QoS augmentation of SIGs, in the following we will give a short description of Quasit policies, also by putting them into their practical applicability context by presenting examples of their possible use within a simple crowdsensing scenario. The considered scenario is that of a smart-city application that combines car-sharing services with urban pollution monitoring. A fleet of cars is equipped with air-pollution meters and are put on disposal to citizens, who can request, use, or share them through a mobile application. While moving through the city, cars report their position and real-time pollution data (through their 3G radio) to a data back-end application running on a Quasit deployment. Similarly, the back-end processes and properly matches citizen requests for car trips with car availability.

- *Output Priority*. By using this QoS policy, it is possible to differentiate the way Quasit assigns resources to parts of SIGs that contribute to produce different outputs. In our reference scenario, users may be provided with gold, silver, or bronze services according to their service membership level: these levels can be mapped on different priorities for the operators responsible for matching their requests with possibly available cars.

- *Processing Cap*. This QoS policy determines a hard constraint on the time available to an operator to process a data sample. When this constraint is violated, the computation is interrupted and a default action executed. For instance, the operator that classifies incoming car requests and maps them to the appropriate subgraphs, needs to complete this process fast, or else assign requests to a default class; in this case default assignment is considered better than too late but more precise assignment.

- *State Fault Tolerance and Consistency*. Both policies determine how the state of an operator is handled by the Quasit platform. A weaker state consistency strategy/replication factor can save resources when partial state loss can be tolerated. For instance, the loss of partial updates on some tiles of the urban pollution map is acceptable in many related applications, especially given the supposed high-update frequency.

- *Queueing Specifications*. This low-level policy controls the way Quasit manages the operators input queues. In our scenario, this policy could be used to set blocking behavior for the input queue of an operator that dispatches matched user-car requests and, at the same time, to define an ordering function that prioritizes only the samples related to requests from gold members.

- *Input Ordering*. It determines whether samples entering an operator marked with this policy are to be processed in their arrival order, or if their processing order should satisfy *happened-before* [23] relations established by preceding operators. For example, a request to modify the destination for a car-shared trip should always be handled after the related request for the trip.

- *Delivery Semantics*. This QoS policy, which may be attached to SIG edges connecting a pair of operators, configures the communication protocol used by the Quasit platform to enable data exchange between the two operators. For instance, for reasons that are similar to the ones discussed above for the State Fault Tolerance policy, pollution-map updates can be transferred using a best-effort communication protocol.

- *Deadline*. This policy controls how the samples in an operator output queue are handled. In particular, by setting a time-based deadline on a channel, the Quasit network management layer is instructed to adopt a network-scheduling policy that tries to ensure that every tuple is transferred from source to destination within a required time threshold after its generation. In our example scenario, a deadline could be set on the graph path that manages application critical operations, such as the management of payments for the car-sharing service via users' credit cards.

As far as we know, the rich variety of QoS modeling options available in Quasit is unique in the stream processing literature. 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.

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 stream processing 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[5].

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 data-centers [14, 17, 21, 26], 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 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 4. 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 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 data-center 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. The granularity of work decomposition and distribution is determined by the defined *simple operators*.

Clients submit SIGs to the QDM, which is responsible of planning and 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*

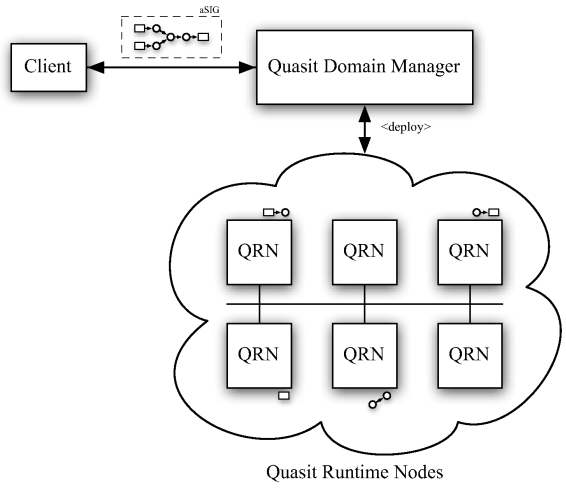

Figure 4. A Quasit domain includes one QDM (conceptually centralized entity with monitoring and management responsibilities) and several QRNs as middleware instances performing the actual stream processing.

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. 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 [18].

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 deeper and more detailed insights 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. This way, most QoS management decisions are taken and enforced locally by node QoS managers, thus relieving the central domain QoS manager from this load and improving the system scalability. Actions by the domain QoS manager are only necessary in a limited set of situations, when global knowledge is needed or when the adaption actions of single local managers are no longer sufficient to avoid QoS violations. For example, it is up to the global QoS manager to decide whether to move an operator from a QRN to another in case of system

overload. This design aims at avoiding the domain QoS manager to represent a bottleneck for system scalability, by demanding as many decision as possible to the local managers. We are planning to perform through measurements to quantitatively evaluate the effectiveness of this solution (see Section 7). However, it is possible that, for very large scale scenarios (e.g, with thousands of nodes), the QoS management duties on the single QDM could become overwhelming: to deal with such cases a possible solution could be a further hierarchical partition of management responsibilities, where the set of all QRNs is divided into separate QoS-management clusters, each supervised by a different cluster head.

4.3. Implementation Insights

Our QDM and QRN components are realized using the Scala[6] 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 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) [27] 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 data-centers 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 [20]: 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. Experimental Evaluation of Quasit Performance

In this section we present some first preliminary results collected while testing our Quasit framework prototype in a relatively small-scale deployment environment. Although the deployment does not reflect the characteristic of our target scenarios completely, its simplicity permits to easily measure and evaluate basic system characteristics, such as the effectiveness of the platform communication and threading mechanisms. We believe that the reported results demonstrate the feasibility and the effectiveness of our approach, and represent an important starting point for a future, large scale evaluation campaign on real-world use cases.

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 Kodak public test set[7]. The size of each sample is approximately 43 KB.

We have modeled the image manipulation process as a pipeline of Quasit operators, whose processing function is implemented as stateless OpenCV[8]-based transformations. The combination of these operators forms a 30 steps pipeline-shaped SIG (as shown in Figure 5) deployed and ran on top of the Quasit framework prototype. All the stages of this pipeline have approximately 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.

The testbed Quasit domain consists of one machine running the QDM component, plus from one up to

[6] http://www.scala-lang.org/, last accessed in August 2013.

[7] kodim23.png, publicly available at http://r0k.us/graphics/kodak/, last accessed in August 2013.

[8] OpenCV, http://opencv.willowgarage.com/wiki/, last accessed in August 2013.

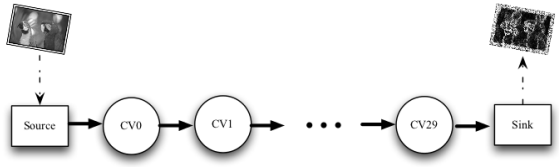

Figure 5. The simple and pipeline-shaped SIG used in this experimental evaluation.

Table 2. Hardware and software configuration of QRN nodes.

Host: Intel Pentium Dual-Core E2160 @ 1.80GHz
Main Memory: 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

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 three main 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.
- A preliminary performance comparison with Apache S4, a state-of-the-art stream processing engine by the Apache Software Foundation.

5.1. Comparison with Ideal Parallel Processing

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 parallel execution), we have also designed a very simple simulator that models our scenario but omits all the overhead associated with middleware-level management of

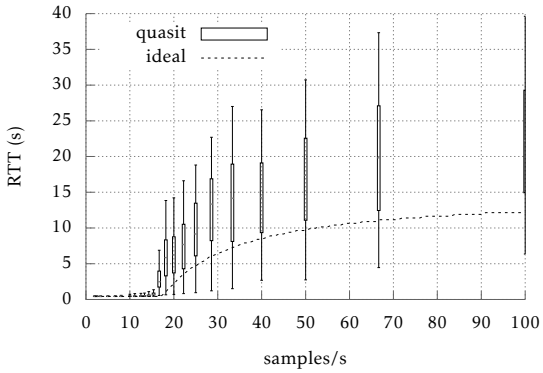

Figure 6. Distribution of sample processing time with 4 QRNs and uniform operator placement. The dashed line represents the performance upper bound in ideal conditions.

operators (including operator scheduling) and inter-QRN network communication. The simulator models a group of parallel 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 6 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 the very limited platform 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.

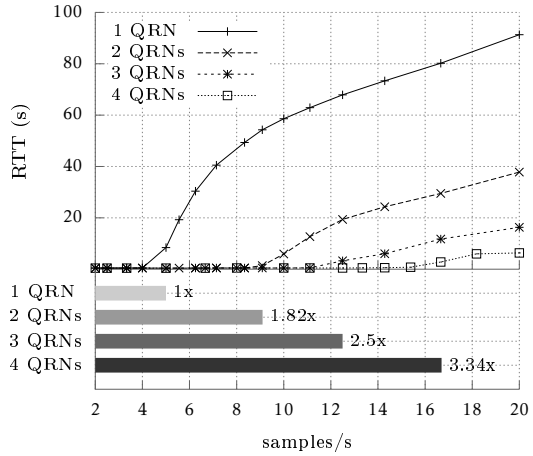

Figure 7. Comparison of average processing times using 1, 2, 3, or 4 QRNs and uniform placement.

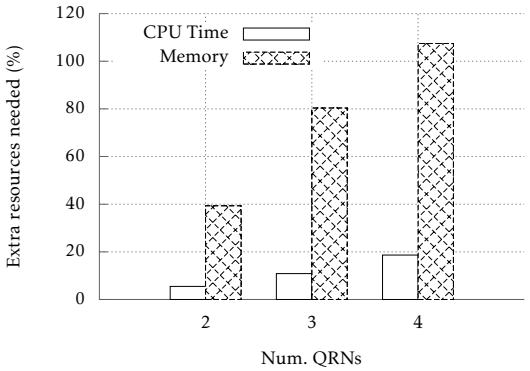

Figure 8. Overhead (in terms of extra resources needed) when adding additional QRNs

Table 3. Critical input rates and speed-up with different numbers of QRNs

# of QRNs	critical input rate	speed-up
1	5 samples/s	1
2	9.10 samples/s	1.82
3	12.5 samples/s	2.5
4	16.7 samples/s	3.34

5.2. Quasit Horizontal Scalability

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 7 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 serve more aggressive input rates, with reasonably limited overhead. Table 3 shows how the *critical sample-rate* (i.e. the data rate at which the system starts to be overloaded and accumulate samples at the operator queues) varies by adding additional QRNs. Clearly, the speed-up values do not grow with a perfect linear trend with the number of QRNs, because of the overhead due to management and network communication, but still the performance degradation is very limited. 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.

In order to estimate the cost of the management overhead when a Quasit deployment is scaled horizontally, we focused on the scenario with the lowest input rate (i.e. 2 samples/s). Note that, in this scenario, a single QRN has enough resources to keep up with the data input rate: by measuring the amount of extra resources needed to process the same data stream in deployments with 2, 3 and 4 QRNs we can effectively estimate the management overhead cause by the additional running processing nodes. Figure 8 shows the percentage of extra CPU Time and memory needed to process the same stream of 500 image samples, when the Quasit deployment is over-provisioned with additional QRNs. The amount of CPU Time required, which in the 1 QRN case amounts in average to 201.81 seconds, increases up to 239.40 seconds in the case of 4 QRNs (less than 20% more). The increase in the amount of memory consumed, instead, is remarkably more significant: if 105.86 MB are consumed on average on a 1 QRN deployment, the 4 QRNs one consumes on average, in the same scenario, about 223 MB, i.e. more than double the resources. This is not really surprising, since the extra nodes running the additional QRNs need to instantiate their own Java Virtual Machines, which in turns load all the classes and instantiate all the objects needed for the management of the QRN itself.

5.3. Preliminary Performance Comparison with Apache S4

Finally, we report here a set of results that compare the performance of our system with Apache S4. The Apache

S4 project[8], initially developed and maintained by Yahoo! [26], is probably the research effort closest to our Quasit proposal. As Quasit, S4 lets users freely define stream analysis graphs and PEs; in addition, inspired by MapReduce, S4 permits to partition streams according to user defined *keys*. The platform instantiates PEs based on the graph layout and on the keys dynamically found in the data, guaranteeing that, within a stream, samples with the same key are always processed by a unique PE instance. According to the project Web site, S4 has been used in several production systems at Yahoo! before being released to the public under open-source license in October 2010; by the end of 2011 it was accepted under the Apache Incubator project umbrella. In the experiments presented here, we used the 0.6 release, code-named *piper*, which we pulled from the project's `git` repository.

Through the S4 API we modeled the same pipelined OpenCV image processing scenario we implemented in Quasit, and executed it on our testbed. Unfortunately, we were not able to control the placement algorithm used by S4 to deploy operators on different nodes, and we had to adopt the default algorithm, which assigns PE instances to nodes according to a hash function applied to stream *keys*; in our pipeline scenario, where the key of each edge connecting consecutive processing steps is constant, this means that S4 will create one PE instance for each processing step. Note that, being the placement of these instances based on the result of a hashing function, it will be, in general, totally unaware of the graph communication characteristics. For this reason, to avoid an unfair comparison, for the following set of results we configured Quasit to use a *random* placement algorithm, which is equivalently unaware of any graph characteristic. As in the previous group of experiments, in each run, we feed the pipeline application deployed on S4 with 500 samples and measure the sample processing time. Again, before starting the measurements, we perform an initial warm-up by generating a preliminary low-rate input sequence. All the reported results are average values over 10 runs for each configuration.

In Figure 9 we show the results for the cluster configuration with four QRNs/S4 nodes, and in Figure 10 we summarize the variation in the average sample RTT for all the tested deployment configurations. It can be observed that Quasit outperforms S4 for what concerns the average sample processing time, thus showing that our prototype exhibits a very limited overhead. Moreover, the difference between the two system is largely more marked in the deployment with just one processing node. It is likely that this is the consequence of

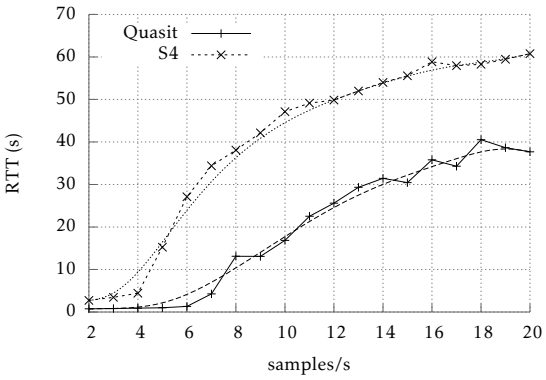

Figure 9. Quasit vs. S4: average sample processing time for increasing data rates with 4 processing nodes and random placement (Quasit) or hash-based placement (S4).

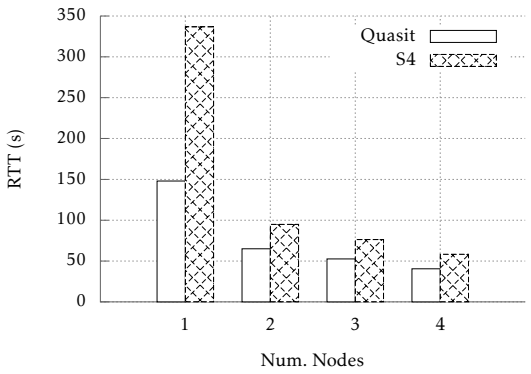

Figure 10. Quasit vs. S4: average sample processing time with 1, 2, 3, or 4 processing nodes and random placement (Quasit) or hash-based placement (S4). The input sample rate is 20 samples/s.

the different threading architecture of the two systems: while Quasit leverages a pool of threads whose size is proportional to the available CPU cores (two on the machines in our testbed) and independent from the number of locally deployed operators, S4 creates a new thread for each data stream in the application graph (in our scenario this corresponds to one thread per local PE instance, plus one on the node where the sink is deployed). This causes higher contention for the available processing resources and greater thread scheduling overhead, in the common cases where the number of "active" components on a single host (operators for Quasit, streams for S4) is significantly bigger than the number of available processing units. Our DDS-based networking solution should also give us some advantage, in terms of serialization space efficiency (DDS serialization format is based on the OMG CDR standard, while S4 uses a custom solution based

[8]Apache S4 project Web site, `http://incubator.apache.org/s4/`. Last visited in August 2013.

on the Kyro[9] serialization framework), but also, and most importantly, in scenarios presenting several *one-to-many* communication patterns: in these situations, the implementation of operator channels over IP multicast could provide significantly reduce network overhead compared to the TCP based solution used by S4 (internally based on the JBoss Netty framework[10]). We are planning extended comparative analysis focusing on the usage of the network in different scenarios to validate the above claims.

As a final remark, it is important to consider that, while in this scenario we used very simple placement strategies (*uniform* and *random*) due to the simplicity of the pipelined processing scenario, in a more general scenario Quasit could effectively exploit additional application-level knowledge, provided in the form of QoS specification attached to part of user graphs, for example by using smarter placement strategies, or by dynamically modifying its thread scheduling mechanisms (e.g., enlarging the thread pool size if operators perform many I/O operations).

6. Related Work

In recent years, there has been an increasing trend shifting the processing load of complex application and services inside data-centers [8] thanks to the ease at which cheap storage and computational resources can be reached on the cloud [5], but also thanks to the flourishing of highly parallel, scalable, and fault-tolerant hardware and software architectures and data processing paradigms. The most popular model for processing large datasets inside data-centers is certainly MapReduce [14], which has received a lot of attention thanks to its ease of use and the diffusion of open source implementations, such as Apache Hadoop[11]. In MapReduce, developers have to model their processing problems 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 [17], 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, [4, 19, 25] leverage a *map-reduce-merge* strategy (originally proposed by [28]) 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 [22] permits to define MapReduce operators through an extension of the SPADE language [16], and to use these operators within an IBM InfoSphere Streams processing graph; DEDUCE jobs can run on either static datasets or, as in the previously cited approaches, sliding windows over streaming data. In [13], 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 Stanford Stream Data Manager [7], and the Aurora [12] projects are two early examples of data stream management systems; coming from the databases community, they introduced the concept of *continuous queries* over data streams by defining specific query languages and algebras [1]. The two systems have a centralized architectures that limit their ability to scale to large data-streams. The Borealis Stream Processing Engine [2, 3] extends its predecessor Aurora, and it leverages the resources of a set of distributed nodes to handle user-defined *query diagrams*, in which a limited set of pre-defined relational-like operators are arranged. 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 [21] 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

[9] Kyro project Web site, http://code.google.com/p/kryo/. Last visited in August 2013.
[10] Netty project Web Site https://netty.io/. Last visited in August 2013.
[11] http://hadoop.apache.org, last accessed in August 2013.

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 [6], the core of IBM Infosphere Streams [16], and S4 [26], a project initially developed by Yahoo! and now maintained by the Apache Foundation[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.

7. Conclusive Remarks and Future Work

In this paper we have introduced Quasit, both a programming model and a framework prototype for scalable, reliable, and quality-aware stream processing. The design of Quasit was guided by the need of having a robust and cost-effective data processing layer, capable of well fitting large-scale deployment scenarios where the awareness of differentiated quality requirements could be exploited to take proper decisions about the most suitable dynamic trade-off among latency, completeness, precision, and accuracy. These characteristics make Quasit especially suited to crowdsensing application scenarios, with i) large variability and unpredictability of input load (with possible frequent peaks, also including very redundant information) and ii) variable quality of the data to analyze and of the contribution that these data can bring to different application goals.

Our first prototype of the Quasit runtime, although still 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), is showing the feasibility of the approach, and is encouraging our further development efforts. In particular, 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. In this context, we plan to extensively evaluate the effectiveness of our distributed and hierarchical QoS management architecture, especially when the scale of applications and data grow.

Acknowledgments. Special thanks go to Professor Cristian Borcea, NJIT. The fruitful and enjoyable discussions on McSense we had together have strongly inspired this paper and the application of Quasit to smart city crowdsensing applications.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik: Aurora: a new model and architecture for data stream management. The VLDB Journal The International Journal on Very Large Data Bases, vol. 12, no. 2, pp. 120–139 (2003)

[2] 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: 2nd Biennial Conference on Innovative Data Systems Research (CIDR), pp. 277–289, VLDB Endowment (2005)

[3] Y. Ahmad, N. Tatbul, W. Xing, Y. Xing, S. Zdonik, B. Berg, U. Cetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, and A. Rasin: Distributed operation in the Borealis stream processing engine. In: 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05), pp. 882–884, ACM New York, NY, USA (2005)

[4] D. Alves, P. Bizarro, and P. Marques: Flood: elastic streaming Map-Reduce. In: 4th ACM International Conference on Distributed Event-Based Systems (DEBS '10), pp. 113–114, ACM New York, NY, USA (2010)

[5] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Kowinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and others: A view of cloud computing. Commun. ACM, vol. 53, no. 4, pp. 50–58 (2010)

[6] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, Y. Park, and C. Venkatramani: SPC: A distributed, scalable platform for data mining. In: R. Grossman and S. Connelly (eds.) 4th international workshop on Data Mining Standards, Services and Platforms (DM-SS), pp. 27–37, ACM New York, NY, USA (2006)

[7] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, and J. Widom: STREAM : The Stanford Data Stream Management System, Technical report, Stanford InfoLab (2004)

[8] L. Barroso, J. Dean, and U. Holzle: Web search for a planet: the Google cluster architecture. IEEE Micro, vol. 23, no. 2, pp. 22–28 (2003)

[9] P. Bellavista, A. Corradi, and A. Reale: The QUASIT Model and Framework for Scalable Data Stream Processing with Quality of Service. In: 5th International Conference on Mobile Wireless Middleware, Operating

Systems, and Applications (MOBILWARE '12), pp. 92–107, Springer Berlin Heidelberg, Berlin, Germany (2012)
[10] P. Bellavista, A. Corradi, and A. Reale: Design and Implementation of a Scalable and QoS-aware Stream Processing Framework: the Quasit Prototype. In: 5th IEEE International Conference on Cyber, Physical and Social Computing (CPSCOM '12), pp. 458–467, IEEE, Besançon, France (2013)
[11] G. Cardone, L. Foschini, C. Borcea, P. Bellavista, A. Corradi, M. Talasila, and R. Curtmola: Fostering ParticipAction in Smart Cities: a Geo-Social CrowdSensing Platform. IEEE Commun. Mag, vol. 51, no. 6 (2013)
[12] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik: Monitoring streams: a new class of data management applications. In: 28th international conference on Very Large Data Bases (VLDB '02), pp. 215–226, VLDB Endowment (2002)
[13] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears: MapReduce Online. In: 7th USENIX conference on Networked systems design and implementation (NSDI '10), USENIX Association, Berkeley, CA, USA (2010)
[14] J. Dean and S. Ghemawat: MapReduce : Simplified Data Processing on Large Clusters. Commun. ACM, vol. 51, no. 1, pp. 107–113 (2008)
[15] EUROCITIES committee. Smart Cities Workshop (2009)
[16] B. Gedik, H. Andrade, K.-l. Wu, P. S. Yu, and M. Doo: SPADE: the System S declarative stream processing engine. In: 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08), pp. 1123–1134, ACM New York, NY, USA (2008)
[17] S. Ghemawat, H. Gobioff, and S.-T. Leung: The Google File System. ACM SIGOPS Operating Systems Rev., vol. 37, no. 5, pp. 29–43 (2003)
[18] R. Guerraoui and A. Schiper: Software-based replication for fault tolerance. Computer, vol. 30, no. 4, pp. 68–74, (1997)
[19] J. Horey: 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, USA (2010)
[20] P. Haller and M. Odersky: Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science, vol. 410, no. 2–3, pp. 202–220 (2009)
[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly: Dryad: distributed data-parallel programs from sequential building blocks. In: 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, vol. 41, no. 3, p. 59–72, ACM New York, NY, USA (2007)
[22] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu: DEDUCE: at the intersection of Map-Reduce and stream processing. In: I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Leger, F. Naumann, A. Ailamaki, and F. Ozcan (eds.) 13th International Conference on Extending Database Technology (EDBT '10), pp. 657–662, ACM New York, NY, USA (2010)
[23] L. Lamport: Time, clocks, and the ordering of events in distributed systems. Commun. ACM, vol. 21, no. 7, pp. 558–565 (1978)
[24] D. Lea: A Java fork/join framework. In: ACM 2000 conference on Java Grande (JAVA '00), pp. 36–43, ACM New York, NY, USA (2000)
[25] D. Logothetis and K. Yocum: Ad-hoc data processing in the cloud. In: Proceedings of the VLDB Endowment, vol. 1, no. 2, pp. 1472–1475, VLDB Endowment (2008)
[26] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari: S4: Distributed Stream Computing Platform. In: 2010 IEEE International Conference on Data Mining Workshops (ICDMW '10), pp. 170–177, IEEE Los Alamitos, USA (2010)
[27] Object Management Group: Data Distribution Service for Real-time Systems, version 1.2. Technical report, Object Management Group (2007)
[28] H.-c. Yang, A. Dasdan, R. Hsiao, and D. Parker: 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, NY, USA (2007)
[29] D. Yang, G. Xue, X. Fang, and J. Tang: Crowdsourcing to smartphones: incentive mechanism design for mobile phone sensing. In: 2012 International Conference on Mobile Computing and Networking (Mobicom '12), pp. 173–184, ACM New York, NY, USA(2012)