Quality-of-Service in Data Center Stream Processing for Smart City Applications

Paolo Bellavista, Antonio Corradi and Andrea Reale

1 Introduction

The wide diffusion of cheap, small, and portable sensors integrated in an unprecedented large variety of devices—from smartphones to household appliances, from cars to fixed monitoring stations—, and the availability of almost ubiquitous Internet connectivity through Wi-Fi hotspots or cellular networks, makes it possible to collect and use valuable real-time information about many fundamental aspects of the environment we live in. If properly understood and used, this information has the potential to bring important improvements to cross-concerning areas that have strong and direct impact on the quality of people's life, such as healthcare, urban mobility, public decision making, and energy management. This continuous collection and exploitation of real-time data from people and objects of the real world is at the foundations of the *Smart City* vision [26], where people, places, environment, and administrations become closer and get connected through novel ICT services and networks. In the last years, several projects from academia, industries and governments have started to work toward the actual implementation of this vision in big urban areas. Examples of these initiatives are the many European funded projects such as European Digital Cities [21], Smart Cities Stakeholder Platform [42], SafeCity [41], or EUROCITIES [23], or industry-led activities, such as the IBM Smarter Cities project [29], or the Intel Collaborative Research Institute for Sustainable Connected Cities [30].

In order to implement novel and useful smart services for the city, it is not only sufficient to collect the raw content of these *Big Data Streams*, but is also crucial to distill interesting and usable knowledge from them. However, the unprecedented

P. Bellavista (✉) · A. Corradi · A. Reale
Department of Computer Science and Engineering (DISI), Università di Bologna, Italy
e-mail: paolo.bellavista@unibo.it

A. Corradi
e-mail: antonio.corradi@unibo.it

A. Reale
e-mail: andrea.reale@unibo.it

© Springer Science+Business Media New York 2015

S. U. Khan, A. Y. Zomaya (eds.), *Handbook on Data Centers*,
DOI 10.1007/978-1-4939-2092-1_35

heterogeneity in data *representation* and *semantics*, and in the *goals* and *quality requirements* of analysis tasks is a hard technical challenge to face while developing applications that deal with huge and continuous streams of information. Distributed Stream Processing Systems (DSPSs) represent very relevant technological support frameworks for the industrial and cost-effective implementation of Smart City applications: for instance, by efficiently leveraging the distributed resources available in data centers with limited impact on the complexity of the application logic, they answer the requirements of *performance* and *scalability* that continuous data streams analysis impose.

In this chapter we analyze the state-of-the-art of DSPSs, with a strong focus on the characteristics that make them more or less suitable to serve the novel processing needs of Smart City scenarios. In particular, we concentrate on the ability to offer differentiated *Quality of Service* (QoS). A growing number of Smart City applications, in fact, including those in the security, healthcare, or financial areas, require configurable and predictable behavior. For this reason, a key factor for the success of new and original stream processing supports will be their ability to efficiently meet those needs, while still being able to scale to fast growing workloads.

The chapter is organized as follows: Section 2 introduces the class of big data analysis platforms known as DSPSs. It does so by providing a simple framework for their description and comparison. Section 3 presents three state-of-the-art and widely used DSPSs and compares their specific characteristics by using the framework presented in Sect. 2. Section 4 focuses on the problem of integrating QoS-aware behavior in DSPSs by emphasizing the reasons why they would be especially useful in Smart City scenarios. Our QoS-aware DSPS, called Quasit, is presented in Sect. 5; Quasit has been specifically designed to allow a rich customization of quality-related stream processing parameters, and is able to enforce them at runtime in a scalable and cost-effective way. Finally, in Sect. 6, we look at a special kind of *weak* QoS specifications, which can be flexibly and adaptively enforced by DSPSs: to this purpose, we present LAAR, a technique for adaptive DSPS operator replication, which allows to trade “perfect” fault-tolerance guarantees off for reduced execution cost, while being able to handle variable load conditions and to offer guaranteed lower bounds on the achievable system reliability.

2 Distributed Stream Processing Systems

A stream processing application is a collection of software components whose goal is to process, analyze, or transform streams of information to produce continuous results in the form of output streams. A *Stream Processing System* (SPS) is a middleware that provides support for both the development and the execution of stream processing applications, and is labeled as *distributed* (DSPS) if it deploys them on a set of distributed computing resources, such as, very relevant nowadays, the ones in a data center.

We propose an original representation model for DSPSs that helps analyzing them according to a simple three layer scheme. The layers are complementary, each

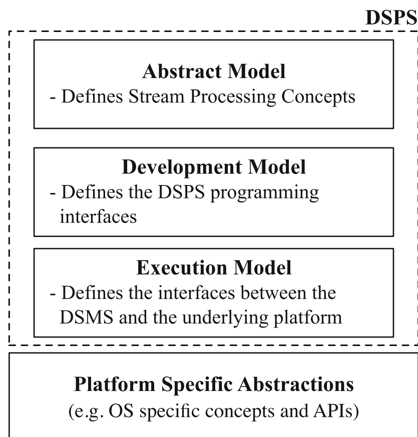


Fig. 1 A three-layer model of Distributed Stream Processing Systems

describing a different aspect of the stream processing system, and are called *abstract model*, *development model*, and *execution model*, respectively (Fig 1).

- *The abstract model* defines the high-level stream processing concepts adopted by the system. For instance, it gives precise definitions of data streams and relevant system events; it determines the characteristics of data processing flows, and the type, role, and granularity of processing components.
- *The development model* defines the set of interfaces that are exposed to developers to build the stream processing components defined in the abstract model. A development model, for example, could map abstract concepts on syntactic constructs of special-purpose stream processing languages, or on ad-hoc Application Programming Interfaces (APIs) and libraries developed for existing general-purpose languages.
- *The execution model* determines how abstract model components are mapped on runtime entities executed by the distributed servers on which the DSPS is deployed. For example, an entire application could be mapped, at execution time, on a single process of the host operating system, or it could be split into several interacting processes.

While the three models may, in theory, largely vary from system to system, in practice, it is easy to identify many recurring aspects among the most common solutions. In the remainder of this section, we discuss the three models and overview how they are commonly realized in existing state-of-the-art solutions.

2.1 Abstract Model

The abstract model of a DSPS defines the high level concepts on which the system is based, including the system-dependent concepts of stream, stream processing application, and the processing workflow that the system adopts. While development and

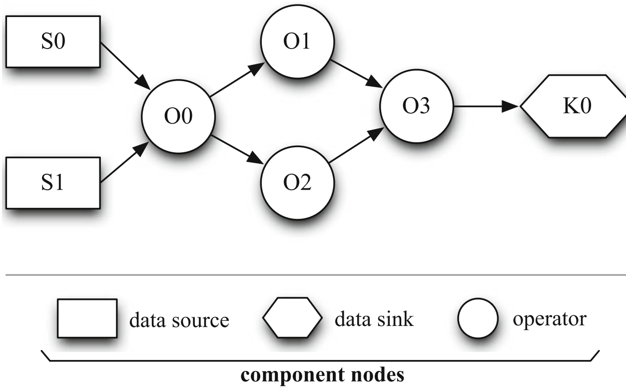


Fig. 2 A generic processing graph in Distributed Stream Processing Systems

execution models usually can be significantly different from one system to another, abstract models tend to be very similar and based on the common abstraction of *processing graph* [2, 4, 6, 10, 16, 24, 35, 44].

A processing graph (Fig. 2) is a Directed and Acyclic Graph (DAG) whose nodes represent data processing and transformation steps, and whose edges represent streams flowing between components. A *stream* is an unbounded sequence of discrete elements, often called *samples* or *tuples*. The *type* of a sample defines its structure, and every stream contains samples all of the same type. Depending on the system, a sample type could be a primitive type—such as an integer or floating point number—or it could be a composite type, similar to a structure in the C language, or, in some cases, to objects of an object oriented type system. Every processing graph is always fed by one or more input streams, and produces one or more output streams as a result. The origin and destination of input and output streams can be highly heterogeneous, such as for example, a file, a network socket, a PUB/SUB endpoint, or a relational database. Since input streams are unbounded, a characterizing feature of stream processing applications is that, once started, they continue to execute forever, unless explicitly stopped.

A graph node can be of three different kinds, i.e., *data source*, *data sink*, and *operator*. A *data source* node identifies a data stream that is conceptually out of the application: 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 application running on the same system. A *data sink* node, conversely, represents the destination of an application output; 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 an application with the input of another one. An *operator* node is associated with one or more input data streams and *generates* one or more output streams. Operators are the core of stream processing applications: they define the set of operations that can be performed on streams. Operators can implement, for example, relational manipulations of single or moving windows of samples, such as projections or joins;

they can perform aggregation or filtering actions, or realize more complex, arbitrary, algorithms. Operators, data sources, and data sinks are collectively called *graph components* or, more simply, components.

Samples are received and produced by stream components on their *input and output ports*, each having its own type, which corresponds to the type of the stream it receives or produces. Every graph component performs its processing operations on data samples according to an asynchronous processing model; conceptually all the components operate in parallel and perform their processing actions as soon as data samples are available at their input ports.

2.2 Development Model

A development model maps the concepts defined in the abstract model on the programming-level constructs offered to developers to write their stream processing applications. These constructs should allow to:

1. Define new applications, by describing which source, operator, and sink components should be instantiated and how they should be connected into a processing graph.
2. Customize component instances, in order to adapt their behavior to specific application needs (e.g., to bind graph source nodes to actual external sources).
3. Develop new components with custom functionalities.

Any DSPS development model should at least define the tools to achieve the first two goals of the list; in fact, in many cases, it is not necessary to create new or custom components, especially in the common scenario where the DSPS comes bundled with collections of ready-to-use components (also referred to as *toolkits*) that can satisfy most common application requirements.

In the available literature, two families of application development models are common. The first includes models whose mappings are based on *special-purpose languages*; the second relates to the exploitation of *general purpose languages* for that.

Special-purpose stream processing languages are usually tightly bound to the system they have been designed for. They normally allow a very concise definition of applications and components, by having stream processing concepts mapped one-to-one to language-level concepts. For example, the Stanford STREAM system [6] defines the so called *Continuous Query Language* (CQL), which permits to develop stream processing applications by writing continuous queries in a syntax that strongly resembles SQL queries. These queries are processed by the underlying system and decomposed in a processing graph of pre-defined operators. If ad-hoc languages permit faster and easier application development, they generally lack the flexibility of general-purpose languages and, more importantly, they force developers to learn new languages and new development processes.

Development models based on *general-purpose* languages, instead, usually have a less steep learning curve, as system-specific stream processing concepts are expressed using familiar constructs offered by common general-purpose programming languages, such as C++, Java, or Python. For example, in Apache S4 [35], operators are defined by writing corresponding Java classes, all subclasses of a common abstract superclass. The developer has to "fill-in" the methods that implement the operator logic, which the system automatically invokes when corresponding events of interest occur. Using general-purpose languages has several benefits, including the possibility to seamlessly reuse existing libraries and software modules in new stream processing applications. However, this usually comes at the expense of conciseness and prototyping speed because those APIs can be verbose and sometimes cumbersome.

2.3 Execution Model

An execution model maps the elements defined in the abstract model and described through the development model onto runtime objects that the hosting platform is able to run directly. An execution model defines:

1. The characteristics of platform specific execution units, and the high-level policies adopted for scheduling local resources, such as CPU and memory.
2. The distribution of the execution units on the cluster of available servers.
3. The mapping of graph edges on communication channels, such as shared memory, pipes, or network sockets.

The first important aspect of an execution model is the mapping of operators, sources, and sinks on host platform concepts, such as processes or threads. With a *process-per-operator* allocation, each operator is individually instantiated as a separate process, with one or possibly more concurrent threads of execution (e.g., one per input port). This grants the maximum isolation, since problems with one component cannot affect other concurrently running ones. With this choice, the local scheduling of resources is demanded to the standard facilities of the host operating system CPU and memory schedulers. The first implementations of the Stream Processing Core (SPC) [4] (at the basis of the more recent IBM InfoSphere Streams system— see the following) used a similar approach, by isolating single components into their own containers, corresponding to standard UNIX processes.

A *process-per-server* allocation creates just one process per server. Within this process, components are hosted as separate software modules, for example as instances of a given class in case of a class-based object oriented implementation. While, on the one side, this arrangement does not grant the same execution isolation that a process-per-operator allocation does, on the other side, it permits a tighter control on resource scheduling policies. For example, every in-process component could be given a dedicated execution thread or, more interestingly, a pool of threads could be used to execute groups of components according to internal policies or QoS

requirements (e.g., for a priority-proportional scheduling of resources). Moreover, the communication of components running within the same process is usually faster and cheaper, as shared memory based channels can be used. The process-per-server allocation is used, for example, by Apache S4 [35] and Quasit [10], which start a Java virtual machine on each cluster server, and deploy sources, operators, and sinks as objects running within the local VM.

Somewhere in the middle between the two previous solutions, the *cluster-of-operators* approach *fuses* subsets of tightly coupled components (e.g., operators with strong reciprocal communication dependencies) into one process. Again, within every single process, very flexible resource scheduling approaches and faster communication channels can be used. Different operator clusters, however, are still mapped onto different processes, granting a better isolation to each group. IBM InfoSphere Streams [24] uses a similar hybrid approach thanks to a technique, called *operator fusion* [33], that groups multiple operators into single execution units automatically.

Knowing what the execution units are, the application processing graph can be rewritten in the corresponding *runtime graph* where nodes represent individual runtime objects (e.g., processes) and edges represent inter-process communication channels. A further role of the execution model is the definition of an *assignment strategy* for runtime objects. An assignment strategy decides the distribution of runtime objects on the available cluster servers: a good solution should take into account the resource requirements of every object (e.g., CPU and memory), the resources availability of each server, and the expected/declared application communication patterns, and it should find an assignment that satisfies the application resource and quality requirements while minimizing its execution cost. The assignment can be static-only, or can have dynamic phases as well. During the static phase an initial assignment is decided based on a-priori knowledge of the application and input streams characteristics. Due to changing load conditions, for example caused by load spikes in some input streams, the initial assignment could be no longer adequate to satisfy the application QoS requirements; in these cases, a dynamic phase can be performed at runtime to adaptively deal with load variations. For example, [39] and [38] propose two-phase algorithms to perform both static and dynamic assignment phases, while [40] tries to find an initial static assignment that maximizes the system robustness to possible variations.

Finally, an execution model should decide how communication channels are implemented at runtime. For in-process communication, *function calls* or *shared memory-based message passing* are the most commonly chosen alternatives. While the first binds the execution thread of the caller to that of the callee, the second allows an independent execution of the two components. For what concerns inter-process communication, the choice depends on whether the channel endpoints reside on the same host or on remote hosts. In the first case, solutions such as system-level shared memory or pipes can be adopted to implement faster and cheaper communication solutions; in case of remote communication, the choice of the protocol depends very much on the desired communication cost and QoS level. For example, if cheap and unreliable communication is enough, UDP-based channels are a possible solution.

3 Platforms for Distributed Stream Processing

In the last decade, the problem of effectively processing continuous information flows has been faced in several projects. In the early 2000s, systems like TelagraphCQ [8, 17], Aurora [1, 16], Borealis [2], and Stream [6, 7] have started recognizing the ineffectiveness of using traditional database management systems (DBMSs) for the real-time analysis of continuous data, and have proposed their own alternative solutions. More recently, as a result of the industrial success of scalable and parallel batch data processing systems like MapReduce [20], solutions such as Map-Reduce-Merge [46] or MapReduce Online [18] have tried to reuse its successful scalable processing model in stream processing scenarios, by enhancing it with continuous and dynamic data analysis capabilities. In this section, we have selected three prominent state-of-the-art DSPSs, and we discuss their design and architectural features under the light of the three-layers modeling framework introduced in the previous section. The three systems described in the following are IBM InfoSphere Streams, Apache S4, and Storm. The particular choice of these systems over the different alternatives available in the literature is motivated by the fact that, to our knowledge, the selected DSPSs are the most widely adopted in real-world large scale production systems, including large data center deployments from important industry players such as IBM, Yahoo! and Twitter: for this reason, we believe that their analysis can provide important insights about the common requirements of real stream processing workloads, including those of Smart City scenarios. It is not the goal of this work to provide an extensive survey of existing stream processing solutions: for a comprehensive work, the interested reader is referred to [19].

In the following three subsections, we briefly overview each selected DSPS by analyzing its abstract, development, and execution model; for each of them, we also emphasize peculiar QoS-related features, when supported.

3.1 IBM InfoSphere Streams

IBM InfoSphere Streams [24] is a DSPS evolved from the SPC research project [4]. In Streams, application processing graphs are defined in an ad-hoc special-purpose Stream Processing Language (SPL) that is used to describe *operators* and their stream connections. The language is very flexible, as it permits to define new data types, or to customize the behavior of existing operators by changing, for example, their number of input/output ports or their output logic. Besides defining simple operators, SPL also enables to combine them in composite ones, which encapsulate more complex behavior.

In addition, the system exposes two sets of general-purpose APIs that can be used to build user-defined custom operators. The first is a mixed C++ and Perl API that, by using a two-steps code generation process, gives them maximum customization flexibility and execution efficiency [25]. The second is a simpler Java API, based on runtime reflection techniques rather than code generation. Due to the cost of

reflection, however, this API is in general less efficient than its C++/Perl counterpart. Operators built through either API can be exported to reusable toolkits and used directly within SPL source files.

At compile time, an optional operator fusion process can be manually or automatically started in order to cluster groups of correlated operators. Each group is then transformed into its corresponding runtime object, called Processing Element (PE), whose execution is mapped onto an operating system process. Hence, InfoSphere Streams follows an operator-per-process or cluster-of-operators approach, depending on whether the fusion step is performed or not. Depending on configuration parameters, operators inside the same process are executed by dedicated threads—in this case they communicate to other in-process operators through message passing—or by shared threads—in this case they communicate via function calls. At the time of writing, the only explicit QoS parameter supported by Streams is a loose form of fault-tolerance, based on checkpointing [32]: periodically or driven by events, runtime objects can save their current state on secondary memory; whenever a crash occurs, that state is restored, but all the processing operations performed between the last checkpoint and the failure are lost.

3.2 Apache S4

Apache S4 [5] is a DSPS initially developed and maintained by Yahoo! [35] and currently part of the Apache Incubator project umbrella.

In S4 processing graphs, there is no distinction between sources, sinks, and operators, but all the components are uniformly modeled and called PEs[1]. PEs can import streams coming from other applications running on the same platform, process them, and possibly export output streams either to external destinations or to other applications concurrently running on the platform. External streams of data (i.e., coming from sources external to the platform itself) can be transformed into internal streams by developing and running special S4 applications, called adaptors.

To develop PEs or adaptors, S4 offers its own Java API. Developer create new PE types by writing classes that inherit from the `ProcessingElement` superclass, whose methods are automatically invoked by the framework whenever new samples to process are available or in a time-driven fashion with customizable rate.

The S4 execution model follows a process-per-server approach: S4 instantiates a JVM container on each cluster server and PE instances are executed within these containers. The VM execution threads are not directly associated with PE instances, but with streams: within a VM container, the platform instantiates one thread for each stream feeding a hosted PE, and this thread executes all the methods of the PE instances served by that stream. Very peculiarly, every S4 stream can be optionally

[1] Note that, while in IBM InfoSphere Streams (Sect. 3.1) the concept of PE belongs to the execution model, in S4, it represents an abstract model concept.

keyed. In a *keyed* stream, every data sample has a unique key: the S4 runtime support dynamically creates a new PE instance for each different key in a stream, so that every instance processes all the stream samples for one key in a sort of functional *map* operation. This allows an easy parallelization of PEs and, consequently, an easy scale-up mechanism. S4 permits to choose inter-VM (and hence remote) communication transports among UDP- or TCP-based ones (by default UDP is employed). Similarly to IBM InfoSphere Streams, the only QoS policy supported by S4 is a weak form of fault-tolerance based on periodic or event-driven checkpointing of PE state.

3.3 Storm

Storm [44] is a DSPS developed by BackType and recently released under the Eclipse Public License by Twitter after its acquisition of BackType. As IBM InfoSphere Streams and Apache S4, Storm is based on the processing graph abstract model presented in Sect. 2.1. In the Storm model, data sources are called *spouts* and operators *bolts*; there is no explicit concept of sink, but destinations can be realized through *bolts* themselves since they can perform arbitrary actions on received samples (including saving them on files or forwarding them to external systems).

According to the Storm development model, the main method to define new spouts and bolts is through a Java API. As in Apache S4, in Storm, custom bolts and spouts are defined by writing classes that extend specific base classes, which, in turn, provide basic functionalities to newly built components. Graph instances are defined by creating instances of component classes and by defining their connection edges via specific API calls.

The Storm execution model is rather articulated. There are three parameters that influence how a particular graph is instantiated on the hosting platform, i.e, (i) the number of worker processes, (ii) the number of per-component tasks, and (iii) the number of per-component threads. The first parameter determines the total number of processes instantiated in the Storm cluster; the second, associated with every spout or bolt, determines the number of instances per component (also called *tasks* in Storm terminology) instantiated across all the cluster; the third determines the total number of threads dedicated to serve a component's set of tasks. At runtime, every worker is instantiated in a different Java VM, which can host one or more tasks (and execute one or more threads) from the same application. When multiple tasks for a single component are running, the routing of stream samples to different component instances is based on a further parameter, configurable at development time, which determines a grouping policy: for example, samples can be randomly shuffled among tasks for load balancing purposes, or can be routed using a modulo hashing of some sample fields.

Very peculiarly, Storm puts a strong focus on fault-tolerance by optionally providing at-least-once processing semantics: this means that it guarantees that every sample produced by any of the graph spouts is processed at least once. To do so, for each *root sample* (i.e., a sample generated by a spout), Storm keeps track of all the

samples whose creation has been *caused by* its processing, and buffers it until all the tracked samples are acknowledged by their final destinations. Given the highly customizable nature of stream processing functionalities, Storm is not able to automatically keep track of *caused by* relationships between samples, but it requires explicit developer intervention for that: at code-level, in fact, Storm developers have to explicitly mark every new sample as caused by another sample if willing to avail of Storm fault-tolerance facilities.

4 QoS-Aware Stream Processing

In every kind of IT infrastructure serving mission-critical application scenarios, such as healthcare, finance, or transportation, it is very important that services behave in conformance to a well-defined *Service Level Agreement* (SLA) that determines the required QoS level. An SLA normally puts constraints on the functional behavior of the service (e.g., it should produce *all and correct* results in normal conditions) but also, and more importantly, constraints on how the service is expected to behave according to a set of performance indicators (non-functional requirements). The range of possible performance indicators is, in general, very large and application-dependent: two common and simple examples are *latency*—measuring the maximum time interval between a service request and the corresponding response—or *availability*—measuring the fraction of time the service generates correct results, even in spite of possible failures. Other indicators can refer to lower-abstraction details of the service, by measuring, for example, platform-specific parameters such as *memory* or *CPU* usage. Every constraint in an SLA that binds a specific performance indicator to some value is said to represent a *QoS policy* for the service.

In general, the implementation of *QoS-aware* services, i.e., services that are guaranteed to deterministically operate according to a set of associated QoS policies, is a very difficult task, and maps to the ability of the runtime platform to allocate (both statically and dynamically) the proper amount of computational resources where they are needed to satisfy the specified quality requirements. The technical challenge is even harder in the case of stream processing applications. In fact, differently from simple request-response or batch-oriented processing scenarios, where characteristics of computational tasks are known a-priori and thus easier to reason about, in stream processing, the properties of input streams (e.g., their data rate) change continuously and their behavior is not completely known in advance and difficult to predict. The consequent high variability in the load that applications have to sustain during long provisioning times, makes it very challenging to implement effective and adaptive resource scheduling techniques.

Nonetheless there is a growing number of real-world applications that has to deal with the analysis of large data streams and that requires, at the same time, predictable performance guarantees. This is often the case in Smart City scenarios, where a common goal is to use the results of stream analysis to trigger real-time feedback actions on real-world aspects of the city itself and of the urban life. These actions

can be responses to emergency conditions, such as the activation of alarms in smart telecare systems [45], or the computation of emergency rescue plans in a smart traffic management system [22], which must be performed in a timely and reliable fashion. To better emphasize the importance of properly handling application-specific QoS requirements, let us briefly expand on this second scenario.

Consider a Traffic Management System (TMS) deployed in a Smart City. In this system every car periodically reports its position and speed to ad-hoc collection points using vehicle-to-vehicle and vehicle-to-infrastructure communications [34]. In their turn, each collection point relays these data to the data center-hosted stream processing application that processes these data in order to realize the TMS services. The TMS generally has the following three high-level functions:

- *Traffic flow control*. By analyzing short term and long term variations in car speeds along different roads, the system adapts the traffic lights timings to current road network conditions.
- *Management of road emergencies*. In case of car accidents, the vehicles involved and other cars passing immediately route messages about the event to on-road collection points, which, in their turn, relay them to the data center application. By analyzing these messages, the TMS detects the emergency condition, notifies the appropriate emergency service (e.g., ambulances), and tries to adapt the traffic flow to the new conditions, for example, by suggesting alternative navigation paths to other drivers (see next point).
- *Real-time navigation*. Cars traveling in the city can query the TMS for advanced navigation services. The system will answer with an always up-to-date route that takes into account road load conditions and possible emergency situations.

The three tasks of the TMS service, although based on the same input data streams, have very different quality requirements. For example, the traffic light timers must be promptly and quickly adapted to new road load conditions, meaning that the related processing actions should be performed with bounded *latency*. Similarly, processing of emergency notifications should be performed within deterministic time limits, in order to allow immediate rescue actions to take place. For the same reason, the management of all the emergency situations must take *priority* over other computations; this is especially useful during periods of high computational load (e.g., during traffic peaks) when the available DSPS resources might not be enough to satisfy all the processing flows. Accident notification messages should be transferred and processed *reliably* becuase the consequences of information loss can be very severe. On the other hand, the analysis of vehicles' position and speed to determine road load conditions can be performed *best-effort*: the related processing tasks can be executed with lower priority; in addition, data loss can be largely tolerated in this case given the implicit spatial and temporal information redundancy present in the corresponding data streams.

This simple but, we believe, very representative example shows how important can be for DSPSs to provide a rich and native support for *QoS-aware* stream processing. By using this type of support, developers of Smart City applications could focus their attention on application-level modeling and implementation problems, while

delegating the realization of complex QoS enforcing mechanisms to the underlying DSPS. However, to the best of our knowledge, the most widely used and state-of-the-art DSPSs have only limited QoS-based configuration capabilities, which in most cases include only the selection of various reliability mechanisms (see Sect. 3).

On the contrary, we claim that QoS should be introduced as a first class concept in DSPSs at all the three abstract, development, and execution layers of our model. *QoS in the abstract model* should permit to specify, with different levels of granularity, the QoS policies required for graphs, single components, or groups of components directly in the application models. At this layer, different DSPSs should define their own quality-related vocabulary and determine which are the performance aspects controllable through their QoS policies, to which specific components they can apply, and how they interact with each other. *QoS in the development model* should define the programming constructs (either as extensions of ad-hoc stream processing languages or as specific APIs for general-purpose languages) that can be used to annotate application code with the quality requirements expressed at the model level. Finally, *QoS in the execution model* should support the execution of applications specified according to the other two layers. In this layer, each different DSPS should map different QoS policies to different mechanisms for runtime admission, monitoring, enforcement, and management, and should develop proper resource scheduling algorithms to satisfy the required QoS specifications.

In the following section, as a practical example of this kind of approach, we introduce Quasit, an original DSPS designed and implemented by following the above QoS-related guidelines.

5 Quasit

Quasit [10, 11] is a distributed stream processing system whose main design goal is to support QoS-aware stream analysis. To do so, it incorporates the concept of QoS at all the abstract, development, and execution model layers. Quasit is designed to run on large data centers made of commodity hardware, exploiting all the available processing power, and automatically handling various types of failures.

The Quasit abstract model is based on the processing graph concepts presented in Sect. 2.1. Originally, every element in Quasit processing graphs, called *streaming information graphs* (SIGs) in Quasit terminology, can be augmented with a QoS specification (a collections of QoS policies applied to that element); collectively, QoS specifications are used to dynamically adapt to variable load conditions and to the quality requirements of different parts of the stream processing flow. The Quasit development model is based on a simple Scala API, which lets developer (i) write, compose, and reuse custom operators, sources, and sinks, (ii) arrange components in SIGs to deploy on the infrastructure, and (iii) define the required QoS configurations for components, channels, or graphs as a whole. The API is designed to support a functional-like programming style that clearly separates operator behavior and state, thus making it easier for the runtime to support advanced QoS provisioning

strategies. The Quasit runtime model maps the SIG components to runtime objects that run on all the available data center resources, and implements the set of QoS mechanisms that make it possible to execute application SIGs while enforcing their QoS requirements.

In the following subsections, we will concentrate on the three model-levels and overview the main ideas behind Quasit QoS-aware stream processing.

5.1 Quasit Abstract Model

The basic modeling unit in Quasit is the *Streaming Information Graph* (SIG), a weakly connected acyclic and directed graph representing the transformations that, applied to one or more input streams, produce an output data stream. Similarly to the model described in Sect. 2.1, three kinds of nodes can be used in a SIG graph i.e., *operators*, *data sources* and *data sinks*.

SIG nodes and edges can be labeled with QoS specifications, which define non-functional configuration parameters or constraints. Depending on the type of node or edge, a QoS specification can consist of several *QoS policies*, each policy influencing a different quality aspect. For example, through QoS specifications on SIG edges, it is possible to control the characteristics of the protocol used to exchange data among nodes they connect, or, through QoS Specifications on operator nodes, it is possible to configure their reliability guarantees.

The processing core of the Quasit abstract model is the *simple operator* component, whose structure is shown in Fig. 3.

A simple operator can be *stateless* or *stateful*. When stateful, the operator processing behavior is defined by the combination of the value of its *state* and its *processing function*; when stateless, by the processing function alone. The processing function is executed asynchronously whenever a sample from any of the operator input ports is available and its result may depend on the current value of the operator state. The role of the processing function is to describe how input streams are combined to produce an operator's output stream, and, if necessary, to update the operator internal processing state.

Quasit also allows to combine operators into more complex ones, by defining *composite operators*: existing operators (either simple or composite) can be arranged in a special SIG type, called *operator definition* SIG (OD-SIG), whose source and sink nodes are *virtual*, i.e., they do not correspond to real streaming data producers/consumers. Quasit defines a mapping between this kind of SIG and the associated composite operators: each data source in the OD-SIG defines a typed input port of the composite operator, and the data sink in the graph determines the type of the operator output port. Without digging into formal details, the behavior of a composite operator is defined by the internal structure of its defining OD-SIG: when a sample arrives to an input port of the composite operator it is processed as if it was processed by the OD-SIG operators graph. This composition mechanism provides

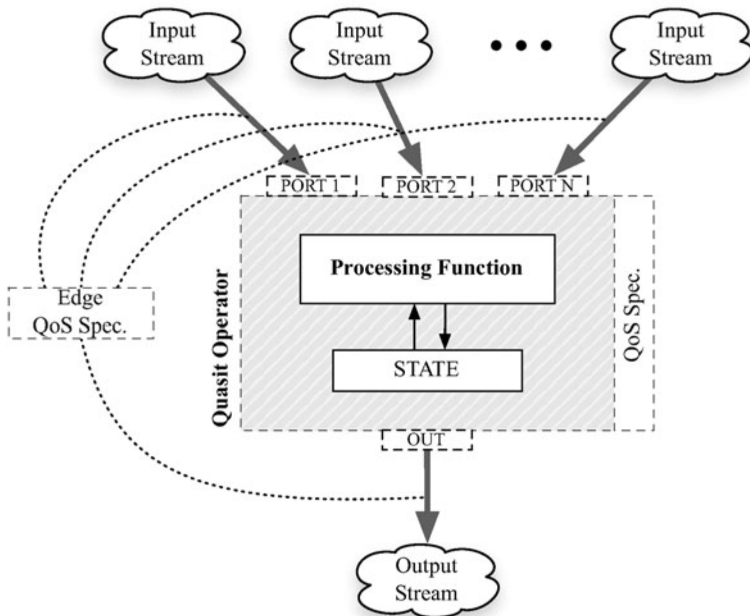


Fig. 3 Structure of a Quasit simple operator

an easy way to create new complex processing functionalities in terms of simpler ones, and promotes sharing and reusing existing and well-tested components.

5.2 *Quasit Development Model*

Quasit offers a very simple Scala API to let developers write their stream processing applications, create new data sources, data sinks and operators (either simple or composite), arrange components in SIGs, and associate QoS specifications to SIG elements.

In order to define new components, developers write *descriptor* classes that contain all the information that the framework needs to instantiate component instances at runtime. Depending on the type of component (i.e., source, sink, simple or composite operator), the descriptor class must extend an appropriate superclass which acts as a sort of "template" for the new descriptor. For example, operator descriptor classes must inherit either from `StatefulOperatorDescriptor[O,S]` or `StatelessOperatorDescriptor[O]`, depending on whether the operator needs to maintain some state between subsequent processing operations or not. Any component descriptor class must implement the appropriate life-cycle methods that are asynchronously invoked when relevant events occur. For example, an operator

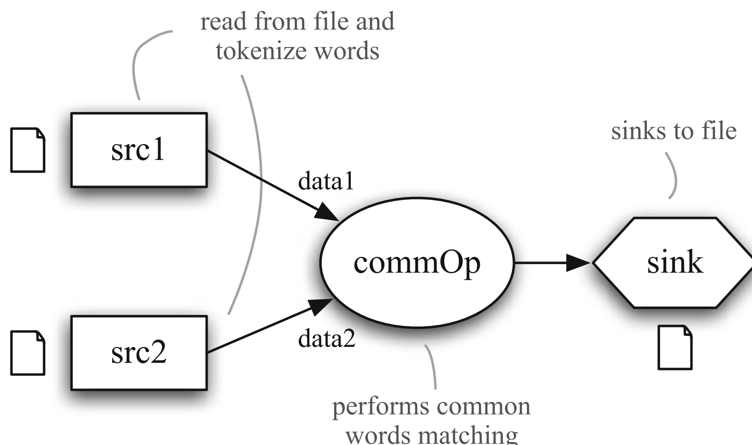


Fig. 4 A Quasit SIG implementing a simple, comm-like application

descriptor class must implement the *processingFunction* method, which realizes the main operator processing logic. This method, in fact, is called whenever samples are available at any operator input port. If the operator is stateless, the optional return value of the processing function is a list of samples to produce in the operator output stream; otherwise, it is a pair of objects, the first being a list of output samples, the second (optional) the new state the operator should transition to. A `SIGDescriptor` describes how components are arranged in the processing graph. It lists instances of component descriptors and the edges that connect them. While instantiating component descriptors, edges, or SIG descriptors, it is possible to associate to each of these elements specific `QosSpecification` objects, which will be used by the Quasit runtime to enforce the required quality levels.

Let us see, with a brief concrete example, how it is possible to create component descriptors and arrange them in SIGs. To keep the discussion self-contained we will consider a very simple scenario, where two sources continuously read lines each from a different file, tokenize them into words and send them to an operator (henceforth referred to as the comm operator) that determines and outputs words found in both the input files; this application can be thought as a sort of distributed and scalable implementation of the UNIX `comm` utility. A representation of the corresponding SIG is shown in Fig. 4.

Listing 1 shows how the descriptor class for the comm operator is defined. `CommOpDescritpor` inherits from `StatefulOperatorDescriptor [WordMsg,Map[String,Short]]`. The two type parameters passed to the base class (i.e., `WordMsg` and `Map[String,Short]` respectively represent the output type of comm operators and the type of their processing state. In fact, `WordMsg` is a simple container for words, while the state of a comm operator is a map that associates every word ever met with three possible values: one, if the word has been found on the first source only, two, if the word has been found on the second source only, and three

if it has been found on both. Every instance of `CommOpDescriptor` has to specify two parameters, i.e., a symbolic name and an instance of `OperatorQosSpec` that will determine the set of QoS policies associated to the operator instance. Lines 3–7 determine the parameters passed to the `StatefulOperatorDescriptor` constructor; in particular, it is interesting to pay attention to the definition of the two operator input ports and their types (line 5) and to the definition of the operator initial state, i.e., an empty map (line 7). The comm operator processing function is defined from line 10 to line 15: by leveraging the expressiveness of Scala partial functions, it is possible to express the actions to perform in case a sample is received from the first data source (`"data1"` port) or the second one (`"data2"` port) very concisely. The private function `processWord` (line 17, determines the actual behavior of the operator, and the return values of its processing function. Note that, for example, once a word is found in both sources, a sample is produced on the output and the operator state updated accordingly (line 33).

```
1 class CommOpDescriptor( name: String, qos: OperatorQosSpec)
2  extends StatefulOperatorDescriptor[WordMsg,Map[String,Short]](
3   name, qos,
4   // Define the operator ports
5  Map("data1" -> classOf[WordMsg], "data2" -> classOf[WordMsg]),
6    // The initial state of the operator is an empty map
7    Map[String,Short]()) {

8
9
10  override def processingFunction = {
11   case (msg: WordMsg, "data1", state: Map[_,_]) =>
12    processWord(msg.word, 1, state)
13   case (msg: WordMsg, "data2", state: Map[_,_]) =>
14    processWord(msg.word, 2, state)
15 }
16
17  private def processWord(word: String, srcMask: Short,
18   state: Map[String, Short]):
19    (Option[WordMsg], Option[Map[String,Short]]) = {
20
21   val wordState = state.get(word)
22  wordState match {
23   case None =>
24     // There is no entry in the map
25     val newState = state + (word -> srcMask)
26     // Produce no output but a new state
27     (None, Some(newState))
28    case Some(mask) =>
29 if ((mask & srcMask) != 0) { // already seen from this source
```

```
30        (None, None)
31       } else {
32   val newState = state + (word -> 3.toShort) // seen from both
33        (Some( new WordMsg(word)), Some(newState))
34       }
35     }
36   }
37 }
```

Listing 1: Definition of a simple Quasit operator.

Listing 2, instead, shows how to instantiate operator descriptors and how to represent a SIG through a SIG descriptor. First, in lines 3 and 4, the two source descriptors are instantiated, pointing to the input files. After that, in lines 7–11, the comm operator descriptor instance is instantiated as well; notice that it is explicitly given the name `"commOp"` and that it is assigned a *queuing* QoS policy, which determines the type of queues used to buffer the operator input samples. After instantiating a file sink (line 14), the actual `SigDescriptor` instance is created in lines 18–29. The SIG descriptor instance is given a unique name, and the graph components are listed one by one through references to their descriptors. Graph edges connecting components are described through a sequence of triples (line 23), each defining an edge through its source node, its target node, and the QoS specification associated to the corresponding stream channel (default in this case). Finally, the SIG descriptor is associated with a SIG-wise QoS specification (lines 28–29): in this particular case, a *fault-tolerance* related policy, called *internal completeness*, is requested. As we will see in more detail in Sect. 6, this policy trades off perfect resiliency to failures and the ability to handle load spikes, while steel guaranteeing that a given fraction of samples are correctly processed (in the example, 70 % of the samples).

```
1 def  main(args: Array[String]): Unit = {
2  // Instantiate the data source descriptors
3  val src1 = new FileSourceDescriptor("src1", "/pathto/f1")
4  val src2 = new FileSourceDescritpr("src2", "/pathto/f2")
5
6   // Instantiate a CommOp descriptors
7   val comOpA = new CommOpDescriptor("commOp",
8      OperatorQosSpec().withPolicy(
9         QueingPolicy(QueueingPolicy.Unbounded,
10             QueueingPolicyKind.Fifo)),
11    )
12
13  // Instantiate the sink
14  val sink = new FileNativeSink[WordMsg]("sink",
     DataSinkQosSpec(),
15    "/pathto/comm.txt")
16
17   // Define the graph
18  val sig = SigDescriptor(
```

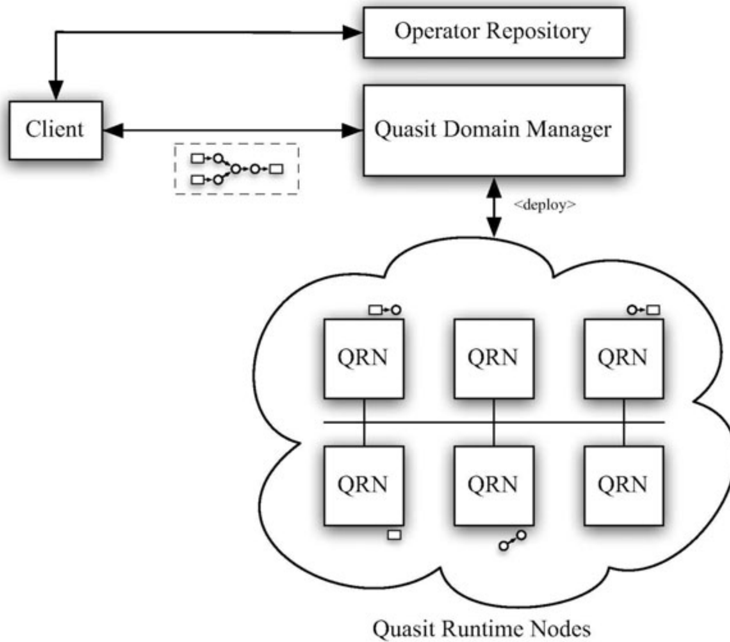


Fig. 5 Distributed architecture of a Quasit domain

```
19      name = "CommSig",
20      dataSources = Seq(src1, src2),
21     dataSink = sink,
22     operators = Seq(comOpA),
23      edges = Seq(
24        ("src1" -> "commOp", "data1", ChannelQosSpec()),
25         ("src2" -> "commOp", "data2", ChannelQosSpec()),
26        ("commOp" -> "sink", ChannelQosSpec()))
27     )
28    ).withSigQosSpec(SigQosSpec()
29      .withPolicy(FTPolicy(FTPolicy.IC, 0.7)))

30
31   ...
32 }
```

Listing 2: Creation of a SIG descriptor instance.

5.3 *Quasit Execution Model*

The Quasit abstract model, combined with the corresponding development model, offers a flexible and intuitive way to express stream analysis needs through the composition of small processing stages, and allows to customize these stages by means of several QoS policies. The Quasit execution model supports the execution of Quasit components at runtime by leveraging the computing power of a cluster of commodity computers within large-scale data centers.

A running Quasit deployment is called *domain*. A domain handles the distributed and QoS-aware execution of one or more user-defined SIGs. Similarly to other scalable data processing architectures (e.g., [20, 24, 31, 44]), the distributed architecture of a Quasit domain follows the *master-workers* pattern, with a central component with management and monitoring responsibilities and several distributed nodes performing the actual data processing operations. Figure 5 shows the three core distributed components running in one Quasit domain:

- Several *Quasit Runtime Nodes* (QRN), the *workers*;
- One *Quasit Domain Manager* (QDM), the *master* node;
- One optional *Quasit Operator Repository* (QOR).

The main QoS-aware execution services of our Quasit framework are provided by the co-operation of the QRN and QDM components. A typical Quasit deployment includes one QDM node and a cluster of QRN nodes, usually interconnected by a high-speed local area network (LAN). A QRN is in charge of providing the execution environment for Quasit *simple* operators and implement *threading*, *networking*, and local *QoS management* services. The QDM has management and control responsibilities over a Quasit domain. It does not take any direct role in stream processing tasks: for this reason, its centralized architecture does not represent a relevant bottleneck to the overall system scalability. Finally, the QOR is a repository of simple and composite operator types, and users can use its services to publish their operator definitions and to search for previously published ones.

In the current Quasit prototype implementation, every cluster server hosts one QRN, which is executed within a Java Virtual machine process (i.e. *process-per-server* model). Within this JVM, operator instances are modeled as distributed actors [3] managed by the Akka Actors framework [27]. All the actors running within the same JVM are managed by a pool of threads of configurable size, which executes operators processing functions at need, i.e., when there are samples to process at their input ports. This threading schema gives tremendous flexibility because it permits easily implementation of custom scheduling strategies, such as, for example, priority-based ones.

Operators deployed on different QRNs are connected via channels realized by leveraging the OMG DDS standard for high-performance PUB/SUB data exchange [36, 37]. Concretely, the PUB/SUB communication module maps the output port of every stream source (either operator or data source) to a unique destination topic and, symmetrically, every input port (of either an operator or a data sink) to a topic

subscription. This solutions provides, at the same time, (i) strong decoupling between data producers and consumers, (ii) reduced space and time overhead thanks to the very efficient serialization mechanisms of the DDS middleware, and— most importantly— (iii) possibility to exploit the very fine-grained control of low-level QoS-related parameters that the DDS standard permits to associate with its topics, readers, and writers.

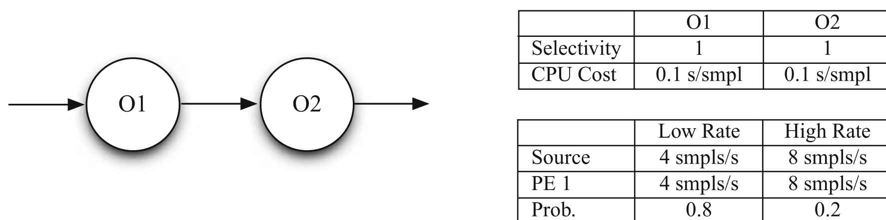
QoS policies defined at model-level on Quasit SIGs are enforced at runtime thanks to a two level QoS-management architecture, realized through the interaction of one *domain QoS manager*, running within the QDM, and several *node QoS managers*, one for each QRN. The domain QoS manager performs global 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.

6 Load-Adaptive Active Replication (LAAR)

We have seen that the nature of stream applications poses several different and hard challenges to platform providers, including the ability to offer, at the same time, performance *elasticity* in spite of load variations, and *resiliency* to failures, while keeping *costs* limited. Handling load fluctuations due to sudden and possibly temporary variations in the data rates of input streams is a very complex task: in general, it maps to the ability to plan and allocate, statically and—more importantly—dynamically, the available computing resources to different parts of the hosted applications.

As stream processing applications usually run for (indefinitely) long time intervals, failures become very likely to occur. Many proposals in the literature have investigated possible *fault-tolerance* techniques—including active replication [14, 43], checkpointing [15, 32], replay logs [9, 28], or hybrid solutions [47] — each providing different trade-offs between *best-case* runtime cost and *recovery* cost. Whichever the adopted technique, maintaining some form of replication at some level (software/hardware components, state, or messages) imposes non-negligible overhead in terms of computing and communication resources.

In this section we present a possible solution to deal with temporary load variations in stream processing applications. This original approach trades off reliability guarantees and execution cost in actively replicated stream processing applications by temporarily claiming computational resources back from the fault-tolerance layer and by using them to handle possible load spikes. Our technique, called LAAR (Load-Adaptive Active Replication) [12, 13], dynamically deactivates and activates redundant replicas of processing components in order to claim/release resources and accommodate temporary load variations. At the same time, LAAR provides a-priori guarantees about the achievable levels of fault-tolerance, expressed in term of an *internal completeness* metric that captures the maximum amount of information that can be lost in case of failures.



	O1	O2
Selectivity	1	1
CPU Cost	0.1 s/smpl	0.1 s/smpl

	Low Rate	High Rate
Source	4 smpls/s	8 smpls/s
PE 1	4 smpls/s	8 smpls/s
Prob.	0.8	0.2

Fig. 6 A simple processing scenario. On the *left*, the application graph. On the *right*, concise characteristics of the application and of its data source

LAAR builds on top and significantly extends existing static replication techniques that have been previously proposed for DSPSs [28]: for every component in the application processing graph, *k* replicas are deployed at runtime. At any moment in time, one of the *k* replicas of each component has the role of *primary*, the others are called *secondary*. Primary and secondary replicas all receive samples from the primaries of their predecessors, and all process them advancing through the same sequence of internal states. However, only the primary outputs samples to the replicas of its successors. LAAR continuously monitors the input rate of application sources. It automatically activates and deactivates replicas in order to satisfy two goals:

1. The application deployment is never *overloaded*;
2. The *internal completeness* constraint expressed in the SLA is satisfied.

For the sake of simplicity, an application deployment is said to be overloaded when, for any host, the total CPU cycles per second that would be needed to execute the components assigned to it is greater than the available CPU cycles per second. Note that, in an overloaded system, samples accumulate at operator or sink input queues (increasing latency) and are eventually dropped when the corresponding buffers fill.

Let us illustrate the basic intuition upon which our approach is based in a minimal application scenario. Consider the application in Fig. 6: it consists of two operators connected in a very simple pipeline; the first operator (O1) processes data from a single data source (not reported in the figure for the sake of simplicity) and forwards its output to O2, which, in turn, sends the results of its computations to an external data sink (also not depicted in the figure). The selectivity of both operators is 1, meaning that for every received input sample they produce one output sample; moreover, it takes 100 ms for both operators to process an incoming sample, considering the CPU architecture of the hosts where the application is going to be deployed. The single data source can produce samples at two different rates, “Low” and “High”: the “Low” rate is 4 samples per second and is active on average for 80 % of the time (0.8 probability), while the “High” rate is 8 samples per second and is active in the remaining time intervals (0.2 probability).

The application is replicated and deployed on two servers, each hosting a copy of each operator, as shown in Fig. 7a. It is straightforward to see that, when the input configuration is “Low”, 80 % of the CPU time available at both hosts will be

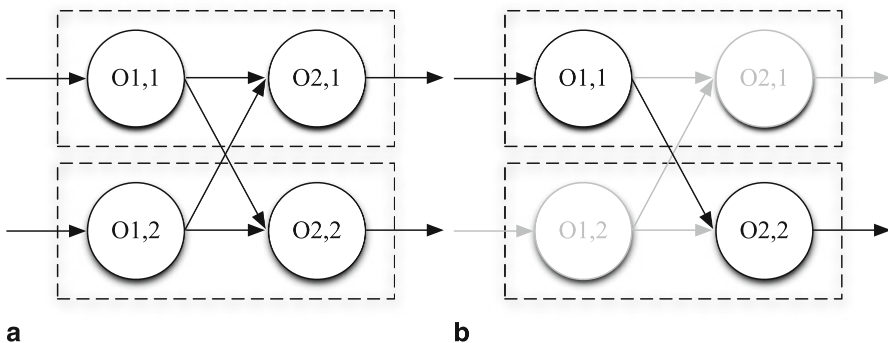


Fig. 7 **a** Replicated deployment of the application of Fig. 6 on two hosts. **b** Dynamic deactivation of replicas by LAAR during a "High" input configuration

occupied for processing samples. More importantly, when the input configuration is "High", the application would need 160 % of the total CPU time available, which—of course—is available only by adding extra resources to the deployment environment (with an increased cost).

The basic idea behind LAAR is to monitor the data sources and, according to the current data rates, dynamically deactivate replicas in order to release the resources necessary to face load variations. Figure 7b, for example, shows how LAAR could deactivate replicas of O1 and O2 during a load peak, so that the total CPU available will become enough to handle the new load.

Figure 8a and b show this behavior in a real stream processing deployment. We implemented and executed the replicated pipeline stream processing application shown in Fig. 7a on a deployment environment consisting of two servers equipped with a single core CPU. Figure 8a shows the execution of the application using static active replication: when the input passes to the "High" configuration (around 50 s from the beginning of the experiment), the CPU of the two hosts saturates, and the output is not able to keep up with the input rate; on the contrary, Figure 8b shows how, by temporarily deactivating replicas during the "High" input configuration, it is possible to save enough resources to allow the output stream to follow the input.

Obviously, if a failure of one of the active operators occurred during a "High" period, part of the input would not be processed as expected. However, the unique and strong aspect of LAAR is its ability to quantify *a-priori* these effects on the overall application reliability. As anticipated, LAAR defines the concept of *internal completeness*, a metric that tries to capture the amount of samples that are guaranteed to be processed in a *pessimistic* failure scenario, i.e., a scenario where all the active operator replicas fail. Without digging into formal details, the Internal Completeness metric (IC) is defined as follows:

$$IC(s) = \frac{\text{no. of samples processed in a pessimistic failure scenario P}}{\text{no. of samples processed with no failures}} \qquad (1)$$

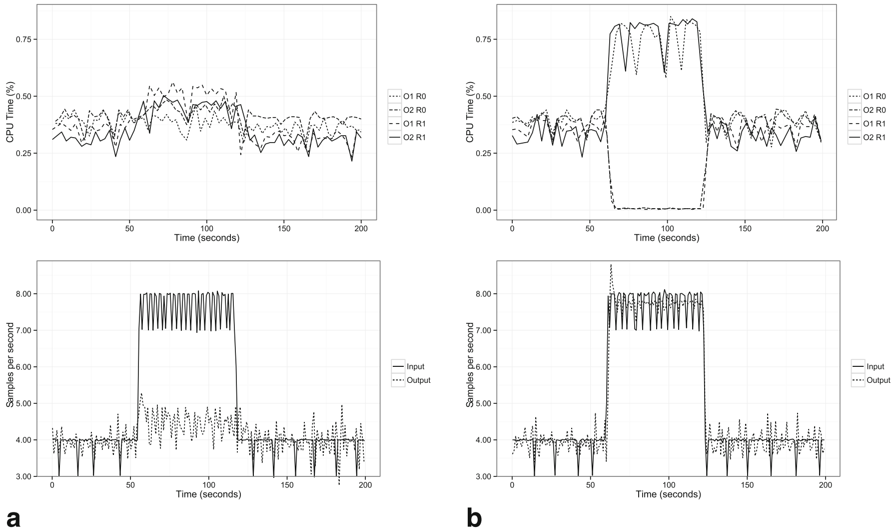


Fig. 8 **a** CPU time used by the two couples of replicated operators—top—and corresponding input and output rate—bottom. **b** CPU time and input/output data rate when O1 replica 1 and O2 replica 0 are deactivated by LAAR

where s represents a particular *replica activation strategy*, which associates the activation/deactivation status of application operators to each possible input rates configuration. For instance, in the example scenario presented above, during a period of T seconds and in absence of any failure, the application would process a total of $T(0.8 \cdot 8 + 0.2 \cdot 16)$ samples (considering both operators). On the contrary, considering a very pessimistic failure scenario P, where the active replica of each operator (respectively $O1, 1$ and $O2, 2$) is crashed all the time, the total number of samples processed would be $T(0.8 \cdot 8 + 0)$, for a total IC value of $\frac{6.4}{9.6} = 0.\overline{6}$. This means that, even in case of failures, at least 60 % of the total processing operations would be correctly performed.

However, finding a replica activation strategy that, at the same time, is able to keep the system in a non-overloaded condition despite load variations and to satisfy a user-defined IC requirement while keeping costs limited, is a very hard problem, especially when the processing graphs are much more complex than the one presented before. In order to solve this problem, LAAR performs a static optimization phase where the problem is modeled as a Mixed Integer Programming (MIP) instance. Although a precise formal description of the problem model is out of the scope of this chapter, in the following we sketch its formulation, in order to help the readers understand the main ideas behind the approach.

$$\underset{s}{\text{minimize}} \quad cost\,(s) \tag{2}$$

subject to:

$$IC(s) \geq SLA\ Constr. \tag{3}$$

$$load(h, s, c) < Thres. \qquad \forall \text{ server } h \text{ and input conf. } c \tag{4}$$

$$nreplicas(o, s, c) \leq 1 \qquad \forall \text{ operator } o \text{ and input conf. } c \tag{5}$$

In the equations above, the *cost* function in the minimization term represents the cost, in terms of resources (e.g., CPU, memory, and bandwidth), for a service provider to run the application during a billing period of length T using replica activation strategy s. Equation 3 constraints IC to satisfy the value required in the application SLA, while Eq. 4 states that each host in the deployment should never be overloaded. *Thres.* is a constant expressing the number of CPU cycles per second available at the deployment servers. The last constraint, expressed in Eq. 5, requires that there is at least one active replica of every operator in every input configuration, to ensure that the measured IC value is always one in absence of failures.

To have a rough idea of the complexity of the above problem, consider that the solution space has a size that is exponential in the number of operators, number of replicas per operator, and number of possible input configurations. In addition, the computation of IC values, resource usage, and server load levels require exponential time with respect to the number of operators, since they depend on the number of samples processed by different operators in different configurations, which in turn recursively depend on the number of samples processed by their predecessors. To deal with this complexity, LAAR solves the problem using an original constraint programming based algorithm, called *FT-Search*, that is able to find optimal or sub-optimal solutions to problem instances of reasonable size (i.e., graphs with tens of operators) in limited time, largely compatible with practical industrial data center constraints and application-specific requirements.

After having found solutions to the above optimization problem before application deployment time, LAAR performs its dynamic replica activation operations at runtime by inserting a special operator in the application graph, which continuously monitors the input rates and, according to the measured values, sends ad-hoc activation/deactivation commands to operator replicas.

If compared to active replication techniques, LAAR is able to handle load spikes by completely avoiding increased latency or sample drops due to full operator buffers. Moreover, by using weaker fault tolerance specifications through the IC metric, it can also reduce the cost of running stream processing applications proportionally to the required fault-tolerance guarantees. A large corpus of experiments, performed on a LAAR implementation built on top of IBM InfoSphere Streams and executed on a 60 cores IBM BladeCenter Cluster deployment, confirms the above claims. In particular, we have executed 100 different artificially generated stream processing applications using four different fault tolerance techniques. A *No Replication* (NR) approach runs the streaming applications without instantiating any operator replica. A *Static Replication* (SR) approach creates two replicas for each operator and keeps them

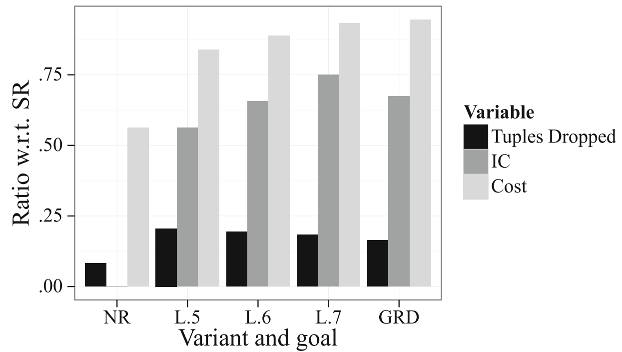


Fig. 9 Comparison of the different replication strategies; average values normalized w.r.t. SR

active all the time, independently on the input configuration. The *LAAR* replication approach uses the previously described techniques to run the streaming applications with three different IC requirements, 0.5, 0.6, and 0.7 (labelled L.5, L.6 and L.7, respectively). Finally, a *greedy* (GRD) approach uses techniques similar to those adopted by LAAR, but instead of deactivating replicas according to the results of a static optimization phase, it uses a simple runtime heuristic (i.e., it deactivates the most resource greedy component first).

Figure 9 shows a concise summary of the results collected in these experiments, by showing the average numbers of samples dropped, the average IC value achieved, and the average cost of different replication strategies as a fraction of the same values measured using the SR approach. It is immediate to see that, by using LAAR, it is possible to control the desired IC guarantees to directly influence the associated deployment cost, which is considered highly valuable and relevant in many business application scenarios.

7 Conclusions

The interest around the Smart City paradigm has been growing at an increasing pace in the last years and it is very likely that, thanks to the technical advances in computing devices and wireless, mobile, and wearable sensing, it will continue to grow in the next years. The efficient and effective exploitation of the unprecedented amounts of real-world data generated and injected every day inside IT infrastructures is a crucial step toward a real improvement in people's quality of life through smart computing technologies. In this context, DSPSs are a key technology for their ability to analyze information "on-the-fly" and produce continuous feedback that can be exploited to adapt real-world processes to their dynamically varying conditions. The tremendous heterogeneity of data and applications, together with their often unpredictable dynamic requirements, pose additional challenges for these systems. Developers of stream processing applications should be allowed to express, in a flexible way, the QoS requirements of their application scenarios, and DSPSs should

understand these requirements and automatically adapt their internal mechanisms to meet them in a way that is as much transparent as possible to the streaming applications and their logic. However, only a few modern DSPSs expose QoS-based customization features, and, in most cases, their are not first-class elements in all the three abstract, development, and runtime models, oppositely from the role we believe they should have. At the best of our knowledge, Quasit represents the most prominent exception, by allowing to express and enforce a large variety of QoS policies at each of the three levels.

We claim that future research on DSPSs should focus on QoS-related open issues with much stronger attention. In particular, it should: (i) improve the existing stream processing models to give application developers the opportunity to integrate rich QoS requirements in their applications; (ii) study efficient mechanisms to implement QoS policies on large scale deployments of DSPSs inside data centers. About this last point, a particularly promising research direction is the development of a novel class of *weak or probabilistic* QoS requirements that, in contrast with more traditional *strong* requirements, give runtime platforms additional degrees of freedom in their enforcement and more possibilities to adapt to highly variable system workloads. We believe (and the first preliminary results already collected are confirming our claim) that the *internal completeness* reliability metric adopted in the LAAR replication technique well represents this new class of QoS requirements for stream processing applications.

Acknowledgements We would like to thank the IBM Research Dublin Lab, and in particular Spyros Kotoulas, for his valuable work and feedback on LAAR (overviewed in Sect. 6), designed and implemented within a joint research collaboration.

References

1. 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, 12, 2, pp. 120–139 (2003).
2. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, 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.Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR). IEEE, Asilomar, CA (2005).
3. Agha, G. A.: Actors: a model of concurrent computation in distributed systems, Ph.D. dissertation, Artificial Intelligence Laboratory, Cambridge MA, USA (1985).
4. Amini, L., Andrade, H., Bhagwan, R., Frank Eskesen and Richard King and Yoonho Park and Chitra Venkatramani: SPC: A distributed, scalable platform for data mining. Proceedings of the Workshop on Data Mining Standards, Services and Platforms (DM-SS 2006). pp. 27–37. ACM, Philadelphia, PA (2006).
5. Apache S4 Project Web Site. Available, http://incubator.apache.org/s4. Last visited in September 2013.
6. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Ito, K., Motwani, R., Srivastava, U., and Widom, J.: STREAM : The Stanford Data Stream Management System, Technical report, Stanford InfoLab (2004).

7. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal. 15, 2, pp. 121–142 (2005).
8. Avnur, R., Hellerstein, and J.M.: Eddies: continuosly adaptive query processing. Proceedings of the ACM SIGMOD international conference on Management of data (SIDMOD 2000). pp. 261–272. ACM, Dallas, TX, USA (2000).
9. Balazinska, M., Balakrishnan, H., Madden, S.R., Stonebraker, M.: Fault-tolerance in the borealis distributed stream processing system. ACM Trans. Database Syst. 33, 1, Article 3, 44 pages (2008).
10. Bellavista, P., Corradi, A., Reale, A.: The QUASIT Model and Framework for Scalable Data Stream Processing with Quality of Service. Proceedings of the 5th International Conference on Mobile Wireless Middleware, Operating Systems, and Applications (MOBILWARE 2012). Springer Berlin-Heidelberg, Berlin, Germany (2012).
11. Bellavista. P., Corradi, A., Reale, A.: Design and Implementation of a Scalable and QoS-aware Stream Processing Framework: the Quasit Prototype. Proceedings of the IEEE International Conference on Cyber, Physical and Social Computing (CPSCOM 2012). IEEE, Besançon, France (2012).
12. Bellavista. P., Corradi, A., Kotoulas, S., Reale, A.: Dynamic datacenter resource provisioning for high-performacne distributed stream processing with adaptive fault-tolerance. Proceedings of the 14th ACM/IFIP/USENIX International Middleware Conference—Demo & Poster Track, ACM, Beijing, China (2013).
13. Bellavista. P., Corradi, A., Kotoulas, S., Reale, A.: Adaptive fault-tolerance for dynamic resouce provisioning in distributed stream processing systems. Proceedings of the 17th International Conference on Extending Database Technology (EDBT 2014), ACM, Athens, Greece (2014). To appear.
14. Brito, A., Fetzerm C., Felber, P.: Multithreading-enabled active replication for event stream processing operators. In: 28th Symposium on Reliable Distributed Systems, pp. 22–31, IEEE, Niagara Falls, NY, USA (2009).
15. Cai, Z., Kumar, V., Cooper, B.F., Eisenhauer, G., Schwan, K., Strom, R.E.: Utility-driven proactive management of availability in enterprise-scale information flows. In: ACM/IFIP/USENIX 7h International Middleware Conference, Springer, Melbourne, Australia (2006).
16. 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. Proceedings of the 28th international conference on Very Large Data Bases (VLDB 2002). The VLDB Endowment, Hong Kong, PRC (2002).
17. Chandrasekaran, S., Shah, M.A., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F.: TelegraphCQ. Proceedings of the ACM SIGMOD international conference on on Management of data (SIGMOD 2003). pp. 668. ACM, San Diego, CA, USA (2003).
18. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce Online. Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI 2010). USENIX Association, San Jose, CA, USA (2010).
19. Cugola, G., Margara, A.: Processing flows of information: From Data Stream to Complex Event Processing. ACM Comput. Surv.. 44, 3, pp. 1–62 (2012).
20. Dean, J., Ghemawat, S.: MapReduce : Simplified Data Processing on Large Clusters. Commun. ACM, vol. 51, no. 1, pp. 107–113 (2008).
21. Digital Cities Project Web Site. Available, http://www.digital-cities.eu. Last visited in September 2013.
22. Djahel, S., Salehie, M., Tal, I., Jamshidi, P.: Adaptive Traffic Management for Secure and Efficient Emergency Services in Smart Cities. Proceedings of the IEEE International Conference on Pervasive Computing and Communicatino (PerCom 2013)—WiP Session. pp. 340–343, IEEE, San Diego, CA, USA (2013).
23. EUROCITIES Web Site. Available, http://www.eurocities.eu/. Last visited in September 2013.

24. Gedik, B., Andrade, H.: A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM InfoSphere Streams. Softw. Pract. Exper. 42, 11, 1363–1391 (2012).
25. Gedik, B., Andrade, H., Wu, K.-L.: A code generation approach to optimizing high-performance distributed data stream processing. Proceeding of the 18th ACM conference on Information and knowledge management (CIKM 2009). p. 847, ACM, Hong Kong, PRC (2009).
26. Giffinger, R., Fertner, C., Kramar, H., Kalasek, R., Pichler-Milanovic, P., Meijers, M.: Smart cities – Ranking of European medium-sized cities. Vienna UT, Centre of Regional Science (2007) Available, http://www.smart-cities.eu/download/smart_cities_final_report.pdf. Last visited in September 2013.
27. Haller, P. and Odersky, M.: Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science, vol. 410, no. 2–3, pp. 202–220 (2009).
28. Hwang, J.-H., Balazinska, M., Rasin, A., Çetintemel, U., Stonebraker, M., Zdonik, S.: High-availability algorithms for distributed stream processing. In: 21st International Conference on Data Engineering, pp. 779–790, IEEE, Tokyo, Japan (2005).
29. IBM Smarter Cities Project Web Site. Available, http://www.ibm.com/smarterplanet/us/en/smarter_cities/. Last visited in September 2013.
30. Intel Collaborative Research Institute for Sustainable Connected Cities. Available, http://www.intel-university-collaboration.net/?page_id=1420. Last visited in September 2013.
31. Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D.: 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).
32. Jacques-Silva, G., Gedik, B., Andreade, H., Wu, K.-L.: Language level checkpointing support for stream processing applications. In: 2009 International Conference on Dependable Systems & Networks, pp. 145–154, IEEE, Estoril, Portugal (2009)
33. Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J.: COLA : Optimizing Stream Processing Applications Via Graph Partitioning. Proceedings of the ACM/IFIP/USENIX 10th International Middeware Conference. pp. 308–327, Springer Berlin Heidelberg, Urbana Champagin, IL, USA (2009).
34. Martinez, F., Toh, C.-K., Cano, J., Calafate, C., Manzoni, P.: Emergency Services in Future Intelligent Transportation Systems Based on Vehicular Communication Networks. IEEE Intelligent Transportation Systems Magazine 2,2, pp. 6–20 (2010).
35. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed Stream Computing Platform. In: 2010 IEEE International Conference on Data Mining Workshops (ICDMW '10), pp. 170–177, IEEE Los Alamitos, USA (2010).
36. OMG: Data Distribution Service for Real-time Systems – Version 1.2, Specification. Object Management Group (2007).
37. Pardo-Castellote, G.: OMG data-distribution service: Architectural overview. Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops, pp. 200–206. IEEE, Providence, RI, USA (2003).
38. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D.: SODA : An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference. pp. 306–325. Springer Berlin Heidelberg, Leuven, Belgium (2008).
39. Xing, Y., Zdonik, S., Hwang, J.-H.: Dynamic Load Distribution in the Borealis Stream Processor. Proceedings of the 21st International Conference on Data Engineering (ICDE 2005). pp. 791–802. IEEE, Tokyo, Japan (2005).
40. Xing, Y., Hwang, J.-H., Zdonik, S.: Providing Resiliency to Load Variations in Distributed Stream Processing. Proceedings of the 32nd international conference on Very large data bases (VLDB 2006). pp. 775–786. The VLDB Endowment, Seoul, Korea (2006).
41. Safe City Project Web Site. Available, http://www.safecity-project.eu/. Last visited in September 2013.

42. Smart Cities Stakeholder Platform. Available, http://www.eu-smartcities.eu/. Last visited in September 2013.
43. Shah, M., Hellerstein, J., Brewer, E.: Highly available, fault-tolerant parallel dataflows. In: ACM International Conference on Management of Data, pp. 827–838, ACM, Paris, France (2004).
44. The Storm Project Web Site. Available, http://storm-project.net/. Last visited in September 2013.
45. Tang, P., Venables, T.: "Smart" homes and telecare for independent living. J. Telemed. Telecare. 6, 1, pp. 8–14 (2000).
46. Yang, H.-c., Dasdan, A., Hsiao, R., Parker, D.: Map-reduce-merge: simplified relational data processing on large clusters. In: Proceedings of the ACM SIGMOD international conference on Management of data (SIGMOD 2007). pp. 1029–1040, Beijing, PRC (2007).
47. Zhang, Z., Gu, Y., Ye, F., Yang, H., Kim, M., Lei, H., Liu, Z.: A hybrid approach to high availability in stream processing systems. In: 30th IEEE International Conference on Distributed Computing Systems, pp. 138–148, Genoa, Italy (2010).