ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Ph.D. Course in
ELECTRONICS, COMPUTER SCIENCE AND TELECOMMUNICATIONS
Cycle XXVI

Examination Sector 09/H1
Scientific Disciplinary Sector ING-INF/05

QUALITY OF SERVICE IN DISTRIBUTED STREAM PROCESSING FOR LARGE SCALE PERVASIVE ENVIRONMENTS

Candidate:
ANDREA REALE

Supervisor:
PROF. ANTONIO CORRADI

Advisor:
PROF. PAOLO BELLAVISTA

Ph.D. Course Coordinator:
PROF. ALESSANDRO VANELLI-CORALLI

Final Examination Year 2014

WEBSITE:
http://middleware.unibo.it/people/ar

E-MAIL:
andrea.reale@unibo.it

ABSTRACT

THE wide diffusion of cheap, small, and portable sensors integrated in an unprecedented large variety of devices and the availability of almost ubiquitous Internet connectivity make it possible to collect an unprecedented amount of real time information about the environment we live in. These data streams, if properly and timely analyzed, can be exploited to build new intelligent and pervasive services that have the potential of improving people's quality of life in a variety of cross concerning domains such as entertainment, health-care, or energy management. The large heterogeneity of application domains, however, calls for a middleware-level infrastructure that can effectively support their different quality requirements.

In this thesis we study the challenges related to the provisioning of differentiated quality-of-service (QoS) during the processing of data streams produced in pervasive environments. We analyze the trade-offs between guaranteed quality, cost, and scalability in streams distribution and processing by surveying existing state-of-the-art solutions and identifying and exploring their weaknesses. We propose an original model for QoS-centric distributed stream processing in data centers and we present Quasit, its prototype implementation offering a scalable and extensible platform that can be used by researchers to implement and validate novel QoS-enforcement mechanisms. To support our study, we also explore an original class of weaker quality guarantees that can reduce costs when application semantics do not require strict quality enforcement. We validate the effectiveness of this idea in a practical use-case scenario that investigates partial fault-tolerance policies in stream processing by performing a large experimental study on the prototype of our novel LAAR dynamic replication technique.

Our modeling, prototyping, and experimental work demonstrates that, by providing data distribution and processing middleware with application-level knowledge of the different quality requirements associated to different pervasive data flows, it is possible to improve system scalability while reducing costs.

PUBLICATIONS

Part of the work in this thesis have previously appeared in the following publications:

[1] P. Bellavista, A. Corradi, and A. Reale, "Effective epidemic dissemination of multimedia metadata in peer-to-peer overlay networks: The metis architecture and prototype," in *Proc. of the 16th IEEE Symposium on Computers and Communications*, (Kerkyra, Greece), pp. 1073–1080, IEEE, 2011.

[2] P. Bellavista, A. Corradi, and A. Reale, "Design and implementation of a scalable and qos-aware stream processing framework: The quasit prototype," in *Proc. of the 2012 IEEE International Conference on Cyber, Physical and Social Computing*, (Besançon, France), pp. 458–467, IEEE, 2012.

[3] P. Bellavista, A. Corradi, and A. Reale, "Quality-aware, reliable, and scalable crowdsensing in smart cities," in *Proc. of the 1st International Workshop on Reliable Cyber Phyisical Systems*, (Irvine, CA, USA), 2012. Informal proceedings.

[4] P. Bellavista, A. Corradi, and A. Reale, "The quasit model and framework for scalable data stream processing with quality of service," in *Proc. of the 5th International Conference on Mobile Wireless Middleware, Operating Systems, and Applications*, vol. 65, (Berlin, Germany), pp. 92–107, Springer Berlin Heidelberg, 2013.

[5] P. Bellavista, A. Corradi, and A. Reale, "Scalable stream processing with quality of service for smart city crowdsensing applications," *EAI Endorsed Transactions on Mobile Communications and Applications*, vol. 13, no. 3, 2013.

[6] P. Bellavista, A. Corradi, S. Kotoulas, and A. Reale, "Dynamic datacenter resource provisioning for high-performance distributed stream processing with adaptive fault-tolerance," in *Proc. of the 14th ACM/IFIP/USENIX International Middleware Conference — Demo & Poster Track*, (Beijing, China), pp. 13:1–13:2, ACM, 2013.

[7] P. Bellavista, A. Corradi, S. Kotoulas, and A. Reale, "Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems," in *Proc. of the of 17th International Conference on Extending Database Technology*, (Athens, Greece), ACM, 2014. In press.

[8] A. Reale, P. Bellavista, A. Corradi, and M. Milano, "Evaluating cp techniques to plan dynamic resource provisioning in distributed stream processing," in *Proc. of the 11th International Conference on Integration of Artificial Intelligence and Operation Research techniques in Constraint Programming*, (Cork, Ireland), Springer-Verlag, 2014. In press.

[9] P. Bellavista, A. Corradi, and A. Reale, "Quality-of-service in data center stream processing for smart city applications," in *Handbook on Data Centers* (S. U. Khan and A. Y. Zomaya, eds.), New York, NY, USA: Springer-Verlag, 2014. In press.

[10] P. Bellavista, A. Corradi, and A. Reale, "Quality of service in wide scale publish-subscribe systems," *IEEE Communications Surveys & Tutorials*, 2014. In press.

AWARDS

For his work on distributed stream processing in collaboration with IBM Research Dublin, the author has been granted a *2013–2014 IBM Ph.D. Fellowship* award.

CONTENTS

1 INTRODUCTION

The deep integration of computing systems into everyday life has been envisioned long ago [1], but it is only in the last years that *pervasive* technologies are assuming a prominent and growingly important role in our society. The reasons behind this phenomenon are manifold, and have to be sought in the convergence of many advances in computing technologies and simultaneous favorable social factors.

One of the driving socio-technical motivations can be identified in the wide availability of cheap, small, and portable computing devices integrated with rich sensing capabilities and heterogeneous network interfaces. Tablet computers, smartphones, or smart watches are very well known examples, and their diffusion can give a quite precise idea of the deep penetration of such devices in our society. All these objects, even those considered to be "low-end", are normally equipped with a surprisingly large number of sensors, such as microphones, cameras, accelerometers, gyroscopes, GPS receivers, or light and proximity sensors [2]. Besides these personal devices, stationary or mobile sensor nodes are often deployed and largely used into other common environments such as vehicular networks [3, 4], utility grids [5, 6], healthcare [7], education [8], or environmental monitoring [9]. At the same time, also thanks to ubiquitous high-speed Internet access, people is changing their way of using the Web and, from being simple consumers, they have become active producers of information [10]. New user-generated content is now continuously created and published on blogs, social networks, or multimedia sharing platforms, and movements like citizen journalism are becoming increasingly popular [11].

This incredible amount of digital information, practically a constant live coverage about our physical and social environment, provides an unmatched opportunity to build new services seamlessly integrated into the physical world fabric and supporting people in their daily actions, with the potential for improving their quality of life by enhancing cross-concerning areas such as healthcare, transportation, decision making, and energy management. *Smart pervasive services* can use the continuous streams of data from the cyber-physical world to give valuable real time insights about what is happening around us; more interestingly, they can exploit this information to act, in a sort of self-corrective feedback loop, on the environment itself.

However, the availability of these big data streams alone is not enough to implement smart and useful applications, because they

are normally unstructured, highly redundant, and often reflect partial and local views of real world events. Other fundamental steps are to collect their content, route it through a scalable data distribution infrastructure, and, most importantly, distill, from the raw data, condensed knowledge that can be more easily leveraged by pervasive services. All this process is at the core of SMART PERVASIVE ENVIRONMENTS (SPEs), which represent the synthesis of ubiquitous sensing, efficient data distribution, smart data analysis, and a possibly large number of heterogeneous pervasive applications.

Yet, to build the infrastructure supporting SPEs can be a very hard task because of the many technical challenges deriving from both the characteristics of the input data streams and the unique processing requirements of these scenarios. Scalability is the most obvious one, with the extraordinary volume of data pushing traditional distribution and management infrastructures to their limits. Additionally, data analysis must often satisfy near real time requirements and results must be promptly produced in response to new data being *pushed* from their sources, since significant delays can reduce or invalidate the usefulness of the processing efforts. The number of possible data sources and their sparse geographical distribution represent other obstacles to overcome, calling for cost-effective mechanisms for efficient data collection and distribution. Furthermore, the large heterogeneity of data sources puts additional requirements on the analysis process, which needs to adaptively deal with different representation formats, non-uniform data semantics, and with missing, incomplete, or erroneous values. Finally, the possibly many SPE applications can have highly differentiated goals, with consequently different requirements that must be properly taken into account at both data streams distribution and processing levels.

The ability to understand and exploit these requirements by offering differentiated QUALITY OF SERVICE (QoS) to different applications is a fundamental feature for SPE-supporting platforms. QoS describes the expected infrastructure behavior with respect to a set of performance metrics, called QoS *properties*. Data delivery guarantees, latency, throughput, priorities, or processing accuracy are common examples. Whichever aspect they regulate, the enforcement of quality requirements always translates to the presence of mechanisms that control, either statically or dynamically, the allocation of distributed networking and computational resources to different tasks so that the promised performance goals are met at runtime. While desirable in many telecommunications and computing scenarios, we are convinced that QoS-awareness is an indispensable property for infrastructures supporting the vision of smart pervasive applications and services. Using the available resources adaptively and according to the actual needs of different applications, in fact, enables them to scale to larger and larger scenarios while efficiently managing the related

cost growth. Moreover, when resources are limited, as they almost always are in real-world settings, the specification of quality requirements permits to assign the right priority to concurrent tasks. This is particularly important, for example, when the rate of the processed input streams changes in in ways that were not expected at application deployment time: these situations, very common in our scenarios where data is produced by external and uncontrolled sources, might cause the resources available for data dissemination and analysis to become suddenly insufficient to handle the whole input load, so it is even more important to allocate them intelligently. For example, think about a pervasive application managing emergency situations compared to another one offering entertainment services: by understanding the different quality expectation of the two applications, the platform would be able to offer prioritized service to the first and most important one.

This thesis studies middleware-level infrastructures for real time distribution and analysis of large data streams generated in pervasive sensing scenarios. In particular, our focus is concentrated on their enhancement through QoS-based behavior, and on the design and implementation consequences of these extensions. We investigate the tradeoffs between guaranteed quality levels, scalability, and runtime costs by reviewing previous state-of-the-art solutions, and by contributing with our own proposal of a QoS centric distributed stream processing system and of a novel fault-tolerance technique that optimizes runtime costs according to application specific QoS requirements.

In the rest of this chapter, we expand our analysis of SPEs and introduce a simple conceptual architecture that models infrastructures for smart pervasive services (Section 1.1). We emphasize the possible benefits that QoS-based behavior can provide at each architectural level and present, in Section 1.2, an example scenario motivating our ideas. Finally, in Section 1.3 we formulate the main research questions that we try to answer in this dissertation, and, in Section 1.4, we conclude the chapter by outlining the main contributions of our work and by presenting the structure of the remainder of the thesis.

1.1 ARCHITECTURE

With the expression *smart pervasive environment* we refer to the collection of hardware, software, and networking elements that support the execution of pervasive services. As described previously, these services operate and act in the cyber-physical world in a self-reinforcing sensing, processing, and actuation loop, which we represent through a simple conceptual model made of three main components and shown in Figure 1.1. At the bottom level, the *sensing* and *actuation* layer interfaces with the cyber-physical world; the *data dis-*

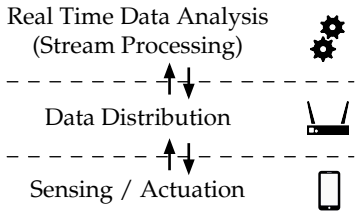


Figure 1.1: An architecture for smart pervasive environments.

tribution layer is responsible of the efficient dissemination of data to and from the sensing layer, while the top *data analysis* level performs continuous processing of the data streams coming from the lower layers and produces results that can be translated to actions eventually fed back downward to the physical world.

1.1.1 Sensing and actuation

Behind our vision of SPEs, there is the large availability of information that describes, continuously and in real time, features and events of the real world. With the expression "sensing data", we refer generically to these data streams. In our scenario, we consider two main types of sensing data sources, i.e., *physical sensors* and *virtual sensors*.

Physical sensors correspond to computing nodes equipped with actual sensor devices and capable of measuring objective physical quantities, such as temperature, light, mechanical vibration, magnetic field magnitude, or concentration of air particulates. Normally, these nodes have one or more network interfaces that are used to communicate sensed data with very limited or no human intervention. Today, physical sensors are ubiquitously deployed as fixed or mobile ad-hoc environmental monitoring nodes, or integrated in other objects like smartphones, cars, or even household appliances.

Virtual sensors encompass all the data sources that, while not measuring real physical quantities directly, nonetheless provide information about important real world aspects. Live social networks feeds are a classic example of virtual sensors (they also have been called social sensors [12]); other examples are RFID streams [13], phone call records [14], or results of crowdsensing tasks [15].

At the other end of the SPE processing loop, there is the actuation process. Once sensing data streams have been distributed, analyzed, and their content have been used to take adaptive feed-back decisions, these decisions must be applied on the physical world. The process of actuation can be much more complex than the sensing process and depends strictly on the specific application scenario. In the simplest cases, it can be performed by using physical actuator devices, like circuit breakers in a smart electrical grid [16], or automatic traffic light controls in a road-network management system [17] (see also

Section 1.2). However, other forms of more complex actuation are also common: for example, data analysis steps can produce accurate management plans that are then put in action by human operators [18], or behavior recommendations displayed on users' smartphones.

1.1.2 Data distribution

Modern sensor nodes are commonly equipped with richer and richer processing capabilities. For instance, average smartphones have multi-core general purpose processors and an amount of main memory in the gigabyte range, allowing many filtering and pre-processing tasks to be successfully performed in place. However, the view of the cyber physical world that a single sensor device builds through its sensing actions is local, partial, and can be highly imprecise due to sensing errors. Furthermore, many data analysis methods (e.g., deep learning algorithms [19]) can be still too demanding for the locally available resources. For these reasons, to make good use of them, sensing data must be routed from the sensing layer to destinations that have a wider view of the global system status and a much greater amount of computational resources. The role of the data distribution layer is to move data efficiently from their sources to the data analysis layer, and, once processing results are transformed in actuation tasks, route them back to actuators in the lower-layer levels.

Building data distribution services is a very hard task, and we identify five main challenges in their realization. First, the conspicuous volume of data produced and exchanged in SPEs can easily push existing infrastructures to their limits, and the distribution service must be also able to deal with possible shortage of network resources. Second, the data production rate can be high and highly variable: it is important that the real time requirements of the processing tasks involving these data are taken into account in the data distribution layer to avoid potential bottlenecks. Third, the exchanged data can be highly variable not only in their content and semantics, but also in their encoding. Given the high heterogeneity of the data distribution network participants (i.e., sensor nodes, actuators, and data processing sites), a big challenge is to provide interoperability despite data variety. Fourth, the data distribution service must support many different and coexisting network access technologies, from high-speed cabled interfaces to low-power and low-speed wireless ones. Fifth, it must efficiently and elastically scale [20] to the geographical distribution of its participants, in order to enable the realization of SPEs at neighborhood, city, or regional scale.

Our opinion is that a key to face these challenge is to allow deep QoS-based configuration of the distribution service. By offering differentiated service levels to different participants and different data flows, the data distribution layer can adapt to their specific require-

ments and optimize the way possibly limited network resources are allocated and used. For example, some data flows may be more tolerant to delays than others, thus allowing a more generous use of network-level batching to optimize channel exploitation. Likewise, data from sources producing highly redundant data streams can be disseminated using lossy channels.

1.1.3 Data analysis

The last but essential layer of our conceptual architecture is the data analysis level. Its role is to process the big data streams generated in the sensing layer and received through the data distribution layer, and produce always up-to-date results that reflect the high-level knowledge extracted from their content. These results can be used either directly to acquire a better understanding of the pervasive environment, or to produce commands sent back to the cyber-physical world with the objective of modifying it according to application goals.

A common trend to face the challenge of processing this huge amount of data is to leverage the computing power of commodity computers inside data centers [21]: by using highly-parallel and fault-tolerant software architectures, extremely complex processing tasks can be performed while keeping costs reasonably limited. Running intensive data processing operations in large data centers brings two important benefits: from a performance point of view, data centers represent the ideal environment for the implementation of highly parallel processing algorithms thanks the availability of very high-speed cluster-local Local Area Networks (LANs); furthermore, management costs can be significantly reduced thanks to extensive use of resource virtualization [22] and workload consolidation [23] techniques. For these reasons, frameworks that handle the complexities of parallel processing on large clusters (e.g., [24–29]) have recently received enormous attention and are currently used in many production scenarios.

Among them, the class of Distributed Stream Processing Systems (DSPSs) includes solutions that are specifically designed for the parallel and scalable analysis of big data streams. In our SPE architecture, data analysis services are modeled as stream processing applications over sensing data that run on top of DSPSs hosted in large data centers infrastructures. DSPSs, differently from other, *batch-oriented*, parallel processing frameworks (a very popular example of this last category being Apache Hadoop [30]), put the assumption of continuous input at the core of their design and implementation principles, and model data analysis tasks as *permanent queries* on these transient data [31]. Moreover, they usually optimize processing latency rather than pure data throughput, making their model particularly attractive for the real time data analysis scenarios of smart pervasive services.

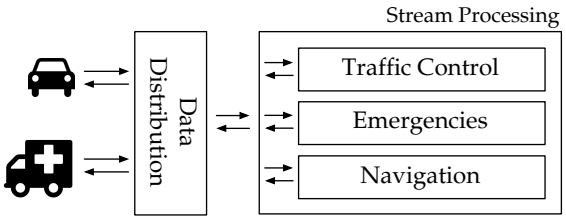


Figure 1.2: Traffic management system data flow.

Like for the data distribution services, we believe that to recognize the different quality requirements of different processing tasks in the data analysis layer can be extremely important to implement cost-effective and scalable SPEs. These tasks, in fact, normally run concurrently on a shared data-center infrastructure and they should be executed according to intelligent resource allocation policies that reflect their specific goals and consequent priorities. For example, tasks requiring bounded processing latency, such as those related to the monitoring of health parameters in smart tele-care services [32], should receive a guaranteed allocation of CPU time, memory, and networking resources, also at the expenses, if necessary, of less time-sensitive services, like, for example, pollution monitoring [33]. When resource allocation is made even more difficult by dynamically changing input load — a typical case in distributed stream processing —, knowledge about application-level QoS requirements can be a valuable information that dynamic schedulers can exploit to adapt to new conditions.

1.2 EXAMPLE SCENARIO

There are countless possible ways in which the large data streams from the sensing layer in a smart environment can be exploited to build useful pervasive services. In this section, we present a simple example that describes the architecture of a smart Traffic Management System (TMS) within an SPE architecture.

The high level goal of this system is to optimize the management of the public road network in a smart city through real time analysis of cars periodical position reports. By using vehicle-to-vehicle and vehicle-to-infrastructure communications [4], cars continuously exchange encrypted packets containing detailed information about their route and, when they pass by wireless data collection points installed in highly trafficked junctions, they relay the collected packets to the data distribution infrastructure. In their turn, the collection points route this information to a data center-hosted stream processing application, that processes it in order to build a global view of the current road network status with the goal of realizing the core TMS functionalities. As shown in Figure 1.2, the TMS implements three

main high-level services:

- *Traffic flow control*. By analyzing short term and long term variations in car speeds along different roads, the system adapts the traffic lights timings according to the current road network conditions.
- *Management of road emergencies*. In case of accidents, the vehicles involved and other cars nearby immediately start sending messages about the event to on-road collection points, which 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 accordingly, for example, by suggesting drivers to take alternative paths (see next point).
- *Real-time navigation*. Cars traveling in the city can query the TMS for advanced navigation services. The system will answer by providing an always up-to-date route that takes into account road load conditions and possible emergency situations.

The three TMS tasks, 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 traffic load when the available network and computational resources might not be sufficient to satisfy all the application flows. Accident notification messages should be transferred and processed *reliably* because 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 in the corresponding data streams.

This simple but, we believe, very representative example, shows how important can be for SPE data distribution and stream processing infrastructures to provide a rich and native support for differentiated QoS. By using this type of support, developers can focus their attention on application-level modeling and implementation problems, while delegating the realization of complex quality enforcing mechanisms to the underlying platforms.

1.3 RESEARCH QUESTIONS AND METHODOLOGY

In this work, we investigate design and implementation issues related to the enforcement of differentiated QoS in SPE middleware infrastructures based on the architecture presented previously in this chapter. The two main research questions that motivate our work are:

- What is the role of quality and quality requirements in SPE environments and how can we model them? What QoS policies should SPEs middleware support and what are the related design and implementation issues? Can we organize these policies into useful taxonomies to help build future large scale SPEs?
- Can we exploit rich knowledge of application-level quality requirements to optimize SPE infrastructures? In particular, is it possible to leverage this information to improve scalability and reduce platform runtime costs?

In this dissertation, we throughly analyze and discuss these questions with an engineering-oriented perspective. Starting from deep technical analyses of existing and state-of-the-art contributions from industry and academia, we build simple and practical models of QoS-aware data streams distribution and processing and draw general and reusable principles for the design and implementation of future QoS-aware systems. We validate our ideas by building prototypes of systems, techniques, and algorithms and by evaluating them through extensive experimental campaigns on small to large scale distributed deployments of realistic application scenarios. We do not use any simulation-based validation method: although they can often provide quick feedback about the effectiveness of new complex solution approaches, we have preferred to base our conclusions on direct and on-the-field performance measurements of real system prototypes, since we believe that the complexities involved in SPE scenarios are very difficult to capture in discrete simulations. Demonstrating its claims with this type of validation methodology, the thesis proposes our answers to the above research questions through several contributions, briefly introduced in the next section.

1.4 THESIS CONTRIBUTIONS AND OUTLINE

In this thesis, we study the technical challenges behind the realization of scalable and cost-effective Smart Pervasive Environments (SPEs), i.e., computing environments that integrate ubiquitous sensing, large scale data distribution, and parallel stream processing to build a new class of services that can improve people's quality of life by acting on the physical world based on real time analysis of sensing data.

In particular, we focus on the middleware-level infrastructure supporting the analysis of big sensing data streams, corresponding to the

data distribution and stream processing levels of the SPE architecture presented previously in this chapter. The main claim of this thesis is that streaming data distribution and processing can be improved by supporting and exploiting a deep knowledge of the QUALITY OF SERVICE (QoS) required by different data flows and applications. We support this claim by providing the following contributions:

- *Chapter 2*. We analyze the requirements of data distribution services for SPEs and identify in the PUBLISH/SUBSCRIBE (PUB/SUB) communication model a promising solution to answer the strong need of scalability, interoperability, and QoS-based configurability coming from the considered scenarios. To support this claim, we survey a selection of widespread PUB/SUB architectures and emphasize the trade-offs between design choices, implementation details, scalability, and supported QoS levels.
- *Chapter 3*. Orthogonally to our study of data distribution service architectures, we survey the state-of-the-art of DISTRIBUTED STREAM PROCESSING SYSTEMS (DSPSs) and investigate their ability to support differentiated QoS for data analysis tasks in SPEs. This study leads to our conclusion that, in order to fully support the requirements of future pervasive services, DSPSs should offers comprehensive solutions for QoS-based data analysis.
- *Chapter 4*. We propose an original architecture for QoS-centric DSPSs, called Quasit, and present the lessons learned while designing and implementing a working prototype of our ideas. The goal of the architecture is to allow deep QoS-based customization of every aspect of the stream processing platform, and our prototype, freely available for download, aims at being a shared platform that researchers can use to develop and validate new QoS mechanisms for distributed stream processing.
- *Chapter 5*. We investigate the possibility to offer dynamically flexible quality guarantees to stream processing applications that can tolerate them, in order to adapt platform runtime costs to their exact requirements. We do so by proposing LAAR, an original fault-tolerance technique for DSPSs that supports the specification and runtime enforcement of customizable consistency levels. Through a thorough experimental evaluation on a large distributed stream processing deployment, we demonstrate the effectiveness of our proposal.

The thesis is concluded by Chapter 6, where we summarize the most important findings of our work and highlight interesting and still open research directions.

2 QUALITY OF SERVICE IN PUB/SUB DATA DISTRIBUTION

The unprecedented amount of data generated in SPEs needs to be effectively distributed before it can be processed, analyzed, and used to build useful applications. We identify three major requirements for data distribution in SPEs.

1. Interoperability.
2. Scalability.
3. QoS-based configurability.

In a highly heterogeneous environment, where sources and destinations with different hardware capabilities and running different software stacks produce and consume information, *interoperability* plays a fundamental role. It enables composition of information and services, encouraging the emergence of reusable components, thus improving the offered services and reducing their cost. *Scalability* is probably one of the most important technical properties to achieve in order to build the large SPEs envisioned in the previous chapter. The cost of running and managing the data distribution infrastructure should increase in a graceful and controlled way as more entities join the system (*participants scalability*), as the systems expands over larger geographical areas (*geographical scalability*), and as the volume of the data exchanged grows (*data scalability*). We believe that the key to achieve scalability lays in the possibility to customize the data distribution service with detailed QoS-related configurations. If properly leveraged, QoS-*based configurability* lets services adapt their protocols to the specific requirements of different scenarios and avoids to uselessly over provision resources that are not strictly needed to satisfy the overall application quality requirements.

The Publish/Subscribe (PUB/SUB) messaging pattern is widely recognized as an important solution to address many of the requirements of flexibility posed by highly distributed systems. In PUB/SUB systems, communications between parties are decoupled in space, time, and synchronization [34]. Decoupling is a fundamental property to implement truly scalable systems: space decoupling means that two parties do not need to know each other to exchange information; time decoupling removes the requirements of parties being on line and active at the same time, while asynchronicity avoids blocking operations on both producing and consuming endpoints, thus promoting a reactive style of interactions [35] that favors scalability. Moreover, PUB/SUB systems have been used often and successfully as

message buses bridging the functionalities of heterogeneous components [36–39] thanks to a communication paradigm that has essentially the advantages of message-based decoupling and to the adoption of open protocols and data representations.

Among the three requirements of scalability, interoperability and QoS-based configurability stated before, the latter is the one that, at least apparently, fits the PUB/SUB model less naturally. In fact, guaranteed behaviour has been more often associated to multimedia systems [40] or to synchronous and point-to-point messaging solutions like REMOTE METHOD INVOCATION (RMI) [41] or Web Services [42, 43] rather than asynchronous, many-to-many ones such as PUB/SUB architectures. In addition, scalable limited overhead and deterministic quality are, in many cases, contrasting requirements from the point of view of design/implementation choices: this is especially true in wide area network deployments, where the satisfaction of these requirements is made even more challenging by the difficulty to estimate and predict network performance at runtime. Nonetheless, in the last decade, several works [20, 44–48] have analyzed and investigated mechanisms and techniques for the design/implementation of scalable PUB/SUB middleware, and many solutions emerged for their ability to satisfy more or less strict quality requirements in highly different application scenarios [49]. The experience made with the development and deployment of these systems has given relevant information on how design/implementation choices can impact on the ability of PUB/SUB middleware to offer different QoS-related performance at runtime.

In this chapter we support the claim that the PUB/SUB model of interaction is the most suitable to satisfy the data distribution requirements of interoperability, scalability, and quality of SPEs. To this purpose, we present a simple and original model for the classification of PUB/SUB systems that considers both functional and non-functional aspects of data interactions, and we survey a selection of widespread PUB/SUB systems from both academia and industry in order to shed light on the adopted trade-offs between design choices, implementation details, scalability, and QoS levels. We finally identify the current issues and limitations of PUB/SUB supports, and we propose directions of investigation for building new systems capable of properly balancing the design and performance trade-offs depending on the target deployment scenarios.

The rest of the chapter is organized as follows. We start by analyzing the previous literature that, like us, has attempted to understand the trade-offs in the implementation of quality-related behavior in PUB/SUB infrastructures (Section 2.1). In Section 2.2, we present our model for PUB/SUB data dissemination that can also describe QoS-aware interactions, and we propose a classification of common PUB/SUB QoS properties that captures their common character-

istics and concerns. A survey of a selection of PUB/SUB systems that emerged for their original support of QoS properties is presented in Section 2.3, followed by a comparative discussion in Section 2.4. Finally, in Section 2.5, we define a set of guidelines for future research in QoS-aware data distribution, and, in Section 2.6, we summarize and conclude the chapter.

2.1 POSITIONING OUR CONTRIBUTION

A conspicuous amount of work has been done in the research field of distributed PUB/SUB systems and middleware in the last 15 years, demonstrating the widespread interest in the topic. In this section we position the surveying work we perform in this chapter by analyzing previous contributions that either give a conceptual grounding to the ideas that we use and develop here, or that are close to our proposal of PUB/SUB modeling and classification.

In their popular and seminal article, Eugster et al. [34] make a thorough comparison of various distributed communication models, and single out the identifying characteristic of the PUB/SUB paradigm. The paper proposes a basic and widely accepted taxonomy of PUB/SUB systems, which classifies them according to their subscription model. The authors also define some basic QoS properties, but very little space is dedicated to the description of design/implementation tradeoffs related to the defined parameters.

Baldoni et al. [45] analyze different PUB/SUB solutions under two orthogonal perspectives that they call *subscription model* and *architectural model*. The article also dedicates a section to one specific quality aspect, i.e., *reliability*, but it does not cope with many other important quality concerns that we believe of primary importance in PUB/SUB data distribution, such as delivery latency, persistence, or ordering.

A detailed survey of routing algorithms for PUB/SUB communication can be found in [46] and [50]; in particular, the two papers focus on the sub-field of *content-based* routing. Although QoS related issues are out of the scope of those articles, their contribution is significant to our work because they perform a detailed discussion of design/implementation details specifically associated with content-based event routing techniques, which were helpful to better understand specific facets of different technical solutions.

In their book chapter, Corsaro et al. [51] directly face the problem of QoS in PUB/SUB middleware: the authors review the requirements and the semantics of three important categories of QoS parameters, i.e., *reliability*, *timeliness*, and *security & trust*. A careful analysis is performed to illustrate how two important standards for industrial messaging systems support the aforementioned parameters. Compared to ours, that work differs in two main directions. First, our survey is signifi-

cantly more comprehensive in the variety of solutions discussed, in particular about PUB/SUB contributions from academia. Second, this chapter has a much stronger focus on proposing guidelines that help to properly understand and improve the quality/complexity trade-offs existing in current state-of-the-art solutions.

Behnel et al. [52] identify and define a set of meaningful QoS policies for PUB/SUB systems and group them in two macro groups, i.e., quality at *global infrastructure level* and quality at *notification/subscription level*. While that work is successful in building a common dictionary and in defining the semantics of many QoS properties commonly available in PUB/SUB systems, it does not investigate how these properties can be effectively achieved in wide-scale PUB/SUB systems with different design/implementation trade-offs, as we aim at doing.

The paper by Mahambre et al. [49] is probably the most similar to the work in this chapter in terms of goals. It surveys state-of-the-art PUB/SUB systems by focusing on their support to different QoS classes. Conversely to [51], the article focuses on solutions coming from the academia and omits to compare significant industrial ones. More importantly, the paper differs from our work in the fact that its primary objective is to give a high-level architectural and conceptual model for PUB/SUB systems that support QoS, rather than to deeply identify technical issues and the associated spectrum of solution guidelines, with technical advantages and weaknesses, for the implementation of scalable and QoS-enabled PUB/SUB systems.

2.2 MODEL

In this section we present a unified model for PUB/SUB systems that captures, at the same time, their functional aspects, i.e., the set of features that relate directly to data distribution, and the properties connected to QoS-based configurability, i.e., their non functional aspects. Our proposal, which takes into account previous PUB/SUB models but originally extends them for what concerns QoS representation, is centered on the concept of *notification space*, an abstract geometrical space that we use to represent PUB/SUB distribution actions and requirements. We relevantly modify and extend the model already introduced by [46] in two main directions. First, we augment the space dimensions with the concept of time, which lets us model dynamic and evolving aspects of data distribution more naturally. Secondly, we present a concise and coherent model of how PUB/SUB actors can offer/request QoS levels in data delivery and how these offers/requests can be effectively matched by the PUB/SUB system.

The presentation is organized as follows. First, in Section 2.2.1, we overview our notification space model and use it to describe the behavior of PUB/SUB middleware at a high abstraction level. Soon after,

we introduce two commonly accepted taxonomies for PUB/SUB systems: later in this chapter, we show that the positioning within these two taxonomies affects many relevant PUB/SUB design/implementation choices and, in particular, the ability to offer QoS-enabled and scalable data distribution. Finally, in Section 2.2.2 we extend the notification space model by supporting QoS requests and offers.

2.2.1 Basic model

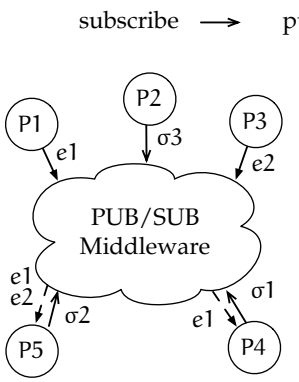
A PUB/SUB middleware is a distributed platform that allows its *participants* to exchange information with each other in the form of *data samples* (or simply, *samples*). A participant enters information in the system by *publishing* and expresses its data interests by means of *subscriptions*. The middleware *delivers* samples to *subscribers* according to their subscriptions. A *sample* e is a set of *key-value* pairs, whose meaning is generally application-dependent. Without loss of generality, we assume key and values to be arbitrary strings of finite length.

The notification space is an m-dimensional space ($m \geqslant k+1$). A point in the space is called *notification* and describes the publication of a sample. The first k dimensions correspond to the k sample keys, while the $(k+1)^{th}$ dimension models the time when the publication has occurred[1]. The remaining $(m-k-1)$ dimensions are optional; they can be used to model behavioral extensions such as possible non-functional aspects of sample publication. As a relevant example, an additional dimension could be used to model the geographical coordinates of a publisher, and thus could enable subscription filtering based on physical proximity. In Section 2.2.2, we use these dimensions to extend the model and represent QoS based interactions.

When a participant *publishes* a sample, a notification n is created and represented as a point in the space whose coordinates correspond to the sample values. In order to *subscribe*, participants select subspaces of interest within the notification space with *subscription filters*, i.e., boolean function expressing constraints over the k dimensions.

Whenever e is published, the system dispatches it to a set of participants P that have specified a subscription filter σ_p matching the sample. A filter is said to *match* a sample when the corresponding notification belongs to the subspace identified by the filter. Any *consistent* implementation of PUB/SUB middleware guarantees that subscribers receive only samples that match at least one of their subscription filters. Let us point out that, for several design/implementation reasons, in many systems it is not necessarily true that a participant receives *all* the samples matching its subscriptions. Figure 2.1 schematically shows a possible PUB/SUB interaction, where circles represent partici-

1 To simplify the discussion, our model assumes the existence of a global clock, but such a clock is used only for modeling purposes and does not have to be available (usually it is not) in PUB/SUB implementations.



Sym.	Content
σ1	{ topic=weather }
σ2	{ topic=weather, city=BO }
σ3	{ topic=news }
*e*1	{ topic: weather, city: BO, forecast: fair }
*e*2	{ topic: weather, city: DUB, forecast: rainy }

Figure 2.1: P2, P4, and P5 subscribe with subscription filters σ1, σ2, and σ3; P1 and P3 publish samples *e*1 and *e*2 that the system delivers to subscribers. The side table describes the represented subscriptions and samples.

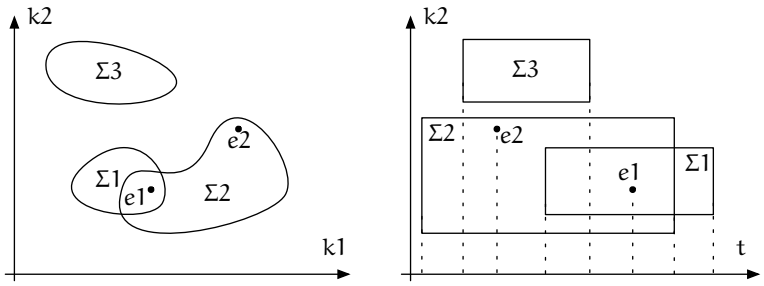


Figure 2.2: Representation of a three-dimensional notification space. On the left a projection of the space on two dimensions (k_1 and k_2); on the right a view of the same space projected on k_2 and the time dimension t.

pants, and arrows represent actions. Figure 2.2 pictures a similar situation in the notification space.

A publisher can optionally declare its publishing intents and describe beforehand the type of samples it will produce through *advertise* actions. An advertisement includes an advertisement filter that, similarly to subscription filters, is a boolean function over the functional and non-functional dimensions of the notification space, and it identifies the subspace where the notifications generated by the publisher samples will fall. Advertisements are a very valuable mechanism to provide PUB/SUB middleware with additional knowledge about the samples that will be generated in the future. This knowledge could be used, for example, to support discovery services or to optimize data routing and matching. Most importantly for the specific goals of our survey, the advertisement mechanism can be leveraged as a building block for modeling advanced interactions between participants, such as QoS requests and offers.

PUB/SUB systems have often been classified according to two alternative taxonomies: the first, model-level, taxonomy organizes systems

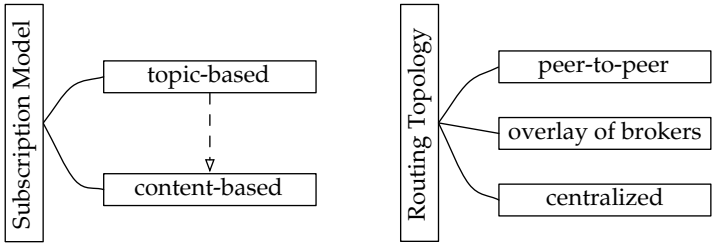


Figure 2.3: Two common taxonomies of PUB/SUB systems. (a) Classification according to the subscription model (the dashed arrow emphasizes the fact that the topic-based model can be seen as specialization of the content-based). (b) Classification based on sample routing topology.

according to the way subscribers express their subscriptions (i.e., their *subscription model*) [34, 49, 51], while the second, architectural-level, taxonomy classifies them based on their distributed deployment (i.e., their *routing topology*) [45, 46]. The two taxonomies are complementary to understand the characteristics of a PUB/SUB system: while the first has a direct impact on the high-level interface of the middleware, the second is useful to correlate these features to lower level and implementation details.

The *subscription model* taxonomy (Figure 2.3a) organizes systems in two main classes[2]:

- *Topic-based*. In topic-based (sometimes also called *subject-based*) PUB/SUB middleware, one of the sample key-value pairs, called topic, is given a special meaning and subscribers specify their interests by providing an equality predicate on the topic value.
- *Content-based*. In content-based PUB/SUB, subscribers can create complex predicates using all the samples fields or only part of them (in the last case they are also referred to as *header-based*).

Topic-based PUB/SUB systems are very diffused in both industry [62–64] and academia [65–68] mainly because of their simplicity; by introducing topics, PUB/SUB systems partition subscribers into groups, each associated to a topic, and including all and only the participants who have subscribed to it. For this reason, the efficient routing of topic-based samples is equivalent to the well-known problem of multicast messages distribution [69]. In contrast, *content-based* subscriptions (e.g., [57, 58, 60, 61, 70]) give participants more flexibility, but can complicate PUB/SUB matching and routing functions [71].

The *routing topology* taxonomy, instead, identifies three main system organization alternatives, as shown in Figure 2.3b.

2 In the literature other classes have been identified, e.g., *type-based* [53–56], or *context-aware* [57–61] systems. Here we omit them because they can be easily mapped on the content-based model, and because their specific peculiarities do not have direct relations with the implementation of quality-aware behavior.

- *Centralized Broker*. In centralized PUB/SUB systems, a conceptually central entity, called *broker*, stores and manages all the subscriptions and acts as the unique data dispatching component.
- *Overlay of Brokers*. The system is organized as a network of peer brokers, i.e., infrastructure nodes realizing the PUB/SUB functions. With this organization, participants connect to one broker that acts as their proxy to the middleware; sample matching and routing are performed in terms of distributed algorithms over the broker network, potentially improving system scalability.
- *Peer to Peer*. Differently from the overlay of brokers network organization, where, even though brokers themselves constitute a PEER TO PEER (P2P) network, the interactions between brokers and participants follow a client/server model, in fully distributed peer-to-peer architectures, there are no intermediary nodes between participants and samples flow directly from publishers to subscribers. All the matching and routing responsibilities are carried on by participants themselves.

Centralized architectures, such as [67] or [72], do not scale well when either the number of participants or the publication rate significantly grow since the broker server has to keep (at least partial) state for all its clients and has to process and route all samples. Notwithstanding that, one central entity with global knowledge of the PUB/SUB network allows easier administration of the available resources and, hence, a simpler implementation of QoS properties.

Topologies based on overlay of brokers [57, 58, 73, 74] enable, in general, improved scalability because the responsibility of matching and routing decisions is spread across the overlay nodes [75]. The presence of a distributed infrastructure allows to leverage geographic and semantic proximity of nodes to optimize routing paths or to implement in-network filtering of samples [56, 58] at the expense of increased complexity in system management and of worse dissemination latency caused by the overhead of middleware-level routing.

P2P topologies [55, 76, 77] are best suited for the dissemination of samples between a limited number of participants and in a small scale geographical deployment. They potentially guarantee small latencies and high throughput given the absence of intermediate hops and thanks to the exploitation of techniques leveraging the co-locality of participants, such as multicast-based message dissemination.

2.2.2 QoS-aware model

The basic notification space model presented in the previous section was intentionally used to represent only the *functional* features commonly offered by PUB/SUB systems. However, in many application scenarios it can be important to have visibility and control over more implementation-related system details, i.e., its *non-functional* aspects.

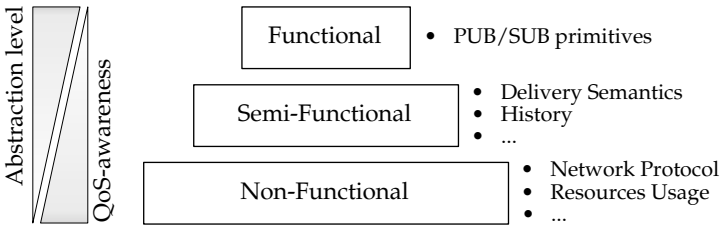


Figure 2.4: QoS specifications configure non-functional properties. By settings what properties are configurable, designers control the level of abstraction offered.

Let us emphasize that functional and non-functional behavior are correlated: choices taken at one level impose constraints or at least guide design/implementation decisions at the other.

This relation is often so strong that it becomes very difficult to draw a line to perfectly separate features of real systems into two different categories. Consider, for instance, delivery reliability. Under one perspective, it could be considered a functional aspect because it alters the semantics of how information is delivered to participants (receive all the samples of interest rather than just a subset of them); from a different point of view, instead, reliability could be seen more as an implementation-related feature that sometimes participants could even be unaware of. We believe that this is the natural consequence of different application scenarios having very different requirements and thus needing diverse levels of visibility and control over the communication process. For this reason, we see the features offered by PUB/SUB middleware better organized as a continuum of levels of increasing abstraction: at the top there are features that are part of the operational semantics; the bottom level includes design/implementation details, sometimes platform-dependent, that strongly affect the performance results achievable.

In this vision, a central role is played by QoS specifications and their emerging visibility. Different specifications can be used to tune, with different granularity, configuration details that permit to achieve the desired level of quality. By using QoS specifications, users are willing to sacrifice transparency from implementation-oriented details (and the accompanying ease of use) to have a deeper control of the mechanisms that realize abstract services in return. PUB/SUB systems designers, by choosing the set of QoS properties to expose, decide the set of low-level configuration aspects exposed to their users, practically setting the maximum visibility they have on middleware non-functional features. Figure 2.4 summarizes this concept; for the sake of representation clarity, in the figure, we have grouped PUB/SUB properties in three groups; the *semi-functional* level collects system properties that represent intermediate abstractions between full visibility of implementation details, the lowest level, and "pure" PUB/SUB semantics, the highest one.

In QoS-based services, the involved parties usually perform a *quality agreement* process to determine the service level to be provisioned at runtime [78]. This process, in the context of PUB/SUB systems, has been modeled through a OFFER – REQUEST (O-R) pattern in the past [49, 63], according to which publishers offer a set of supported quality properties, and subscribers choose the desired service level among these possibilities. We claim that this basic agreement model misses to properly emphasize the fundamental role of the middleware, which must decide whether to dedicate resources, possibly distributed, to QoS provisioning and confirm its decision to clients; hence we explicitly refer to a generalized three-way OFFER – REQUEST – CONFIRM (O-R-C) agreement model in all the cases where middleware resource management is critical in order to achieve the required quality.

To represent QoS agreements and enforcement in the proposed notification space model, we exploit its optional non-functional dimensions, that we call, in this context, QoS *dimensions*. In the notification space, a QoS dimension q_i is associated with a QoS property and its admissible values represent possible values of the property. For instance, if we consider *delivery reliability*, possible discrete values are *best-effort* or *reliable*. The introduction of the QoS dimensions modifies the operations described in Section 2.2.1 as follows. *Publish* actions now allow the publishing participant to specify QoS parameters that influence the delivery of its generated samples. When publishing data, it can specify a set of values for each QoS dimension; this will represent its QoS offer. Note that, now, a publish action corresponds to multiple points in the notification space. With *subscribe* actions, subscribers can request a QoS-level for the delivery of samples belonging to the subscribed space: this is achieved by extending subscriptions with an additional filter made of equality constraints on the QoS dimensions. Together with the subscription filter, these constraints identify a subspace of the extended notification space. *Advertise* actions are similarly modified and permit publishers to declare the QoS properties they intend to offer beforehand. Within this extended model, we say that a sample *matches* a subscription if one of the notifications generated by the sample publishing action fall within the subscription subspace *and* — at the same time — if the middleware confirms the QoS agreement.

In the following subsection, we survey and classify a selection of QoS properties for PUB/SUB data delivery that we consider central for PUB/SUB middleware design/implementation choices. This selection has a twofold rationale: on the one hand, we aim at representing what is most commonly supported in state-of-the-art academic and industrial PUB/SUB systems. On the other hand, we include also a few QoS properties that, although uncommon in currently widespread solutions, we believe will be central in the design of next generation PUB/SUB architectures.

2.2.3 Classification of QoS properties

We propose a clean and simple three dimensional classification scheme of QoS properties that emphasizes common features, trade-offs, and design/implementation problems. The three dimensions look at QoS properties each from a different perspective:

- *Granularity*. Is the property enforceable with the granularity of a single publication, or does it make sense only considering sequences of samples (i.e., *flows*)?
- *Agreement mode*. Does the QoS property require O-R-C agreements or does it involve only the end parties (O-R)? In other words, does the enforcement of the property require resources from distributed components other than participants?
- *Quality Domain*. What aspect of the PUB/SUB functionality does the property regulate?

The *granularity* dimension analyzes the scope of enforcement of a QoS property. Many QoS properties cannot be defined in the context of an isolated sample. For instance, consider a subscriber that needs to receive data at fixed intervals and wants to be sure that publisher and middleware can satisfy its requirement: the concept of period intrinsically relates to a sequence of samples and cannot be properly applied to isolated ones. We identify two types of granularity:

- *Sample Granularity*.
- *Flow Granularity*.

The former includes QoS properties that are defined with respect to the delivery of a single sample; the latter, instead, comprises the properties that *only* apply to temporal sequence of samples, or *flows*.

The *agreement mode* specifies which quality agreement process (O-R or O-R-C) is required to establish and enforce the property at runtime. Let us remark again that the most important difference between properties agreed with or without middleware involvement is that they may require additional resources from intermediary components other than source and destination endpoints. It is easy to understand that it is critical to accurately consider this difference during the design of quality-supporting mechanisms.

Finally but not less significantly, the *quality domain* dimension classifies QoS properties according to the specific aspect of data distribution they regulate. We further identify five groups within it:

- *Delivery semantics*.
- *Time-related constraints*.
- *Persistence*.
- *Adaption indications*.
- *Security*.

Table 2.1: Three-dimensional taxonomy of PUB/SUB QoS properties. (LH: Limited History; FH: Full History).

	Sample Granularity		Flow Granularity	
	O-R	O-R-C	O-R	O-R-C
Delivery	Reliability[a]	Reliability[a]	Ordering	—
	Uniqueness[a]	Uniqueness[a]		
Time Related	Latency	Lifespan	—	Periodicity
		Freshness		
Persistence	Not Ack. (net.)	Not Ack. (client)	LH (net.)	LH (client)
			FH (net.)	FH (client)
Adaption Ind.	—	Priorities	—	Probabilistic
Security	Confidentiality[a]	Integrity[a]	—	—
	Confidentiality[a]	Integrity[a]		

[a] The agreement model is implementation-specific (see the property description for details).

The boundaries between these categories are sometimes weak, as QoS properties are often closely related to each other; for instance, this may occur because one property is a prerequisite of another, or because there is a trade-off between the two (e.g., ensuring one property might hinder the enforcement of the other). Furthermore, it is sometimes difficult to strictly confine a cross-concerning QoS property in only one group. In our presentation, we will stress situations where categories are partially overlapping.

Table 2.1 summarizes the QoS properties surveyed in the remainder of this section and their position within our simple three-level classification. For the sake of descriptive fluency, the following analysis groups the properties based on their *quality domain*.

Delivery Semantics

This class groups QoS properties that determine the characteristics (and related guarantees) of data delivery. Recalling Section 2.2.1, the basic PUB/SUB delivery guarantee is the following: if a sample is delivered to a subscriber, then at least one of its subscriptions will match the sample. We already emphasized that this means that a subscriber is not guaranteed to receive *all* the data of interest: the PUB/SUB middleware has the only duty of routing an arbitrary set of samples to it, ensuring that this set does not include non-matching samples. This is an undesirable and oversimplified behavior in many scenarios. What normally happens is that a PUB/SUB system tries to route all the samples with *best-effort* semantics. However, this is often insufficient for applications that need always-deterministic behavior. In the following of this subsection, we will discuss the QoS properties related to the extension of the above basic guarantees.

DELIVERY RELIABILITY We identify as *delivery reliability* properties the set of functions that try to ensure that *every sample is delivered to all the subscribers that have matching subscriptions*. This property is called *liveness* in [79, 80] and [51]. Liveness can be hindered by any kind of failure, in and between PUB/SUB system components (e.g. network, software, or hardware issues). In PUB/SUB implementations, replication of links, nodes, and messages is the well-known countermeasure to deal with these fault situations. The most commonly adopted techniques can be coarsely classified in two classes: probabilistic and deterministic. On the one hand, with probabilistic approaches (e.g., [81–83]), multiple copies of samples are sent through redundant nodes and links, increasing the chances that at least one copy arrives correctly at destination. On the other hand, with deterministic methods (e.g., [64, 84–86]), acknowledgments are used to detect when messages are lost and to trigger retransmissions. In PUB/SUB systems that route samples via an overlay of brokers, it might be also necessary to reconfigure and repair the overlay topology when transient or permanent failures of brokers occur [87–89]. Note that, when retransmission is used, some forms of message buffering and/or *persistence* (see also the following subsections) are needed at some point in order to make the data to retransmit available. Reliability influences the delivery of single samples (*sample granularity*), and *may* need the intervention of intermediate components, for instance, for in-network buffering and persistence [67, 80]. In such cases the agreement follows the O-R-C pattern.

UNIQUENESS With *uniqueness*, we identify the property that guarantees that subscribers receive samples no more than once. Several reasons can lead to duplicate deliveries: for instance, it can happen when data is routed through multiple paths for redundancy, or when retransmissions occur due to elapsed timeouts or lost acknowledgments. The simpler and probably most common approach is to detect duplicates at the destination by attaching a unique identifier to every sample and by having destination endpoints record the history of already received identifiers [55, 61]. Duplicate avoidance is performed at single publication-level (*sample granularity*) and may require resources from components other than publishers and subscribers, for example, if duplicates detection is performed along routing paths [83].

The combination of duplicate avoidance and guaranteed delivery can be used to single out three classes of delivery semantics. Traditionally, they have been used to describe the invocations in REMOTE PROCEDURE CALL (RPC) or REMOTE METHOD INVOCATION (RMI) supports, but their meaning is very easily extensible to the context of PUB/SUB. They are:

- *At most once*: duplicate samples are discarded, but data delivery is not guaranteed.
- *At least once*: delivery of samples is reliable, but subscribers could receive duplicate data.
- *Exactly once*: at-most-once and at-least-once semantics are satisfied at the same time.

ORDERING This property regulates the order in which subscribers receive samples. In the literature, a classification very similar to the one defined by Defago et al. [90] for ordered multicast is often used; this is scarcely surprising because the one-to-many dissemination of a sample can be modeled as a multicast message to all the subscribers having matching subscriptions.

- *No order*: a subscriber receives samples in any order, regardless of their publication time.
- *Publisher order*: if one publisher s publishes sample e_1 before e_2, then every subscriber receiving both e_1 and e_2 will receive them in their publishing order. This type of ordering is often also referred to as First In, First Out (FIFO).
- *Causal order*: if the publication of a sample e_1 *happened before* the publication of e_2 (according to the *happened-before* relation defined in [91]), then every participant receiving both samples will receive e_1 before e_2.
- *Total order*: Given any two samples e_1 and e_2, if the system delivers e_1 before e_2 to any subscriber, then all the subscribers receiving both e_1 and e_2 will do that in the same order. Total order does not imply causal or publisher order.

Ordering has *flow granularity*. Generally, the implementation of publisher order does not involve parties other than publishers and subscribers (usually it is implemented leveraging publisher specific sample sequence numbers [55, 80]). However, implementing causal or total order may require additional resources possibly offered by the PUB/SUB middleware; thus, it usually requires a more expensive three-way agreement.

Time-related constraints

Samples are often generated in response to real-world phenomena that subscribers need to be informed about to perform proper reactions. For instance, a sample could represent an alarm ringing or the completion of a commercial transaction, or it could describe a periodic sensor reading. In this section we review the most relevant time-related QoS properties that PUB/SUB middleware can implement.

DELIVERY LATENCY The latency property expresses an upper bound on the time elapsed between sample production and delivery. If the

middleware guarantees to deliver samples with latency τ ($\tau > 0$), it means that if a publisher p publishes a sample e at time t_p, then any subscriber s interested in e will receive it within the time interval $(t_p, t_p + \tau]$. This property generally applies to single samples and requires the agreement of the PUB/SUB middleware to be granted because it needs the availability (and possible pre-allocation) of sufficient distributed resources [92].

LIFESPAN Sometimes also referred to as TIME TO LIVE (TTL), the lifespan property sets temporal frames of validity for samples. For instance, in an e-commerce scenario, a store may be willing to publish its commercial promotions, some of those having limited validity in time: after the deadline, to deliver related samples would induce a useless consumption of resources. A lifespan of l time units associated to a sample e means that if the sample is published at time t, then, after time $t + l$, the PUB/SUB middleware has the option to save some resources and not deliver it anymore [55, 64]. It is important to remark the difference between the latency and lifespan requirements: while the former expresses a hard constraint in terms of delivery delay, lifespan represents a publisher-side indication about the application-level validity of some data. This information is typically exploited to optimize the use of distributed resources. The property granularity is single sample. It is specified by publishers and not subject to negotiation by subscribers. The agreement associated with this property is always confirmed by the middleware because its enforcement does not consume resources.

FRESHNESS While lifespan expresses a publisher-side application requirement, *freshness* represents its subscriber-side counterpart. A subscriber can use this property to specify that it does not want to receive samples older than a time threshold f, where f is the value of the freshness property. As the lifespan property, a freshness specification can be useful to influence the PUB/SUB middleware decisions for what concerns the usage of its routing resources [55, 64]. For this reason, freshness requires a simple two-way agreement. Its granularity is at the level of single sample.

PERIODICITY In many scenarios, samples are generated periodically. This is the case, for example, of a publisher associated to a sensor making its readings available to interested destinations. The *periodicity* parameter expresses the expectations of publishers and subscribers with respect to the timing of periodic data. In particular, a publisher can associate an advertisement (and hence the corresponding flow) with a *publishing period* p_p: in this way, it commits itself to publish a new sample at most within p_p time units after its last publication. Conversely, a subscriber can request to receive new data at least once

every p_d seconds. Periodicity specifications can serve mainly for two purposes. On the one hand, they can provide information about the publication rate that the middleware can use for optimization purposes. On the other hand, they can be used to save resources: in fact, if the delivery period requested by a subscriber is significantly greater than the publishing period, the system could decide not to dispatch data that is not strictly necessary to satisfy the subscriber QoS requirements [55, 67]. Periodicity has flow granularity. If specified on the subscriber side, it does not require any middleware resource to be enforced; on the contrary, depending on its implementation, publisher side periodicity might require middleware confirmation to guarantee that the required data rate is sustainable.

Persistence

An important parameter affecting the QoS of PUB/SUB systems is persistence. A sample can be generated by a publisher, propagated toward its subscribers, and removed by an intermediary node as soon as it is routed toward the next hop. Although this is the cheapest solution in terms of resource consumption, it may be undesirable sometimes. In fact, the persistence cross-cutting concern is crucial in several scenarios: it may be necessary to realize *delivery reliability* through AUTOMATIC REPEAT-REQUEST (ARQ) mechanisms [93]; it may be used as the basic mechanism to implement an *history* that participants are allowed to browse [67], or it could be used to offer *permanent subscriptions* that allow publishers to receive samples published while they were off line [58, 64].

In defining persistence-related QoS properties, there are three main concerns to take into account, which deal respectively with the persistence mode, the persistent storage technique, and the distributed architecture that realizes the storage.

PERSISTENCE MODE Depending on the goal of persistence, we identify three main persistence modes. With *non-acknowledged only* persistence, samples are persisted until they are acknowledged by destination subscribers [64, 94]. This solution is mainly used for reliability purposes. Differently, with *limited history* persistence, only the last r samples generated by each publisher are persisted, where r is said to be the history size [55, 67]. Finally, a *full history* approach aims at saving the complete history of samples [55]. This last mode may appear equivalent to the *limited history* strategy when considering limited storage space. However, the difference between the two categories stands on a conceptual level: in *limited history* there is an explicit intention to preserve only recent history; full history persistence, instead, aims at keeping as much data as resource availability allows.

STORAGE TECHNIQUE This property characterizes the persistence services according to the storage techniques and supports used to make data persistent. With *in-memory* storage, samples are kept in a buffer allocated in main memory. If the storage node crashes, all the data it was responsible for are lost. On the contrary, if a *distributed cache* is used, persistence is obtained via main-memory storage replicated across a set of nodes[3]. In this way, even though each process keeps data in volatile memory, the storage as a whole is resilient to data losses. Finally, samples can also be saved in stable storage, relational databases [95], or distributed non-relational systems [96].

STORAGE ARCHITECTURE The storage architecture of a PUB/SUB system determines how the storage solution is implemented on its distributed infrastructure. A fundamental decision is where to deploy the persistence service and where to keep the persistent data. If data are saved at the publisher side [55, 84, 85] the responsibility of retransmission or retrieval of old samples always involves the original publisher, who must be on line to make its data available. Moreover, the capacity of the persistent storage for a publisher is strongly coupled to its resource availability. On the opposite, with subscriber side persistence [84, 85, 97], samples are stored on the subscriber nodes. This allows faster access to old data and relieves applications from the duty of implementing their own storage. However, this solution is not appropriate to implement middleware-driven retransmission policies and persistent subscriptions. Finally, samples can be stored *in the network* [58, 80, 97], i.e., data is persisted by the middleware infrastructure of intermediate nodes so that no responsibility is given to participants, which are free to connect and disconnect from the system without compromising data availability. This is probably the most common solution in industry-adopted architectures.

Limited or full history persistence are only applicable to flows, while persisting only non-acknowledged data is meaningful with single sample granularity as well. The agreement type strongly depends on the implementation of the persistence strategy and in particular on data placement: for instance, in-network persistence requires the middleware to grant sufficient distributed storage resources; differently, if storing responsibility is held only by the end parties, the O-R agreement is sufficient.

Adaption indication

Adaption indications are a special class of QoS properties that do not influence low-level implementation mechanisms directly, but are

3 These nodes can be either be PUB/SUB participants, routing elements, or dedicated components.

rather high-level concise properties that guide the dynamically adaptive behavior of the PUB/SUB middleware infrastructure. These QoS properties are used to let PUB/SUB administrators/deployers express their requirements by means of simple and coarse-grained indications and to let systems configure their inner mechanisms accordingly.

PRIORITIES In many systems, subscribers or publishers can specify priorities for their data [55, 64, 95]. A priority specification is an integer value chosen from an arbitrary (and usually very limited) range of possible values (e.g., ten priority levels). Thanks to priority specifications, the middleware can perform differentiated management of samples based on their relative importance. Possible implementations of the priority idea in PUB/SUB systems can involve differentiated routing queues for WEIGHTED FAIR QUEUEING (WFQ) algorithms [98]. Another idea is to use priorities as hints to decide which samples to sacrifice in case other QoS requirements (e.g., delivery latency) can no longer be satisfied. Priorities have sample granularity, and, since priorities represent hints rather than constraints, their enforcement does not usually require an O-R-C agreement.

PROBABILISTIC SPECIFICATIONS Probabilistic specifications can enhance the semantics of QoS properties by allowing to reformulate requirements and offers in probabilistic rather than deterministic terms. Consider, for instance, reliability or latency properties: in a deterministic view, reliability guarantees that the ratio between the number of unique samples published and the number of interested subscribers receiving them is exactly one, while latency ensures that exactly every sample is received within a given time interval. The idea of probabilistic QoS starts from the consideration that a PUB/SUB system could benefit from the opportunity of being requested weaker and non-strictly mandatory quality specifications. Going back to the reliability and latency examples, a subscriber could request the delivery of some kind of sample to be guaranteed with probability $p < 1$, or it could specify a CUMULATIVE DISTRIBUTION FUNCTION (CDF) that describes the desired behavior for data distribution latency [81–83]. The main advantage of supporting this kind of specification is the derived flexibility in the implementation of QoS mechanisms, which permit a smarter and less constrained exploitation of resources at runtime. For instance, in the case of reliability, if at some point 100% of samples have been delivered correctly while a 90% success rate was requested, the middleware could voluntary drop some data, e.g., to save resources to dedicate to another resource-greedy flow.

Security

Security is a relevant non-functional aspect which contributes to determine, to some extent, the quality provided by a PUB/SUB system.

For the sake of briefness and in order to keep the focus of the survey well targeted, we consider security aspects of PUB/SUB middleware not central to the scope of this chapter. Nonetheless, given the criticality of these concerns in production scenarios, it is worth to report here a concise overview of the most relevant security properties emerging from the literature. PUB/SUB security relates to two main problems: creating mechanisms that let subscribers verify the authenticity of samples, and allowing the production and dispatching of samples that can be read only by authorized subscribers. These two aspects correspond to the classical security concerns of *integrity* and *confidentiality*, respectively.

The effective implementation of security mechanisms is still an open problem in PUB/SUB research. The main difficulty rises from the fact that achieving integrity and confidentiality partially contrast with the decoupling principles of PUB/SUB delivery. Integrity, in fact, assumes that a subscriber knows the publisher in order to be able to check authenticity of samples. Similarly, participants must share some secret for easy solutions to confidentiality. Moreover, encryption makes routing data based on their content a very hard task.

In general, both integrity and confidentiality can be established with sample granularity. For both, the type of agreement depends on whether PUB/SUB intermediaries, if present, need to dedicate computing resources to enable secure communication. If, for instance, secrets are exchanged by interacting parties with out-of-band mechanisms, then, obviously, the involvement of PUB/SUB middleware components is not necessary for the agreement. Other solutions, such as [99], allow routing nodes to perform content matching in a way that keeps data undisclosed at the price of increased processing complexity: in that case a three-way agreement ensures that it is possible to allocate sufficient resources for the required security features.

2.3 SYSTEMS SURVEY

In this section, we provide an in-depth technical overview of how quality of service properties have been successfully implemented in relevant and state-of-the-art architectures for PUB/SUB data distribution over the last years. Given the important amount of work available in the field and in order to keep this work concise and focused, our technical investigation is based on a thorough analysis of a short selection of representative examples from industry and academia. Three main goals have guided this selection process:

- Choose systems that support a representative selection of most relevant, diffused, and original PUB/SUB QoS properties.

Table 2.2: List of surveyed PUB/SUB systems.

System	Subsection
Data Turbine (RBNB) [67, 86]	2.3.1
Java Message Service (JMS) [64, 100]	2.3.1
DCRD [83]	2.3.2
IndiQoS [92, 101]	2.3.2
Costa et al. [84, 85]	2.3.3
Gryphon [80, 93, 102]	2.3.3
Data Distribution Service (DDS) [55, 63]	2.3.4
STEAM [76, 103]	2.3.4

Table 2.3: Classification of surveyed PUB/SUB systems.

	Centralized	Overlay	Peer-to-peer
Topic based	RBNB	DCRD	DDS
	JMS	IndiQoS	STEAM
Content based	–	Costa et al.	–
		Gryphon	

- Present significant architectures that allow us to discuss problems and trade-offs for the achievement of quality aware data distribution.
- Cover as many application and deployment scenarios as possible, ranging from small ad-hoc networks, to large wide area networks with fixed and mobile participants.

While we recognize the important influence that some well known, seminal PUB/SUB architectures (e.g., Hermes [56], PADRES [70], REBECA [60], or SIENA [57]) had on the design of some aspects of the PUB/SUB solutions discussed in this section (for example, the basic ideas of their routing algorithms), we decided not to include them in our systems selection because they do not fully satisfy the rationale stated above by lacking, most significantly, the implementation of original quality-aware strategies for data distribution.

To facilitate the discussion and the comparison between different systems, we organize this survey in four parts, each corresponding to a different cell in the bi-dimensional grid that organizes systems according to their subscription model and routing topology. Table 2.2 lists the systems surveyed in this chapter, while Table 2.3 shows their position within the grid that organizes this section.

This thorough system analysis will pose the basis for the discussion in Section 2.4, where, starting from the comparison of the selected systems and extending the analysis to other similar architectures, we will derive general principles about the trade-offs in the practical implementation of QoS in PUB/SUB data dissemination.

2.3.1 Centralized and topic based

The first group of systems includes those using a central broker to perform routing and management tasks and having a topic-based data model. This is certainly the simplest logical organization considered in this section. However, we show that the simplicity behind this organization favors an easier implementation of sophisticate quality guarantees. The systems that we describe in this group are Data Turbine [67, 86] and the JAVA MESSAGE SERVICE (JMS) [64, 100].

Data Turbine (RBNB)

Data Turbine, based on the RING BUFFER NETWORK BUS (RBNB) architecture, is a middleware for the dissemination of data originated by distributed and stationary sensors; it is used in production in several environment monitoring projects. Data publishers organize their samples into RBNB channels, each being a logical container for a data stream from one source. The RBNB subscription model is a restriction of the topic-based one, where a topic is mapped one-to-one to a data source. Data Turbine is designed around a centralized architecture, with the RBNB server at its core: samples flow from sources to the server and from the server to sinks. Federations of servers is possible trough so called *routing* or *mirroring* configurations.

A central feature of the Data Turbine solution, is its support for ring buffer persistence. As servers receive data from their clients, they store the last n samples received on each channel on RING BUFFER OBJECTS (RBOs), special data structures implemented as a mix of in-memory and secondary storage. All the communications channels offer exactly-once delivery semantics, and some basic support for timely sample delivery is available via *monitor* subscriptions, which guarantee that subscribers always receive "sufficiently" recent data by automatically (and arbitrarily) dropping unnecessary samples. Samples are delivered according to publisher order thanks to explicit timestamps assigned by sources.

Java Message Service (JMS)

The JAVA MESSAGE SERVICE (JMS) is a standard defined in the JAVA SPECIFICATION REQUEST (JSR) 914 [100] and part of the Java Platform, Enterprise Edition (Java EE). It defines an APPLICATION PROGRAMMING INTERFACE (API) for the implementation of message-oriented middleware targeted at business enterprise applications. JMS defines point-to-point and PUB/SUB communication models. Given the scope of this chapter, we focus only on the latter. The JMS architecture is based on the concept of message broker, which mediates all the actions of PUB/SUB participants. The specification does not enforce any deployment architecture for JMS brokers, so it is up to the different vendors to implement more or less complex solutions.

Although very simple in its data model and in the topologies used by many of its implementations, the JMS standard supports a rich range of QoS parameters, demonstrating the strong request for quality guarantees coming from the enterprise realities that JMS targets. The delivery semantics can be influenced by controlling the acknowledging behavior of the underlying distribution protocol and by deciding whether samples are saved in persistent storage or main memory before delivery to participants. *Durable subscriptions* set up the system for storing samples so that off-line subscribers can receive them once the are again on line. JMS also supports priority indications, specified with single sample granularity; however, the standard does not require that JMS-compliant implementations actually use these values. Finally, through the TTL property, publishers can set a sample expiration time, which can be exploited by the JMS broker to avoid delivering messages that are no longer valid.

2.3.2 Overlay and topic based

Topologies based on overlays of brokers try to overcome the scalability limitations introduced by the presence of a single broker. The price to pay is the increased complexity in their routing algorithms, which also leads to more complex mechanisms for QoS enforcement. For the systems discussed in this group, this complexity is partially reduced thanks to the fact that only one data dimension, i.e., the topic, is used for routing purposes. The systems chosen to represent this category are DCRD [83] and IndiQoS [92, 101].

DCRD

The Delay-Cognizant Reliable Delivery (DCRD) protocol is a data delivery protocol for PUB/SUB broker overlay networks that, similarly to [81] and [82], try to maximize the probability of timely delivery with respect to user-specified latency requirements despite possible link failures or packet loss. DCRD is designed to work on topic-based PUB/SUB systems and does not rely on any particular structural organization of the broker network.

In DCRD, periodically, every broker measures and stores, for any possible link–destination pair (l, d), the routing delay and delivery ratio expected (in probabilistic terms) in case a sample for d is routed via l. Whenever a broker receives a sample to dispatch, it first filters out all its output links whose expected routing delay is greater than the time left to deliver the sample, and forwards it on the link that minimizes the ratio between the expected delay and delivery ratio. The authors show that this choice is optimal with respect to the protocol goals. No QoS parameter other than latency is supported, but it is easy to imagine an extension that supports the probabilistic specification of the desired level of reliability.

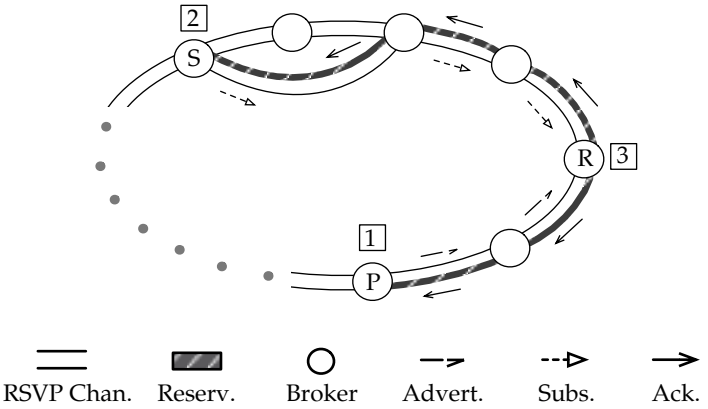


Figure 2.5: IndiQoS reservation process in a simplified DHT overlay. 1. The publisher broker P sends an advertisement message to the rendezvous node R. 2. The subscriber broker S sends its subscription to R. 3. R sends back acknowledgment messages that reserve resources along its path.

IndiQoS

The main goal of the IndiQoS system is to provide QoS-aware routing in type-based PUB/SUB systems. Publishers must announce the characteristics of their future publications through advertisements that include a QoS profile representing the offered quality level; symmetrically, subscriptions include participants QoS requests. IndiQoS topology is built over a graph of brokers organized according to a DISTRIBUTED HASH TABLE (DHT) [104], which is used to route advertisement and subscription requests, and it uses what is commonly known as *rendezvous* routing [56, 66, 68] to dynamically build non-optimal sample distribution multicast trees rooted at the brokers of publishing participants.

IndiQoS focuses on timeliness-related QoS parameters and in particular to low-level features influencing delivery latency, all enforced with flow granularity. The system leverages the INTEGRATED SERVICES (INTSERV) architecture [105] and the RESOURCE RESERVATION PROTOCOL (RSVP) [106] to reserve enough resources along the data distribution branches sot that the required latency dissemination bounds are met (Figure 2.5).

2.3.3 Overlay and content based

Overlays of brokers that route data based on all content fields provide maximum flexibility to users and maximum decoupling between PUB/SUB participants. However, the absence of a dimension that, like the topic, partitions data consumers in disjoint groups makes the implementation of QoS features a very hard technical challenge. We discuss the work by Costa et al. [84, 85] and the Gryphon system

[80, 93, 102] as examples of implementation of QoS-mechanism in overlay and content-based PUB/SUB networks.

Costa et al.

Costa et al. describe an approach that realizes reliable data delivery in content-based PUB/SUB through epidemic-style protocols, designed to work on top unreliable routing protocols based on subscription forwarding [57]. The goal is to provide *at least once* delivery semantics in spite of message loss and link or broker failures.

The proposal includes three protocol variants. All of them assume that every broker keeps a limited history of the last routed samples. The first variant implements a proactive and push-based recovery strategy, in which brokers periodically gossip the list of samples in their local history and can respond, upon receiving these messages, by requesting samples that they are missing. Differently, the second and third variants implement a reactive and local loss detection strategy, in which brokers detect missing data by looking at the sequence of sample identifiers. Periodically, they try to retrieve those missing samples by propagating ad-hoc recovery gossip messages. The two proposed strategies differ in the direction along which gossips are propagated: the second variant directs requests toward the publisher (increasing the chances to find valid copies of the sample), while the third sends them toward subscribers, potentially reducing the pressure on "popular" publishers.

Gryphon

Gryphon is a PUB/SUB middleware by IBM Research putting together the results of several efforts in different sub-areas of content-based routing. Gryphon models flows through *information flow graphs* that describe the logical topology of the data distribution network.

In the original Gryphon proposal [93], data dissemination is best-effort. A later article [80] proposes a variant of the system supporting publisher-order and exactly-once delivery semantics despite failures of links and brokers. The fault-tolerant architecture is based on a synchronous abstract model built around the concepts of *knowledge* and *curiosity*, which extend the *information flow graph* model. This model is implemented in the Guaranteed Delivery (GD) protocol. In particular, the combination of knowledge and curiosity states is implemented through a system of Acknowledgement (ACK) and Negative Acknowledgement (NACK) messages that propagate in the information graph triggered by two configurable timeouts (the Gap Curiosity Threshold (GCT) and the Ack Expected Threshold (AET)). By regulating the duration of GCT and AET, a combination of *publisher-driven* and *subscriber-driven* liveness can be obtained.

2.3.4 Peer-to-peer and topic based

This last group of systems includes two examples of PUB/SUB middleware architectures where data flows from participant to participant without the mediation of any infrastructure. Both these systems follow a topic-based data model: we are not aware of P2P systems realizing relevant QoS-related behaviors adopting a content-based subscription model. The systems we review are the OBJECT MANAGEMENT GROUP (OMG) DATA DISTRIBUTION SERVICE (DDS) [55, 63, 107] and STEAM [76, 103].

Data Distribution Service (DDS)

DDS is an OMG standard that defines the interfaces and the behavior of PUB/SUB middleware for soft real-time and mission-critical applications. Central to the DDS middleware architecture is the concept of QoS that publishers and subscribers can use to specify their requirements and let the middleware adapt accordingly. The DDS specification is organized in different layers, each providing an increasingly higher level of abstraction to DDS users: we concentrate mainly on the DATA-CENTRIC PUBLISH SUBSCRIBE (DCPS) layer, which realizes the PUB/SUB semantics. The DCPS specification, being an open standard, does not mandate any specific topology for the deployment of DDS implementations, but RTI Connext DDS [108] (also known as RTI DDS) and PrismTech OpenSplice DDS [98], two of the DDS market leaders, rely on a completely decentralized peer-to-peer architecture that enables direct data exchange between participants.

All DDS system entities (e.g., topics, publishers, and subscribers) are associated with a list of QoS policies that regulate some of their behavioral aspects. The standard defines twenty-two policies, and vendors have added many others in their custom implementations. In this subsection we discuss those that we believe most significant.

Through the *history* QoS policy publishers and subscribers[4] decide whether to store locally only the last n samples received or all of them (up to a configurable resource limit). The *durability* policy determines whether subscribers will receive samples that were published before their creation, and reliable data delivery is achieved through the use of the *reliability* property. This property strongly interacts with the history policy, because the reliability protocol retrieves messages from the publisher history queue for retransmission purposes. The *presentation* policy can be used to configure ordering guarantees, supporting publisher order and a form of total-order based on publisher-side timestamps. About timeliness-related parameters, the *latency budged* and *transport priority* policies complementary regulate the urgency

4 The DDS specification uses the concepts of *data writer* and *data reader* to refer to the entities we defined as *publishers* and *subscribers* in this chapter. The specification uses the terms publisher and subscriber to refer to slightly different concepts.

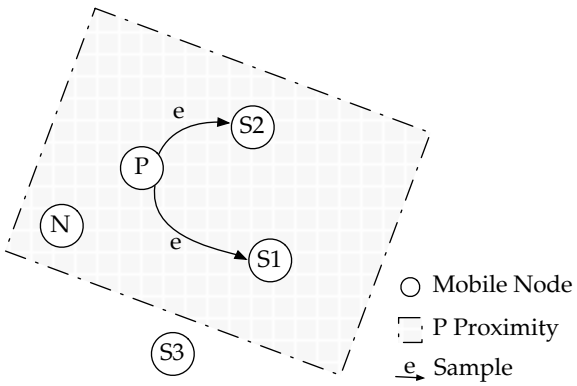


Figure 2.6: A STEAM publisher P publishes samples whose topic is of interest for S1, S2 and S3. However, only S1 and S2 are delivered *e*, since S3 is not physically located in the proximity defined by P.

and importance of data samples. In particular, the latency budget allows publishers and subscribers to express the maximum amount of time for a sample to be dispatched, i.e., its urgency, and the transport priority indicates the relative importance of a sample within those dispatched through the same transport. Publishers can also declare a validity interval for a sample, after which it is considered stale and discarded, through the *lifespan* property, while a *time based filter* policy specification allows subscribers to request a minimum separation interval between consecutive data deliveries. Finally, for periodic publications, the *deadline* property specifies the timing expectations of publishers and subscribers.

STEAM

STEAM is a PUB/SUB middleware designed for location-aware data distribution in wireless and mobile local area networks. Both publishers and subscribers are mobile nodes moving in the same physical environment and sharing information via PUB/SUB exchanges. A distinctive aspect of the system is that it allows expressing location-dependent subscriptions: as shown in Figure 2.6, publishers can define geographical *proximities* for their samples, and the system will deliver them only to subscribers physically located in their proximity. Subscriptions and data delivery are handled by a *proximity group* communication service [109] that permits to send multi-hop wireless multicast messages to nodes in the same proximity group.

STEAM delivery semantics, in the implementation described in [76], are basically best-effort, because the proximity group multicast is implemented with IP multicast. However, in [110], the authors propose to use STEAM on top of TBMAC, a time division and layer 2 protocol that provides deterministic upper bounds on the time needed to access the communication medium. Together with accurate estimations of the time required by subscription matching algorithms, this

Table 2.4: Summary of supported PUB/SUB QoS properties w.r.t. subscription model.

	Reliability	Persistence	Latency	Priorities	Fresh.
Topic Based	RBNB	RBNB	DCRD	JMS	JMS
	JMS	JMS	DDS	DDS	DDS
	DDS[a]	DDS[a]	STEAM		
			IndiQoS		
Content Based	Gryphon	—	—	—	—
	Costa et al.				

	Periodicity	Pub. Order	Causal Order	Total Order
Topic Based	RBNB	DDS	—	RBNB
	DDS			DDS
Content Based	—	Gryphon	—	—

[a] See system description for detailed semantics.

Table 2.5: Summary of supported PUB/SUB QoS properties w.r.t. routing topology.

	Reliability	Persistence	Latency	Priorities	Fresh.
Centralized	RBNB	RBNB	JMS	JMS	—
	JMS	JMS			
Overlay	Gryphon	—	IndiQoS	—	—
	Costa et al.		DCRD		
Peer-to-Peer	DDS[a]	DDS[a]	STEAM	DDS	DDS
			DDS		

	Periodicity	Pub. Order	Causal Order	Total Order
Centralized	RBNB	—	—	RBNB
Overlay	—	Gryphon	—	—
Peer-to-Peer	DDS	DDS	—	DDS

[a] See system description for detailed semantics.

extension enables the implementation of QoS mechanisms related to the *timeliness* of data delivery. For instance, [103] suggests a deadline-based scheduling algorithm. Sørensen et al. [111] leverage the original STEAM proposal to build CORTEX, a framework for context-aware PUB/SUB interactions. CORTEX includes a dedicated component for QoS admission control and provisioning, based on the Timely Computing Base (TCB) framework [112], which allows CORTEX to compute the maximum delivery latency expected to be supported [113].

2.4 DISCUSSION

In the previous section, we have surveyed the quality-related features of a selection of relevant QoS-aware PUB/SUB systems. Our primary goal was to point out the relationships between their deployment architecture, their routing techniques, and their support to different QoS parameters. Tables 2.4 and 2.5 summarize the main characteristics of these systems, and classify them according to their subscription model, their distributed architecture, and their QoS support.

In this section we try to sum up what emerged from our analysis, by emphasizing the practical design/implementation elements that characterize the complex set of trade-offs between middleware design and quality aspects. To make these trade-offs come forth clearly, we will analyze how certain design choices influence the QoS properties that can be easily (or feasibly) granted. To this purpose, whenever possible, we will also discuss possible extensions to the surveyed systems that could enrich them with additional QoS-related features, by carefully considering the need of a limited impact on their existing design and architecture. The remainder of this section is organized in five parts, each examining one of the different quality aspects central for PUB/SUB middleware with rich QoS support, i.e., *delivery semantics*, *persistence*, *latency*, *priorities & weak timing indications*, and *ordering*.

2.4.1 Delivery semantics

Our survey work shows that the most widely supported QoS property is undoubtedly *delivery reliability*. Many of the surveyed systems implement *guaranteed delivery* with exactly-once semantics through the use of application-level dissemination protocols with positive or negative *acknowledgment mechanisms*: this is the case of DataTurbine, Gryphon, JMS, and DDS. IndiQoS, instead, exploits TCP/IP links between brokers in their overlays to offer exactly-once semantics in spite of lossy communication channels; however, all of them do not take special countermeasures to deal with extended link or broker failures. DHT-based PUB/SUB overlays, such as the ones used by IndiQoS, can leverage the auto-reconfiguration features usually provided by DHT services to re-establish delivery paths in case of link or node failures; they could be easily extended with simple application-level ARQ protocols to issue retransmissions after broken paths are repaired, and thus obtain stronger reliability guarantees. If, instead, the overlay does not implement self-organizing features, proper overlay reconfiguration procedures need to be executed before retransmission protocols can operate [114].

We have seen that other forms of *weaker reliability* can be envisioned, and are actually implemented by some systems: the common aspect of these solutions is that, although they implement dedicated tech-

niques to increase the chance of sample delivery, they do not try to achieve 100% success ratio. In [81] and [115], for example, the probability of successful delivery is increased by using path and message *redundancy*. Costa et al., instead, exploit the inherent resilience of *gossip-based protocols* and uses epidemic-style message exchanges to retrieve lost samples through an out-of-band protocol. A very similar approach has been proposed by Malekpour et al. [97], which adopt a Bloom filters [116] based subscription matching algorithm to identify and re-route lost samples. A further approach, used in DCRD, leverages information about single-hop link reliability (inferred from historical data) to build *probabilistic reliability models* of the network that are exploited in their routing algorithms.

STEAM and its CORTEX [111] extension do not provide reliability-related guarantees, but use a special-purpose proximity-based group multicast to disseminate samples with best-effort semantics in MOBILE AD-HOC NETWORKS (MANETs). However, on top of it, reliable multicast protocols for ad-hoc networks could be adapted: many alternatives have been proposed in the literature, for example in [117] and [118].

2.4.2 Persistence

Only DataTurbine, JMS, and DDS offer QoS policies that control data persistence directly; the fact that all of them are widely used production systems highlights how the ability to explicitly specify persistence-related parameters is a central feature in real world scenarios.

Because of the strong relationship between *reliable delivery* and *persistence*, the majority of the systems offering some form of delivery reliability also implements dedicated mechanisms to persistently store published data. An exemplar case is the relation between DDS history and reliability policies: the size of the history kept by publishers closely interacts with the type of reliability requested by the endpoints to determine the delivery guarantees and the semantics of publisher operations. Similarly, in the JMS specification, the semantics of reliability properties and the provided delivery guarantees depend on the chosen type of persistence: for instance, if in-memory persistence is used, reliability is not guaranteed in case of server crash.

In PUB/SUB systems based on broker overlay topologies, the choice of the *storage architecture* can impact even more on reliability performance. For instance, the gossip protocols by Costa et al. leverage *in-network persistence* and show that distributing storage responsibilities on nodes other than end-participants can help implementing fast and lightweight reliability solutions. Similarly, Gryphon provide its broker nodes with limited history caches, used by intermediate brokers to reply to retransmission requests in place of source nodes, thus simplifying and alleviating their responsibilities. These examples emphasize another important aspect to consider while designing in-network

storage, i.e., the *placement of historical data*. Storing them close to their destinations allows faster retrieval of lost information but is generally more expensive because samples might need to be persisted on a larger number of branches to be accessible by all subscribers. On the contrary, keeping them close to their source increases the probability of them being on paths shared by several destinations, but at the same time it increases the expected retrieval time and the load on shared storage nodes. Cross-considerations using *sample priority* awareness could help in placement optimization: urgent data could benefit from being persisted close to destinations, while resources could be saved by storing less important samples near their sources.

2.4.3 Latency

In many mission-critical application scenarios a very important and desirable feature is the ability to *request and offer upper bounds on delivery latency*. However, especially in large scale systems, this is a very difficult goal to achieve. A first consideration is that the minimum delivery delay possible depends on four main factors: i) the *number of routing hops*, ii) the *algorithmic complexity of determining each routing step*, iii) the *geographical distance between source and destination*, and iv) the *efficiency in combining single-hop links*. While the contribution of the first three factors is likely to be relatively easy to estimate, several difficulties are associated with the fourth aspect. The problem strictly relates to the best-effort semantics of the underlying IP network and to its intrinsically unpredictable performance. This is the main reason why the majority of PUB/SUB solutions do not support *hard latency requirements*. The only example of this kind of support is provided by IndiQoS, which builds on the IntServ model to offer guaranteed timing bounds for its distribution service. In general, any PUB/SUB system based on brokers (either through a distributed infrastructure or a centralized organization) could support some forms of latency requirements via resource reservation on routing nodes. However, designers should consider that, in real world wide-area networks spanning over multiple domains, it is often impossible to control resource allocation, thus making this kind of approaches ineffective.

DCRD (and, similarly, [81] and [82]) recognize this intrinsic difficulty and adopt a more flexible approach that optimizes the probabilistic expectation of samples being delivered according to their latency requirements, by leveraging the knowledge of more easily measurable single-hop latencies. Analogously, CORTEX uses the model and services provided by TCB [113] to predict routing delay bounds.

Notwithstanding the recognized value of supporting bounded delivery latency, most PUB/SUB solutions operate *best-effort*. Peer-to-peer architectures, where the actual data exchange occurs directly between source/destination pairs without the help of intermediate compo-

nents, generally offer lower (although still not deterministic) routing delay and higher throughput. DDS, STEAM, and CORTEX are all examples of that. In addition, DDS enables the fine-grained configuration of low-level details of its transport protocol, thus providing ways to further influence its delivery latency.

2.4.4 Priorities and weak timing indications

Although very different in terms of semantics, QoS properties like *priority* or weak timing indications such as *freshness*, *lifespan*, or *periodicity*, can be effectively used for the common purpose of optimizing resource usage trough the exploitation of application-level awareness.

Both JMS and DDS support, in different variants, freshness, lifespan, and priority properties. DDS also lets publishers and subscribers specify periodicity offers and requests. DataTurbine offers an abstraction similar to subscriber-side periodicity through the notion of *monitor subscriptions*: the difference is that periodicity is not explicitly chosen by subscribers, but automatically determined by the middleware depending on network status.

Let us remark once again that an important common element of these properties is that PUB/SUB middleware usually exploits them as optimization hints rather than dissemination constraints. For this reason, we believe they could be beneficially implemented, with the necessary adroitness, on almost any PUB/SUB solution, independently on its specific routing architectures. For example, in STEAM, the membership of proximity groups could be dynamically modified according to the value of these properties. Information like priority level or expiration time could be exploited also to optimize batching and consolidation techniques used to aggregate multiple samples or acknowledgments in single transmissions, like Gryphon and JMS do.

2.4.5 Ordering

Some form of *ordering* is supported by almost all the surveyed PUB/SUB systems, but most solutions only implement simple variations of publisher order: in fact, since the publisher itself is a serialization point for all its samples, ordering enforcement is straightforward.

Only two solutions, i.e., DDS and DataTurbine, support totally ordered delivery by using timestamps generated at data sources. Vector clocks [91] or their variants could be adopted to realize causal order in small deployment environments with limited numbers of participants: however, as the size grows, the complexity associated with their use is likely to become unmanageable.

2.5 DIRECTIONS FOR FUTURE RESEARCH WORK

Relevant research results on PUB/SUB systems have been achieved in the last years, leading to a wide set of different solution and implementation guidelines. Many authors have recognized the importance of augmenting PUB/SUB middleware by means of QoS guarantees, and the variety of QoS-related configuration parameters supported in successful industrial standards like JMS and DDS clearly confirms the relevant need for QoS-based customization capabilities in enterprise scenarios. However, techniques for the effective implementation and deployment of PUB/SUB QoS have been only partially explored, especially for what concerns their scalability along the three dimensions of *volume of event data*, *number of participants*, and *network size*.

As motivated in Chapter 1, we believe that, in the near future, there will be a growing number of application scenarios where the possibility of using QoS handles to customize the behavior of the data distribution and processing infrastructures will be fundamental. Our analysis of the state-of-the-art PUB/SUB middleware has shown that, as long as the scale remains constrained to small and locally controlled networks, well-known techniques can be adopted to provide relatively strong QoS guarantees with predictable and limited cost. But as the scale grows and the possibility to have full control on physical resources decreases, the provisioning of deterministic QoS becomes harder and significantly more expensive. However, in our opinion, there are many applications that do not necessarily need to pay the full price of strong QoS guarantees, but could take high benefit from different and more flexible models that provide intermediate alternatives between *hard* QoS enforcement and no QoS at all. We claim that novel PUB/SUB QoS models and implementation techniques that leverage *soft and approximated* QoS indications are very promising for the near future. New systems should support the *relaxation of global and strict quality constraints*, and let users control the trade-off between scale, quality, and runtime dissemination costs by choosing the level of uncertainty that they can sustain. In this perspective, we envision two main research directions to improve current PUB/SUB systems through the exploitation of soft QoS constraints: *probabilistic QoS specifications* and *locality-aware QoS management*.

As it emerged from our survey, there is already an important and recent research trend that is starting to explore some first *probabilistic approaches to* QoS *modeling and provisioning* in PUB/SUB middleware [81, 83, 115, 119]. Expressing guarantees through probabilistic specifications allows systems to benefit from additional degrees of freedom in their event routing choices, which can be dynamically leveraged to optimize delivery quality and resource consumption. Developing new smart and optimized event routing techniques that exploit this improved flexibility is maybe the most relevant and promising re-

search direction to face, but many less obvious and nonetheless challenging related issues are open, including the need for efficient resource prediction algorithms and for effective models that map these predictions on reliable estimations of the service quality achievable.

Exploitation of locality is another central design principle that we believe should be extensively considered to develop PUB/SUB solutions suitable for the new scenarios of unprecedented large scale. This principle can be leveraged in many forms, with the common goal of *relaxing global system properties and constraints* in favor of simpler but more scalable local ones. For instance, visibility of data should be defined by taking into account the geographical co-location or the interaction patterns of PUB/SUB participants, and by defining layered logical domains, where lower layers have a more detailed view of local events but a narrower visibility scope, while upper layers have a more abstract and concise knowledge about a wider visibility scope. Similarly, we believe that QoS management and provisioning themselves should be based on locality considerations: these could be used to determine partitions of very large scale networks into smaller logical units, each managing its own independent QoS control and provisioning; in this way, *loosely-coupled* QoS *administration domains* could decide autonomously (and possibly dynamically) the extent of QoS guarantees to offer to their local participants, and dynamically negotiate the quality of inter-domain communication. Where necessary, this idea of *federated* QoS *management* can be further evolved to the more complex perspective of *multi-level* QoS *management*, where big independent domains can be further partitioned by need.

2.6 SUMMARY AND CONCLUSIONS

In this chapter, we claim that the PUB/SUB communication paradigm can properly satisfy the strong requirements of *interoperability* and *scalability* that emerge from SPEs if properly extended with the possibility to configure QoS parameters that regulate the behavior of the underlying data distribution infrastructure. To support this claim, we propose a model that unifies the functional and non-functional aspects of PUB/SUB behavior under a single *notification space* abstraction. In addition, we survey and classify a selection of common QoS properties supported and supportable by PUB/SUB middleware implementations, emphasizing the technical aspects that regulate the trade-offs between strong quality guarantees and system scalability. In particular, we propose a detailed survey of eight PUB/SUB solutions that emerged in the literature for their support to QoS-aware interactions. Our survey offers several technical insights about the extent to which QoS is explicitly or implicitly supported in academic and industrial PUB/SUB solutions.

Our analysis shows that there are several open opportunities for future research, especially in the efficient implementation of differentiated QoS levels in large-scale scenarios. We envision two main promising research directions: i) the development of more flexible and adaptive models based on probabilistic QoS, and ii) a deeper exploitation of the locality principle for effective data distribution and QoS enforcement. Both these development directions are motivated by our claim that, in many modern real-world application scenarios, protocols, algorithms, and decisions based on global system knowledge should be often sacrificed in favor of mechanisms and techniques leveraging partial and incomplete knowledge with local focus, in order to enable scalable, even though approximated, management decisions and optimization.

3 QUALITY OF SERVICE IN DATA STREAMS PROCESSING

In the previous chapter, we have discussed QoS-aware PUB/SUB systems as a promising communication paradigm and architecture for the distribution of data in large scale SPEs. Delivered by the data distribution infrastructure, endless flows of sensing data need to be analyzed and transformed from row information to usable knowledge. However, the characteristics of these data pose several hard challenges that need to be solved. In the literature, the *three 'V's* mantra [120] has been used often to summarize and communicate effectively the unique features of these data (e.g., in [121–123]). The three 'V's stand respectively for *volume*, *velocity*, and *variety*. The *volume* of the data has two orthogonal consequences: i) data cannot be feasibly handled by a single storage or processing location but they need to be distributed across multiple sites; ii) it is not possible to store data for off-line analysis but they must be continuously processed on-the-fly. Tightly linked to this latter aspect is data *velocity*: new data is constantly produced at different and increasing rates; in this context, processing platforms must provide sustained data throughput for indefinite periods of time and handle expected and unexpected variations of the input data rates (i.e., load peaks) through dynamic load adaption mechanisms. Finally, data can be extremely heterogeneous in their representation, semantics, and value: processing platforms need to deal with this *variety* by supporting extensible sets of data formats and by providing customization mechanisms that can be used to adapt to scenario-specific data characteristics.

In the last ten years, distributed and scalable processing systems like MapReduce [24] or Dryad [25] have emerged and are having an important industrial success, also thanks to the wide availability of industrial-quality open-source implementations [30]. These solutions, which have been often referred to as Many-Tasks Computing (MTC) [124, 125] or Data-Intensive Scalable Computing (DISC) [126, 127] systems in the literature [128], share the common goal of processing large volumes of data by leveraging in parallel the distributed resources provided by commodity hardware in large data centers [21]. We believe that the reason for their success is twofold: on the one hand, by adopting a *shared-nothing* architecture [129], they provide easy scalability when the operations on the input can be partitioned in a number of independent tasks; on the other hand, they are usually programmed using familiar general-purpose languages that reduce the framework learning curve, making their adoption faster.

In this chapter, we concentrate on one particular category of DISC systems, i.e., DISTRIBUTED STREAM PROCESSING SYSTEMS (DSPSs). Differently from *batch-oriented* DISC systems, designed to analyze very big but fixed data sets, DSPSs have the primary goal of providing scalable tools for for processing continuous and theoretically unbounded flows of data, i.e., *data streams*. DSPSs represent promising architectures to address the data analysis requirements of the emerging large scale scenarios scenarios described in Chapter 1, where it is often necessary to produce near-real-time feedback in response to real world events. In such scenarios, DSPSs must handle a possibly very large number of incoming data streams, serving the needs of a usually high number of applications executing concurrently. We have seen that these applications normally expect very different quality levels from the data processing service. Think about an application that monitors road traffic to detect accidents [17] (see also Section 1.2) and about another managing distributed energy smart meters data [130]: while both applications require limited and controlled *latency*, it is very likely that the guarantees required by to the first are stronger and have priority on those given to the second.

Similarly to what we have done in Chapter 2 for our PUB/SUB systems analysis, in this chapter we discuss the state-of-the-art of DSPSs and their ability to provide configurable QoS to applications requiring it. The remainder of this chapter is organized as follows. In Section 3.1 we position our work by reviewing the contributions in the literature that, before us, have partly analyzed DSPSs and their quality-related characteristics. Then, in Section 3.3, we propose a general model that captures at the same time, functional DSPS features and QoS-related characteristics; in the same section, using a simple bi-dimensional classification, we present common or possible QoS properties that DSPS either do or can feasibly implement, and we emphasize their mutual interactions. Under the light of this classification, we survey, in Section 3.4, a collection of relevant DSPSs that emerged in the literature for their support to unique QoS-related features, and we discuss the related implementation/cost trade-offs in Section 3.5. Before concluding the chapter, in Section 3.6, we sum up the lessons learned from our surveying work and propose design and research directions that we envision for the development of future DSPSs.

3.1 POSITIONING OUR CONTRIBUTION

The problems and issues related to continuous data stream processing have been widely studied in the last decade, and the very recent industrial success of scalable systems for data-intensive applications of the last few years has created a renewed interest in this research topic. Several works have already tried to build common models and

to indicate design/implementation principles for stream processing systems, each focusing on different aspects of the problem space. In this section, we position the work presented in this chapter within the existing literature and we point out common aspects and unique original contributions of this work compared to previous material.

Sakr et al. [131] discuss state-of-the-art approaches for satisfying the requirements of data-intensive applications in virtualized data center environments and analyze the existing trade-offs between the achievement of guaranteed performance levels, including latency and consistency, and the associated runtime costs. Differently from us, their contribution focuses mainly on batch-oriented systems rather than streams-oriented ones. In addition, we also propose a simple but comprehensive QoS-aware model that provides a consistent framework that we use to compare DSPSs.

In their work [132], Cugola and Margara survey the large domain of Information Flow Processing (IFP) systems, i.e. every kind of information management system that deals with continuous flows of data, including active databases [133], Complex Event Processing (CEP) systems [134], and DSPSs. Within the last category, which is the subject of our work, the survey focuses especially on Data Stream Management Systems (DSMSs), which, as thoroughly explained in the next section, represent only a part of the larger DSPS domain. The authors present a very general model that, as the one we propose in this chapter, is based on a layered architecture. However, while their model aims at covering a heterogeneous set of very different system types and, for this reason, is very general, ours is specifically tailored to DSPSs, and tries to capture with a more precise detail their unique features also by focusing focus on their specific QoS-related characteristics.

A very recent book chapter by Sandra Geisler [135] provides a thorough overview of DSPS features and architectures. Like [132], the attention is mainly centered on DSMSs and the work misses to discuss some interesting aspects peculiar of recent scalable stream processing engines. Interestingly, the survey dedicates an entire section to the analysis of QoS in DSMSs, where the author proposes a simple distinction between *application-based* and *system-based* quality dimension, the first group relating to data processing reliability, the second to system-wide performance indicators. In our work, we do not use this classification perspective because we believe that such a distinction misses to capture the strong relationships between the implementation of data processing reliability and consistency mechanisms and the achievable platform runtime performance.

3.2 DATA-INTENSIVE SCALABLE COMPUTING

We use the term DATA-INTENSIVE SCALABLE COMPUTING (DISC) to refer to the wide class of systems for distributed data management and analysis that achieve scalability to data volumes by exploiting the parallelism opportunities offered by large cluster of computing nodes connected in high speed LANs.

Systems belonging to this class share four cardinal characteristics:

- Shared-nothing distributed architecture.
- Processing operations divided in smaller computational *tasks*.
- Execution of tasks across multiple cores and nodes.
- Minimized inter-task and inter-node communication.

Shared-nothing architectures describe distributed systems that share nothing else than (usually asynchronous) communication channels, as opposed to shared-memory or shared-disk architectures [129]. They are characterized by a simplified distributed organization and enable better scalability because they require less or no synchronization to access shared resources. In order to leverage at best the parallelization opportunities offered by this kind of architectures, DISC systems decompose their workload in small and distributable *tasks*. This can be done either by partitioning input data into independent groups (*data parallelism*), by partitioning subsequent processing steps (*pipelined parallelism*), or by combining the two strategies. Tasks are executed across all the computing resources in a way that tries to minimize their coordination. A similar goal is to isolate the execution of tasks as much as possible, with minimum inter-task data exchange; on a further-level, communication between different cluster nodes should be reduced with even greater care.

We propose a simple and original classification of DISC systems based on a two levels hierarchy (Figure 3.1) . The first classification level distinguishes between *batch-oriented* and *streams-oriented* systems. As the names suggest, batch-oriented systems are designed for off-line analyses of static data sets, while streams-oriented systems are built for handling endless flows of data and for executing *permanent queries* on them, i.e., queries that update their results as new data is available. While the optimization goal of the first group of systems is usually *high throughput*, in the second case it is also very desirable to obtain *low latency* responses [136]. The second classification dimension relates to the data model that systems adopt. In particular we consider *structured* versus *semi-structured* data models: structured models organize data according to formal representations, such as the relational one [137], and normally define more or less complex query algebras that are used to interrogate the available data; on the contrary, semi-structured models are based on more flexible data representations, like key-value bags, and usually allow the definition of

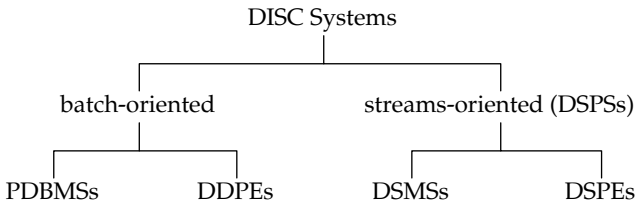


Figure 3.1: A two levels classification of DISC systems.

custom functions for data analysis, for example, built using general purpose programming languages.

The first examples of batch-oriented systems with a strong data model can be traced back to the late 80s [138, 139] with seminal PARALLEL DATA BASE MANAGEMENT SYSTEMS (PDBMSs), such as Gamma [140] or Teradata [141]. Based on the relational model, the core idea of these systems is to work on horizontally partitioned data tables and to rewrite and distribute STRUCTURED QUERY LANGUAGE (SQL) queries across these partitions. As today, modern commercial PDBMS implementations, such as Microsoft Parallel Data Warehouse [142], and the Database Partitioning Feature in IBM DB2 Enterprise Server [143] are still based on the same core concepts.

Batch-oriented systems with a semi-structured data model began to spread in the first decade of the 2000s. Their data model flexibility allows easier processing of semi-structured or unstructured data, and their user interfaces, usually based on main-stream general-purpose programming languages, promote a quicker developer adoption. We call this class of systems DISTRIBUTED DATA PROCESSING ENGINES (DDPEs). Compared to PDBMSs, DDPEs have a much stronger focus on boosting system scalability, often also at the expense of transactional properties — such as ATOMICITY, CONSISTENCY, ISOLATION, AND DURABILITY (ACID) — that, instead, are commonly supported by PDBMSs [144]. The MapReduce processing model, promoted by Google in [24], is probably the best known example of DDPE. In MapReduce, data is modeled as a collection of key-value pairs, and tasks, which can be of only two types — either *map* or *reduce* tasks —, define isolated computational units that process key-based subsets of input data. Also following the success of MapReduce, many other DDPEs have emerged. Some of them generalize the MapReduce concepts and offer processing structures that go beyond the simple map and reduce schema; some examples are Dryad [25], Nephele [26, 145], or Hyracks [146].

In the same period, another class of systems began to emerge, with contributions coming especially from the Data Bases community. These systems, called DATA STREAM MANAGEMENT SYSTEMS (DSMSs), recognize the different requirements posed by the management of continuous data flows as opposed to those offered by traditional or parallel DATA BASE MANAGEMENT SYSTEMS (DBMSs) [147]. Among those, there are the ability to act on *live* data by removing the requirement

of storing information before processing it, the ability to offer *minimal latency* by reducing the platform overhead, the creation of *formal models and query algebras* that define operations for streaming data, and the ability to provide *repeatable, consistent, and fault-tolerant processing* [31]. At the foundations of DSMSs there is the idea of letting users write permanent queries on input data streams by using ad-hoc query languages that usually resemble SQL [148–150], decompose complex queries into *query graphs* made of simpler and atomic tasks, and finally distribute components of the query graphs on cluster nodes [151, 152]. Noteworthy examples of DSMSs are TelegraphCQ [153, 154], STREAM [155, 156], and Aurora/Borealis [27, 157–159].

Finally, we call Distributed Stream Processing Engines (DSPEs) those streams-oriented architectures that do not follow a rigid data model or adopt a formal query algebra; similarly to DDPEs, these systems are better suited for scenarios where the information delivered by streams has weak or no structure. Differently from DSMSs, DSPEs do not have the realization of strong consistency properties among their main requirements, but they principally aim at offering theoretically unlimited scalability and minimal processing latency. Probably trying to exploit its widespread industrial success, many authors have proposed adaptations of the MapReduce model to continuous stream processing scenarios (e.g., [160–163]), usually based on the execution of sequences of MapReduce jobs on data stream windows. However, a large collection of literature have also proposed "standalone" DSPEs, with their own processing model and programming interfaces specifically designed for data streams analysis scenarios. As explained in more detail in the rest of this chapter, most of these solutions model stream processing problems as *processing graphs* made of user-defined *operators*, i.e., components implementing isolated and well defined functionalities and deployed as distributed tasks on data center resources. Given their enormous flexibility, these systems are perfectly suited to answer the novel requirements of SPE data processing applications, a fact that is also demonstrated by the widespread academic interest and industrial adoption that DSPEs are gathering in the last few years. Open source frameworks like Storm [29], Apache S4 [164], and Samza [165], or commercial solutions like IBM InfoSphere Streams [28, 166] are only a few examples of well-known and largely used DSPEs available today.

In the following, we concentrate on stream-oriented systems by analyzing in deeper detail their peculiar models and characteristics and by focusing on existing or possible quality-related features. Given the many common characteristics, we use the term DSPS to refer to DSMSs and DSPEs collectively: we use the more specific term only when we want to emphasize that some characteristic or property only applies to systems belonging to one of the two groups.

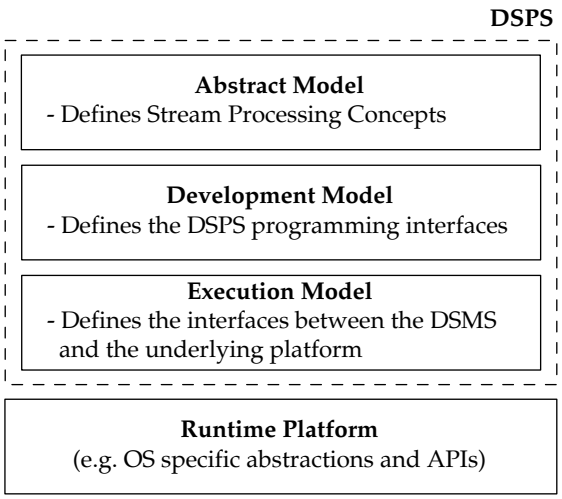


Figure 3.2: A three-layers model of Distributed Stream Processing Systems.

3.3 MODEL

In this section, we propose an original framework for the modeling of DSPSs. This framework helps to organize the description of DSPSs by emphasizing their common characteristics and unique features, and we will use it to this purpose throughout the rest of this thesis. In Section 3.3.1, we introduce the architecture of the model, based on three description layers — called respectively abstract, development, and execution layers — and, in Section 3.3.2, we enrich it with the ability to describe QoS-related features. Under the light of this discussion, we survey and classify, in Section 3.3.3, common or envisioned QoS properties for DSPSs.

3.3.1 Basic model

We propose an original representation model for DSPSs that helps analyzing them according to a simple schema based on three layers. The layers are complementary: each of them describes a different aspect of the stream processing system, and they are called *abstract model*, *development model*, and *execution model*, respectively (Figure 3.2).

- *The abstract model* defines high-level stream processing concepts. 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 given to developers to build the stream processing components of the abstract model. A development model, for example, could map system-specific concepts to syntactic constructs of special-purpose stream

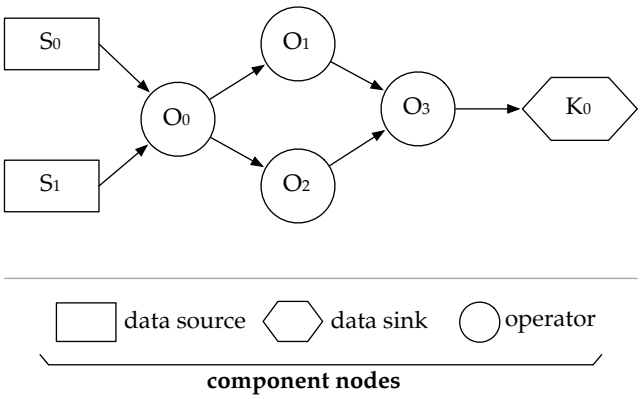


Figure 3.3: A DSPS processing graph.

processing languages, such as formal query languages, or to ad-hoc APIs and libraries for general-purpose languages.

- *The execution model* determines how abstract model components are mapped to runtime objects (e.g., OPERATING SYSTEM (OS) entities) that are then executed by the distributed hosts where the DSPS is deployed. For example, an application could be mapped on just one process of the host OS at execution time, or it could be split into several interacting processes.

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

Abstract model

The abstract model of a DSPS defines the high level concepts on which the system is based, including the system-dependent definitions of stream, stream processing application, and the processing work flow that the system adopts. While development and execution models usually present very significant differences among different systems, abstract models tend to be very similar and based on the common abstraction of *processing graph* (e.g., [27–29, 156, 157, 164, 166, 167]).

A processing graph (Figure 3.3) is a directed graph whose nodes represent data processing steps, and whose edges represent streams flowing between components. A *stream* is an unbounded sequence of discrete elements, often called *tuples* (more common in DSMSs) or *samples* (more common in DSPEs). 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 programming language, or, in some cases, to objects of an object oriented type system. A process-

ing graph is 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, a file, a network socket, a PUB/SUB endpoint, or a relational database. Since input streams are theoretically unbounded, a characterizing feature of stream processing applications is that, once started, they execute forever unless explicitly stopped. In some cases it is useful to reason about limited portions of streams: a finite temporal sequence of samples belonging to a stream is called *trace*.

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 generated outside the application: its role is to abstract from the actual nature of the stream producer, and it can represent either an external stream source or the output of another application running concurrently 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 or storage, or to connect the output of one 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 [150]; they can perform aggregation or filtering actions [168], or implement more complex and arbitrary USER DEFINED FUNCTIONS (UDFs) [26, 29]. 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 component performs its processing operations on data samples according to an asynchronous processing model; conceptually they all operate in parallel and perform their processing actions as soon as data samples are available at their input ports.

Development model

A development model maps the concepts defined in the abstract model to programming-level constructs that are used to program stream processing applications. These constructs should allow to:

1. Define new applications by describing sources, operators, sinks, and their connections in a processing graph.
2. Customize component instances by specializing their behavior for particular application needs (e.g., to bind graph source nodes to actual external sources).

3. Develop new components implementing UDFs (optional).

Any DSPS development model should at least define the tools to achieve the first two goals stated above; in fact, in many cases, it might be not necessary to create new or custom components, for example, because the system comes bundled with collections of ready-to-use components (often known as *toolkits* in DSPEs [168]), or because streaming queries are built via ad-hoc query languages, as it is often the case for DSMSs [150].

In the available literature, two families of application development models are common. The first includes the models where the mappings are based on *special-purpose languages*; the second family uses *general purpose languages*. The use of one model rather than the other is very often dependent on the system being a DSMS or a DSPE. However, DSPEs offering simplified stream processing languages [166, 168, 169] that can be used complementary with respect to their operator definition APIs are not uncommon.

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, STREAM [156] defines the Continuous Query Language (CQL), which permits to develop stream processing applications by writing continuous queries in a syntax that strongly resembles SQL. These queries are processed by the underlying system and decomposed in a processing graph of pre-defined operators. If applications written in an ad-hoc language are usually simpler to write, they lack of the flexibility of general-purpose languages and, more importantly, they require developers to learn new languages and new development processes.

Development models based on *general-purpose* languages, instead, have a less steep learning curve because system-specific stream processing concepts are defined through familiar constructs offered by main-stream programming languages, such as C++, Java, or Python. For example, in Apache S4 [164], operators are written as standard Java classes that inherit from a common abstract superclass defined by the framework. The developer has to "fill-in" a few methods, and the runtime takes care of automatically invoking them when events of interest occur. Using general-purpose languages has several benefits, including the possibility to re-use pre-existing libraries and software modules seamlessly inside custom stream processing applications. However this usually comes at the expense of conciseness and prototyping speed, as APIs can be verbose and sometimes complex.

Execution model

An execution model maps the elements defined in the abstract model and described through the development model to runtime objects (or

tasks) that run natively on the platform hosting the stream processing framework. An execution model defines:

1. The platform-specific execution units on which DSPS elements are mapped, and the scheduling policies for local resources.
2. The distribution of the execution units on cluster 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 concepts native to the host platform, such as OS processes or threads. With a *process-per-operator* allocation, operators are instantiated individually as separate processes with one or more concurrent threads of execution (e.g., one per input port). That schema grants maximum isolation because any problem occurring to one component does not affect other concurrently running ones. Normally, when this architecture is used, the local scheduling of resources is demanded to the standard facilities of the host OS CPU and memory schedulers. The first implementations of the STREAM PROCESSING CORE (SPC) [28] 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. All the components are hosted as separate software modules within this process, for example, as instances of a some class in case of a class-based object oriented implementation [164]. While, on the one side, this arrangement does not grant the same execution isolation as the process-per-operator allocation, on the other side, it gives tighter control on resource scheduling policies. For example, every in-process component could be executed by a dedicated thread, or, more interestingly, they could share a pool of threads scheduled according to internal policies or QoS requirements (e.g., for a priority-proportional scheduling of resources). Another advantage of this architecture relates to the fact that communication of components running within the same process is usually more effective, thanks to channels based on shared memory. The process-per-server allocation is used, for example, in Apache S4 [164] and Quasit [167], which start a JAVA VIRTUAL MACHINE (JVM) [170] on each cluster server and deploy sources, operators, and sinks as objects running within the local JVM.

Somewhere between the previous solutions, the *cluster-of-operators* approach *fuses* subsets of tightly coupled components into one process. For example, operators with strong reciprocal communication dependencies are good candidates for fusion. Again, within every process, very flexible resource scheduling approaches and faster communication channels can be used. Different operator clusters, however, are still mapped to different processes, isolated from each other. IBM InfoSphere Streams [166] uses a similar hybrid approach through

a technique called *operator fusion* [171] that groups multiple operators into single execution units.

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 communication channels. A further role of the execution model is the definition of a *placement strategy* for runtime objects. A placement strategy decides the distribution of runtime objects on the available cluster servers: a good solution should take into account the resources requirements of every object, the resources availability of each server, and the expected/declared application communication patterns, and it should satisfy the application 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, 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 incrementally deal with load variations. The works in [172] and [173] are examples of algorithms performing both static and dynamic assignment phases, while [174] tries to find an initial static assignment that maximizes the system robustness to possible load variations.

Finally, an execution model should decide how communication channels are instantiated at runtime. For in-process communication, *function calls* or *shared memory-based message passing* are the most common alternatives. While the first binds the execution thread of the caller to that of the callee, the second allows independent execution of the communicating parties. 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 OS pipes can be adopted for 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, unordered, and unreliable communication is enough, UDP-based channels are a commonly adopted solution.

3.3.2 QoS-aware model

In Information and Communications Technology (ICT) infrastructures serving mission-critical applications, e.g., in the areas of health-care, 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 constraints functional and non functional runtime parameters according to a pre-

defined set of performance indicators. The range of possible performance indicators is, in general, very large and application-dependent: before, in this chapter, we already introduced two common and simple examples of general and high-level metrics such *latency* — measuring the maximum time interval between a service request and the corresponding response — and *throughput* — measuring the average number of samples per time unit that the DSPS can process. Other indicators can refer to non-functional aspects such as *availability*, i.e., a measure of the fraction of time the service is up and running, or to lower-level details like *memory* or *CPU* usage. Every constraint that, in an SLA, binds some performance indicator to its required value is said to represent a QoS *property* for the service.

In all kind of distributed scenarios, the implementation of QoS-*aware* services, i.e., services that are guaranteed to deterministically operate according to a set of QoS properties, is a very difficult task, and maps to the ability to allocate, statically and dynamically, the proper amount of computational resources to different processing components. The technical challenge is even harder in the case of stream processing: differently from simple request-response or batch-oriented services, where characteristics of computational tasks are known a-priori and easier to reason about, in stream processing, the properties of input streams (e.g., their data rate) change continuously and they can be partially or completely unknown in advance and very difficult to predict. The consequent uncertainty that platforms have to deal with during long provisioning times makes the implementation of effective and adaptive resource scheduling techniques a very challenging task.

Nonetheless, there is a growing number of real-world large data streams analysis applications that requires predictable performance guarantees. For example, think again about the SPE scenarios described in Chapter 1, where the results results of stream analysis is used to trigger real time feedback actions on physical aspects of the cyber-physical world. These actions can be responses to emergency conditions, such as the activation of alarms in smart tele-care systems [32], or the computation of emergency rescue plans in a smart traffic management system [17], which obviously need to be performed in a timely and reliable fashion.

We strongly believe that the described scenarios call for a strong integration of QoS in novel DSPSs, and that QoS-awareness should be developed 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 properties required for graphs, single sources, operators, or sinks, 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 properties, to

which specific components they can apply, and how they interact with each other. QoS *in the development model* should define the syntactic constructs 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, DSPSs should map QoS properties to proper mechanisms for runtime admission, monitoring, control, enforcement, and management, and should develop resource scheduling algorithms that can be used to successfully satisfy the required QoS specifications.

In the next subsection, we survey a selection of QoS properties for DSPSs that have been successfully proposed and implemented in the literature or that we believe could be successfully realized in future stream processing framework.

3.3.3 Classification of QoS properties

In this section, we describe an original bi-dimensional classification of QoS properties for DSPSs. Similarly to what we have done for the classification of PUB/SUB QoS properties in Chapter 2, our classification tries to make common features, quality trade-offs, and issues related to the implementation of different properties emerge clearly from their position in the taxonomy.

The classification axes we consider are:

- *Enforcement mode*. Considering the distributed deployment of a DSPS in a cluster setting, can a QoS property be enforced autonomously and locally by cluster nodes, or does it need distributed coordination?
- *Quality Domain*. What aspect of the DSPS functionality does the property regulate?

The *enforcement mode* dimension captures the fact that some properties can be implemented and enforced without any distributed coordination, while others might need more or less complex interactions between cluster hosts. In the first case, think about a property that defines the *queuing behavior* of processing components by setting queues capacities and related blocking semantics; in the second, consider, for example, a latency requirement on a sequence of operations performed by components distributed on many computing nodes. Understanding the *enforcement mode* associated to QoS properties is very important, especially when considering data center deployments of DSPSs on a possibly very big number of computing nodes. It is trivial to recognize, in fact, that QoS properties that only require a *local enforcement* tend to scale better than those that need a *distributed enforcement*, since their complexity can grow rapidly with the number of distributed participants.

Table 3.1: Bi-dimensional taxonomy of DSPS QoS properties.

	Local enforcement	Distributed enforcement
Processing semantics	Parallelism	Ordering
	Queuing	
Load management	Priority[a]	Priority[a]
		Latency
		Throughput
Fault tolerance	—	Availability
		Consistency

[a] The enforcement mode is implementation-specific (see the property description for details).

The *quality domain* dimension groups different properties according to the high-level goal that they aim at achieving. We identify three main groups within the quality domain dimension:

- *Processing Semantics.*
- *Load Management.*
- *Fault Tolerance.*

As it was the case for our PUB/SUB QoS properties classification in Chapter 2, to isolate one quality domain group from the other completely can be difficult because QoS properties from one group have often special interactions with those of another group; for example, they can alter each other's semantics, have mutual requirements, or thwart each other's enforcement. In the following description of QoS properties, we try to highlight every possible cross of concerns among different QoS properties or quality domains.

Table 3.1 organizes the properties that we survey in the remainder of this section according to their quality domain and enforcement mode. For the sake of discussion clarity, in the following, we group the analyzed properties by their quality domain.

Processing semantics

QoS properties in this group assist the stream processing platform in configuring and managing operators or, more specifically, their corresponding runtime objects, according to application-driven requirements. Properties in this class must be used carefully because their setting can also influence other performance parameters: to mention just one example, queuing details are strongly correlated to perceived latency.

PARALLELISM We define two different type of parallelism specifications: *instance parallelism* and *task parallelism*. Both consist of an integer value $p \geqslant 1$ attached to an operator o. The value of the task parallelism property specifies how many tasks should be instantiated and

deployed for the operator at runtime (e.g., how many processes in a *process-per-operator* mapping, see Section 3.3.1). With an instance parallelism specification, instead, a user can configure how many threads should execute corresponding operator tasks. When task parallelism is used, it is possible to specify a *data partitioning function*. This function, defined over the domains of the operator input streams and producing as output a discrete number of possible values, is used by the platform to route data samples to different tasks of the same operator. The platform guarantees that any two input data samples e_1 and e_2 for which the evaluation of the data partitioning function produces the same value k are routed to the same operator task.

An instance parallelism $p > 1$ means that several execution threads can process a task's input samples concurrently: in that case, it is up to the developer to make sure that the state of the operator is correctly protected against concurrent accesses. If the right trade-off between concurrency benefits and synchronization overhead is chosen, using this property can provide important advantages especially in multi-core architectures [29]. Task parallelism can bring the same type of performance advantages, and also provides an additional opportunity to distribute the operator load across multiple nodes [175]. In this case, the states of different operator tasks are isolated and do not influence each other: this fact, can be acceptable or not depending on the specific semantics of the operator. In many *data parallel* problems, for example, the processing state relative to different groups of input data are inherently independent (e.g., a streaming word count problem [176]). Once an application is deployed on the cluster, parallelization of operators does not require global coordination at runtime.

QUEUING In most DSPS implementations [29, 159, 164, 166, 167], operators follow an asynchronous processing pattern, by executing their data processing operations reactively in response to the arrival of data samples at their input ports. In order to decouple processing components from data distribution channels, most implementation use ad-hoc input and output sample queues that are interposed between the network and the operator tasks. At runtime, the DSPS extracts samples from output queues, routes them to local or remote destinations, and puts them into destinations input queues. Asynchronously with respect to this process, it also assigns execution threads to tasks with non-empty input queues. The *queuing* QoS property lets users have a more precise control on the configuration of tasks input and output queues. For example, by setting a maximum queue size value, it is possible to limit the maximum amount of memory that tasks will use to hold unprocessed samples. Note, however, that bursty input can cause queues to rapidly fill: in that case, unless techniques for queue *back pressure* are implemented [154, 177], some samples will have to be dropped in order to avoid blocking operations. When dropping

data is unavoidable, *data dropping policies* can be specified: trivial policies are to discard newly arriving samples or random ones, but other more complex ones can be devised, for example, to drop data according to application specific priorities [178]. Note also that longer queues inevitably cause longer processing *latencies* [179]. Queuing specifications act locally to operator tasks, so they do not require any distributed enforcement protocol.

ORDERING The ordering QoS property configures the ordering guarantees provided to the operator or sink it is attached to. Guaranteeing consistent data processing order can be useful, for example, in case *repeatability* of results is a requirement or if the order of samples has an application-relevant meaning that must be preserved. We consider three possible values for this properties:

- *No order*.
- *Producer order*.
- *Total order*.

When no order is used, samples are processed as they arrive on any operator input-port. Note that the arrival order does not necessarily reflect the sample production order, for example, if an unordered protocol, such as UDP, is used to deliver samples. On the contrary, this can be guaranteed by using producer order. From an implementation point of view, producer order can be realized by using an ordered transport protocol (e.g., TCP) for transferring samples from a producer's output queue to its consumers' input queues. Note also that producer order does not impose any relation between samples received from different input ports. That guarantee can be achieved, instead, by using a total order QoS property, which ensures that samples arriving to a task from all its input ports are ordered deterministically. A possible implementation approach for total order is, for example, based on global time stamps that producers include in their samples [180]. Neither producer nor total order can be implemented locally if any of the operator sources is running on a remote host.

Load management

In batch-oriented DISC systems the characteristics of the input data are usually known a-priory, but the same cannot be claimed for DSPSs. The data rate of input streams can vary sensibly during the lifetime of an application temporarily or permanently. Management of time variable input load poses several hard challenges for the allocation and scheduling of distributed resources. In fact, if, at some point, a DSPS does not have enough resources to handle the input load and to keep up with the data production rate, input queues start to grow with two possible consequences: if they fill, some samples have to be dropped at some point in the processing graph, leading to *data loss*; if,

on the other hand, they are long enough to sustain the duration of the load peak, the system will have to pay the price of increased latency. In this section, we discuss QoS properties that can be used to regulate how to handle load variation according to user-defined requirements.

PRIORITY In stream processing applications, it is not rare that different processing flows have different importance from the point of view of applications. Consider, for example, the TMS scenario presented in Chapter 1: in that scenario it is clear that the management of emergency situations should take absolute priority over other services, such as car navigation. This must be true even and especially when the system is under heavy load and when the available processing resources are not enough to execute every application processing functionality successfully. The priority QoS property can be exploited to impose an application-aware partial ordering between different processing tasks; the system will use these hints to allocate resources to tasks accordingly, and to take appropriate actions in case of resources shortage due to a sudden growth of input load. A priority QoS property consists of a value usually chosen from a short set of possibilities (e.g., in the 0–9 range) and can be associated either to single operators or to data sinks. Sink priorities are used to influence the scheduling of resources for all the runtime objects that produce data ultimately going to the prioritized sink [181]. On the contrary, when associated to operators, priority properties influence only the execution of that operator's tasks. Although their usefulness is not as easy to understand as sink priorities, operator priorities allow a very fine grained specification of task resource requirements that can aid resource usage optimization. If not explicitly given by users, priority values can also be inferred statically or dynamically leveraging other orthogonal QoS properties such as latency [157] or custom utility functions [182]. Priorities decide how tasks running on the same server share its resources at runtime: their enforcement does not require distributed coordination. However, global knowledge of priorities can help task placement algorithms to decide the mapping between tasks and available cluster servers.

LATENCY Latency QoS specifications express timing bounds on DSPS operations. A latency specification can be attached to an operator, a graph edge, a connected sequence of operators [169] or, more simply, to a data sink [157]. Operator latency specifies the maximum time interval elapsing from the moment a sample is put in an task's input queue to the moment the sample is extracted from the corresponding output queue. In this definition, both queuing and sample processing times must be taken into account. Latency on an edge constraints the time taken to transport a sample from a source task output queue to the input queue of the target operator task: note that this process

could involve network-level buffering and routing that can add significant time overhead. Latency can also be specified on connected sequences of operators: in this case the upper bound refers to the sum of latencies of operators and edges involved in the sequence; finally, a latency specification on a data sink refers to the sum of the latencies of all the sequences that eventually lead to that sink node.

Latency is probably the performance DSPS metric most strongly perceived by users because it is simple to understand and has a direct and critical impact on almost every application scenario. However, to implement guaranteed latency bounds in DSPS is probably one of the toughest challenges to face when realizing QoS-based behavior. First, there are many concurrent sources of delay that can influence sample processing time at runtime, such as I/O operations, adverse network conditions, data batching policies, synchronization points, or coordination protocols for fault-tolerance [136]. Second, DSPSs are normally hosted by non-real-time OSs that do not provide mechanisms to control strict time-based operations, as real-time schedulers do [183], and to coordinate the interactions between middleware-level and OS-level scheduling is a non trivial task. Third, tracing the causality relationships between source input samples and output samples is a rather complex tasks, especially when UDFs are used to represent operator behavior [29]. Dynamic adaptation of priorities based on changing latency requirements [154, 177], static and dynamic task placement algorithms [172, 184], or continuous resource re-scheduling for load adaption [185] are common optimization-based solutions that are used to enforce latency constraint. An alternative approach is to claim the resources needed to guarantee low-latency processing by voluntarily dropping some data, through *load-shedding* techniques [186, 187]. Especially in the case of sequences of operators, all the servers where involved operators are deployed must coordinate to guarantee timely sample processing, and hence distributed enforcement is required.

THROUGHPUT In a DISC system, throughput defines the average number of samples processed per time unit. In batch-oriented systems, where all the input data is available before processing starts, throughput is the primary performance evaluation metric, and techniques to improve it have been widely studied [188, 189]. When thinking about DSPSs, the definition of what throughput means is less obvious because the maximum number of tuples processed per time unit depends not only on the characteristics of applications and platforms but also on the current input streams data rate. We define throughput of a DSPS the maximum number of output tuples per time unit that the system can produce as the input data rate grows indefinitely. A throughput QoS property can be used, at deployment time, to guide tasks instantiation processes and resource assignment or, at runtime,

to initiate adaptation procedures [175, 190]. A general way to improve throughput is by using batching techniques at both processing and networking levels: at processing level, it consists in letting a task process a group of samples rather than one sample at time before giving control back to the local thread scheduler; at networking level, it consists in grouping several samples into single packets to avoid the overhead due to network communication protocols. However, note that batching techniques tend to deteriorate latency [169]. As latency, implementing controlled throughput requires the proper orchestration of all system components, and hence distributed enforcement.

Fault tolerance

Fault-tolerance is the ability of a system to continue to offer its service in spite of possible hardware, software, or network failures that might occur [191]. Stream processing systems are designed to run for indefinitely long amounts of time on a large number of distributed nodes made of commodity hardware: in such a setting, the likelihood of failures are very high. For this reason, DSPSs should provide appropriate fault-tolerance mechanisms, and, most-importantly, they should clearly define the guaranteed service-level they provide when different types of failures occur. From another perspective, users should be given tools to express fault-tolerance requirements for their applications, and QoS-aware DSPSs should fulfill these requirements.

AVAILABILITY When distributed DSPS components fail, some of the hosted applications will stop producing results or will produce partial or incorrect output. The availability QoS property measures the ability of a system to quickly restore faulty components and restart steady-state operations. Note that, according to this definition, to be available a system does not necessarily need to restore its internal state as it was before the fault: the availability concept only captures the ability to overcome failures and measures the speed of this *recovery process*. Many solutions have been proposed in the literature to implement high availability in DSPSs [192]: in *active* and *passive replication* the system deploys one primary replica and one (or possibly several) secondary replicas for each operator task proactively; upon detection of primary failures, it replaces it with one of the secondary copies. An alternative and reactive approach is to instantiate new tasks only after failures occur. In general terms, proactive approaches tend to provide shorter recovery times, and reactive ones, while cheaper in terms of runtime overhead, are usually slower. Availability policies have a strong relationship with the *consistency* QoS-level that the platform offers (see next paragraph): as a rule of the thumb, the stronger the consistency guarantees, the more expensive the availability guarantees are. Implementing available systems, re-

quires distributed monitoring and recovery, which rule out the possibility of local QoS enforcement.

CONSISTENCY Consistency deals with the ability of DSPS systems to mask the effects that failures may have on application outputs. Failures of processing nodes or network might cause operator processing state or stream data to be lost, an event that can easily alter application results. Note that any system providing some form of guaranteed consistency is necessarily available: in fact, a platform must recover from failure situations first to possibly hide their effects. We identify five main types of consistency guarantees:

- *Best-effort consistency*.
- *At least once processing consistency*.
- *Exactly once processing consistency*.
- *Repeatable consistency*.
- *Weak consistency*.

All the consistency guarantees are defined with respect to *executions* of the DSPS on some finite portions of an application's input streams, i.e., a set of input traces (see Section 3.3.1). System providing *best-effort consistency* are those systems that, while not giving any type of strong guarantee about their behavior in presence of failures, still try, in some way, to reduce their negative effects. With at least once processing consistency, the DSPS guarantees that all the samples in the input traces are processed at least once, and that their possible influence on the application state is not lost in case of failures. Similarly, exactly once processing consistency ensures that all the traces samples are processed exactly once, and their effect on the application state is preserved in case of failures. Repeatable consistency is even stronger: it guarantees that any two executions on the same input traces produce exactly the same output traces, also in case of failures. Note that the realization of protocols for repeatable consistency is a very hard task: in fact, it is not only sufficient that all the samples are processed once, but their processing order must be completely deterministic, even between samples arriving on different operator input ports. For this reason, repeatable consistency requires *totally ordered* processing semantics. With the weak consistency expression, finally, we refer to a class of consistency guarantees, rather than a specific one. Members of this class are characterized by their common attempt to find a trade-off between strong consistency guarantees (such as at least once, exactly once, or repeatable) and best-effort or no guarantees at all, for example, by expressing fault-tolerance in probabilistic terms, or by explicitly specifying the amount of information that is guaranteed not to be lost in case of failures.

State checkpointing techniques are widely used to implement best-effort consistency [193, 194]; fault-tolerance techniques like upstream backup [195, 196] or active and passive replication [197, 198] can

Table 3.2: List of surveyed DSPS systems.

System	Subsection
Aurora/Borealis [27, 158]	3.4.1
STREAM [155, 156]	3.4.1
TelegraphCQ [154, 199]	3.4.1
IBM InfoSphere Streams [166]	3.4.2
Nephele Streaming [169, 200]	3.4.2
Storm [29]	3.4.2

realize, depending on implementation details, either at least once or exactly once consistency. In Chapter 5, we introduce a particular instance of weak consistency systems that uses an ad-hoc *internal completeness* metric to define fault-tolerance related guarantees. Implementation of consistency is a complex task and, as availability, it requires the coordination of many system components, i.e., a distributed enforcement.

3.4 SYSTEMS SURVEY

In this section, we provide a technical overview of a selection of existing DSPSs, and we discuss their design and architectural features under the light of the three-layers modeling framework introduced in Section 3.3.1. In our analysis, we focus in particular on system features that realize quality-related behavior, and try to emphasize the related design and implementation aspects. The selection of systems that we include in this section is guided by reasons similar to those that motivate the choice of PUB/SUB systems in Chapter 2. In particular, our goals are:

- Present systems that support a representative selection of relevant, diffused, and original DSPS QoS properties.
- Analyze the benefits and drawbacks of different implementation architectures for what concerns the achievement of quality-aware data processing.
- Cover both DSMSs and DSPEs and highlight their differences for what concerns their QoS-aware services.

Table 3.2 summarizes the DSPSs that we discuss in the rest of this section. We organize our analysis in two subsections: in the first we discuss systems belonging to the DSMS group and in the second those that can be classified as DSPEs (Table 3.3). Let us remark that it is not our goal to provide an extensive survey of existing stream processing solutions: for a comprehensive work, the reader is referred to [132].

Table 3.3: Classification of surveyed DSPS systems.

STREAM MANAGEMENT SYSTEMS	STREAM PROCESSING ENGINES
Aurora/Borealis	IBM InfoSphere Streams
FIT	Nephele Streaming
STREAM	STORM

3.4.1 Data stream management systems

The most important DSMS proposals emerged after novel stream processing scenarios brought to light the limitations of traditional DBMSs in handling continuous queries over streaming data. Mostly coming from data bases communities, these systems usually focus on QoS properties related to *processing consistency* and *reliability* which are very important problems also in the case of traditional data bases. In this section we analyze the three most influential DSMSs, i.e., Aurora/Borealis [27, 158], STREAM [155, 156], and TelegraphCQ [199].

Aurora/Borealis

Aurora [157, 158] is a centralized stream management systems jointly developed by Brandeis University, Brown University, and Massachusetts Institute of Technology. Borealis [27, 159] brings together the stream management functionalities of Aurora with a distributed architecture inspired by Medusa [201]. In Aurora and Borealis, users build stream processing applications according to a so called *boxes and arrows* model, which corresponds to our processing graph abstract model with operators chosen from those defined by an ad-hoc STREAM QUERY ALGEBRA (SQUAL).

Quite interestingly for the goals of our work, Borealis has been a common development platform for the experimentation of several QoS properties in the context of DSMSs. The original Aurora model permits to express QoS requirements on the output streams using a three-dimensional quality model that measures output *latency*, percentage of samples *drop*, and the presence of desired *values* in the output data. These QoS requirements are not expressed as strong constraints, but as functions that map the measured output performance to a value from 0 (maximum QoS violation) to 1 (perfect QoS satisfaction). Borealis extends this model by permitting to associate these measures not only to output sinks, but also to inner operator boxes. The QoS requirements are used to guide the initial placement of tasks on cluster nodes, the scheduling of resources at runtime, and possible dynamic operator rescheduling. In [187], a *load shedding* algorithm for Borealis is proposed that selectively drops samples to handle load peaks and maximize the system *throughput*, weighted according to user-defined priorities on output sinks. The DELAY, PROCESS, AND CORRECT (DPC) protocol [196] has been more recently proposed to enable

low latency and *eventual repeatable consistency* in spite of failures. To avoid an excessive increase of processing latency in case of failures, and while the system is not fully recovered yet, DPC produces *tentative* samples, i.e., possibly partial or incorrect results. These tuples are guaranteed to be eventually corrected once the system is restored. The user can influence the trade-off between availability and consistency by controlling two parameters: a latency requirement and the maximum number of tolerable tentative tuples.

STREAM

The Stanford Data Stream Management System, or STREAM, [156] is a general-purpose DSMS supporting continuous queries over continuous data streams or static data sets. Its development model is based on the CQL algebra and query language [150], designed to resemble SQL for easier adoption.

Although STREAM is a *centralized* data streams manager with a multi-threaded process doing all the work, we include it anyway in this survey because it implements several interesting QoS-related features that have been seminal to more recent DSPSs. After being submitted, CQL queries are compiled into a corresponding processing graph of operators, whose execution is managed by the local scheduler. STREAM adopts a *chain scheduling* algorithm [202] that aims at minimizing memory usage by giving execution priority to chains of operators that maximize the reduction in the length of internal sample queues. If, instead, achieving low latency is more important than saving main memory, a very simple FIFO scheduler can be used in alternative. STREAM also guarantees totally ordered processing of tuples [203]. To do so, it puts reordering buffers in front of operators, which store all the tuples older than τ until an *heart beat message* with time stamp τ is received. At this point, they reorder their content and feed it to the corresponding operators. After that heart beat, no tuples older than τ will be passed to the processing stage. The timing of heart beats can be customized according to user defined parameters, for example, by setting a *skew bound* that defines the difference in the time stamps from two different input streams that is expected to be found. Additionally, STREAM supports a set of user indications, called *adherence parameters*, that permit to define specific characteristic of input data streams used to optimize resource usage. For example, an adherence value of type "ordered-arrival k-constraint" attached to an input data stream specifies that, considering a single sample attribute other than the time stamp, for any sequence of $k + 1$ samples $e_0 \dots e_k$, the value of that attribute in e_k will be strictly greater than the attribute value in e_0 [204]. For what concerns load adaptation, STREAM implements *load shedding* [205], i.e., it inserts automatic drop operations inside the processing graph in order to retrieve sufficient

resources to handle load spikes while minimizing the inaccuracy of results due to data loss.

TelegraphCQ

TelegraphCQ [154, 199, 206], developed by UC Berkeley and the IBM Almaden Research Center, exposes an ad-hoc query language which, syntactically, is a subset of SQL; SQL operations are mapped at runtime on a set of continuous and non-blocking operators (or *modules* according to TelagraphCQ terminology). The runtime model follows a process-per-server model, with local TelegraphCQ instances connected by the so called Flux component [207] that implements data routing functionalities and replication mechanisms.

The systems has a strong focus in realizing adaptive mechanisms for query processing. For example, its Eddy components [177] can dynamically reorder the sequence of operators samples go through based on load variations detected at runtime. *Instance parallelism* is supported via Flux: in particular, Flux partitions data across the clustered Telegraph instances in order to balance servers' load, and it implements mechanisms to dynamically adjust these partitions to keep the cluster balanced in presence of short or long-lasting load variations. TelegraphCQ also supports content-based priority specifications through its Juggle component [208] that reorders samples produced by sources according to their content. Finally, for what concerns fault-tolerance, TelegraphCQ supports availability through active replication and exactly once processing consistency (both implemented in Flux).

3.4.2 Distributed stream processing engines

Compared to DSMSs, DSPE implementations are more interested in realizing highly parallel and scalable solutions rather then providing a formal query model or strict processing guarantees. Nonetheless, especially in the recent years, many solution have emerged offering richer QoS-related features that respond to the growing need for quality coming from real-world processing scenarios. In this section we discuss two widely used industrial DSPEs — IBM InfoSphere Streams [28, 166] and BackType (now Twitter) Storm [29] — and a relevant proposal coming from academia — Nephele Streaming [169, 200].

IBM InfoSphere Streams

IBM InfoSphere Streams [166] is a DSPE evolved from the SPC research project [28]. 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. In addition to SPL, the system offers two sets of general-purpose APIs

that can be used to build UDF-based custom operators. The first is a mixed C++ and Perl API that, via a two-steps code generation process, gives them maximum customization flexibility and execution efficiency [209]. 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 APIs can be used directly from SPL source files. At compile time, an optional operator fusion process can be manually or automatically performed in order to cluster groups of correlated operators [171]. Each group is then transformed into its corresponding runtime task, called Processing Element (PE), whose execution is mapped onto an OS process. Hence, InfoSphere Streams follows a cluster-of-operators approach if the fusion step is enabled, or operator-per-process otherwise. 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.

InfoSphere Streams supports a fault-tolerance mechanisms based on state checkpointing [193], which guarantees availability and best-effort consistency: periodically or in response to the reception of particular samples, tasks can save their state on secondary memory; whenever a crash occurs, that state is restored but all the processing operations performed between the checkpoint and the failure are lost. To the best of our knowledge, the commercial version of the system does not perform dynamic load management, but distributes tasks on the available cluster resources based on an off-line only profiling and placement algorithm [171]. At runtime, the system monitors and detect possible load imbalances and suggests alternative tasks placements [173]; dynamic PE migration operations, however, must be manually triggered by system administrators. In the context of the research precursor of InfoSphere Streams (SPC), Amini et al. [185] have proposed a dynamic resources micro-scheduling algorithm that dynamically changes local PE resource allocation based on the current system load; we are not aware whether this algorithm is implemented in the commercial system or not.

Nephele Streaming

Nephele is a DISC system developed mainly by TU Berlin in the context of the larger Stratosphere project [210]. Originally, Nephele was designed to support the needs of batch-oriented scenarios, offering a generalized graph-based processing model [26] that flexibly extends many core ideas previously proposed in the MapReduce framework [24]. In this section we discuss its very recent extension, Nephele streaming [169, 211], that modifies the platform by adding support for continuous data stream processing. Nephele applications are de-

veloped by using ad-hoc APIs (Java or Scala at the time of writing) to create operators and connect them in processing graphs. Processing graphs are then deployed on a cluster of servers and executed by a set of *task managers* each corresponding to an OS process (process-per-server model).

The main contribution of Nephele streaming for what concerns QoS enforcement is the implementation of two complementary techniques that dynamically optimize the system in order to satisfy user *latency* requirements attached to sequences of operators. The first technique, called *adaptive output buffer sizing*, trades high throughput off for improved latency by adjusting the size of operator output buffers according to runtime estimations of the maximum time tuples can sit into output buffers and still avoid latency violations. The second mechanism, called *dynamic task chaining*, works locally to single task managers where different tasks normally run in separate threads. The idea of dynamic task chaining is to remove the additional latency due to input/output queuing and execute, when possible, sequences of consecutive task within the same thread; also in this case, latency gains are paid with possible throughput degradation since using task chaining reduces the number of opportunities for pipelined parallel task execution.

Storm

Storm [29] is a DSPE developed by BackType and recently released under the Eclipse Public License by Twitter after its acquisition of BackType. In Storm processing graphs, 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. According to the Storm development model, the main method to define new spouts and bolts is through a Java API: custom bolts and spouts are defined by writing classes that extend specific base classes, which in turn provide core stream processing functionalities to newly built components.

The Storm execution model is rather articulated and, originally, the system allows to configure many parts of its execution engine. For instance, the way a processing graph is instantiated on the cluster can be influenced by three parameters, i.e., i) the number of worker processes, ii) the number of tasks per operator (*task parallelism*), and iii) the number of per-component threads (*instance parallelism*). The first parameter determines the total number of processes that will be created on the Storm cluster; the second, associated with every spout or bolt, determines how many tasks of each component are instantiated; the third determines the total number of threads that should serve a component's set of tasks. At runtime, every worker is instantiated in a different JVM, which can host one or more tasks (and execute one or more threads) from the same application. Very peculiarly, Storm

Table 3.4: Summary of supported DSPS QoS properties.

	Parallelism	Queuing	Ordering	Priorities
DSMS	TelegraphCQ	—	Borealis	Borealis
			STREAM	TelegraphCQ
DSPE	Nephele	Streams	Storm	—
	Storm			

	Latency	Throughput	Availability	Consistency
DSMS	Borealis	Borealis	Borealis	Borealis
		STREAMS	TelegraphCQ	TelegraphCQ
DSPE	Nephele	Nephele	Nephele	Storm
			Storm	

puts also a strong focus on fault-tolerance by optionally providing at-least-once processing consistency. To do so, for each *root sample* (i.e., a sample generated by a spout), Storm keeps track of all the other samples that are *caused by* processing operations that involve it directly or indirectly. The root sample is kept in stable storage until all the samples caused by it are acknowledged by their final destinations. Given the highly customizable nature of stream processing functionalities, Storm cannot keep track automatically of *caused by* relationships and requires explicit developer intervention: at code-level developers have to explicitly mark every new sample as caused by another sample if they will to avail of Storm fault tolerance facilities. Building on top of this functionality, Storm implements what it calls *transactional topologies*, i.e., processing graphs implementing exactly once processing consistency.

3.5 DISCUSSION

In the previous section, we have surveyed six DSPS solutions selected for the unique QoS-related features they implement. Table 3.4 shows the main QoS properties supported by these systems and organizes them based on their belonging to the DSMS or DSPE classes. In this section, we summarize our analysis by focusing on the three QoS quality domains identified in Section 2.2.3 and by discussing their implementation in state-of-the-art DSPS systems.

3.5.1 Processing semantics

Task parallelism is a fundamental tool to enable scalability in very large stream processing scenarios. The creation of several tasks per operator permits to partition the analysis of large data streams across multiple processing sites and helps avoiding computational bottle-

necks. Task parallelism is becoming a standard technique in DSPEs: in our survey, both Nephele [26] and Storm [29] support it by also letting users express custom data partitioning criteria. Besides these two systems, many other modern DSPE solutions implement task partitioning, including, for example, Apache S4 [164], Apache Samza [165], and Google MillWheel [180]. The finer grained instance parallelism property is less widely supported, probably because it influences directly low level implementation mechanisms that systems usually prefer to hide from final users. Among the surveyed systems, only InfoSphere Streams and Storm permit to use that property to some extent, the first by allowing to choose whether a fused operator should be executed by a dedicated thread or not, the second by letting users choose how many threads to create per operator (while still hiding the threads-to-tasks allocation policy). With the exception of TelegraphCQ, which offers some basic horizontal partitioning methods with its Flux component [207], most DSMSs do not support neither task or instance parallelism. The reason for this must be probably sought in the fact that these systems are often the evolution of simpler centralized platforms that were designed to work in scenarios where input load is successfully handled via basic pipelined-parallelism.

Direct control of input and output queues is not allowed in most DSPSs: we are not aware of any system that permits to specify the length or type of input and output sample queues except InfoSphere Streams that provides SPL syntactic elements to accomplish this task. In other solutions, the behavior of input queues is only indirectly and automatically controllable by setting other QoS properties, such as latency: for example, in Nephele Streaming, the length of output buffers is adjusted according to user latency requirements; similarly, the load shedding features in Aurora/Borealis or STREAM involve input queues by dropping their part of their content in case of overload, but their final goal is to achieve better latency performance.

Most systems, finally, support producer order: this is normally implemented by mapping processing graph edges on network channels based on ordered transport-level protocols. However, just a few systems offer totally ordered processing: in our survey, only the DPC extension of Borealis, STREAM, and Storm transactional topologies guarantee total order. Forcing a total order between samples coming from many sources is, in fact, a non-trivial task. On the one hand, it is not always obvious how to choose a meaningful attribute that determines the order of samples from different sources: a commonly adopted mechanism is to leverage producer-side time stamps, but this might not always be a good choice since time stamps by different producers might be largely de-synchronized. On the other hand, to implement cross-stream ordering always involves a pre-buffering stage where out-of-order samples are kept and re-ordered before being pushed to downstream operators: this operation introduces ad-

ditional delays on the stream processing operations, which is a normally undesirable effect. For these reasons, total ordering is normally available only if strictly required to guarantee other properties, such as *repeatable consistency* [29, 165, 196].

3.5.2 Load management

Properly handling load variation is fundamental in any DSPS, and it is one of the main reason why the first DSMSs were developed, overcoming the limitations of DBMSs. However, the way this is done can vary a lot from system to system.

Priorities are an easy to understand and easy to use method to express preferences among different processing operations or different application outputs. Surprisingly, they are not widely supported in the literature we have surveyed. TelegraphCQ supports a form of content-based priorities through its Juggle [208] component, while in Aurora/Borealis, although supported, priorities are only used in relation to another load-management technique, i.e., load-shedding.

Load shedding is supported by Aurora/Borealis, Stream, and TelegraphCQ [212], and has been widely studied also outside these systems [213–215] because it provides a simple mechanism to deal with increasing input loads using a limited amount of resources. While very successful in DSMSs, we are not aware of DSPEs offering load-shedding features out-of-the-box. However, given the simplicity of the mechanism, we believe that load shedding can be feasibly implemented in ad-hoc operator components for any DSPS and used in any scenario where deterministic dropping of samples is acceptable.

Although latency and throughput are two of the declared objectives of DSPSs, only Aurora/Borealis and Nephele Streaming support the specification of explicit latency requirements and none of the systems in our survey supports direct throughput specifications. Instead, latency and throughput are often considered optimization goals rather than hard constraints. This derives from the fact that implementing strong latency bounds on non-real-time operating systems, where DSPSs usually run, can be a very hard, if not impossible, task: platform-level schedulers need to cope with the presence of OS-level schedulers and have to share CPU and memory resources with other services that run concurrently to the DSPS in ways that are usually not directly controllable.

3.5.3 Fault tolerance

Given the importance of fault tolerance in distributed systems that are supposed to run for indefinitely long amount of times, techniques for availability and consistency are common in DSPSs: the only system presented in this chapter that does not discuss about its fault-

tolerance mechanisms is STREAM. The guarantees provided by different systems and fault-tolerance protocols are very different among each other: each tries to balance the trade-off between recovery time, processing guarantees, and runtime cost. For example, active operator replication [195], which can provide exactly once processing semantics and almost instantaneous recovery, comes also at the price of a very high runtime cost. On the opposite end, solutions based on state checkpointing [164, 193] are very cheap to maintain, but they cannot give any guarantee other than best-effort consistency and are associated with slower recovery times. The strongest form of consistency, i.e., repeatable consistency, stands at the extreme end of this consistency–cost trade-off and is offered only by a few systems [29, 165, 196]. In fact, the strong synchronization constraints between incoming input streams that are required to implement the necessary totally ordered processing semantics can have a severe impact on the achievable system performance in terms of throughput and latency.

3.6 DIRECTIONS FOR FUTURE WORK

Our survey has shown that, over the years, DSPS solutions have introduced several quality-related features in distributed stream processing, covering almost completely the space of possible QoS properties classified in Section 3.3.3. Among the surveyed systems, Aurora/Borealis is the one realizing the largest number of QoS-related features; most of the other systems, instead, focus on just a few QoS properties each and do not offer comprehensive QoS configuration possibilities. Most importantly, none of the systems we are aware of elevates QoS to a first-class stream processing concept: even when they are present, QoS features are very often hidden or not directly controllable by the final user. However, limiting QoS visibility also limits the possibility for users to customize DSPS services for their very own stream processing requirements, which, in turn, reduces the opportunities for platforms to optimize resource usage accordingly and to achieve better scalability in large scale processing scenarios.

We believe, instead, that novel DSPSs should be designed and implemented around the concept of QoS-based configurability, and that QoS should be considered as a central entity at all the abstract, development, and execution model levels. Moreover, platforms should expose complete visibility of their QoS features and clearly explain the associated scalability–costs trade-offs so that users can always choose the one that best adapts to the requirements of different stream processing applications. Experienced users should be allowed to make this choice by directly configuring the platform QoS, or else smart middleware layers built on top of the stream processing system should be allowed to make their adaptive configuration choices. As antic-

ipated in our QoS-aware model presented previously in this chapter (Section 3.3.2), the abstract model should let developers associate QoS-specifications to every abstract-level entity, such as sources, operators, sinks, edges, or entire graphs. Similarly, the development model should define ad-hoc syntactical constructs or APIs for the description of QoS specifications at application development time and, most importantly, the execution model should define how different properties map to runtime mechanisms that take care of their enforcement.

The new class of QoS-centric DSPSs should struggle to support the widest possible range of QoS properties, from high level indications such as priorities or latency requirements, which are normally easier to understand and use by novice users, to very low level configuration details, such as parallelism, queuing, or networking specifications, which can help more experienced users to customize platform services to their own needs. In addition, we believe that there are still many open research directions to explore in the area scalable QoS-centric DSPSs, in particular for what concerns the definition, implementation, and validation of novel *weak* QoS properties. This important class of properties should propose new and flexible trade-offs between cheap and highly scalable best-effort QoS properties and more expensive strong QoS properties that, while guaranteeing always deterministic and consistent behavior, may hinder system scalability. Differently from best-effort QoS, weak QoS should still offer well-defined behavior guarantees but it should be possible to enforce these guarantees more flexibly when compared to those provided by strong QoS properties. *Probabilistic* QoS is probably the best example of weak QoS properties: to express guarantees in probabilistic rather than deterministic terms provides the platform with additional degrees of freedom for what concerns how and when to use possibly expensive QoS enforcement mechanisms.

3.7 SUMMARY AND CONCLUSIONS

In this chapter, we analyze the possibility to use Data-Intensive Scalable Computing (DISC) systems to face the scalability challenges posed by the novel Big Data analysis scenarios emerging from SPEs. In particular, given the streaming nature of SPE data flows and the near-real-time processing requirements of SPE applications, we focus on *stream-oriented* DISC systems, i.e., Distributed Stream Processing System (DSPS), and we analyze the current state-of-the art with particular attention on system features related to QoS-based configurability. After proposing an original model for DSPSs based on a layered three-levels architecture, we classify commonly supported and envisioned QoS properties for these systems. We then survey six important state-of-the-art DSPSs, noteworthy for their unique support to QoS-related

features, and emphasize design and technical aspects related to the implementation of scalable mechanisms for the runtime enforcement of different QoS properties.

Through our analysis we identify two main directions of future work for building novel and QoS-centric DSPSs. First, although many works have faced the problem of QoS-based behavior in DSPSs, there are still no solutions offering a comprehensive QoS framework that lets users understand and choose the most suitable quality–cost configuration for their peculiar stream processing requirements. Second, only a few systems have explored QoS properties providing *weak* or *probabilistic* QoS guarantees: we believe that, by using this kind of properties, new flexible and adaptive QoS enforcement mechanisms can be realized, with the potential of greatly improving systems scalability. In the next two chapters we present our efforts in the two directions above: in Chapter 4 we present Quasit, an original QoS-centric model and framework for distributed stream processing, while in Chapter 5 we introduce LAAR, a novel technique for partial fault tolerance that lets user trade platform runtime costs off for consistency.

4 A FRAMEWORK FOR QUALITY OF SERVICE AWARE STREAM PROCESSING

In the previous chapter, we have defined our model for DSPSs and we have introduced and motivated a set of requirements for stream processing in SPEs. All of them orbit around the concept of QoS-centric DSPSs, i.e., scalable processing frameworks that put QoS-based configurability at the core of their functionalities. In this chapter, we present our contribution toward the development of this novel class of systems by presenting Quasit, an original stream processing model that is able to express and support application-driven quality requirements in order to achieve the most suitable trade-off between efficiency, scalability, and differentiated QoS. We present the main principles behind the Quasit stream processing model and we report our design and implementation experience in building the prototype of a distributed and scalable platform supporting the execution of Quasit stream processing applications on clusters of off-the-shelf multi-core computers. We discuss the Quasit modular architecture and present its prototype implementation that integrates several state-of-the-art technologies (e.g., actor-based threading and DDS-based inter-process communication) and combines their strengths in order to offer a *scalable*, *lightweight*, *easy to use*, and *easy to extend* system. The source code of our Quasit prototype is freely available for download from the Quasit project Web site [216].

The rest of the chapter is organized as follows. After comparing Quasit with other related systems in the literature, in Section 4.2, we explain the main design principles that guided our design and implementation work. Then, in Section 4.3, we present the Quasit stream processing model using the three-layer description framework introduced in Chapter 3. In Section 4.4, we outline the architecture of our DSPS, and we discuss the most relevant aspects of its implementation in Section 4.5. After presenting an experimental analysis of our framework in Section 4.6, we report the main lessons learned from the design and development of the Quasit platform. Guidelines for future Quasit extensions and final considerations conclude the chapter.

4.1 RELATED WORK

Google MapReduce [24] and Microsoft Dryad [25] are very popular solutions supporting scalable data processing within large data centers. Albeit their relevant differences, both systems offer a simplified computational model that permits to define custom and application-specific data analysis algorithms, and both implement a runtime platform that executes these user-defined tasks on data center infrastructures simplifying the management of complex details such as concurrency, networking, or fault tolerance. Both MapReduce and Dryad are batch-oriented DISC systems (see Section 3.2), hence designed for data crunching scenarios where the whole input data is statically known.

Nonetheless, given the remarkable industrial success of MapReduce, and also thanks to the availability of the open source Apache Hadoop [30] implementation, many authors have extended the framework trying to adapt its model to continuous processing problems [161–163, 217, 218]: all these approaches represent operations on data streams as continuous sequences of batch jobs on windows of the input. Being based on an inherently static data model, however, these solutions struggle to describe highly dynamic stream scenarios, and their implementations do not always offer adequate performance for applications that call for low processing latency.

Being designed to work with streaming data, Quasit shares several aspects with traditional DSPSs. More specifically, our system falls in the category of Distributed Stream Processing Engines (DSPEs) (Section 3.2) because i) it focuses on scalability rather then strict data consistency, and ii) it does not define a formal query model but lets developers build their own general-purpose stream processing operators. Quasit extends the set of common DSPE features by putting QoS at the center of its processing model in order to let users have a very fine grained control over the runtime behavior of their applications.

The DSPEs sharing the largest number of characteristics with Quasit are Apache S4 [219], Storm [29], and IBM InfoSphere Streams [166]. Like all of them (and as the vast majority of DSPSs presented in Chapter 3), Quasit stream processing problems are modeled through processing graphs, and custom operators, sources, or sinks are defined using a general purpose API. Unlike Apache S4 or Storm, which adopt task parallelism greater than one by default (see Section 3.3.3 of Chapter 3), and similarly to Streams, Quasit uses a simpler base approach, by instantiating only one task per operator by default. While different DSMSs and DSPEs implement, with different success, various QoS-based mechanisms, they do not provide a consistent and comprehensive framework that allows the definition and enforcement of rich QoS specifications: Quasit is the only DSPS we are aware of whose processing model is specifically designed to allow extensive user-driven customization of processing behavior by means of QoS specifications.

4.2 DESIGN PRINCIPLES

Before presenting the details of the Quasit processing model and distributed architecture, in this section, we illustrate the principles that stand as the foundation of our design and development work. We summarize these principles as a list of five attributes for our Quasit DSPS solution, which are:

1. QoS-*centric*.
2. *Scalable*.
3. *Modular*.
4. *Reusable*.
5. *Intuitive*.

We have already discussed about the importance of QoS awareness in stream processing and, in the previous chapter, we have also shown how previous state-of-the-art system have contributed, often only partially, toward this objective. Quasit has the ambitious goal of realizing a QoS-*centric* stream processing solution that, at the same time, summarizes in a single and consistent model previous QoS-related contributions and offers a platform that can be used to develop, experiment, and compose novel stream processing QoS mechanisms.

Scalability is obviously a highly desirable property in any distributed system. With Quasit, we aim at being scalable on three different and complementary levels, with a *scalable model*, a *scalable architecture*, and a *scalable implementation*. Model scalability refers to the availability of a set of abstractions that is simple enough to describe and solve small problems without excessive developer effort, but also sufficiently expressive so that the same set of abstractions can be used to deal with more complex and large scale problems. Architectural scalability means that Quasit design should be easily extensible with new functional and non functional features. On the third level, the Quasit platform implementation should allow the scalable execution of both small and extremely large stream processing applications and, whichever the scale, use the computational resources available in data center environment transparently and effectively. Implementation scalability should be both *vertical* and *horizontal*: Quasit implementations should scale up seamlessly when using increasingly powerful processing nodes (e.g., with faster or additional CPUs or CPU cores) — vertical scalability — and scale out when adding more servers to the data center deployment — horizontal scalability.

Modularity should drive both the design of the stream processing model abstractions and the implementation of the runtime platform that executes that model. Modular abstractions enable model scalability by means of composition mechanisms, and a modular design and implementation promote the construction of platform extensions by helping potential platform developers to understand the system architecture and its implementation details.

A modular and component-based design has the additional advantage of fostering component *reusability*. We aim at offering a collection of features that let application developers define, share, and reuse self-contained stream processing components. This should make the platform adoption curve more gentle and accelerate the application development life cycle.

Finally, all the parts of the framework directly exposed to developers, i.e., its abstract and development models, should be *intuitive* to use. The offered abstractions should aim at simplicity, which should be reflected with an execution model and programming APIs easy to manage and understand. Complexity should be hidden to developers whenever possible, but, at the same time, it should be possible to have full QoS-based visibility when necessary.

With these high-level principles in mind, in the next sections of this chapter, we present our Quasit architecture and implementation: the reader will realize that most of our design and development choices are motivated by one or several of these guidelines, and we try to underline with the most adequate emphasis where and how particular aspects of our platform were motivated by these general principles.

4.3 THE QUASIT MODEL

In this section, we present our original Quasit model for QoS-centric distributed stream processing. The presentation is organized by following the layered DSPS description framework presented in Chapter 3. In particular, in Section 4.3.1, we introduce the main Quasit stream processing concepts and analyze their relationships. The tools that developers can use to build instances of the abstract model are presented in Section 4.3.2, while, in Section 4.3.3, we discuss the execution model of a Quasit deployment on a set of distributed data center servers.

4.3.1 Abstract model

Quasit is used to process one or more input data streams, perform arbitrary transformations on their content, and produce as output other data streams, which can be fed to other systems for storage or further processing. A data stream is a temporal sequence of data samples, all of the same type, whose content is a set of key–value attributes; the *data type* of a stream is the type of all its samples.

The basic modeling unit in Quasit is the Streaming Information Graph (SIG), a weakly connected Directed and Acyclic Graph (DAG) that represents the information flow and the transformations that, applied to one or more input streams, produce one data stream, the output of the SIG. The nodes of a SIG represent data transformation

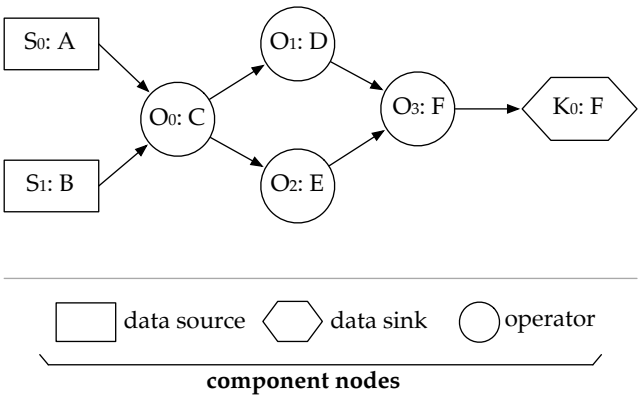


Figure 4.1: Structure of a Quasit SIG with two data sources, one sink, and four operator nodes. Sources S_0 and S_1 produce two data streams of typeA and typeB respectively; Operator O_0 processes them and generates a typeC data stream, which is, in turn, received by O_1 and O_2 that output typeD and typeE data streams. Finally, the typeF data stream generated by O_3 goes into data sink K_0 of the same type.

stages, while its edges represent communication dependencies. Figure 4.1 shows an example SIG; note that, at this level of abstraction, a SIG corresponds exactly to the concept of processing graph introduced in Section 3.3, with the only notable addition of having typed nodes and edges and only one data sink. A SIG node can be of three different kinds, i.e., *operator*, *data source* or *data sink*.

- A *data source* node has no ingoing edges and at least one outgoing edge; a SIG contains at least one data source node.
- A *data sink* node has exactly one ingoing edge and no outgoing edges; any SIG contains exactly one data sink node.
- An *operator* node has one or more ingoing edge and at least one outgoing edge, and defines a data transformation; a SIG usually contains at least one operator node.

A *data source* node identifies an external data stream and its role is to abstract from the actual nature of the stream producer (for example, it can represent an external stream source or the output of another Quasit SIG). We say that the data source *generates* that data stream. A *data sink* node, conversely, represents the destination of the SIG output data stream; data sinks can be used either to redirect output streams to other systems for further processing or storage, or to connect the output of a SIG with the input of another one. An *operator* node is associated to one or more input data streams and it *generates* exactly one output stream. SIG edges represent communication dependencies between components; every edge is associated to the data stream generated by its source node, and all the edges going out from the same node are associated to the same data stream. Nodes, edges and the SIG as a whole are conventionally given a type as well: the

type of a source or operator node is the type of the data stream it generates, the type of a data sink is the type of the data stream it receives, an edge has the same type of its source node, and the type of a SIG is the type of its only data sink. Every element of a SIG (nodes, edges, or the graph as a whole) can be labeled with an optional QoS specification. QoS specifications allow users to enrich their processing graphs with additional information about non-functional requirements.

The Quasit abstract model is based on very simple concepts, but their unique property of being extensively composable makes the model description quite articulated. In the next paragraphs, we provide additional details on the SIG-based processing model and focus on the main processing component, i.e., the *Quasit operator*, and on the mechanisms and semantics of SIG-level QoS specification.

Extended abstract model

The goal of our abstract model is to provide a simple and small set of abstractions that permit to describe from small to large scale problems flexibly and uniformly. Figures 4.2–4.5 schematically show the elements of our abstract model.

There are two possible kinds of SIG, called *attached* SIG and *operator definition* SIG (OD-SIG), respectively (Figure 4.2). As shown in Figure 4.3, an attached SIG is defined as a SIG whose source nodes are all *attached sources*, that is, source nodes that are connected to actual data streams (as opposed to *virtual sources*, see the following). An OD-SIG, instead, is a special type of graph that, as the acronym suggests, is used to define new complex operators starting from simpler ones, i.e., composite operators; in an OD-SIG all the sources and sinks must be *virtual*, meaning that they are not attached to actual data streams but only represent abstract endpoints. The presence of virtual data sources and sinks permits to define the input expectations and to declare the output type of an OD-SIG.

Figures 4.4 and 4.5 show the hierarchy of possible nodes that can appear in a SIG. As anticipated, sources and sinks can be virtual or attached; virtual sources and sinks appear in Operator Definition SIGs (OD-SIGs) and act as place holders for real data streams producers and destinations. An *attached source* defines an actual stream producer, which, in the simplest cases, corresponds to a *native source*, i.e., a data producer external from the Quasit DSPE. Attached sources are at the base of the first of the two SIG reusability mechanisms that we designed in Quasit, i.e., *graph concatenation*. To support this mechanism, we model any attached SIG to be also an attached source; this permits to use the output of any stream processing graph as the input of another one. There is only one type of attached sink, that is, *native* ones, corresponding to external stream destinations (e.g., a file system, a DBMS, or a PUB/SUB endpoint). Finally, operators can be either *simple* or *composite*. Simple operators are atomic stream processing units

directly defined by users through an ad-hoc operator API; composite operators, instead, can be used to implement more complex processing behavior by assembling OD-SIGs using other existing simple or composite operators. *Operator composition* is the second SIG reusability mechanism built in Quasit, and it permits to reuse existing stream processing components and build new and more complex ones in the form of composite operators.

Graph concatenation and operator composition are at the foundation of the scalability of our abstract processing model: by exploiting these features, the same small set of concepts can easily model from very small processing graphs made of a few simple operators to very large scale SIGs. In the next paragraph we concentrate on the Quasit operator, the core of Quasit stream processing functionalities.

Operators

Quasit operators can be *simple* or *composite*, and both types can be either *stateful*, if their behavior depends on their processing history, or *stateless*, otherwise. A *simple operator* logically consists of several sub-components, shown in Figure 4.6.

- *Ports.* An operator defines at least two typed *ports*: one or more *input ports* and exactly one *output port*. Input ports model the operator input requirements, while output ports represents its output contract.
- *State.* An operator can be either *stateless* or *stateful*. When stateful, its output depends also on its state, i.e., an object that summarizes previous processing history.
- *Processing Function.* The processing function is the reactive core of the operator. It is a user-defined function asynchronously invoked by the Quasit framework as samples arrive at its input ports. If the operator is stateless, the outcome of the function only depends on the data sample being processed; if stateful, it also depends on the most recent value of the operator state. The output of the processing function is a tuple of two elements. The first is a sequence of samples to generate in the operator's output stream, while the second describes the updated operator state. In other words, by defining the processing function, developers specify the set of transformations that, applied to the input, produce its output and state transitions.

This operator processing model is purely event-based: an operator produces output and/or changes its state only in response to incoming data; this model encourages simpler operator design, and also fosters scalability by permitting a large number of operators to share processing resources very efficiently. Furthermore, the sharp separation between the behavior of the operator, expressed through its (stateless) processing function, and its state gives Quasit great *flexibility* in taking

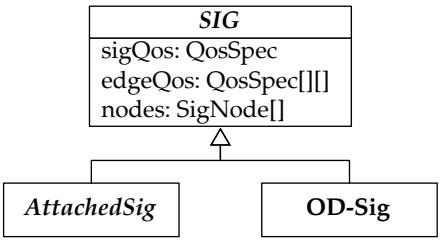


Figure 4.2: Quasit abstract data model: SIG

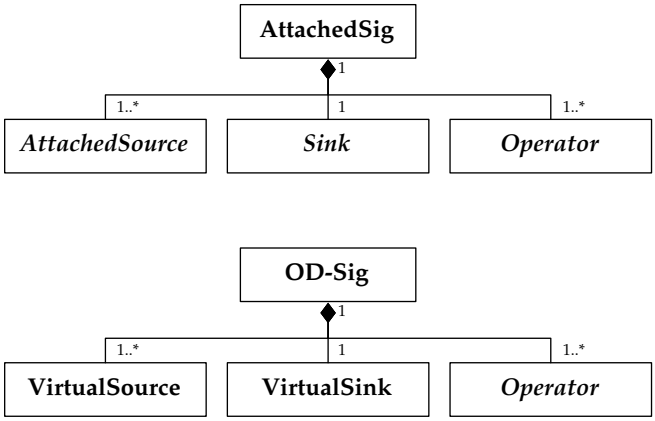


Figure 4.3: Quasit abstract data model: Attached vs. OD-SIG

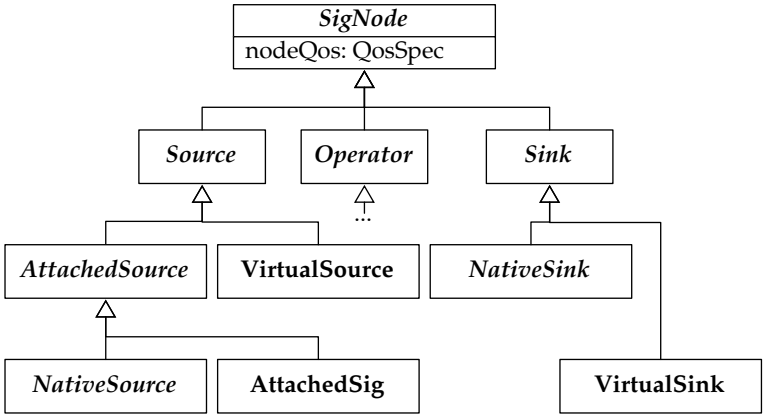


Figure 4.4: Quasit abstract data model: Sources and Sinks

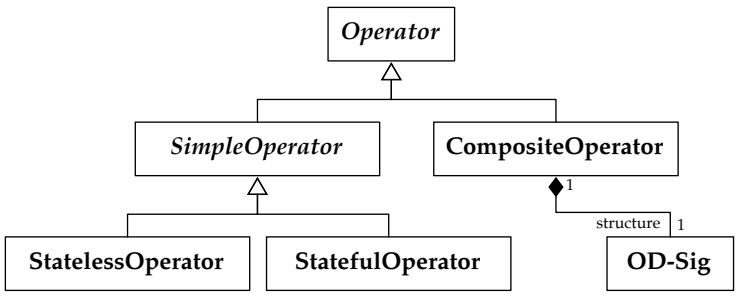


Figure 4.5: Quasit abstract data model: Operator

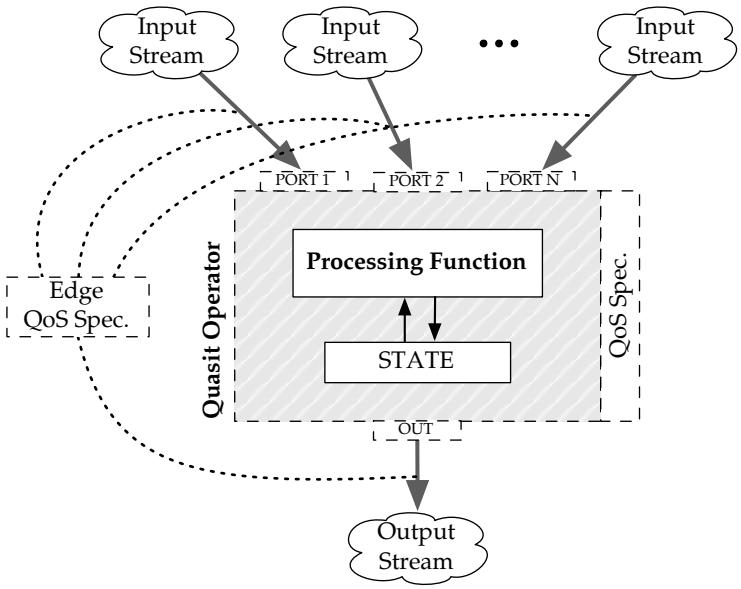


Figure 4.6: Structure of a Quasit *simple operator*.

transparent management decisions at runtime, in order to effectively support the execution of operator components. For instance, complex and differentiated state persistence policies can be implemented more easily by keeping state and processing logic completely separate.

As introduced previously in this section, *operator composition* is the mechanism that enables to reuse existing components as building blocks for creating, through OD-SIGs, more complex and powerful ones, i.e., *composite operators*. An OD-SIG completely defines the execution characteristics of the composite operator. Virtual source (sink) nodes are used to describe the input (output) ports of the composite operator, and the remaining nodes in the graph and their connections define how the composite operator behaves in response to new samples arriving at its input ports: when a data sample comes in one of the composite operator input ports, the framework will handle it as if that sample is processed by the OD-SIG operator graph.

Let us remark that, since composite operators can be used as nodes in an OD-SIG, operator definitions can be nested at will (infinite loops are avoided by not allowing recursive operator definitions). Operator composition permits to encapsulate complex behavior into modular units, with obvious reusability advantages.

QoS support

The most ambitious goal of Quasit is to provide extensive support for QoS-based configuration. For this reason any element of the abstract model can be enriched with an optional QoS *specification*, defining a set of non-functional configuration parameters or constraints. A QoS specification is a set of QoS *policies*, each policy influencing a different

Table 4.1: Quasit QoS policies.

Element	QoS Policy	Possible values
Data Sink	Priority	*Priority value*
Operator	Processing Cap	*Time threshold*
Operator	State Fault Tolerance	*Replication factor*
Operator	State Consistency	*Lazy*, *Snapshot*, *Strong*
Operator	Queuing Spec.	*Input queues size*, *Scheduling policies*
Channel	Delivery Semantics	*Best Effort*, *At most once*, *At least once*, *Exactly once*, *Probabilistic*
Channel	Deadline	*Time threshold*

quality aspect. The set of values specified for different QoS policies on different SIG elements guide the platform for the allocation of resources at runtime according to the required quality levels.

The set of possible policies that can be used in QoS specifications for the various elements is open and extensible: platform developers can easily add their own at the abstract model level and implement the related enforcement mechanisms on the underlying runtime. Table 4.1 reports a short list of Quasit QoS policies, by concisely showing their applicability scope and their possible values.

In order to provide readers with an overview of the practical aspects that can be regulated through QoS-based augmentation of SIGs, in the following we give a short description of these policies, and put them into their practical applicability context by presenting examples of their possible use within a simple SPE 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 made available to citizens, who can request, use, or share them through a smartphone 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, which processes and matches citizen requests for car trips with car availability.

- *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. Starting from a data sink to which this policy is attached and moving backwards through the graph, every operator is assigned a priority value, which is used by the local resource scheduler to assign computational resources. 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 available cars.

- *Processing Cap*. This QoS policy sets 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. The default action could be, for instance, to produce a predefined output, or to discard any partial result. The policy is mostly useful when, for some computation, controlled timing is more important than result completeness. For instance, the operator that classifies incoming car requests and routes them within the appropriate branches of the Quasit SIG 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. According to the values provided for these properties, Quasit can provide no consistency guarantee, best-effort consistency, or at least once processing consistency (see Section 3.3.3). For instance, when an operator is configured with strong state consistency and a replication factor of 2 and whenever its processing function updates its state, the operator blocks until the update is persisted and replicated at least on two different nodes, thus providing at least once guarantees. 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.
- *Queuing*. This low-level policy controls the way Quasit manages the operators input queues. This policy allows to configure i) the size of input queues, by practically influencing the trade-off between processing latency and sample drops, ii) the queue behavior when it becomes full, and iii) an ordering function that determines how samples are dequeued before being processed by the corresponding operator. 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 samples related to requests from gold members.
- *Delivery Semantics*. This QoS policy, associated to SIG edges, configures the communication protocol used by the Quasit platform to enable data exchange between operators connected by that edge. For example, if at-most-once delivery semantic is set, the destination operator will keep a list of already-received data samples and discard possible duplicates. For instance, pollution-

map updates can be transferred using a best-effort communication protocol because of information redundancy.

- *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. To handle cases when the deadline cannot be satisfied, a fall-back policy can be selected (e.g., discarding late tuples). 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.

A final but important aspect to emphasize is that Quasit QoS policies regulate heterogeneous aspects of the underlying platform, and are often defined at different abstraction layers: while some policies, such as priorities, model more closely application-level expectations, some others, like queuing specifications, are more related to system level optimization parameters. We believe that the organization of QoS policies in an hierarchy of increasingly platform-specific configuration opportunities, on the one hand, lets experienced users perform a very fine tuning of the system behavior, while, on the other hand, offers, to less experienced ones, the possibility to use simpler and coarser grained indications that the system can use to configure lower level parameters accordingly.

4.3.2 Development model

According to the classification in Section 3.2, Quasit is positioned in the group of DSPEs. Developers of stream processing applications can freely define their own data samples types and can build arbitrary processing graphs made of operators performing arbitrary work on the typed samples.

Quasit exposes three related sets of APIs:

- The *type definition* API.
- The *operator* API.
- The *graph* API.

The type definition API allows to build custom stream processing types that model application-specific data samples. It is based on the OMG Interface Definition Language (IDL) [220], which, with a simple and C-like syntax, permits to define rather articulated type systems in a way that is fully decoupled from the specific implementation language. The operator API is an object-oriented API that exposes classes and methods used by developers to build their own stream process-

ing components, i.e., custom native sources or sinks, and simple operators. All the functionalities implemented in this API turn around the concept of *descriptor*: users write source, sink, or operator descriptor classes, whose instances contain all the information that Quasit requires to instantiate the actual components at runtime. Finally, the graph API publishes a set of functionalities that permits to assemble Quasit component descriptors in directed graphs corresponding to attached SIGs or OD-SIGs: bundled into an execution archive, the first type of SIGs represent stream processing applications ready to be deployed on a Quasit cluster; as described before, OD-SIGs can be used, instead, to define new and arbitrarily complex composite operators. QoS-based augmentation of components is supported by both the operator and graph APIs. In the second, it is possible to associate instances of ad-hoc QoS specification classes to every component to define the quality-levels required by the application being built. Complementarily, in the first, it is possible to set the most appropriate default QoS values for the operators being designed, or to disable unwanted QoS policies values.

Currently, there is only one implementation for the operator and graph APIs, built for Scala [221]. We choose this general-purpose programming language as the first developer interface to Quasit for three reasons: i) its elegant syntax permits to map Quasit abstract concepts on programming constructs very concisely and effectively; ii) it encourages a functional-like programming style with strong separation between state and behavior — the same separation we foster in the operator model (recall Figure 4.6); iii) it integrates seamlessly with the rest of the platform that, as we explain later in this chapter, is developed using the same technology. However, given the strict relationship between Scala and the Java programming language, it is possible to invoke the Scala APIs from Java programs as well with only a few special restrictions. An example of how to use the Quasit operator and graph APIs is reported later in this chapter (Section 4.6).

4.3.3 Execution model

Like other systems for data-intensive scalable computing in data centers [24, 25, 29, 164, 166, 222], the Quasit distributed architecture follows a *master-workers* model, where a logically centralized node (the *master*) implements management and coordination tasks, while a possibly large number of *worker* nodes perform data processing tasks. In particular, Quasit user-defined SIGs are deployed and executed by a set of Quasit Runtime Nodes (QRNs), which are monitored and managed by one Quasit Domain Manager (QDM), as shown in Figure 4.7. The set of QRN 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 and QRN failures, and

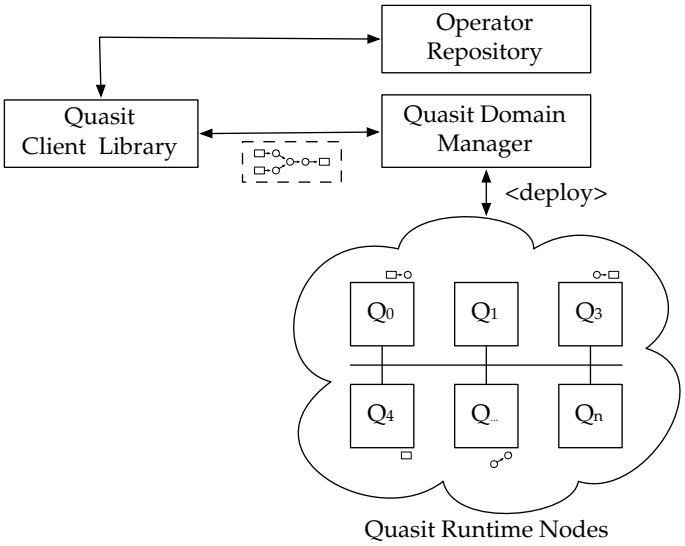


Figure 4.7: Distributed architecture of a Quasit domain.

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 LAN, as typically occurs in data center scenarios [223].

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 workers and executed in parallel. 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 continuously monitoring their distributed execution. As soon as a new SIG is received, the QDM decides an initial graph partitioning by running an *operator placement algorithm* that determines how the graph components are executed by the available QRNs. Different operator placement algorithms can have different optimization goals, including, for example, execution cost, processing latency, or resiliency to failures. Custom placement algorithms can be easily plugged in Quasit by defining the implementation of ad-hoc object oriented interfaces. At the time of writing we support two simple placement algorithms: a *uniform* algorithm, which distributes operators fairly among the QRNs according to the graph topology, and a *random* algorithm, used mainly as reference for comparison.

A QRN implements a QoS-aware execution container for Quasit operators providing scheduling and communication support. Reflecting the operator model, the QRN execution model is *asynchronous and event-based*. Every operator is associated with an input queue that contains samples received from any of its ports, and the container assigns the available execution threads to operators having at least one message queued, executes their processing function and possi-

bly updates their state, and produces new samples on their output stream. At OS-level, every QRN corresponds to a distinct process (a JVM process), and there normally is only one QRN per cluster server. Simple operators are instantiated as JVM objects and their processing functions are executed by a shared pool of threads whose size is configurable through a platform-level *threading* QoS policy (if not overridden, the platform creates two threads per available CPU core by default). The choice of a *process-per-server* model (see Section 3.3.1) is motivated by two main reasons: first, hosting all local components in a single OS process makes it possible to implement flexible resource allocation and meta-scheduling policies, for example by handling in a very fine-grained way the management of the available execution threads, and second, this model permits to minimize the communication overhead for co-located operators, enabling very efficient sample exchange through JVM-local shared memory.

Finally, in order to foster component reuse, Quasit permits developers to publish custom simple or composite operators to a special component, the QUASIT OPERATOR REPOSITORY (QOR). The QOR is optional and conceptually separated from the rest of the Quasit deployment, and actually not strictly needed for a Quasit domain to work. As the name suggests, its role is to offer a directory service that developers can use to look up for existing operators.

4.4 ARCHITECTURE

In this Section, we dig into the architecture of the main Quasit runtime components, i.e., the QDM and QRN runtime services, and the optional QOR. We describe the modular internal organization of the Quasit platform and explain the main high-level choices that characterize its design. The presented architecture is designed to be as much general as possible, independent from specific implementation technologies. For this reason, here we concentrate only on the discussion of the responsibilities and interactions of the main modules, and we delay the description of their actual realization to the next section.

4.4.1 Quasit domain manager

The QDM service has management and control responsibilities over a Quasit domain, but it does not take any direct role in stream processing tasks. For this reason, although there is only one QDM per Quasit domain, its centralization does not represent a relevant bottleneck to the overall system scalability. The QDM functionalities are implemented through the interactions of the modules shown in Figure 4.8.

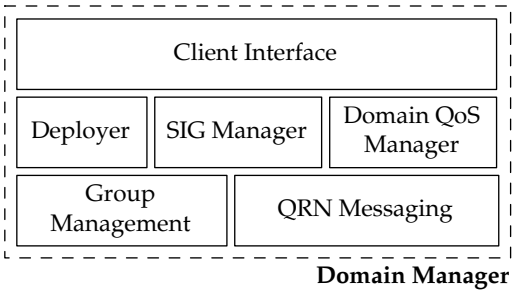


Figure 4.8: QDM Architecture.

The *client interface* implements and exposes the protocol that is used (through the Quasit client library) to administer and interact with the domain. While the client interface exposes the QDM services to final users, the QRN *messaging* regulates the data exchange between between the QDM and the QRNs in its domain. Via this interface, the QDM sends, for example, operator deploy messages, subscribes to QRN monitoring information, issues QoS adaptation commands. The *group management* module keeps a consistent and always up-to-date view of the domain members and provides hooks to register for notifications in case of membership changes. The SIG *Manager* module maintains information about the SIGs currently running in the domain and about the status of their associated operators. To do so, it interacts with the QRN messaging module and subscribes to periodic status information sent by QRNs. The *deployer* module is only used at SIG deployment type and it is responsible of running the chosen operator placement algorithm that determines the allocation of sources, sinks and operators to available QRNs. Finally, the *domain* QoS *manager* has central QoS coordination responsibilities. Like the SIG manager, it subscribes to monitoring data from the domain QRNs, and, according to their values, it orchestrate QRN adaptation actions.

4.4.2 Quasit runtime node

The QRN provides Quasit operators with a rich execution environment, by implementing *threading*, *networking*, and *fault-tolerance* functionalities. By design, many model-level concepts like SIGs or composite operators are completely transparent to QRNs, which are, instead, only aware of *simple* operators. The QDM, which, on the contrary, has a full view of all the Quasit abstract model elements, hides these details from the QRNs: every management action, including requests to deploy new components, is expressed in terms of simple operators by resolving, when necessary, composite operators to their atomic units through OD-SIG rewriting. The reduction of the abstractions available at QRN-level to just the essential ones has helped tremendously in keeping the implementation of the service simple and flexible.

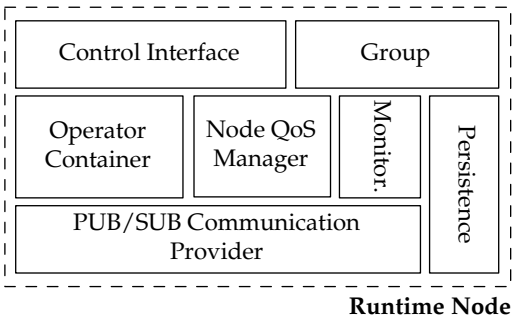


Figure 4.9: QRN Architecture.

Like the QDM, the QRN has a modular architecture that defines a set of interacting components, as shown in Figure 4.9. The remote interactions between the QRN and the QDM are coordinated by a *control interface* module, which listens for commands from the QDM and dispatches them to the internal modules. While the control interface takes care of managing one-to-one interactions with the domain manager, the *group membership* module is responsible of group communication services. When started, it announces the presence of the QRN to the domain and keeps a local membership view up to date. The *operator container* is the core component of the QRN. It controls all the mechanisms behind the execution of operators: for example, it creates local operator instances in response to QDM commands and executes them according to the Quasit execution model. To handle updates in the operators processing state, the operator container leverages the services of the *persistence* module. This latter module implements the persistence mechanisms that are used according to the state consistency policies required by different operators in order to make their state persistent and available also in spite of failures. Inter-QRN communication services are implemented by the PUB/SUB *Communication Provider*, which offers a QoS-aware PUB/SUB abstraction through which operators exchange their streams. The choice of a PUB/SUB communication model provides several advantages, the most important ones being i) complete transparency with respect to physical operator placement, which enables easy implementation of operator relocation policies, ii) clean modeling of one to many stream channels, by having all the recipients of one stream subscribe to it, and iii) a convenient way to enable SIG composition, by publishing a SIG's output on a PUB/SUB endpoint and allowing subscriptions from other SIGs. The *monitoring* module continuously collects the status of local operators together with up to date information about physical server resources availability, and it forwards these data to monitoring subscribers, which include the domain manger (through the control interface) and the *node* QoS *manager*. This last component controls that the required quality-level is met for what concerns locally running components: in case violations are detected, it tries to solve the problem autonomously

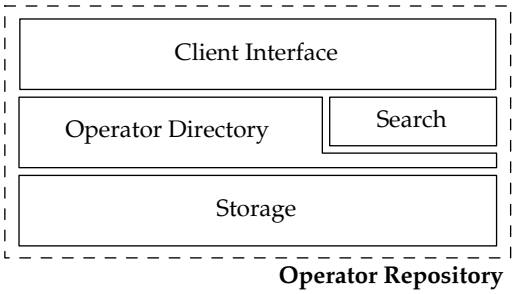


Figure 4.10: QOR Architecture.

before escalating it to the attention of the QDM global QoS manager, which has a broader view of the whole cluster.

4.4.3 Quasit operator repository

The QOR is an optional component, whose goal is to implement a repository of user-defined simple and composite operators, and a directory service where user can publish, search, and retrieve existing stream processing components. Interactions with the QOR take place at application development time, and, hence, its services do not have any influence on the actual stream processing operations. Figure 4.10 shows the high-level design of this component.

The *client interface* module implements the protocol(s) that expose the repository functionality to both users, for interactive usage, and client libraries, for programmable access. At the lower level, a *storage* module has the task of making the archives containing operator definitions always persistent and available, for example by saving them on a local or distributed file system structure or DBMS. In the middle, the *operator directory* manages and organizes the meta-data of the operator archives stored in the lower-level layer. Interacting with the directory, the *search* module provides directory search services, thus letting users search for existing operators by name, functionalities, supported QoS levels, or via custom and user-defined attributes.

4.4.4 QoS management

QoS policies defined on Quasit SIGs are enforced at runtime thanks to a two levels QoS-management architecture, sketched in Figure 4.11, and realized through the interaction of a *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 implement and enforce the requested policies locally.

In order to provide a better insight about this management scheme, let us briefly examine its role in the process of deployment and ex-

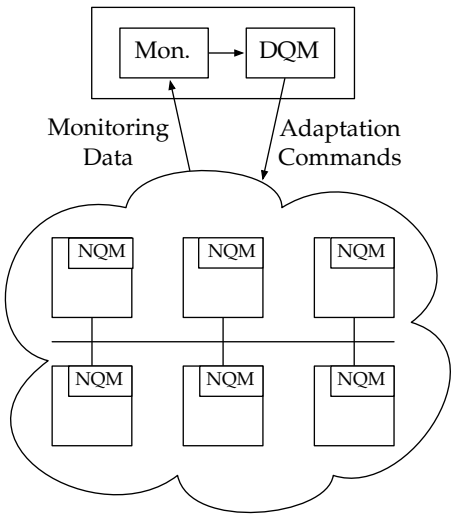


Figure 4.11: Two levels QoS management architecture.

ecution of a SIG. At *deployment time*, the domain QoS manager, after having verified the self-consistency of the SIG QoS policies, performs a translation phase, during which user-level policies are transformed to platform configuration parameters that 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 associated network queues. Node QoS managers use these data to set up the initial configuration for the operator instances 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 local operators, and try to autonomously adjust their configuration to avoid possible violations of the required quality levels; for example, they can reallocate their local resources by giving a greater share to operators with higher priority (thus, penalizing the less important ones). At the same time, they also forward monitoring data to the domain QoS manager, which uses them to take authoritative decisions in case adaption actions of single local managers are not sufficient to avoid QoS violations; for example, it can decide to move an operator from a QRN to another when the latter has more resources to allocate to its execution.

4.5 IMPLEMENTATION

We have implemented the model and architecture presented in the previous sections into a first system prototype. Its goal is to provide the basic system functionalities that can be used as a starting point for the implementation, validation, and evaluation of techniques for

```
..
val op = new StatefulOperator(name, outEP,
    initialState, pf)
  with ProcessingTimeQos[O]
  with SnapshotStateQos[O,S]
  with AsyncStateQos[O,S]
...
```

Listing 4.1: Scala trait mix-ins are used to instantiate an operator based on its QoS configuration.

the enforcement of quality-based behavior in Quasit. The prototype is released under a permissive BSD license [224], and its codebase can be freely downloaded from the project Web site [216].

The core of the Quasit framework is implemented in Scala [221], a relatively young programming language developed at École Polytechnique Fédérale de Lausanne, which unifies principles coming from the functional and object-oriented programming paradigms under an elegant and concise syntax. Our decisions to use Scala rather than any other general purpose language is motivated by four considerations.

First, Scala is natively designed to support concurrent applications: for example, the language and its standard library promote the usage of immutable data structures, that, being *thread-safe* by definition, simplify the design of concurrent code and make it less error prone. Our Quasit framework prototype makes heavy use of multithreading, and we were able to profitably leverage the functional programming paradigm and immutable data structures to keep our implementation simple, robust, and highly efficient. Second, some advanced object-oriented mechanism available in Scala, such as *traits* and *mix-in* composition, were particularly useful to implement stackable QoS-based behavior: in fact, operators are instances of types that are defined at runtime by mixing-in a set of traits that depends on their associated QoS policies; as shown in the small code snippet in Listing 4.1 (directly extracted from the Quasit codebase), we encapsulate inside traits — composable units that define reusable behavior — the code implementing operator specific QoS enforcement mechanisms; when needed, the runtime invokes this code transparently by leveraging principles borrowed from aspect oriented programming [225]. Third, the elegant and concise syntax of the language allowed us to create, with relatively few effort, fluent and intuitive APIs that lets developers model and manage their stream processing applications in a very natural way. An example showing the usage of the operator API is presented in Section 4.6. Finally, another important motivation for our choice stands in the strong integration of Scala and the Java ecosystem: Scala code runs on the solid and widely supported JVM and is completely compatible with Java at the bytecode level; this means that it is possible to exploit all the large corpus of

existing Java libraries seamlessly from Scala code and that, conversely, Java code can use Scala APIs with little or no effort.

Despite these many positive aspects of the language that drove our choice on Scala, we also want to highlight an important limitation that emerged from our experience, and that is likely related to the young age of the technology: although the language itself is quite stable and mature, it still lacks the great volume of documentation and resources that is, for example, available in the Java or .NET world. For what concerns performance, we did not observe any macroscopic issue, but we have not concentrated our research efforts on accurate Scala performance profiling yet. However, we expect the possible performance overhead to be very limited compared to a pure Java implementation, as also confirmed by the benchmarks in [226].

In the rest of this section, we delve into some in-depth technical details about the current prototype implementation. In our discussion, we report our experiences and highlight the lessons learned from our work, with a strong focus on the issues related to the integration of different state-of-the-art technologies in a complex system as Quasit.

4.5.1 Quasit domain manager

The QDM is the central management component of a Quasit domain. Notwithstanding is central role in a Quasit deployment, our current QDM implementation is relatively simple, especially if compared to the QRN (see the next subsection). In fact, our QDM implementation exposes a limited set of functionalities to external users, which are:

- Submit, start and stop new stream processing applications.
- Monitor the status of currently deployed applications.
- Shut down the Quasit domain.

These simple functionalities are realized by the QDM client interface through a very minimal and ad-hoc request-response protocol that the QDM implements on top of TCP.

Group management is implemented by using a DDS-based membership management system: the QDM is the only subscriber of a special membership topic, where QRN reliably submit their presence; a heart-beats based mechanism (built in DDS) ensures that membership status is properly updated when QRNs appear or disappear from the domain. Membership information contain up-to-date names and addresses of available QRNs, which are used by the manager to contact them at runtime through the QRN messaging module. Like the client interface, this latter is based on a minimal ad-hoc point-to-point request–response protocol, again built on top of standard TCP connections. Information about the running SIGs and operators are kept in an in-memory cache and are currently not saved on permanent storage. The QDM also subscribes to monitoring information from its QRNs.

As it will be explained in more details in the next subsection, QRNs publish these data using standard JAVA MANAGEMENT EXTENSIONS (JMX) interfaces [227], which are accessed by the QDM through remote RMI based connectors [228]. As anticipated in the previous section, we currently implement only two placement strategies in the deployer module: the first *uniform* deployment strategy is based on the topological ordering of the processing graph decomposed into its simple operator components, and tries to distribute operators uniformly among the available QRNs while minimizing inter-QRN communication in a best-effort manner. The second one, called *random*, trivially computes random placements of operators: it is usually used only for comparison purposes. In the current prototype, the QoS management module is extremely simple: it just records the monitoring data from the QRNs, but it does not implement any global QoS control mechanism yet.

To deal with possible QDM failures, Quasit adopts a traditional *primary-backup* replication scheme [229]: given the relatively limited number of requests per second that can change the QDM state, this simple solution does not cause important performance penalties.

4.5.2 Quasit runtime node

The QRN is the most complex and advanced service of the current Quasit prototype. Since all the stream processing operations are executed within its boundaries, we implemented the component by trying to optimize its runtime performance as much as possible, while still avoiding, when possible, to sacrifice design cleanliness. Given the complexity of the QRN implementation, we decompose its description in four parts, each covering a different aspect.

Communication and networking

There are three main communication concerns for a QRN, the first being the support to point-to-point QDM–QRN interactions, the second the realization of inter-QRN communication for streaming data exchange, and the third the implementation of a group membership service. This list does not include the publication of monitoring data, which will be discussed separately later.

As already discussed while describing the QDM implementation, the first problem is solved by implementing a simple and TCP based communication protocol in the QRN control interface module.

The PUB/SUB communication module takes care of the second concern. Since most of the network traffic in a domain is originated by the channels between operators placed on different QRNs, it is of critical importance to develop a communication solution that reduces at minimum the data exchange space and time overhead. A concurrent requirement is to avoid strong endpoint coupling, in order to support operator mobility and location transparency. As anticipated,

our solution to solve the trade-off between these partly contrasting requirements has been to model inter-component communications with a PUB/SUB abstraction, which perfectly answers our decoupling requirements, and to implement it by using the OMG DDS standard for high-performance PUB/SUB data exchange [55, 63]. Concretely, the PUB/SUB communication provider module maps the output port of every stream source to a unique topic and, symmetrically, every input port (of either an operator or a data sink) to a topic subscription.

By implementing the operator channels via DDS, we had the opportunity to exploit a second but not less important benefit: in fact, the DDS standard allows a very fine-grained control of its low-level behavior by giving the possibility to associate topics, readers, and writers with a number of options in a rich set of differentiated QoS policies. Based on these basic DDS mechanisms, we have built the Quasit QoS policies associated to stream channels. On the other hand, the choice of DDS also comes with one major drawback, related to system usability: in order to provide type safety and low spatial overhead, DDS needs IDL descriptions of the used data types, which must be coded at application development time and preprocessed by an IDL compiler. This can be sometime a cumbersome process, so we are currently exploring the forthcoming *dynamic topic types* DDS extension [230], which should allow to bypass the IDL compilation step.

The group module is also implemented through a DDS-based solution. At startup, QRNs write their membership information to a dedicated topic, and DDS takes care of keeping this information globally consistent and up to date through a protocol based on periodic exchange of heart beat messages.

Operator management

The core task of QRNs is to provide an execution environment for instances of Quasit simple operators. Figure 4.12 shows a view of the QRN component that highlights the threading architecture used to process the flow of samples coming from the PUB/SUB module. One *demultiplexer thread* waits for data samples by using the DDS *wait set* component, which offers synchronous I/O multiplexing services similar to the standard POSIX `select()` system call [231]. As samples arrive, the thread dispatches them to interested operators by moving them to their message queues. The actual processing tasks are performed by a pool of threads that share the responsibilities of de-queuing samples according to a pluggable scheduling policy and running operator processing functions. Note that this asynchronous execution service suits perfectly the event-based abstract operator model presented in Section 4.3.1. The size of the thread pool is a configurable parameter and should be set according to the number of available CPU cores.

In the current version of the Quasit prototype, operator instances are implemented as concurrent actors [232, 233] managed by the

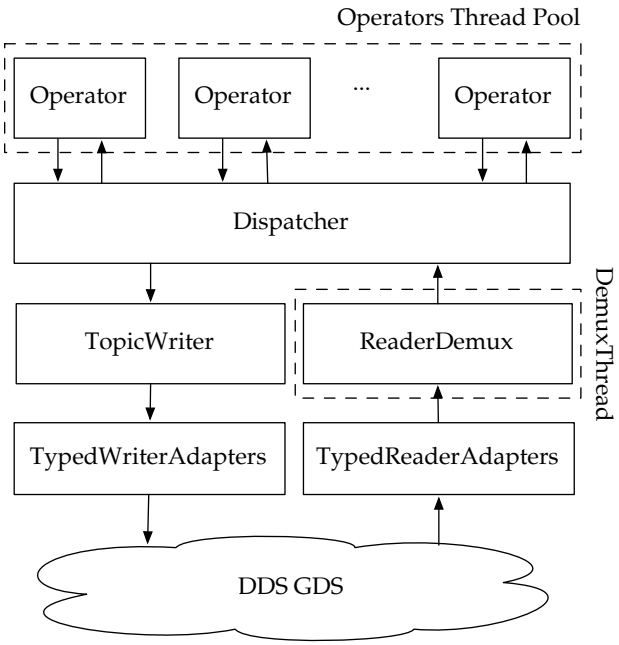


Figure 4.12: Data flow and threading model in a QRN instance.

Akka [234] actor framework. Actors are parallel and lightweight execution units that interact according to a message-oriented communication pattern. Every actor works in an environment that is ideally isolated from every other system actor: they share no state and no synchronization point, and act only in reaction to received messages. Akka implements these ideas on the JVM by providing development APIs and runtime services to build large scale actor systems. This actor model fits particularly well our processing scenario, where a graph of lightweight components (the operators) execute their *processing function* in response to the arrival of stream data samples. We associate an Akka actor to every Quasit simple operator; incoming data samples are dispatched to these actors' mailboxes and, by leveraging the Akka actor scheduler, they are processed by the operators' processing functions. On top of Akka, Quasit builds automatic message routing (i.e., developers do not have to specify the destinations of an operator's output in its processing function, which, instead, is automatically inferred from the SIG topology), advanced state management policies (handled by the QRN persistence module), and custom scheduling policies (by default, the standard Akka fair scheduler based on the *fork-join* framework [235] is used). Although Akka offers a network service based on JBoss Netty [236] that enables remote interactions between actors deployed on different servers, we do not use it for inter-QRN data exchange, as explained previously. Our DDS-based implementation, in fact, grants several advantages, including the fine-grained QoS configuration capabilities and the possibility to leverage IP-multicast for common one-to-many stream dissemination patterns.

Monitoring and QoS

In our prototype, the monitoring module is implemented through the coordinated use of standard JMX monitoring tools [227] and of the cross-platform monitoring and reporting SIGAR [237] library. The first permits to access JVM-level information (including detailed information about threading and heap memory allocation), while the second offers a very fine grained access to low level system information, such as global server CPU and memory usage, and OS-level status data. The collected information is published through a Quasit-specific JMX management interface and made available both to local and remote subscribers [228].

The node QoS manager is responsible of implementing local QoS enforcement mechanisms. At the time of writing, there is no global QoS coordination implemented in Quasit, and all the available polices are implemented with local scope only. In practice, in the current prototype, the role of the QoS manager translates in the definition of proper QoS traits that are applied to operator instances and that wrap ad-hoc the QoS mechanisms needed to enforce the associated QoS polices at runtime (recall Listing 4.1) and in the configuration of communication channels through the choice of proper QoS polices for the corresponding DDS endpoints. For example, priority based scheduling of locally running operators is managed by decorating operator instances with a `PriorityQoS` trait that specifies their priority levels (in a 0 to 9 range — 0 being maximum priority), and the priority scheme to use (currently *absolute* and *fair-share* priorities are implemented).

Fault tolerance

The QRN does not keep any global state by itself, but manages the state of locally running operators. Thus, guaranteeing fault tolerance despite its failures is the same as providing fault tolerance for the state of the operators that it runs. Differently from QDM, operator state dynamics can be extremely fast causing the realization of exactly-once or repeatable consistency guarantees (see Section 3.3.3) to be very expensive. This had to be taken into account carefully by our fault-tolerance solution in order to avoid severe negative effects on the overall framework performance. For this reason, and in accordance with the Quasit general approach, the strategies used to replicate or make operator state persistent are configured through ad-hoc QoS specifications. By default, the state is neither saved on disk nor replicated, but through appropriate combinations of QoS policies it is possible to specify the *fault tolerance back-end* (i.e., distributed in-memory replication or disk-based persistence) used, the *replication factor*, or the *checkpointing policy* (i.e., persist every state update or just take periodic snapshots), and whether the persistence mechanism should be *synchronous or not* with respect to the execution of operator pro-

cessing tasks. The persistence module offers specific hooks to request these fault-tolerance services and hides their actual implementation mechanism from the rest of the platform. At the time of writing, the implementation of the persistence module is still undergoing: we are testing the Hadoop Distributed File System (HDFS) [238] as a way to persist, with configurable replication guarantees, the operator data on the cluster disks, and the Redis key-value store for in-memory data replication [239].

4.5.3 Quasit operator repository

At the time of writing, there is no working implementation of the QOR service yet. Although the QOR has an important role in the overall Quasit architecture because it enables easy component sharing and promotes their reuse, its implementation has been delayed in favor of the more important QDM and QRN services.

Nonetheless, we have begun to explore possible technological solutions that would help to realize the operator repository service quickly and reliably. The main idea is to use Maven repositories [240] as the main building block for the QOR service. Maven is a comprehensive tool for the management of the entire development life cycle of software projects that, among a long list of other features, permits to publish so called *artifacts* (i.e., software archives) to remote repositories. Repositories can be accessed by other developers, which can use them to retrieve published software components through an HTTP interface. Common implementations of Maven repositories [241] implement artifact versioning systems, simple Web-based browsing, and minimal search functionalities (e.g., search-by-name). Our idea is to re-use these existing repository services by mapping operator components to publishable Maven artifacts, and to build the extended QOR features on top of these basic functionalities. For example, the *client interface* module should allow both interactive and code-based exploration of the repository, also by leveraging an extended *search* module that permits to search for operator components not only by name, but also by other attributes such as its description or the type of its input and output ports.

4.6 EXPERIMENTAL EVALUATION

In this section, we present some first experimental results that we obtained while testing our 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 mech-

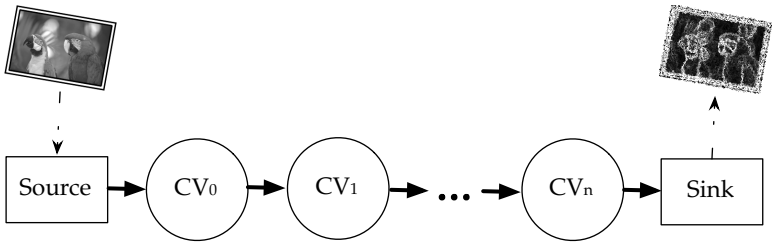


Figure 4.13: The simple and pipeline-shaped SIG used in this experimental evaluation.

anisms. 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 goals of our evaluation are the following:

- Present a possible use-case scenario for our system, by showing an example of the Quasit operator API (Section 4.6.1).
- Measure the management overhead of our distributed processing solution compared to to an ideal parallel processing scenario (Section 4.6.2).
- Demonstrate Quasit ability to scale up just by adding additional computing resources to a domain (Section 4.6.3).
- Compare the performance of the Quasit prototype with S4, a state-of-the-art stream processing platform developed by the Apache Software Foundation (Section 4.6.4).

4.6.1 Scenario description

The considered application scenario relates to the processing of a video stream, whose key frames need to be continuously analyzed through a series of image manipulation steps modeled as a pipeline of OpenCV [242] transformations, as shown in Figure 4.13. We chose this scenario because it is simple enough to illustrate the operator API concisely and, at the same time, it is representative of many common practical stream processing applications having the goal of extracting data from image streams or, more generally, of identifying specific patterns or sequences within incoming data streams. In order to avoid any bias in the measurement of the processing performance due to differences in the content of the video key frames, we artificially build the stream as the endless repetition of the same 192x128 24 bits per pixel lossless PNG image, chosen from a public image data set by Kodak [243].

Using our model, we represent the OpenCV transformation stages as stateless Quasit operators. Simple operators (either stateless or stateful) are described by instances of the `OpDescriptor` class, which encapsulates the information necessary for the runtime platform to instantiate and run the actual operator components. `OpDescriptor`

```
8  abstract class OpenCVOpDescriptor (
9    // Specific instances of the operator
10   // will initialize its name and QoS Spec
11   override val name: String)
12     extends StatelessOpDescriptor[ImageData](
13       name = name,
14       // Default QoS specification
15       qos = OperatorQoSSpec().withPolicy(
16         QueuingPolicy(QueuingPolicy.Unbounded,
17           QueuingPolicyKind.Fifo)),
18       // Declaration of the input ports
19       ports = Map("data" -> classOf[ImageData])
20     ){
21
22   override def init() {
23     OCVUtils.loadNativeLibs()
24   }
25
26   // abstract: implemented by subclasses
27   def processImage(in: IplImage): IplImage
28
29   override def processingFunction = {
30     case ("data", f: ImageData) =>
31       try {
32         val ipl = OCVUtils.imgToIpl(f)
33         val tranformed = processImage(ipl)
34         val img = OCVUtils.iplToImg(transformed)
35         Some(img)
36       } catch {
37         case e: Exception =>
38           warn("Error processing sample", e)
39           None
40       }
41   }
42 }
```

Listing 4.2: Base descriptor class for OpenCV-based operators.

instances are created by extending the appropriate base classes and then creating actual instances, as usual, with the `new` keyword. To maximize code reuse, we concentrate operations common to different OpenCV operators in the abstract `OpenCVOpDescriptor` class, from which all the concrete OpenCV operator descriptor classes inherit. The code of this base class is shown in Listing 4.2. Let us briefly analyze it, in order to show, with a concrete example, how our stream processing model maps on the operator API. Line 11 defines a class parameter which gives operator instances their names; line 15, associates a default QoS specification to all the OpenCV-based operators (in the example, it specifies that these operators should use unbounded FIFO message queues), while line 19 declares the only input port of our OpenCV operators, named *data*, and accepting streams of `ImageData` samples. The most important part of the class is the

definition of its processing function (lines 29–41). The body of a processing function is an instance of a Scala *partial function* built through a sequence of `case` statements [244]: whenever an input sample is ready to be processed, the block corresponding to the case alternative matching both the port name and the sample type is executed. In the example, if its operations are successful, the processing function returns the resulting sample wrapped inside an `Option[ImageData]` instance[1] (line 35); in case of errors, the function logs them and returns the `None` object (line 39) to indicate that no output should be produced. Recall that our OpenCV operators are stateless: this is the reason why no state value is involved in the processing function; differently, there would have been an additional element in the case pattern tuple (corresponding to the current state of the operator) and the returned object would have included a value representing the new operator state.

As one can easily see from the code, the defined processing function converts the image data from the external format (`ImageData`) to the corresponding OpenCV internal representation (`IplImage`) and vice-versa, and demands the actual image manipulation work to the `processImage` abstract function, which is defined by concrete subclasses. The `init` method, called as soon as an operator instance is created and before it receives any data sample, is used, in the example, to ensure that the OpenCV native libraries are correctly loaded.

We created four concrete subclasses of `OpenCVOpDescriptor`, respectively implementing *open*, *close*, *smooth*, and *dilate* functionalities, and we arranged their instances in a pipeline of forty operators. Although this very particular arrangement of image elaboration steps has no practical meaning from a computer vision perspective, it must be clear that our primary goal was to test our Quasit prototype in an artificial but yet realistic scenario.

The testbed consists of one machine running the QDM component plus from one up to four different physical servers 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 4.2.

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).

1 In the Scala standard library, the `Option[T]` class is used to model objects that can either have a value of type `T`, or no value at all.

Table 4.2: Hardware and software configuration of QRN nodes.

Configuration Details
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

4.6.2 Ideal parallel processing

In this section, we quantitatively evaluate the overhead imposed by the Quasit middleware for the distributed execution of stream processing applications compared to the maximum speed-up achievable thanks to an ideal parallel graph execution. To this purpose, we have designed a very simple simulator that models our pipelined processing scenario but omits the overhead associated with middleware-level operations (including operator scheduling) and inter-QRN network communication. The simulator models a group of parallel workers arranged in a pipeline, whose number reflects the number of available CPU cores across all the QRNs. The OpenCV transformations of our scenario are distributed to workers evenly, reflecting the original processing pipeline. In the simulations, we measure the average time needed to process an image sample completely as the input rate grows, and we compare the results with the performance data obtained on a real deployment environment with 4 QRNs (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 result (recall that, in our deployment, the server hosting the external data stream is the same receiving the SIG output stream). Figure 4.14 shows the distribution of the measured RTTs in the four QRNs deployment and the average processing time in the "ideal" simulated scenario for growing data generation rates..

In both cases, the processing time increases abruptly as soon as the available processing resources are 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 absent in the simplified simulated scenario.

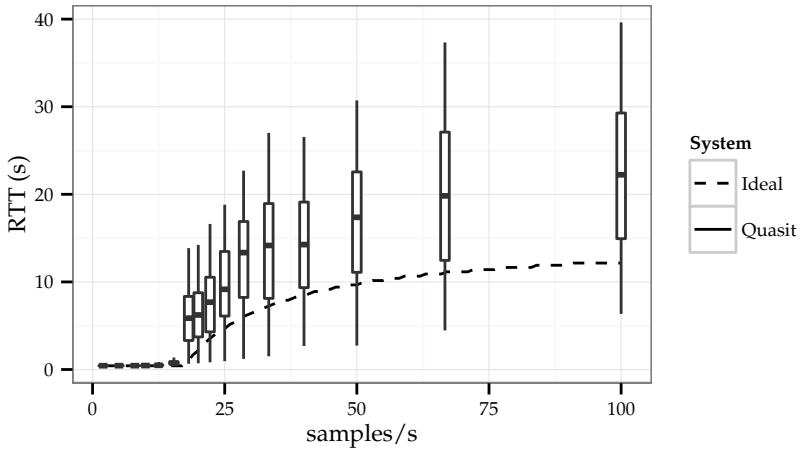


Figure 4.14: Distribution of sample processing time with 4 QRNs and uniform operator placement. The dashed line represents the performance upper bound in ideal conditions.

Table 4.3: 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

4.6.3 Horizontal scalability

About our second evaluation goal of verifying the ability of Quasit to scale when 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, and four QRNs. In all cases we have deployed the graph using the uniform placement strategy.

Figure 4.15 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 4.3 shows how the *critical sample-rate* (i.e., the data rate at which the system starts to be overloaded and accumulate samples at operator queues) varies by adding additional QRNs. Clearly, the speed-up values do not grow with a perfect linear trend with the number of available processing servers because of management overheads and network communica-

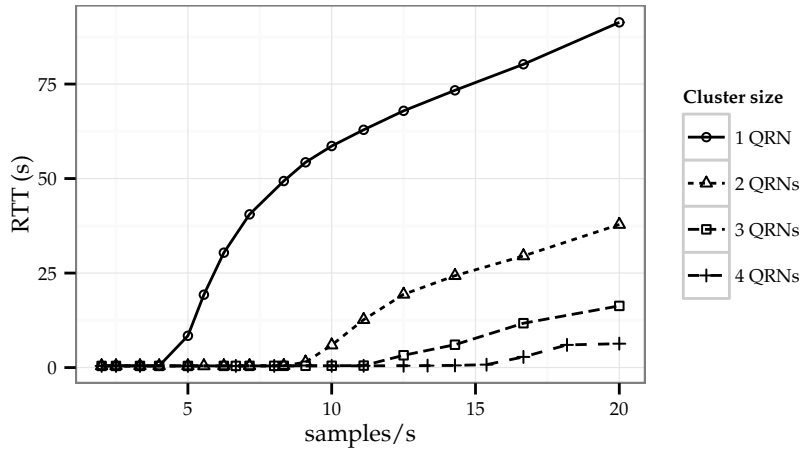


Figure 4.15: Comparison of average processing times using 1, 2, 3, or 4 QRNs and uniform placement.

tion, but still the performance degradation is very limited. Let us also note that the system ability to scale horizontally also depends on the characteristics of the application graphs: for this reason, Quasit fosters a SIG design made of many fine grained components, 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 experiment with the lowest input rate (i.e., 2 samples/s). Note that, in that set-up, just one QRN is enough to keep up with the data production pace: 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 processing servers. Figure 4.16 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 about 223 MB, i.e. more than double the resources. This is not really surprising, since the extra servers need to instantiate their own JVMs, which in turn load all the classes and instantiate all the objects needed for the management of the QRN itself.

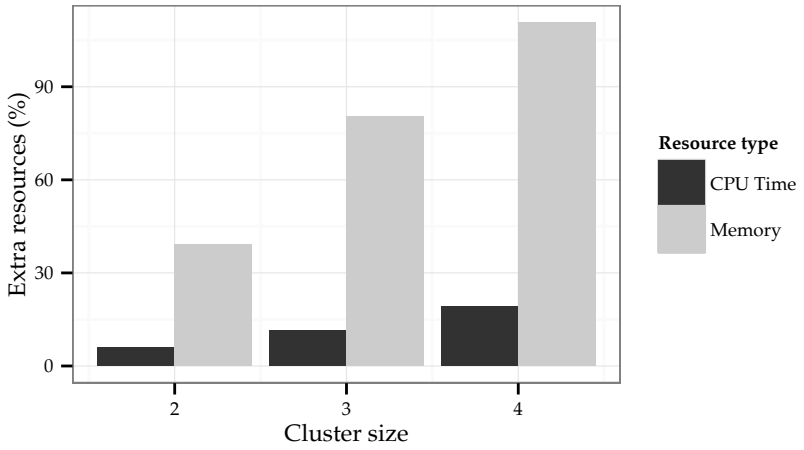


Figure 4.16: Overhead (in terms of extra resources needed) when adding additional QRNs

4.6.4 Apache S4

Finally, we report here a set of results that compare the performance of our system with Apache S4. The Apache S4 project [219], initially developed and maintained by Yahoo! [164], is probably the research effort closest to our Quasit proposal. As Quasit, S4 lets users freely define stream analysis graphs made of processing nodes called PEs in S4 terminology (see Section 3.4); in addition, inspired by MapReduce, S4 permits to partition streams according to user defined *keys* (task parallelism). The platform instantiates PEs based on the graph layout and on the keys dynamically found in the data and guarantees that, within a stream, samples with the same key are always processed by the same 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 an 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 have modeled the same pipelined OpenCV image processing scenario implemented in Quasit, and we have executed it on our testbed. Unfortunately, we were not able to control the S4 PE placement algorithm, so we used the default algorithm, which assigns PE instances to servers according to a hash function applied to stream *keys*. Note that the placement obtained through this algorithm will be, in general, totally unaware of the graph communication characteristics. For this reason, to avoid an unfair comparison, we configured Quasit to use the *random* placement algorithm, which is equivalently unaware of graph characteristic. Once again we feed the pipeline application deployed on S4 with 500 samples per experi-

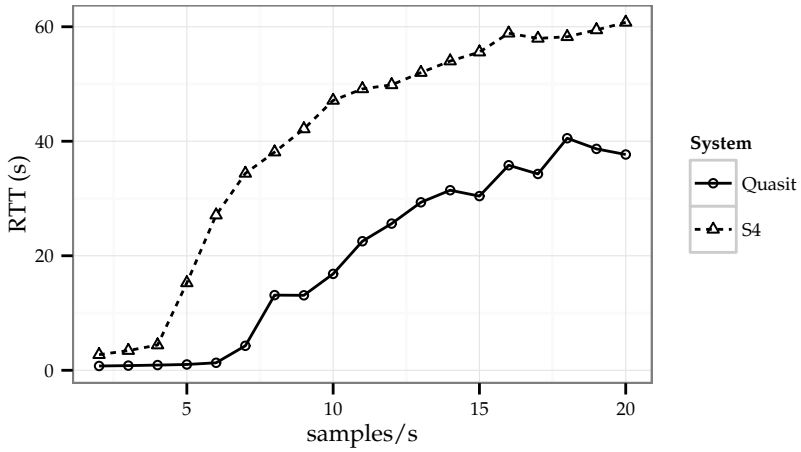


Figure 4.17: Quasit vs. S4: average sample processing time for increasing data rates with 4 processing servers and random placement (Quasit) or hash-based placement (S4).

ment run, and we 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 4.17, we show the results for the cluster configuration with four QRNs/S4 servers, and, in Figure 4.18, 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. We believe that this is the consequence of 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). 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 Common Data Representation (CDR) standard [245], while S4 uses a custom solution based on the Kryo serialization framework [246]), but also, and most importantly, in scenarios presenting several *one-to-many* communication patterns: in these situ-

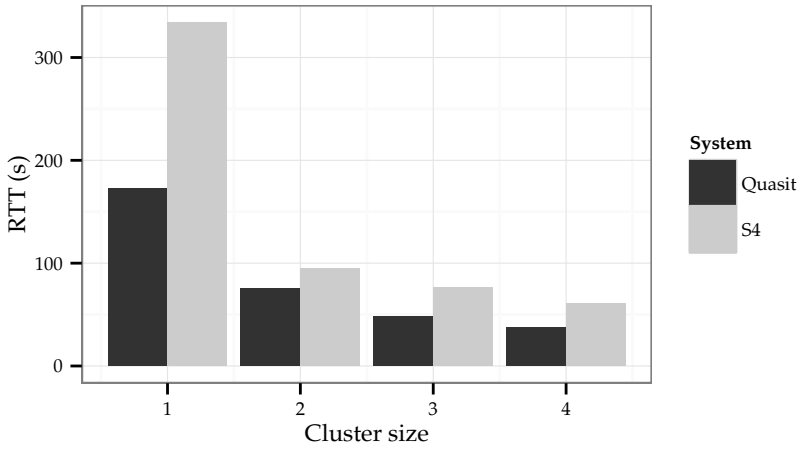


Figure 4.18: Quasit vs. S4: average sample processing time with 1, 2, 3, or 4 processing servers and random placement (Quasit) or hash-based placement (S4). The input sample rate is 20 samples/s.

ations, the implementation of operator channels over IP multicast can significantly reduce network overhead if compared to the TCP based solution used by S4 (internally based on the JBoss Netty framework [236]). We are planning an extended comparative analysis focusing on network utilization 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, to perform smarter operator placement, or to modify its thread scheduling mechanisms dynamically (e.g., enlarging the thread pool size if operators perform many I/O operations).

4.7 LESSONS LEARNED

The design of the Quasit architecture and the realization of its prototype required an important engineering effort in order to satisfy all the requirements deriving, either explicitly or implicitly, from the design principles discussed in Section 4.2. In this section, we summarize our experience by reporting in three points the most important lessons learned during our work on Quasit.

- *Clearly separate state and behavior to foster reusability*. In Quasit, operators are implemented as purely functional and reactive components whose behavior is defined by a processing function that is stateless even when the operator as a whole has stateful be-

havior. We applied this decoupling principle whenever possible in our prototype: not only purely functional behavior is easier to understand and debug, but, most importantly, it can be easily composed. Complex functionalities can be defined by "mixing and matching" smaller pieces of well-tested code expressed in terms of pure functions. Moreover, this type of design encourages a clear and compact definition of the state of different components, which permits to have a better understanding of their possible state transitions and to implement advanced state management functionalities more easily.

- *Adopt a reactive threading model*. As confirmed by our experimental evaluation, the choice of multiplexing the execution of operators on a pool of threads with limited size is one of the main advantages of the Quasit architecture with respect to other similar systems. In fact, with this approach, the number of context switches is minimized and the parallelism of multi-core processors is exploited at the best. Our conclusion is that this execution schema should be always preferred when the number of active components is sensibly greater than the number of available execution cores.
- *Use immutable data structures*. Our Quasit prototype implementation confines state mutations in a few selected places, and uses immutable data structures whenever possible. In highly concurrent environments, immutable data structures provide easy and fast thread-safety being lock-free by definition. Additionally, limiting the places where mutations can occur also limits the number of possible race conditions and helps to write correct thread synchronization mechanisms and to detect concurrency problems early.

4.8 FUTURE WORK

The Quasit prototype is under development and its current implementation is still not ready for everyday use in production environments. However, our initial experimental results encourage us to persist in our efforts to build a stable and usable platform within a short time frame. Our work will go in two main directions:

- Extend the Quasit processing model, and complete and enrich its implementation platform.
- Use the platform as a ground layer for the implementation, validation, and comparison of new stream processing QoS policies and enforcement algorithms.

For what concerns the extension of the platform, one of our short-term goal is to adapt and include common stream processing QoS

policies from state-of-the-art systems (see Section 3.4) in the Quasit processing model, and to provide the possibility to easily configure them through our API. Once we reach stable prototype and APIs, we plan to continue our experimental evaluation on an extensive set of large scale real world applications, by trying to single out possible performance bottlenecks in very large deployment environments.

However, the most ambitious goal of the Quasit platform remains to provide middleware researchers and developers with a common infrastructure and playground for the development and experimentation of novel models and implementation techniques for QoS-based behavior in DSMSs. We will work toward this goal by trying to improve our modular system architecture further, in order to make the development of new modules and their integration in our platform easy and intuitive. As a first step in this direction, we plan to freeze and standardize the interfaces and semantics of our current architecture and make the results publicly available.

4.9 SUMMARY AND CONCLUSIONS

In this chapter, we have introduced a novel model for QoS-centric data streams processing and presented our experiences in the design and implementation of Quasit, the platform providing a distributed, scalable, and highly configurable execution environment for data stream processing applications written according to the proposed model.

Similarly to many other existing DSPSs, the Quasit processing model is based on the processing graph abstraction introduced in Chapter 3: Quasit processing graphs are called SIGs, and they model the composition of simpler data processing steps, represented by the reusable *operator* component. Uniquely, every element of Quasit SIGs can be decorated with QoS specifications, i.e., collections of policies regulating the expected runtime behavior of the component. QoS specifications can be used by the platform to optimize its resource allocation policies and to provide applications with the required quality-level. The Quasit architecture is highly modular and is based on the principle of strong separation of responsibilities between its different modules. Its prototype implementation exploits several state-of-the-art technologies in order to implement a simple, efficient, and low-overhead platform. We have reported an initial set of experiments that evaluate our system in a controlled but realistic video processing scenario and that compare its performance with the Apache S4 stream processing framework. The results show that Quasit can offer horizontally scalable performance with limited management overhead, also if compared to a widely used state-of-the-art DSPE such as S4.

We are continuing our work on the Quasit prototype, enriching it with new QoS policies and with mechanisms for their enforcement.

We are also planning a more extensive evaluation campaign to verify the performance of our system in much larger deployment scenarios, also testing its ability to serve the processing needs of real-world workloads. In the long run, we hope that Quasit could become an useful research platform for the experimentation of new mechanisms for the enforcement of QoS in DSPSs, and be stable and efficient enough to support the execution of real world and large scale workloads.

5 ADAPTIVE FAULT-TOLERANCE IN DISTRIBUTED STREAM PROCESSING SYSTEMS

In this thesis, we have encountered more than one example of the growing number of applications that require continuous processing of high-throughput data streams, and we have seen that these applications often require specific quality-of-service levels to achieve their goals; yet, due to the high time-variability of stream characteristics, it is often inefficient to statically allocate the resources needed to guarantee application SLAs.

Experience with Cloud services [247] has shown that the possibility to offload the management of computing infrastructures to third parties represents an attractive opportunity for both developers and cloud providers. However, in a cloud environment, the nature of stream processing applications poses several hard challenges, including the ability to offer, at the same time, performance *elasticity* in spite of load variations and *resiliency* to failures while keeping *costs* limited. From a provider perspective, one major problem lies in the necessity to handle load fluctuations due to sudden and possibly temporary variations in the rates of data streams feeding the hosted applications. If not handled properly, in fact, load peaks can lead to increased processing latency due to data queuing and to data loss due to queue overflows. To avoid these effects, it is necessary to allocate the proper amount of additional resources for the overloaded applications, either statically or dynamically when load variations are detected [182, 185, 190].

Another typical requirement for stream processing applications is the implementation of *fault-tolerance* techniques. In fact, since they usually run for (indefinitely) long time intervals, failures are unavoidable. Many proposals in the literature have investigated possible fault-tolerance approaches — including active replication [197, 198], checkpointing [193, 248], replay logs [195, 196], or hybrid solutions [249] — each providing different trade-offs between runtime cost in absence of failures (*best-case*) and *recovery* cost. Whichever the adopted technique, maintaining some form of replication at some level (software/hardware components, state, or messages) is a significant overhead in terms of computing resources.

In a large class of applications, however, perfect fault tolerance is not always required, while it is very important to effectively manage temporary load variations. This is very common, for example,

when dealing with SPE-generated big data streams. In this context, in fact, large data streams are produced by many distributed sources — e.g., mobile phones, ad-hoc sensing devices, or vehicles — that continuously capture and transmit sensed environmental features. These data need to be analyzed in real-time, and results must be promptly delivered to let appropriate control actions be performed. In this kind of scenarios, controlled information loss is usually tolerable, given the common partial information redundancy or overlap of input streams. Consider, for instance, an application used to control traffic light signals based on periodic reports of vehicles' positions, among other factors. During high traffic conditions (i.e., high system load), it is clearly preferable to compute on incomplete information than delay control decisions, given the high redundancy in reported positions. At the same time, during low traffic conditions, processing events with accuracy is still important.

In this chapter, we investigate the possibility to trade-off reliability guarantees and execution cost, and use the conserved resources to handle load variations. We propose a novel method, called Load-Adaptive Active Replication (LAAR), that dynamically deactivates and activates redundant replicas of application operators in order to claim or release resources and accommodate temporary load variations. Our technique provides a-priori guarantees about the achievable fault-tolerance levels, expressed in terms of an *internal completeness* metric that captures the maximum amount of information that can be lost in case of failures. This is possible thanks to an off-line optimization phase that determines the most appropriate runtime *replica activation strategy*. In the remainder of this chapter we propose a detailed study of the problem and present our solution approach. We show that LAAR can be suitably implemented as a middleware-level layer on top of existing stream processing platforms, and we present general architectural and design guidelines about how to do it efficiently. As a working proof-of-concept, we describe an implementation of LAAR on top of IBM InfoSphere Streams [166] and we discuss experimental results about the performance of LAAR on a 60-core IBM BladeCenter cluster deployment.

The rest of the chapter is organized as follows: after reviewing the related literature in Section 5.1, we present the considered stream processing service model in Section 5.2. In Section 5.3, we model the optimization problem at the core of our technique, and we describe the off-line and Constraint Programming (CP) -based solution strategies the we use to solve it in Section 5.4. The runtime architecture enforcing our LAAR technique is presented in Section 5.5, followed by an extensive experimental evaluation in Section 5.6. Some final remarks and ideas for future extension of this work conclude the chapter.

5.1 RELATED WORK

The analysis in Chapter 3 has already highlighted that an effective management of load variations is very important in DSPSs, a fact that is also demonstrated by the existence of relevant solution strategies presented in the previous literature. In fact, unless DSPS deployments are over-provisioned with resources (an usually undesired solution because highly cost ineffective), even short variations in sources input rate can cause increased processing *latency* due to operator queues getting longer, or random tuples drops when queues fill up.

A common and very simple solution is to allocate enough resources to sustain the load for most of the time and then to avoid (or limit) the growth in latency or random data drops by introducing *load shedding* mechanisms [155], which selectively drop tuples at strategic points in the processing flow to maximize some quality measure [187] or to minimize the amount of data lost [250].

More sophisticated solutions try to dynamically adapt to load variations while avoiding to drop any data. A first common approach is to move operators between processing hosts to re-balance the system and accommodate new load conditions [172, 182, 251]. In [185], the authors develop a dynamic resource allocation algorithm that automatically re-distributes resources among operators to maximize the expected throughput. More recently, a similar approach has been proposed by [190]. All these solutions effectively manage to handle load variations when the available resources are still sufficient to handle the total load or, from another perspective, when there is no hard limit on the runtime cost of the solution.

In [252], the authors propose a dynamic priorities mechanism for the Stream MapReduce system [253] that automatically reduces the execution priority of tasks replicas during load spikes. Similarly to our proposal, this mechanisms permits to gather the resources necessary to handle the extra load; differently from LAAR, the proposed solution does not provide hard guarantees about the possible information loss in worst-case failure scenarios and does not adapt the applications runtime cost to their required fault-tolerance guarantees.

In our work, we deal with this problem from a different and original perspective. Instead of handling load variation by sacrificing latency (queuing), completeness (load shedding), or increasing cost (resource over-provisioning), we collect the resources needed to cope with changing load conditions by leveraging the flexibility of weaker reliability requirements. LAAR guarantees that these requirement are enforced and minimizes the application execution cost accordingly.

The use of constraint programming to manage replicas in distributed systems has been previously explored by Michel et al. [254], who propose a CP model that solves the problem of deploying replicas on distributed nodes to minimize the communication cost in EVENTUALLY SE-

RIALIZABLE DATA SERVICE (ESDS) systems. Our optimization problem is sensibly different, as we do not deal with the assignment of replicas to computing resources, but we decide their dynamic activation strategy. In [255], the authors solve a combined assignment and scheduling problem for CONDITIONAL TASK GRAPHS (CTGs). Similarly to this work, the CP model includes stochastic elements, but they are used to describe the probability that branches in the task graphs are actually used at runtime.

5.2 SERVICE MODEL

In this section, we introduce a PAAS-based [247] service model, for the commercial-relevant scenario where *service providers* host customers stream processing applications according to a set of SLA parameters that define the expected runtime behavior of hosted applications and the associated costs and pricing plans. By presenting this model, we set up the basic terminology that we use throughout the remainder of this chapter, and we state the fundamental assumptions of our LAAR dynamic fault-tolerance approach.

In our model, stream processing services are regulated by customer–provider *contracts* composed of (i) the *stream processing application* to be executed on the platform, (ii) an *application descriptor* that characterizes the application components and the application input (e.g., its statistical properties, see the following), (iii) an SLA determining the targeted runtime quality requirements, and (iv) a *pricing plan* that defines the economical conditions under which the provider runs the customer application with the requested quality of service.

The *stream processing application* (or, hereinafter, simply application) is described through the processing graph abstraction presented in Section 3.3.1, i.e., consists of a set of software components organized in a DAG. The software components are one or more *operators*, at least one *data source*, and at least one *data sink*. An operator transforms one or more input data streams — theoretically unbounded sequences of structured tuples — into another stream (its output); data sources and data sinks retrieve input from external sources and write tuples to external destinations, respectively. The processing graph arranges operators, data sources, and data sinks as vertices of a DAG, connected by edges representing communication channels. Recall that this data-flow based processing model is very general and can be mapped on the majority of state-of-the-art DSPSs from academia [27, 156, 158, 167] and industries [25, 164, 166, 256].

The *application descriptor* is a document summarizing, with a set of concise attributes, the computational behavior of operators and the expected characteristics of application input streams. Similarly to what has been done in the literature (e.g., [171, 174, 187, 195, 250,

251]), application descriptors summarize operator behavior by using the metrics of *port selectivity* and *per-tuple* CPU *cost*. To be more specific, we associate every graph edge going into an operator to a selectivity value and a per-tuple CPU cost value: selectivity represents the weight of the contribution of an input data stream on the data rate of the operator output stream; per-tuple CPU cost is the number of CPU cycles (on a given processing architecture) required on average to process a tuple from the stream associated to the edge. For simplicity of mathematical derivation, we adopt a linear load model, i.e., we assume that the output rate and the total CPU load of any operator can be expressed as a linear combination of the streams data rates and the operator selectivities and per-tuple CPU costs respectively. With considerations similar to those in [174], our solution can be extended to nonlinear models as well. In the following discussion, we assume that operator selectivities and per-tuple CPU costs are either provided by the customer or extracted through a preliminary *profiling* step [209]. The application descriptor also includes the expected characteristics of the external data sources: for each data source, the descriptor contains the probability distribution function describing the probability of the source to produce data at different tuple rates. We assume that the continuous space of possible tuple rates for each data source has been properly transformed in advance into a finite number of discrete data rates through, e.g., binning techniques [257]. Again, this information is specified by the customer or else inferred from a set of example input traces that she provides.

A Service Level Agreement (SLA) is a set of clauses specifying the desired runtime quality characteristics of the application. Two possible examples of SLA clauses are *maximum latency*, putting an upper bound on the time taken to produce an output after all the input data generating it has been received, or *fault-tolerance*, defining a guaranteed application behavior in case of failures.

Finally, the *pricing plan* determines the provider monetary revenue for running the customer application instance. We assume *continuous processing* applications, i.e., applications that run for an indefinite amount of time. As a result, we consider a *time-based, fixed* billing plan, according to which the customer pays a flat fare per billing period T. This fare depends on the characteristics of the application, of its input streams, and on the agreed SLA.

The service provider is expected to deploy and allocate computing resources so that the constraints imposed by SLA clauses are satisfied at runtime as long as, within each billing period T, *the characteristics of the external data streams reflect those specified in the contract*; if they do, the provider has to pay a penalty in case of SLA violations. The provider is interested in satisfying the quality requirements imposed by the SLA, while minimizing resource utilization. We assume that the service provider does always her best to avoid SLA violations.

5.3 LOAD-ADAPTIVE ACTIVE REPLICATION

LAAR is a novel and adaptive active replication method for data streams processing platforms that lets applications adapt to changing load conditions by temporarily trading perfect reliability for computational resources. It can provide *guaranteed* fault-tolerance levels, measured in terms of an upper bound on the information loss in case of failures, called *internal completeness* and defined in Section 5.3.3.

Similarly to traditional active replication techniques [195], LAAR deploys k replicas of every operator in the application processing graph: at any moment, one of the k replicas has the role of *primary*, the others are called *secondary*. Primary and secondary replicas all receive tuples from the primaries of their predecessor operators, and all process them advancing through the same sequence of states. However, only the primary outputs tuples to the replicas of its successors. When a primary fails, one of the secondaries is elected as the new primary; once the failed replica is recovered, its state is synchronized with the non-failed ones before it becomes active again as secondary.

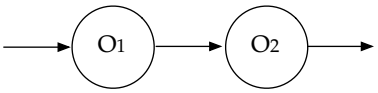
Originally, LAAR monitors the input rate of its application sources, and it dynamically and automatically activates and deactivates replicas in order to satisfy two goals:

1. The application deployment is never *overloaded*.
2. A required *internal completeness* constraint is satisfied.

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 operators assigned to it is bigger than the available CPU cycles per second. Note that, in an overloaded system, tuples accumulate at input queues of operators (increasing latency) and are eventually dropped when the corresponding queues fill.

5.3.1 LAAR in a simple application

Before presenting an in-depth analysis of the LAAR model and its fault-tolerance guarantees, we illustrate the basic intuition behind our approach in a minimal application scenario. Consider the application in Figure 5.1: it consists of two operators connected in a very simple pipeline; O_1 processes data from a single data source (not reported in the figure for the sake of simplicity) and forwards its output to O_2, 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 tuple they produce one output tuple; moreover, considering the CPU architecture of the deployment hosts, both operators require 100 milliseconds to process an incoming tuple. The single data source can produce tuples at two different rates: "Low" and "High". The "Low" rate is 4 tuples per



	O_1	O_2
SELECTIVITY	1	1
CPU COST	0.1 s/tup	0.1 s/tup

	LOW RATE	HIGH RATE
SOURCE	4 tup/s	8 tup/s
O_1	4 tup/s	8 tup/s
PROB.	0.8	0.2

Figure 5.1: A simple processing scenario: application processing graph (left), concise characteristics of the application operators (top-right) and of its data source (bottom-right). For simplicity, data source and data sink are not shown.

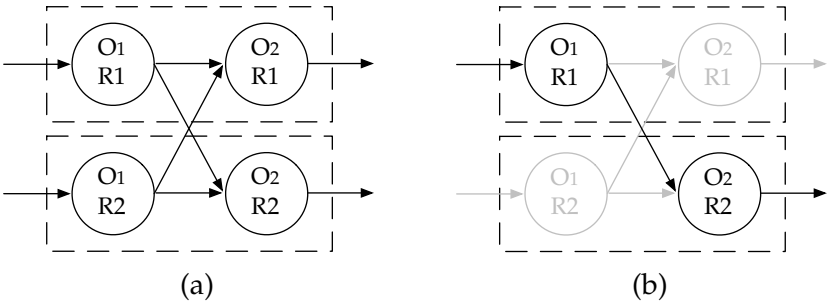


Figure 5.2: (a) Replicated deployment of the application of Figure 5.1 on two hosts. (b) Dynamic deactivation of replicas by LAAR during a "High" input configuration.

second and is active on average for 80% of the time (0.8 probability), while the "High" rate is 8 tuples per second and is active in the remaining time intervals (0.2 probability). The application is replicated and deployed on two hosts, each hosting a copy of each operator, as shown in Figure 5.2a. It is straightforward to see that, when the input configuration is "Low", 80% of the CPU time available at both hosts will be occupied for processing tuples. 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 (with an increased cost).

The basic idea behind LAAR is to monitor the data sources and, according to the current data rates, to dynamically deactivate replicas in order to release the resources necessary to face load variations. For example, Figure 5.2b shows how LAAR could deactivate two replicas of O_1 and O_2 during a load peak so that the total CPU available will become enough to handle the new load.

Figure 5.3 shows this behavior in a real deployment. We implemented, deployed, and executed the replicated pipeline application in Figure 5.2a on an IBM InfoSphere Streams deployment consisting of two hosts equipped with a single core CPU. Figure 5.3a reports the

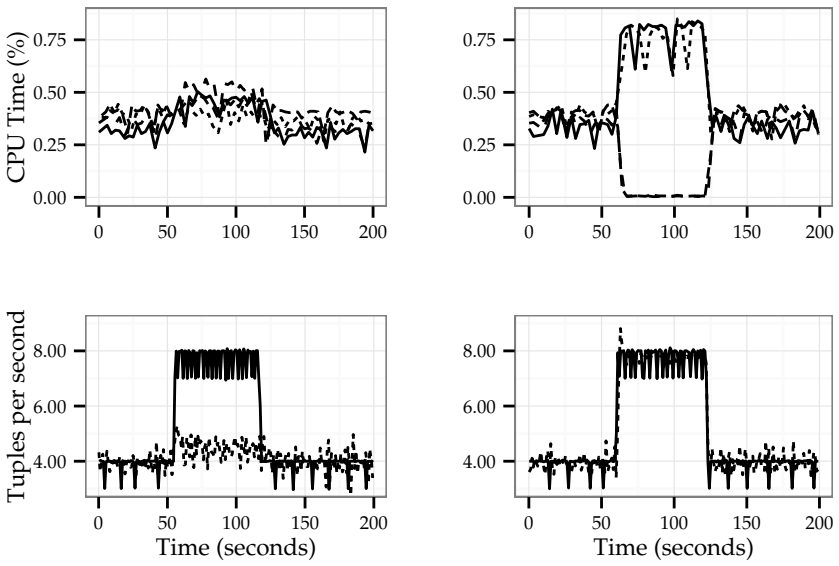


Figure 5.3: (a) CPU Time used by the replicated operators — top —and corresponding input and output rate — bottom. (b) CPU time and input/output data rate when O_1 replica 2 and O_2 replica 1 are deactivated by LAAR. In the top graphs, different line styles correspond to different operator replicas; in the bottom graphs, the solid line corresponds to the input rate, the dashed one to the output rate.

CPU usage and input/output rates of the application in time when static active replication is used: when the input passes to the "High" configuration (around 50 seconds from the beginning of the experiment), the CPUs of the two hosts saturate, and the application is not able to keep up with the input rate; on the contrary, 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 (Figure 5.3b).

Obviously, if a failure of one of the active operators occurs during a "High" period, part of the input would not be processed as expected. As we will clarify in the remainder of this section, the unique and strong aspect of LAAR is its ability to quantify *a-priori* these effects on the overall application reliability.

5.3.2 Model and definitions

An application A consists of a set of components: a set I of data sources, a set P of operators, and a set O of data sinks, which collectively define the set $X = I \cup P \cup O = \{x_i\}$. The components in X are

arranged in a directed acyclic application graph $G = (X, E)$. The set of edges E is described by the function:

$$pred : X \mapsto \mathcal{P}(X) \tag{5.1}$$

which, for each component x_i, identifies the set of predecessor components $\{x_j\}$ so that $x_j \in pred(x_i) \Leftrightarrow (x_j, x_i) \in E$.

The characteristics of operators are summarized by the selectivity function δ and the per-tuple CPU cost function γ: for each couple (x_i, x_j) so that $x_i \in I \cup P$ and $x_j \in P$ and that $(x_i, x_j) \in E$, $\delta(x_i, x_j)$ is the selectivity of operator x_j with respect to the tuples it receives from x_i, and $\gamma(x_i, x_j)$ is the per-tuple CPU cost for operator x_j to process tuples from x_i.

Every data source $x_i \in I$ can produce output at one rate among a finite set of input rates R_i. The Cartesian product $C = R_1 \times \ldots \times R_t$, where t is the number of sources, is the set of all the possible *input configurations*. As anticipated in Section 5.2, we assume to know $P_C : C \mapsto [0, 1]$, the probability mass function associated to the probability distribution of different input configurations in time. The output rate of data source $x_i \in I$ in a particular input configuration c is indicated as $\Delta(x_i, c)$. In absence of failures, it is straightforward to derive the expected output rate of each operator in any input configuration c; for uniformity of notation, we also indicate this value as $\Delta(x_i, c), x_i \in P$.

We assume that an operator placement algorithm among the many described in the literature (e.g., in [171] or [174]), computes a *replicated* assignment of k replicas of each of the operators in P to a set of hosts $H = \{h_i\}$. We indicate the replicated set of operators as:

$$\widetilde{P} = \{\tilde{x}_i^l\} \tag{5.2}$$

For simplicity of notation, we will use the symbol $\tilde{x}_i^l$ to indicate the l-th replica of operator x_i. The assignment is represented by the function:

$$\vartheta : \widetilde{P} \mapsto H \tag{5.3}$$

which maps every operator replica to the host where it is deployed. For convenience, we also define $\vartheta^{-1} : H \mapsto \mathcal{P}(\widetilde{P})$ such that:

$$\vartheta^{-1}(h) = \{\tilde{x}_i^l \in \widetilde{P} : \vartheta(\tilde{x}_i^l) = h\} \tag{5.4}$$

A *replica activation strategy* is a function:

$$s : \widetilde{P} \times C \mapsto \{0, 1\} \tag{5.5}$$

that associates every operator replica – input configuration pair to one of the two possible active/inactive states.

5.3.3 Internal completeness metric

By activating and deactivating operator replicas according to the current input configuration, LAAR dynamically modifies the resilience of applications to failures. In order to measure the effect of LAAR on fault-tolerance guarantees, we define the INTERNAL COMPLETENESS (IC) metric. Intuitively, given a failure model that describes how hosts and operators are expected to fail and a *replica activation strategy* s, internal completeness measures the fraction of total tuples that is expected to be processed during the billing period T in case of failures, compared to those that would be processed in absence of failures.

Let us examine the no-failure scenario (best-case) first: the total number of tuples that is statistically expected to be processed by the application operators during T is:

$$\text{BIC} = T \cdot \sum_{\substack{c \in C, \\ x_i \in P, \\ x_j \in pred(x_i)}} P_C(c) \cdot \Delta(x_j, c) \tag{5.6}$$

BEST-CASE INTERNAL COMPLETENESS (BIC) is the summation of the contributions of all the application operators in all the possible input configurations, weighted by their probability to occur in T.

FAILURE INTERNAL COMPLETENESS (FIC) measures the expected number of tuples processed given a failure model ϕ and a replica activation strategy s. It is defined as:

$$\text{FIC}(s) = T \cdot \sum_{\substack{c \in C, \\ x_i \in P, \\ x_j \in pred(x_i)}} P_C(c) \cdot \phi(x_i, c, s) \cdot \widehat{\Delta}(x_j, c, s) \tag{5.7}$$

$$\widehat{\Delta}(x_i, c, s) = \begin{cases} \Delta(x_i, c) & \text{if } x_i \in I \\ \phi(x_i, c, s) \cdot \sum_{x_j \in pred(x_i)} \delta(x_j, x_i) \widehat{\Delta}(x_j, c, s) & \text{if } x_i \in P \end{cases} \tag{5.8}$$

The function $\phi(x_i, c, s)$ depends on the chosen failure model and describes the probability that at least one replica of operator x_i is alive and active when the input configuration is c and the replica activation strategy is s. $\widehat{\Delta}(x_i, c, s)$, instead, represents the expected output of PE x_i under failure model ϕ, when the input configuration is c and the replica activation strategy is s; note that the definition of $\widehat{\Delta}$ is recursive, as the number of tuples produced by a PE depends not only on its possible failure status (described by ϕ) but also on the number of tuples produced by its predecessor (Equation 5.8).

INTERNAL COMPLETENESS (IC) is the ratio between FIC and BIC:

$$\text{IC}(s) = \frac{\text{FIC}(s)}{\text{BIC}} \tag{5.9}$$

We choose the IC metric over other possible metrics (e.g., output completeness or average replication factor) for two main reasons. First, IC is easy to understand and measure. Second, and most relevant, IC captures not only the completeness of the application output (at the data sinks) but also the divergence, in a scenario with failures, of the state of operators compared to a failure-free scenario, under the assumption that this divergence is proportional to the amount of tuples that are not processed.

5.4 REPLICA ACTIVATION PROBLEM

In LAAR, the information in the application descriptor is used to compute — off-line and before application deployment — a replica activation strategy that fits the application fault tolerance requirements.

The cost minimization problem that is solved to determine the appropriate replica application strategy, given an application descriptor, is called *replica activation problem* and is defined as follows:

$$\underset{s}{\text{minimize}} \quad cost(s) \tag{5.10}$$

subject to:

$$IC(s) \geqslant G \tag{5.11}$$

$$\sum_{\substack{\tilde{x}_i^l \in \vartheta^{-1}(h), \\ x_j \in pred(x_i)}} \gamma\left(x_j, x_i\right) \Delta(x_j, c) s(\tilde{x}_i^l, c) < K \qquad \begin{matrix}\forall h \in H, \\ \forall c \in C\end{matrix} \tag{5.12}$$

$$\sum_{l=1}^{k} s\left(\tilde{x}_i^l, c\right) \geqslant 1 \qquad \begin{matrix}\forall x_i \in P, \\ \forall c \in C\end{matrix} \tag{5.13}$$

The *cost* function in the minimization term represents the cost, in terms of resources, for a service provider to run the application using replica activation strategy s and the replicated assignment defined by ϑ. In this work, we model the bandwidth available for cluster-local communication as an abundant resource (a common assumption in data center contexts), and our cost function as the total CPU time used by an application in a billing period T. It is defined as follows:

$$cost(s) = T \sum_{\substack{c \in C, \\ \tilde{x}_i^l \in \tilde{P}, \\ x_j \in pred(x_i)}} P_C(c)\, \gamma\left(x_j, x_i\right) \Delta(x_j, c) s(\tilde{x}_i^l, c) \tag{5.14}$$

and is the summation over all operator replicas $\tilde{x}_{i,h}$ of their consumed CPU time.

Equation 5.11 constraints IC to satisfy the requested SLA value G, while Equation 5.12 states that each host in the deployment should never be overloaded; K is a constant expressing the number of CPU cycles per second available at the deployment hosts. The last constraint,

expressed in Equation 5.13, requires that there is at least one active replica of every operator in every input configuration, and it ensures that the measured IC value is 1 in absence of failures.

5.4.1 Failure model

In order to solve the replica activation problem, LAAR considers a simplified failure model ϕ, based on the following assumptions:

1. In any failure scenario, all PE replicas fail except one.
2. Unless all the replicas are active at some point in time, the non-failed replica is chosen among the inactive ones.
3. Once failed, replicas never recover.

or, more formally:

$$\phi(x_i, c, s) = \begin{cases} 0 & \text{if } \sum_{l=1}^{k} s(\tilde{x}_i^l, c) < k,\ \tilde{x}_i^l \in \widetilde{P} \\ 1 & \text{otherwise} \end{cases} \tag{5.15}$$

The so defined model will in general overestimate possible failure conditions (it is highly unlikely that all PE replicas fail at the same time) and their consequences (normally failures would be recovered): for these reasons, we also refer to it as *pessimistic failure model*. However, this choice of ϕ provides two fundamental benefits:

- Since it overestimates the likelihood and effects of failures, the IC value computed using this model is a lower bound to the real IC that will be observed on the actual application deployment (see Section 5.6).
- Its mathematical formulation simplifies the computation of IC values for different possible replica activation strategies and, hence, the optimization complexity.

Note that the solution space of this problem is still very large, as for every application there are $2^{|P|\cdot|C|\cdot k}$ possible replica application strategies. Note also that, in cost function (Equation 5.14), the IC constraint (Equation 5.11), and the hosts CPU constraints (Equation 5.12) depend on $\widehat{\Delta}(x_i, c, s)$ (Equation 5.8), which is a recursively defined exponential term. Hence, to find algorithms that can find optimal or good enough solutions to this problem is a major technical challenge. In the next section, we present a detailed study of the properties of the replica activation problem, and we present and compare three original CP algorithms for its solution.

Let us remark again that the optimization phase is performed off-line with respect to application execution, so its complexity does not cause any direct overhead on the application runtime cost, which is,

instead, minimal. In Section 5.5, we describe how LAAR on-line counterpart can be implemented as a thin middleware layer requiring little modifications to existing data streams processing architectures already supporting (static) active replication.

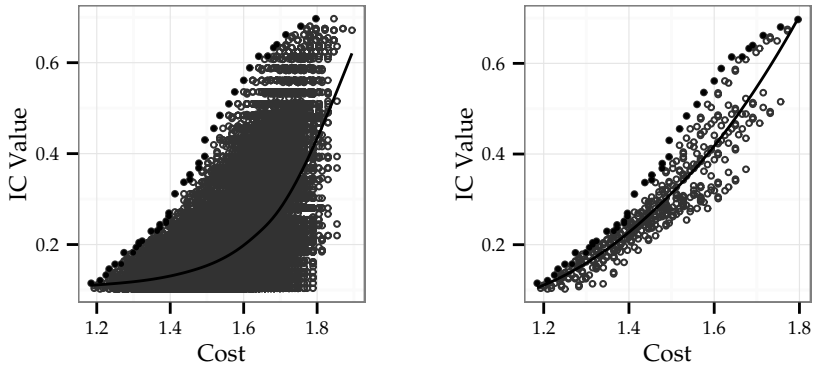
5.4.2 Constraint programming solutions

We have developed three different CP-based algorithms to solve instances of the LAAR replica activation problem. For implementation simplicity, all the three solution variants limit the search space by considering only two-fold replication ($k = 2$), practically restricting the problem solution space size to $3^{|P|\cdot|C|}$. We believe that this is not a strong restriction, since, in our experience, stream processing applications are very rarely deployed using a replication factor greater than 2. The three algorithms we have studied are:

- A trivial CP solver, called *basic*.
- A search algorithm based on Large Neighborhood Search (LNS) techniques [258].
- A scalable decomposition-based algorithm.

Before presenting them in more detail, let us first describe the characteristics of the solution space of our replica activation problem that guided our algorithm design. To perform this analysis, we developed a straightforward implementation of the model of Equation 5.10 on the commercial IBM ILOG CP Optimizer solver [259], and we used it to get a better understanding of the problem structure.

Looking at the problem from a user perspective, cost (Equation 5.14) and IC (Equation 5.9) are the most important parameters because, together, they determine the cost-quality trade-off for running stream processing applications with LAAR. Intuitively, since the basic mechanism to mask the effects of failures is to activate more replicas, requiring higher IC will correspond in general to higher runtime costs. Figure 5.4a gives some insight about the shape of the problem solution space when considering together cost and IC: it shows the space of possible feasible solutions of a problem instance consisting of 24 operator replicas distributed on 6 computing hosts and IC constraint set to 10%. The continuous black line is a *loess* regression [260] of the solution points and confirms that, as a general trend, the cost of solutions is proportional to their IC value. However, the graph also shows that there is a very large number of sub-optimal solutions (empty circles) and that higher costs do not necessarily imply higher IC. Recall that the IC value does not only depend on the number of active replicas, but also on the particular choice of active operators and on the topology of the application processing graph. As a consequence, a wrong choice of replica activation strategy can easily lead to a useless waste of resources.



(a) (b)

Figure 5.4: Cost–IC relationship in the solution space of a problem instance consisting of 12 operators (2 replicas each) deployed on 6 hosts. Empty circle represent sub-optimal solutions, while the continuous line is a regression of the solution points. Without (a) and with partial filtering of sub-optimal solutions (b).

However, an important fraction of sub-optimal solutions can be discarded quickly with simple considerations. For example, think about of a pipeline of operators where a first operator (O_1) feeds a second one (O_2). Given the pessimistic failure model described by Equation 5.15, having, in any input configuration, two active replicas of O_2 and, at the same time, only one active replica of O_1 does not contribute to the overall IC value because, in case of failures of the O_1 replica, O_2 would not receive any sample to process; however the solution cost would still be higher than that of a non-replicated deployment. This is not only valid for pipelines but can be generalized for any graph shape: in particular, any feasible replica activation strategy s_α that, in some input configuration c, has two active replicas for some operator x_i whose predecessors all have only one active replica is sub-optimal with respect to a corresponding feasible replica activation strategy s_β that differs from s_α only for the fact that x_i has just one active replica. This relation can be used to add a new constraint that performs a *partial sub-optimal solution filtering* and removes obviously sub-optimal solutions to the replica activation problem. We formulate this constraint as follows:

$$\begin{aligned} \exists\, x_i \in P, c \in C \text{ s.t. } \sum_{l=0,1} s\left(\tilde{x}_j^l, c\right) = 1 \qquad \forall x_j \in pred\,(x) \\ \implies \sum_{l=0,1} s\left(\tilde{x}_i^l, c\right) = 1 \end{aligned} \tag{5.16}$$

Figure 5.4b shows the solution space of the same problem instance of Figure 5.4a after the filtering based on Equation 5.16. Obviously, this important reduction in size has a significant impact on the time needed to solve the problem. Figure 5.5 summarizes the average search

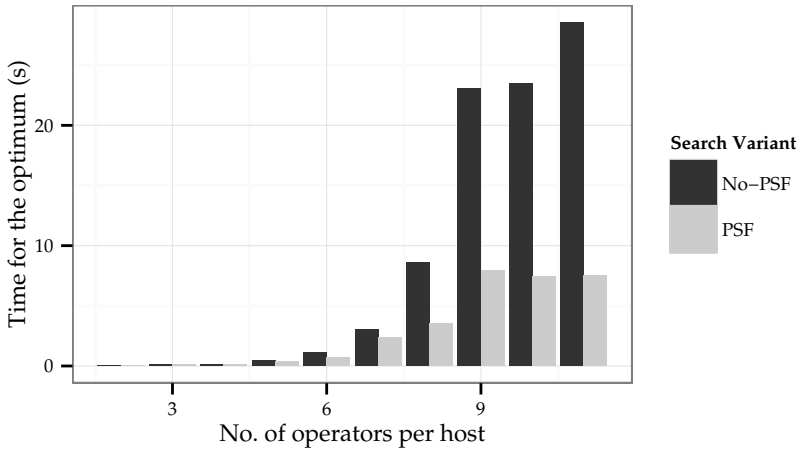


Figure 5.5: Comparison of average time to find the optimum with (PSF) and without (No-PSF) partial sub-optimal solution filtering.

time needed to find the optimum solutions for a batch of small problem instances, in which graphs of 2 to 11 operators are deployed on 4 hosts with two replicas per operator. As the graph complexity increases, the benefit of the additional constraint in Equation 5.16 becomes more and more evident.

The first *basic* solution strategy is nothing but the trivial implementation of the model in Equation 5.10 with the additional constraint in Equation 5.16 on the ILOG CP Optimizer solver. Since its realization is a straightforward transcription of the above model and constraints on the solver, we do not detail it further here. On the contrary, in the following two paragraphs, we introduce the second and third search strategies, i.e., the LNS-*based strategy* and the *decomposition-based* one. In Section 5.6.1, we report an evaluation of the three strategies that analyzes their trade-offs between solution time and quality.

LNS-*based strategy*

The basic idea behind Large Neighborhood Search (LNS) strategies [258] is to start from an initial solution and then proceed through incremental improvement steps that focus on *large neighborhoods* of the current best solution. We developed a strategy to solve our replica activation problem that is based on these concepts. The algorithm starts from a solution found either by using the basic solver just presented, or by leveraging a simple *greedy* algorithm that starts by activating two replicas of every operator for every input configuration and then deactivates the most resource hungry operator iteratively, until all the non-overloading condition constraints (Equation 5.12) are met. The advantage of the greedy approach is its ability to find a solution very quickly. This solution will not necessarily be feasible because it could

violate the constraint in Equation 5.11; however, this infeasibility can be often rapidly corrected through local moves.

Similarly, we have developed two alternative methods to choose the variables to relax during every iterative improvement round. In the first (*simple random*), they are chosen completely random; in the second (*weighted random*), every search variable is assigned a weight that depends on its corresponding input configuration, so that variables associated to resource-hungry input configurations (typically corresponding to load peaks) have more chances to be relaxed. The idea is that, since highly demanding configurations usually require the highest number of operator replicas deactivated in order to satisfy the constraint in Equation 5.12, the corresponding search variables have, in general, stronger influence on the satisfiability of the IC requirement.

Note that, differently from the basic search strategy, the LNS-based one does not detect optima, and, when the greedy approach is used to look for an initial solution and none is found, it cannot even conclude whether the problem has any solution or not.

Decomposition-based strategy

The third solution strategy we present in this chapter decomposes the problem in a number of orthogonal sub-problems along its $|C|$ different input configurations. The goal of this decomposition-based approach is to improve solution scalability, especially for problem instances with a large number of input configurations.

Let us consider once again the formulation of the replica activation problem in Equation 5.10. Separating the CPU constraints (Equation 5.12) and the minimum replicas constraints (Equation 5.13) is trivial, because each of them involves only terms relative to a single input configuration c. The search variables s can be equally easily separated by considering $|C|$ different replica activation strategies s_c such that:

$$s(\tilde{x}_i^l, c) = s_c(\tilde{x}_i^l), s_c : \widetilde{P} \mapsto \{0, 1\} \tag{5.17}$$

The IC constraint (Equation 5.11) can be, instead, rewritten as follows:

$$\begin{aligned}
&\frac{FIC(s)}{T} \geqslant \underbrace{\frac{BIC}{T}(G)}_{G'} \\
&\Leftrightarrow \sum_{c \in C} \underbrace{\sum_{\substack{x_i \in P, \\ x_j \in pred(x_i)}} P_C(c) \cdot \phi(x_i, c, s) \cdot \widehat{\Delta}(x_j, c, s)}_{\mu_c(s_c)} \geqslant G' \\
&\Leftrightarrow \sum_{c \in C} \mu_c(s_c) \geqslant G'
\end{aligned} \tag{5.18}$$

Similarly, considering Equation 5.14, the minimization term (Equation 5.10) can be written as:

$$\min \sum_{c \in C} \underbrace{\sum_{\substack{\tilde{x}_i^l \in \tilde{P}, \\ x_j \in pred(x_i)}} P_C(c)\,\gamma\left(x_j, x_i\right)\Delta(x_j, c)s(\tilde{x}_i^l, c)}_{\lambda_c(s_c)}$$

$$\Leftrightarrow \min \sum_{c \in C} \lambda_c(s_c) \tag{5.19}$$

Note that, while the CPU and minimum replicas constraints can be evaluated and satisfied considering each input configuration c separately, the IC constraint and the cost minimization expression cannot; nonetheless, they both can be expressed as a sum of $|C|$ non negative terms, and each of this terms can be evaluated separately for different values of c.

Our decomposition approach consists in defining $|C|$ sub-problems $prob_c$, one per input configuration; the solution of each problem is a partial replica activation strategy s_c that satisfies at least the corresponding CPU and minimum replication constrains (Equations 5.12 and 5.13). The sub-problems' optimization goal and possible additional constraints, instead, depend on the particular phase the decomposition algorithm is in. Algorithm 1 sketches, in pseudo-code, the main steps of the decomposition-based solver.

The algorithm starts by maximizing the $\mu_c(s_c)$ values of each sub-problem (Phase 1, lines 1–9). Note that, after this phase is complete, and if a solution is found for every sub-problem, an upper bound on the possible IC for the original problem can be obtained using Equation 5.18: through it, it is possible to test immediately whether the original problem admits solutions (line 7) and output an initial sub-optimal solution when it does. During Phase 2 (lines 10–22), this initial solution is improved by working separately and iteratively on each sub-problem. At every iteration, the problem providing the minimum contribution to the overall IC, weighted by its contribution to the cost (line 12), is chosen as a candidate for improvement, and the algorithm tries to decrease its cost while ensuring that the obtained $\mu_c(s_c)$ value still allows to satisfies the overall IC requirement (line 13). This iteration is repeated until no improvement can be obtained from any sub-problem. In Phase 3, finally, the partial replica activation strategies are combined and the result returned as output.

Like the LNS-based strategy, this algorithm can decide whether the problem is feasible, but does not recognize optimal solutions. For particularly complex problem instances, it might be necessary to set a time limit for Phase 1 to avoid to block the solver for too long; in such cases, the solution obtained after this initial phase is no longer guaranteed to be an upper bound on the obtainable IC, and so the lack of an initial feasible solution can no longer be used to prove the

ALGORITHM 1: Decomposition-based Search Strategy

input : $\{prob_c\}$: the |C| decomposed subproblems.
output: A replica activation strategy s, or None if no solution found

Phase 1: /* μ_c maximization */
  foreach $prob_c$ **do**
    $s_c^{max} \leftarrow$ maximize μ_c in $prob_c$
    if s_c^{max} *is* None **then return** None $\mu_c^{max} \leftarrow$ maximum μ_c for $prob_c$
    $\lambda_c^{max} \leftarrow$ cost value corresponding to s_c^{max}
  end
  if $\sum_{c \in C} \mu_c^{max} < G'$ **then** /* Feasibility test */
    return None
  end
Phase 2: /* optimization */
  foreach $prob_c$ **do** $\mu_c^{cur} \leftarrow \mu_c^{max}$; $\lambda_c^{cur} \leftarrow \lambda_c^{max}$ **while** *no* $prob_c$ *can be further improved* **do**
    $c' \leftarrow \max_c (\lambda_c^{cur}/\mu_c^{cur})$ /* Choose prob. to improve */
    $\mu_{c'}^{limit} \leftarrow G' - \sum_{\substack{c \in C \\ c \neq c'}} \mu_c^{cur}$
    Post $\mu_{c'} \geqslant \mu_{c'}^{limit}$ as constraint on $prob_{c'}$
    Post $\lambda_{c'} < \lambda_{c'}^{cur}$ as constraint on $prob_{c'}$
    $s_{c'}^{cur} \leftarrow$ `findFirst(`$prob_{c'}$`)`
    if $s_{c'}$ *is* None **then**
      $prob_{c'}$ cannot be improved further
    else
      Update $\mu_{c'}^{cur}$ and $\lambda_{c'}^{cur}$ according to $s_{c'}^{cur}$
    end
  end
Phase 3: /* End */
  s $\leftarrow$ Combine all the s_c^{cur}
  return s

unfeasibility of the entire problem. Let us note, finally, that all the sub-problem optimization steps can be performed with any of the presented optimization techniques. Section 5.6.1 shows that using the LNS-based strategies can significantly improve solution performance.

5.5 RUNTIME ARCHITECTURE

LAAR has been designed to be integrated with little effort with existing platforms that already offer static active replication and that support the model described in Section 5.2. The work flow used to deploy a LAAR-enabled application is schematically shown in Figure 5.6. The application descriptor, the IC SLA requirement, and the application itself (see again Section 5.2) are fed to two different components. The first implements one of the optimization algorithms described in the previous section and produces a replica activation strategy. The second component, i.e., the *Application Preprocessor*, modifies the original application to produce the *extended application*, which enhances the original user application with LAAR functionalities. In particular, as

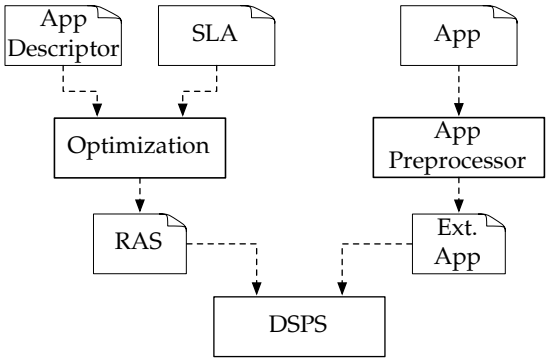


Figure 5.6: Deployment of a LAAR application.

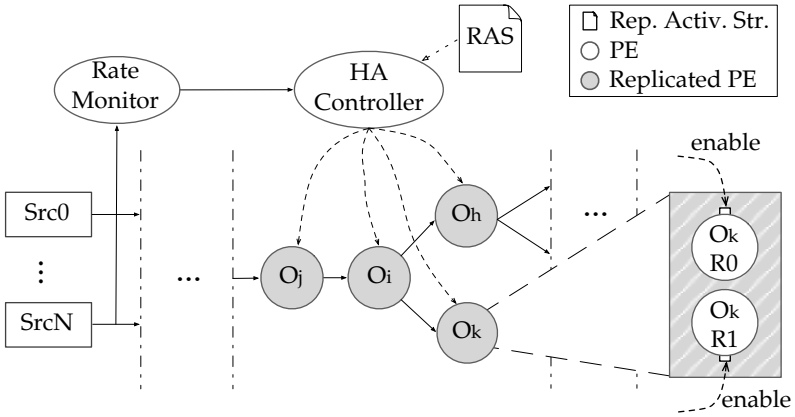


Figure 5.7: Structure of extended processing graphs.

shown in Figure 5.7, two special operators are added to the original processing graph — the *Rate Monitor* and the *HAController* —, and the behavior of application operators is extended in order for them to understand and accept *activation* and *deactivation* commands. The extended application is finally deployed on the actual system.

At runtime, the *Rate Monitor* operator periodically measures the data rates from sources and outputs this measurement result. The High Availability Controller (*HAController*), initialized at startup with the chosen replica activation strategy, receives the sources data rates from the Rate Monitor and, according to their values, it chooses the appropriate replica activation state based on the current input configuration. To achieve that quickly and effectively, it uses an R-Tree [261] -like data structure that selects the input configuration that is spatially closer to the current data rates and whose components are all greater than the corresponding actual rates. This choice guarantees that the chosen replica configuration will never underestimate the actual system load. Whenever a change in the replica configuration occurs, the HAController reliably sends activations or deactivation commands to operator replicas.

Application operators behavior is also slightly modified to make them accept commands from the HAController. When deactivated, they immediately stop processing their input and transit into an *idle*, resource-saving state. On the contrary, when activated again, they re-synchronize their state with one of the active replicas and restart processing their input. Since this process is almost identical to the recovery of crashed operators in traditional static replication systems [195], we will not detail it further.

Let us emphasize again that, since they do not require any particular platform-dependent functionality, both the Rate Meter and HAController can be implemented transparently on top of existing stream processing platforms as standard operators. For what concern the enhancements needed on application operators, they are minimal and can be implemented, for example, by dynamically proxying user provided components. In Section 5.6.2, we describe how we implemented this architecture on IBM InfoSphere Streams.

5.6 EXPERIMENTAL EVALUATION

In this section, we present the results of the experimental evaluation we performed to assess the goodness of our LAAR solution. The section is organized in two parts. In the first, we concentrate on the off-line part of the problem, i.e., the static optimization process that, starting from stochastic knowledge of application characteristics, finds solutions to the associated replica activation problem; we compare the three CP algorithms presented in Section 5.4.2, and highlight their different properties. In the second part, instead, we move our attention to the dynamic part of LAAR, i.e., on the on-line mechanisms that regulate the activation and deactivation of application operator replicas to guarantee the required IC level. We introduce a prototype implementation of these mechanisms on top of IBM InfoSphere streams (developed during the research collaboration with IBM Research in Dublin that resulted in the design of LAAR), and we present an extensive experimental evaluation on a 60-cores cluster that demonstrate how our system is actually able to regulate the trade-off between consistency levels and execution cost, while always satisfying the given SLA constraints.

5.6.1 Off-line optimization

Our evaluation and comparison of the optimization algorithms presented in Section 5.4.2 has two main objectives:

- Compare the speed of the three strategies and the quality of the solutions they can find within fixed time limits.

- Evaluate the scalability of the solutions (in particular of the decomposition-based strategy) as the problem size grows.

For the first part of this evaluation, we consider a batch of 20 different stream processing applications with processing graphs of 96 operators each. Every application has three data sources, each producing output at two possible data rates (for a total of 8 input configurations), and is associated with a replicated deployment (two replicas per operator, 192 replicas in total) on 24 computing hosts. The IC constraint in the related replica activation problem is set to 0.50. We choose these applications as we believe their complexity to be well representative of real world stream processing application deployments.

We compare the optimization algorithms in the following variants:

1. Basic solver with partial sub-optimal-filtering (*BASIC*).
2. LNS-based strategy using BASIC for the initial solution and simple weighted random relaxation method (*L_SRW*).
3. LNS-based strategy using a greedy initial solution and weighted random relaxation method (*L_GRW*).
4. LNS-based strategy using a greedy initial solution and simple random relaxation method (*L_GRS*).
5. Decomposition strategy using BASIC for Phase 1 (*DEC_S*).
6. Decomposition strategy using LNS_GRW for Phase 1 (*DEC_L*).

All the experiments are executed on a machine with an AMD Phenom II X6 1055T @2.8 GHz processor and 8 GB of main memory. The ILOG CP Optimizer is configured to use only one worker (single threaded solution), and its search time limit is set to 300 seconds wall time.

Due to the complexity of the problems, for no instance it was possible to demonstrate the optimality of the solutions that we found; however, we were able to find feasible solutions for all instances except four. Here, we report only the most interesting results: the interested reader will find more detailed results, together with the downloadable replica activation problem instances used in this evaluation, in our on-line repository [262].

The bar plot in Figure 5.8 compares the various search algorithm variants to BASIC, which we choose as the base line solution method; the plot analyzes two parameters, i.e., the time to find the first feasible solution (FTIME) and its cost (FCOST), and it is obtained by normalizing, separately for each variant and problem instance, the values of FTIME and FCOST with respect to the results obtained in BASIC. The figure shows the average of these values along with the associated standard error. The plot shows that the best results are provided on average by the LNS-based search variants using a *greedy* initial solution: in all but two cases they are able to find a first feasible solution much more quickly than BASIC. In general, the *weighted random* relaxation method seems to provide some advantage with respect to *simple random* relaxations, with times up to 66% shorter. However,

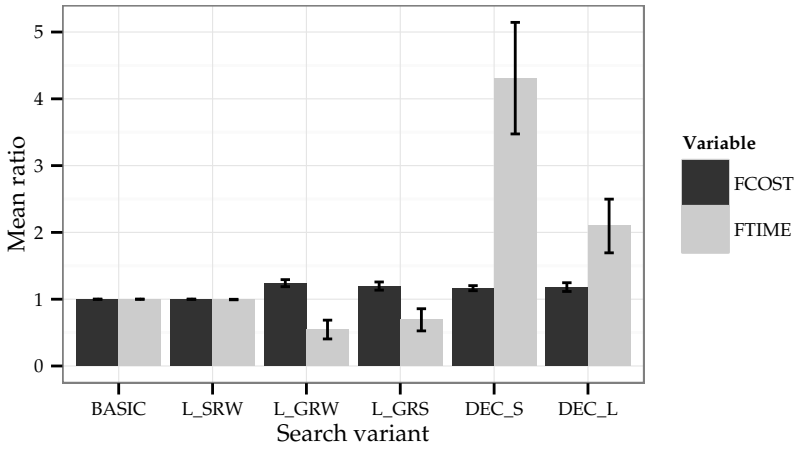


Figure 5.8: Mean time to find an initial feasible solution and associated solution cost. All the results are normalized w.r.t. the BASIC variant.

if the time needed to find a first feasible solution is normally very small, the quality of that solution is better than BASIC in only two cases and more expensive in all the others (20% and 24% on average for the simple random and the weighted random, respectively). The decomposition-based solutions perform remarkably worst than the other ones: this is due to the fact that this approach needs to find an initial solution for as many problems as the number of configurations (eight in this case) before being able to produce a first complete solution; moreover, this starting solution tries to maximize the obtained IC value, and thus is generally associated with an high cost (recall Figure 5.4b).

Similarly, in Figure 5.9, we compare the quality of the best solutions that our algorithms can find within the 5 minutes time limit. It is immediately clear that the two decomposition-based variants are able to find solutions that are at least as good or slightly better than those found by the BASIC variant. In more detail, in the only six instances where the decomposition variants finds solutions worst than BASIC, that solution is at most 23% more expensive; at the same time, they can save considerable amounts of time (43% on average). The results show also that, on the one hand, DEC_L can find good solutions much faster than DEC_S (4% to 70% faster), probably due to the initial speed-up given by the LNS greedy strategy used to find the starting solutions for Phase 1; on the other hand, the solutions found by DEC_L tend to be a little more expensive than those found by DEC_S (from 1% to 24%): that is explained by considering that the use of the BASIC solver in Phase 1 usually gives tighter IC upper bounds, which , in turn, permit to use looser constraints on the μ_c^{limit} values in Phase 2. The solutions found by the LNS-based variants, finally, are in all but one case worst than those found by BASIC, with a cost in-

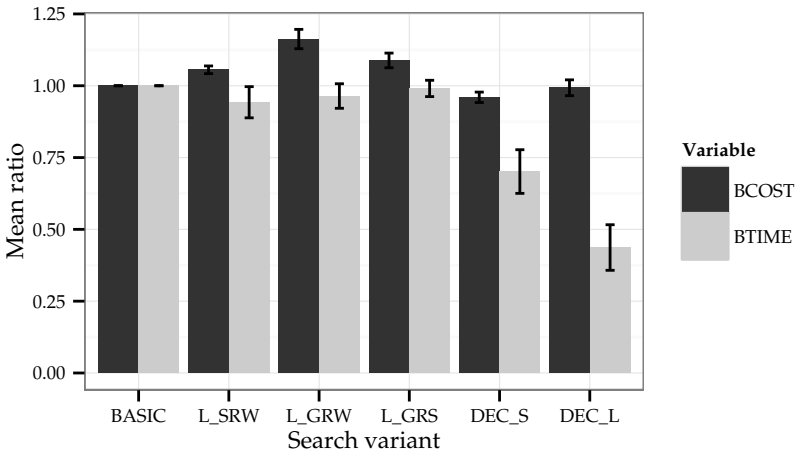


Figure 5.9: Comparison of the time needed to find the best solution within the time limit and associated solution cost (average value and standard error). All the results are normalized w.r.t. the BASIC variant.

crease that goes up to 57%. Among the three LNS-based variants, the one that provides the best results is L_SRW, thanks to its better (although slower) initial solutions, with costs that are between 1% and 13% more expensive, and solution times that are 75% faster to 26% slower than BASIC.

Finally, we have evaluated the scalability of the decomposition-based strategy when the number of input configurations grows. In order to do so, we started from an application graph with 32 operators and one data source, with a replicated deployment (64 operators) on 8 hosts, and we randomly generated 40 different applications, for each, customizing the number of possible data source rates. The result is a set of 40 different instances of the replica activation problem sharing the same processing graph and deployment, but with a progressively growing number of input configurations (from 2 to 80 by steps of 2). We tried to solve these instances by using the BASIC and the DEC_S search variants. Figure 5.10 shows the time taken by the two strategies to find their best solution as the number of input configurations grows. The results for the BASIC variants grow very quickly, and, for instances with more than 18 configurations, it not possible to find any solution within the time limit. On the contrary, by using DEC_S, the solution time grows much more slowly, and it is possible to solve all the problems up to 80 input configurations.

The presented results show that all the three LAAR optimization algorithms have their own strengths and limitations, and choosing one rather than the other depends on specific problem characteristics and on the desired properties of the solution process and of the solution itself. In fact, the LNS-based variants represent an appropriate approach

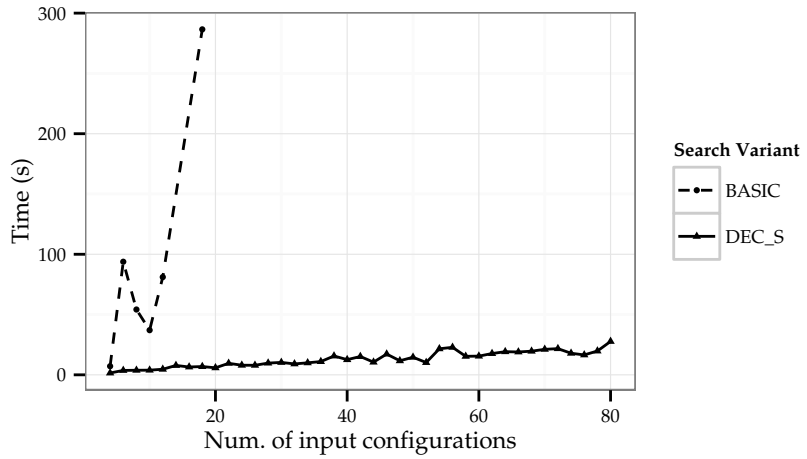


Figure 5.10: Comparison of BASIC and DEC_S average solution time when the number of input configuration grows.

when finding quickly a "good-enough" feasible solution is the main concern. On the other hand, when more time budget is available, or when the scale of the problem makes the other approaches not viable, the decomposition-based algorithm can find good solutions and scale particularly well for instances where data sources produce input at many possible rates. Finally, the straightforward implementation on the ILOG Optimizer provides the most consistent behavior across all the possible scenarios when used in combination with the partial sub-optimal solution filtering constraint.

5.6.2 On-line execution

We have implemented LAAR on top of an enterprise deployment of an industrial-strength stream processing system, IBM InfoSphere Streams, and we tested it on a set of artificially generated stream processing applications. In this section, after describing the main aspects of the implementation on Streams and the rationale behind our synthetic application generation process, we present the performance results obtained by running these applications on a 60-core cluster.

LAAR on IBM InfoSphere Streams

Already introduced in Section 3.4.2, IBM InfoSphere Streams [166] is a DSPE evolved from the SPC research project [28]. In Streams, applications are written in an ad-hoc SPL language that is used to describe *operators* and their stream connections. At compilation time, operators are transformed into their runtime counterparts, i.e., Processing Elements (PEs), each executed, after application deployment, in its own process on the host system. The mapping from operators to

PEs is usually many-to-one, as the Streams compiler can *fuse* [171] several operators into single PEs to minimize context-switching and communication overheads. At the time of writing, the only form of fault-tolerance supported natively by Streams is checkpointing [193].

In order to use LAAR, which leverages active replication support from existing platforms, we implemented a minimal active replication system on top of Streams, based on operator proxying. The same proxying technique is also leveraged to implement the replica activation and deactivation mechanism needed by LAAR at runtime. In more detail, in the application preprocessing step, the application SPL sources are modified by creating two replicas of every operator and by introducing, for each replica, a special *HAProxy* operator. This operator intercepts the input and output streams of the proxied replica and has the following functions:

- Accept *activate* and *deactivate* commands from the LAAR HAController. When active, HAProxy forwards all the input to the proxied operator replica and all its output to all the replicas of its successors; when inactive, all the input is ignored and no output tuple is forwarded.
- Send periodic heartbeat messages to the proxies of the replica's successors to indicate that the replica is alive.
- Receive heartbeats and input tuples from all the replicas of the proxied operator predecessors and forward only data from the current primary to the proxied operator.

Each proxy and its corresponding operator replica are fused into a single PE using the *partition co-location* setting.

Rate Meter and HAController PEs are also inserted in the operator graph, the latter customized with the path to a JSON file describing the replica activation strategy to use at runtime.

Experimental setup

We generated a corpus of different stream processing applications on which to test and validate our LAAR approach. To this purpose, we developed and used a generator that builds synthetic stream processing applications from a set of descriptive parameters. The output of the generation is an application descriptor, which is then transformed into a corresponding InfoSphere Streams application. Every PE in the generated application is mapped onto a deterministic Streams operator that behaves according to its concise attributes. More precisely, tuple processing is simulated through busy wait cycles of configured length, and selectivity is implemented by producing an output tuple after receiving, from an input port, a number of tuples equal or greater than an integer multiple of its selectivity. These operators are *stateless*, with no particular semantics associated with their output.

Our deployment environment consists of a 60-core IBM BladeCenter cluster. Each node is equipped with one Intel Xeon X5690 processor and 96 gigabytes of primary memory. The cluster runs an instance of InfoSphere Streams v.2.0.0.4, with one of the servers hosting Streams management services only, and the remaining dedicated to the execution of PEs. In all the experiments, we used applications composed of 24 PEs — 48, considering the twofold replication (1 PEs per CPU core) — deployed on the available servers to minimize inter-host communication. During the execution of the experiments, we periodically query Streams about the current status of all the PEs and log this information.

We present the results of a set of experiments on 100 generated applications. The application graphs have an average outgoing node degree between 1.5 and 3, and the operators are generated with port selectivity values uniformly distributed between 0.5 and 1.5. An external source produces tuples at two possible input rates (labeled "Low" and "High"), both chosen from a uniform distribution between 1 and 20 tuples per second. The PEs per-tuple CPU cost parameters are randomly generated ensuring that i) the deployment is not overloaded when all replicas are active and the input configuration is "Low" and that ii) it would instead be overloaded when all replicas are active and the input configuration is "High". Every experiment runs on a 5 minute long input trace, with the High input configuration being active for one third of the trace. All the PEs are configured with one queue for each input port, long enough to hold 2 seconds of tuples in the "High" input configuration; once a queue is filled up, new tuples are dropped.

For every application, we run experiments using six different replication approaches, henceforth referred to as *variants*. The first three variants use our LAAR approach: we used the decomposition-based search algorithm to obtain replica activation strategies for three different IC requirements of 0.5, 0.6, and 0.7, labeled in the following as L.5, L.6, and L.7, respectively. In order to compare LAAR with other possible static an dynamic replication techniques, we also run experiments using the following other three variants:

- *Non Replicated* (NR). The application is deployed on the available resources with no PE replication. A NR variant is obtained starting from the PE activations for the "High" input rate from the D.5 variant, and modifying them to make sure that only a replica of each PE is ever active. The obtained activations are used for both the possible input configurations. This simple procedure permits to quickly obtain a non replicated deployment over all the cluster resources that guarantees that the system is never overloaded.
- *Static Replication* (SR). For every PE, both replicas are active all the time independently of the current input configuration.

- *Greedy* (GRD). A dynamic replica activation strategy is derived using the following greedy algorithm: starting from a static active replication setting, for every input configuration, redundant PE replicas are iteratively disabled until every host is non overloaded; at each algorithm iteration, an overloaded host is chosen, and the replica that consumes the most CPU is chosen for deactivation. A simple heuristic is used to prefer the deactivation of upstream PE first.

Evaluation on BladeCenter cluster

We present an evaluation of the LAAR approach compared to the different replication variants considering the following two edge failure models: i) No failure ever occurs (referred to as *best-case* scenario); ii) a replica of each PE is permanently crashed throughout the experiment according to the pessimistic failure model presented in Section 5.4.1 (referred to as *worst-case* scenario); iii) during the experiment, a random server crashes and is recovered after some time. Many of the results in this section are presented through box plots, which show how metrics of interest are distributed across executions of different applications when different dynamic replication variants are used. Specifically, each box in a box plot shows the 25^{th}, 50^{th} and 75^{th} percentiles of the population of measured values. The ends of the whiskers represent the smallest (biggest) sample within 1.5 times the inter-quartile range, and circles represent outliers. We do not differentiate by other graph parameters (e.g., average node degree) since our experiments have shown that they do not affect the performance results relevantly.

Figure 5.11 (top) shows the distribution of the total CPU time used to process all the input traces when using the considered variants in a best-case scenario. To compare measurements from different applications, the results are normalized with respect to the value measured when a non-replicated deployment is used (NR). As expected, static active replication (SR) is the variant using the highest amount of CPU time to process the same trace, with the overhead due to active replication being between 61% and 90% (note that this is not 100% since the deployment cluster does not have, by design, enough resources to handle the load peak). As expected, the greedy (GRD) variant is the second most expensive one because it deactivates "just enough" replicas for the system not to be overloaded. The three LAAR variants result to be the cheapest solutions in terms of resource use and, most interestingly, the cost of each of them is proportional to the IC value requested. This is a very important feature of our solution because it gives a direct way to correlate the desired reliability level to its runtime cost.

Figure 5.11 (bottom) analyzes, in the same best-case scenario, the ability of the studied variants to efficiently use the available resources by adapting to changing input configurations. In particular, we mea-

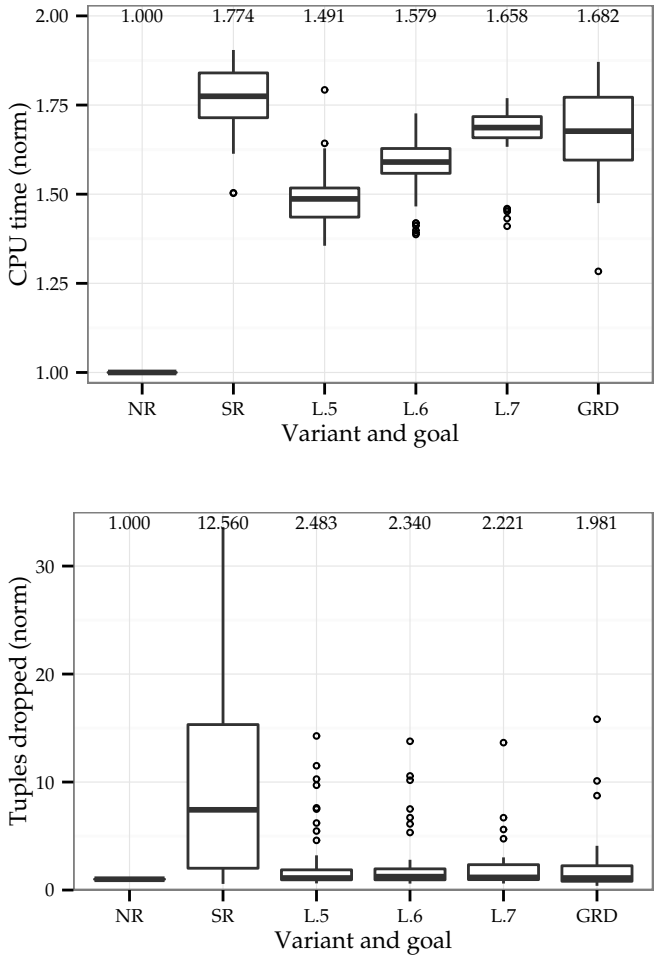


Figure 5.11: Distributions of the total CPU time used — top – and total number of tuples dropped — bottom — in a best-case experiment scenario, normalized w.r.t. the NR variant. Labels correspond to mean values.

sured the number of tuples dropped due to queues filling up during the experiment. Note that static replication, not having any form of dynamic adaptation to the input rate, can drop up to 33.6 times more tuples compared to the non-replicated deployment, with a very high variance due to the different characteristics of different applications. In all the dynamic variants, instead, the amount of tuples dropped is much smaller, even if it is not zero mainly because of the effects of glitches on the input rate (a phenomenon that could be smoothed by carefully tuning PE queue lengths).

Another important element we looked at for the evaluation of our approach is the applications output tuple rate during load peaks. In fact, this value directly depends on resource availability and on the ability of the platform to effectively use these resources. For each different application, we measured the average output data rate in the

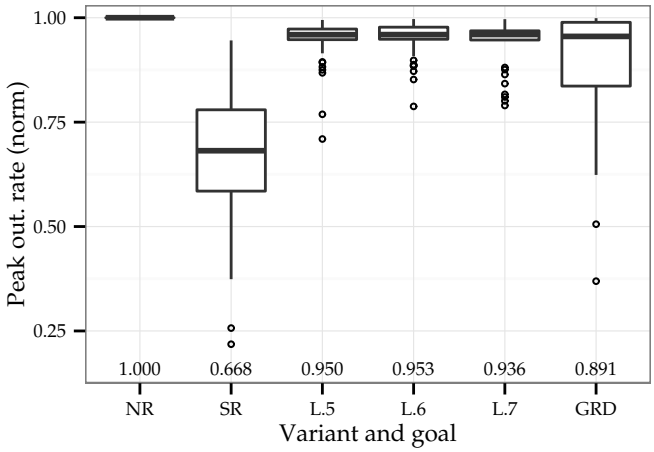


Figure 5.12: Output data rate during load peak, normalized w.r.t. the NR variant. Labels correspond to mean values.

over-provisioned and non-replicated configuration (NR) and used it as reference to evaluate the other variants. Figure 5.12 shows the distribution of the output data rate ratios measured for all the applications and variants. When static active replication (SR) is used, the applications output rate is on average 33% slower than the NR variant (up to 63%), while, when LAAR is used, the rate is at most 9% slower. The greedy variant (GRD) is in general better than the static one, but, differently from LAAR, it is not able to provide a consistent behavior across different applications, with an output rate ratio that is from 2% to 38% slower than NR.

We ran the same applications also assuming the pessimistic failure model defined in Section 5.4.1, in order to verify whether our system was actually able to satisfy the promised IC requirements in real stream processing deployments. The graph at the top of Figure 5.13 shows the distribution of the normalized total number of samples produced by PEs in the worst-case scenario for all the replication variants. While the NR variant fails to produce any output (recall that in this failure model one active replica of each PE is permanently failed), the three LAAR variants are able to produce a fraction of output tuples that satisfies their respective IC requirement, except in a very limited number of cases, in which the violation is still never bigger than 4.7%. On the contrary, the GRD variant, while performing well in many cases, is not able to provide a consistent behavior across different application, with measured IC values that can be as big as 0.95 but also as low as 0.35.

Figure 5.14 summarizes the presented results showing the average numbers of tuples dropped, the average IC value, and the average cost of different replication strategies compared to the static active replication variant. It is immediate to see that LAAR permits to con-

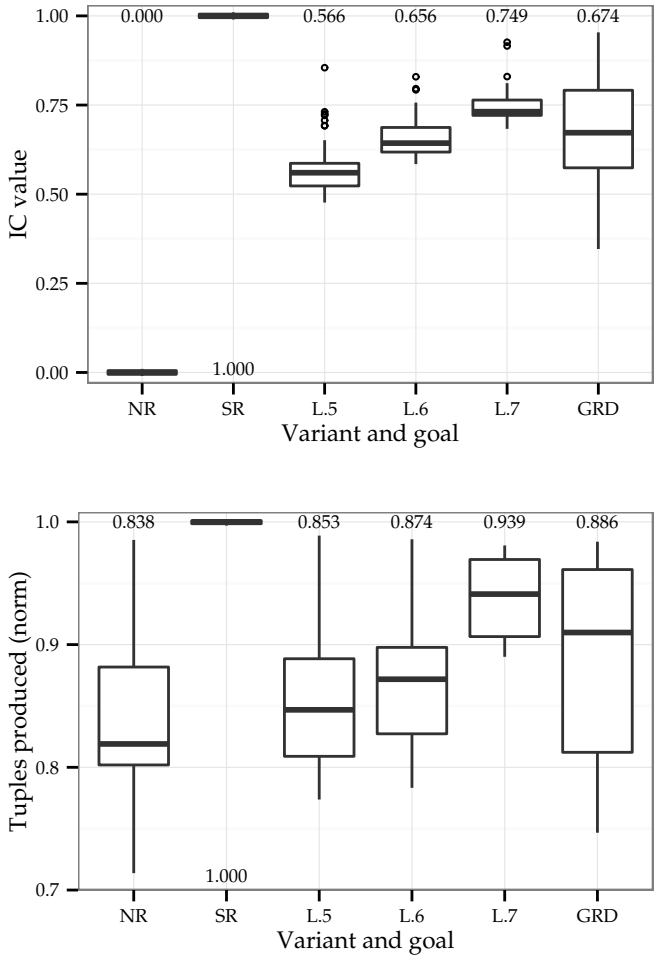


Figure 5.13: Total number of samples processed simulating the pessimistic (worst-case) failure model — top — and a single server crash model — bottom — normalized w.r.t. the failure-free NR variant (SR is introduced for ease of comparison). Labels correspond to mean values.

trol application execution costs by tuning the desired IC guarantees, a crucial property in our business application scenario.

Finally, in order to evaluate the system behavior when more realistic failure scenarios are considered, we re-executed a randomly sampled subset of 40 applications, using a *single host crash-failures with recovery* failure model. In practice, during each experiment run, we randomly crash one of the PE hosting servers. The failure lasts for 16 seconds, i.e., the time needed, according to the experiences in [263], for Streams to detect it and migrate failed PEs to another host. We force these failures to occur only during "High" input configurations, because it is the case when LAAR provides the weaker fault-tolerance guarantees (thus disfavoring our solution). Figure 5.13 (bottom) shows the IC values measured for the different variants in this

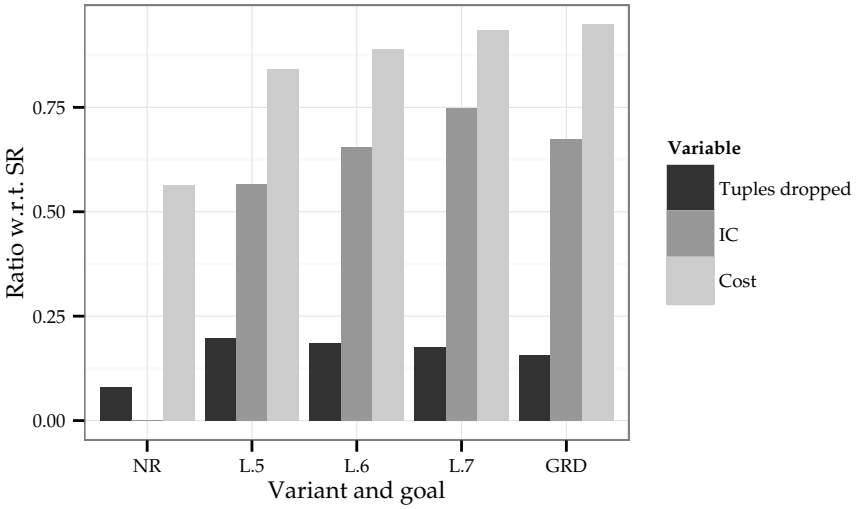


Figure 5.14: Summary: comparison of the different variants (mean values normalized w.r.t. SR)

scenario. As expected, the IC measured for the LAAR variants is much higher than their guaranteed values, given the less pessimistic failure model. Note that the results for L.5 are similar to those measured for the no-replication variant: recall, in fact, that the NR deployment is derived from L.5 by deactivating the replicas which have still two active replicas in the "High" configuration (which are usually just a few). Once again, the GRD variant confirms its unpredictable behavior in the way it responds to failures.

5.7 FUTURE WORK

The good feedback received from our LAAR experimental evaluation encourage us to continue our exploration of the possibilities given by weaker fault-tolerance techniques. In particular, we will continue our work on LAAR and try to improve both the off-line optimization phase and its on-line enforcement mechanism.

On the one hand, for what concerns the LAAR optimization problem, we plan to:

- Investigate the use of alternative and more realistic failure models with the goal of providing tighter lower bounds on the guaranteed IC values.
- Extend the replica activation problem formulation by developing a penalty model associated to IC violations considered as minimization terms rather than strict constraints.
- Conduct a deeper examination of the replica activation problem and try to correlate specific problem characteristics (e.g., patterns in the graph shape, deployment properties) to the performance of different search strategies.

On the other hand, we want to improve and validate our runtime LAAR implementation further. Specifically, we consider the following opportunities particularly worth of being explored:

- Extend the platform experimental evaluation to complex real-world applications, evaluating its impact on the semantics of applications output in different scenarios.
- Integrate LAAR with other DSPSs, including Quasit, and verify its actual portability.
- Study the effects that unexpected input configurations have on the LAAR quality guarantees, and consider to develop on-line algorithms that adapt statically computed replica activation strategies to new runtime conditions.

Let us finally note that, although we have designed and developed LAAR with the stream processing context in mind, the presented principles are applicable to the much larger domain of distributed data flow systems that can tolerate weaker fault-tolerance levels through dynamic active replication. An important research direction will be the validation of this claim by applying and evaluating our technique on a broader set of data management scenarios.

5.8 SUMMARY AND CONCLUSIONS

Stream processing service providers may need to temporarily provision additional resources to hosted applications during load spikes, if avoiding to drop or delay application tuples is a requirement. In this chapter, we have presented LAAR, a novel technique for dynamic active replication that can reduce the costs of provisioning these resources by enforcing weaker fault tolerance guarantees for applications that can tolerate them. In particular, LAAR is able to temporarily gather CPU resources by dynamically activating and deactivating operator replicas according to the current system load. The runtime replica deactivation decisions are guided by a so called *replica activation strategy* which is built by solving an off-line optimization problem that uses limited stochastic knowledge of application characteristics. We have presented three different algorithms that solve this problem using several constraint programming techniques, and we have shown that each of them has its own advantages and limitations. Finally, to evaluate the effectiveness of our technique, we have implemented LAAR on top of IBM InfoSphere Streams and deployed it on a 60-cores cluster. Our evaluation shows that LAAR can effectively trade runtime costs off for perfect consistency while still enforcing completely predictable and guaranteed fault-tolerance levels.

We are continuing our work on LAAR in two main directions. First, we are trying to improve the underlying optimization model making

less restrictive assumptions about IC satisfaction constraints, and including the possibility of IC violations in a penalty-based cost model. Second, we are studying the still unexplored problem of dealing with incomplete or partially wrong statistical descriptions of applications, by trying to develop smart on-line algorithms that compensate to off-line optimization imprecision. Let us finally note that, although we have presented LAAR in the context of stream processing, we deem it applicable to the much larger domain of distributed data flow systems that can tolerate incompleteness.

6 CONCLUSIONS

In this dissertation, we have presented our work about Quality of Service (QoS) provisioning in large scale middleware infrastructures supporting the vision of Smart Pervasive Environments (SPEs). In this chapter we summarize our contributions by reporting our most important findings, and point out open research directions for future extensions of our work.

6.1 MAJOR CONTRIBUTIONS

In Chapter 2, we have identified the quality requirements of middleware services supporting the distribution of pervasive sensing data, and we have proposed an original QoS-based model for PUB/SUB systems that satisfy these requirements. Based on this model, we have presented a detailed technical survey of many state-of-the-art PUB/SUB solutions analyzing their specific offers of QoS enforcement mechanisms. Starting from the comparison of their characteristics, we have highlighted the general model, design, and technical aspects that regulate the relationships between quality guarantees, system scalability, and runtime data distribution costs in PUB/SUB data exchange. From a proper understanding of these trade-offs, we have derived promising guidelines for future research in the field of QoS-based large scale data distribution, and we have provided practical and general principles useful for the development of future PUB/SUB systems extensions that effectively address the technical challenges of scalability and quality.

Chapter 3 has expanded this modeling and surveying work to the area of scalable data center stream processing. We have proposed a simple yet powerful description framework for DSPSs based on the orthogonal analysis of their abstract, development, and execution models, and, through it, we have discussed the semantics and technical implementation details of the most relevant QoS provisioning features currently available in DSPS solutions. Once again, the goal of our analysis has been to put forth the prominent design and technical challenges, together with existing and envisioned solutions, for the runtime enforcement of different QoS properties.

This study has led to the definition of the requirements for a QoS-centric DSPS supporting the needs of pervasive environments. We have presented an original stream processing model and architecture answering these requirements in Chapter 4 by introducing Quasit,

a novel distributed platform providing a scalable and highly configurable execution environment for stream processing applications. Uniquely, Quasit permits to decorate from coarse to fine grained elements of stream processing applications with QoS specifications, i.e., collections of quality requirements regulating their expected runtime behaviour, and uses them to optimize its resource allocation policies. The reported experimental evaluation of the Quasit prototype has shown that Quasit offers scalable performance and limited management overhead, also if compared to the widely used industrial-level Apache S4 DSPS.

Our last contribution, presented in Chapter 5, is an on-the-field investigation on the promising opportunities offered by flexible QoS enforcement mechanisms that trade perfect guarantees off for reduced costs. We have experimented with this particular class of QoS properties by proposing a novel technique for dynamic active replication, called LAAR, that allows applications to request custom fault-tolerance levels, measured through an ad-hoc consistency metric. Through a wide set of experiments on real-world sized data center deployments, we have shown that LAAR is able to temporarily gather CPU resources by dynamically activating and deactivating operator replicas according to the current system load and, by doing so, it can effectively trade-off runtime costs with perfect consistency while still enforcing completely predictable fault-tolerance levels.

On the whole, our all out surveying, modeling, implementation, and experimental work demonstrates that, by providing data distribution and processing middleware with application-level knowledge of the quality requirements associated to different data processing and distribution flows, it is possible to optimize significantly the utilization of computational and network resources. The principles at the core of this improvement are i) always to avoid over-provisioning resources when and where they are non-strictly necessary, and ii) to adapt their allocation dynamically also in response to changing system conditions. By rigorously applying these two simple but fundamental principles in every architectural and implementation aspect of the streams management infrastructure, it is possible to improve systems scalability by making a smarter use of the available resources, and, at the same time, to keep the growth of execution costs always limited and proportional to the overall quality of service supplied.

6.2 FUTURE RESEARCH DIRECTIONS

In this thesis we have done a thorough cross-layer analysis of the many problems related directly or indirectly to the design and implementation of scalable and QoS-aware middleware supports for SPE. We have explored several very important research aspects of the prob-

lem, but we have also identified many still open research questions worth pursuing. In this section, we concentrate the primary directions that, in our opinion, should drive future research efforts in four simple principles:

- *Offer domain-specific* QoS *policies.* In this thesis, we have identified, modelled, and classified several QoS policies for the data distribution and processing layers of emerging SPE architectures. In choosing which policies to consider, we were guided by the goal of representing general policies that could be useful in the largest possible variety of SPE applications, while we purposefully did not consider domain-specific policies, i.e., policies only meaningful in specific application domains. One well representative example is the set of QoS parameters that regulate middleware behavior according to energy-related considerations, very relevant, for example, in WIRELESS SENSOR NETWORKS (WSNs) [264] data streaming scenarios. We believe that future research should narrow the investigation scope also to specific application areas and identify and study their most relevant QoS parameters. This work could lead to further optimizations in SPE infrastructures and improve their adaptability and scalability.
- *Stratified* QoS *modeling.* We have remarked many times the role of QoS-awareness as a mean to enable better infrastructure scalability through intelligent resource management. The more details users can provide about their applications quality requirements — or, in other words, the finer the grain of QoS-based configuration possibilities —, the more the opportunities to exploit this external knowledge for runtime optimization purposes. However, as more and more customization knobs are provided, there is an increasing risk of a quickly exploding complexity in the system interfaces offered to infrastructure users. Future research efforts should study extensively how this complexity can be managed. Providing sensible *default* values for QoS specifications is an obvious solution that can work in simple scenarios, but less simplistic solutions must be sought for the more general case. A possible viable approach is, in our opinion, the definition of *stratified* QoS models, where a large and very fine grained set of QoS policies defined at the lowest stratum is gradually abstracted by increasingly less, simpler, and more abstract policies at higher levels. A major challenge, here, will be to define meaningful mappings from higher to lower strata, and to consider the possibility to dynamically modify these mappings through adaptive runtime mechanisms.
- *Partial and probabilistic* QoS. Our experience with LAAR (Chatper 5) has only begun to scratch the surface of the many opportunities that, in our opinion, lay in weak, approximated, partial, and stochastic models for QoS enforcement. As new large

scale pervasive services are being proposed, it is becoming evident that not every application needs traditional perfect and strict quality guarantees all the time. There are many possible application-specific reasons that justify partial quality guarantees, including low task criticality, information redundancy, or domain-specific tolerance to approximated processing results. The challenge is to identify all these opportunities and to define algorithms that can intelligently exploit the additional degrees of freedom offered by imperfect service level requests. If properly leveraged, as we have shown in our LAAR use case, this new kind of QoS specifications have the potential to greatly improve system scalability by providing even greater resource management flexibility.

- *Hierarchical* QoS *enforcement.* While we have shown that the exploitation of the differentiated quality characteristics of different application can improve scalability through better resource allocation, we recognize, on the other hand, that the global enforcement of strong quality guarantees, such as perfect fault-tolerance or strict processing latency bounds, can become harder and harder as scale grows. This is especially true when a large number of possibly geographically dispersed distributed participants are all involved in the enforcement of quality-based behavior: in these cases, to run distributed algorithms for global and strict QoS admission control, monitoring, and enforcement can be severely expensive. We are convinced that a possible solution approach to successfully face this kind of challenges stands in the development of new QoS management architectures based on hierarchical models that make extensive use of locality principles. For examples, algorithms based on this idea could use traditional and strict QoS enforcement techniques within local clusters of components of controlled size, and then scale up by treating local clusters as hierarchical units where QoS is handled through looser and more scalable methods.

6.3 FINAL REMARKS

The actual realization of the vision of smart pervasive environments is an exciting perspective with the potential of bringing tremendous improvements to our society with the promise of novel pervasive services assisting us in every aspect of our lives. While the increasingly deep integration of ICT services in the physical world has been the main enabling factor motivating this vision, we believe that the intelligent and efficient exploitation of the resulting information flows will be the key to its success. This is confirmed by the growing interest in "Big Data"-related initiatives from both academia and industry

and by the important financial investments in that area for the years to come. Within the "Big Data" movement — an umbrella expression often used to refer to the wide set of cross-concerning technological areas of data storage, distribution, analysis, and visualization — the management of moving data still presents many important open challenges. Our contribution has proposed an original solution approach for dealing with big data streams distribution and analysis issues, which identifies the intelligent exploitation of differentiated quality of service as the key factor for achieving scalability and cost effectiveness. We hope that our work was successful in proposing general and simple solution principles that can be used fruitfully as starting points for new research efforts toward the realization of future large scale smart pervasive environments. Finally, let us note that, while the scope of our research was focused on the specific application scenario of smart pervasive environments, we believe that most of the presented results can be easily and successfully used in other scenarios with similar requirements of efficient and scalable distribution and analysis of real time data streams, such as the management of large scale telecommunication infrastructures or smart grids deployments.

ACRONYMS

ACID	Atomicity, Consistency, Isolation, and Durability 49
ACK	Acknowledgement 34
AET	Ack Expected Threshold 34
API	Application Programming Interface 31
ARQ	Automatic Repeat-reQuest 26
BIC	Best-Case Internal Completeness 126
BSD	Berkeley Software Distribution
CDF	Cumulative Distribution Function 28
CDR	Common Data Representation 112
CEP	Complex Event Processing 47
CP	Constraint Programming 118
CPU	Central Processing Unit
CTG	Conditional Task Graph 120
CQL	Continuous Query Language 54
DAG	Directed and Acyclic Graph 82
DBMS	Data Base Management System 49
DISC	Data-Intensive Scalable Computing 45
DCRD	Delay-Cognizant Reliable Delivery 32
DCPS	Data-Centric Publish Subscribe 35
DDPE	Distributed Data Processing Engine 49
DDS	Data Distribution Service (OMG Standard) 35
DHT	Distributed Hash Table 33
DPC	Delay, Process, and Correct 67
DSMS	Data Stream Management System 47
DSPE	Distributed Stream Processing Engine 50
DSPS	Distributed Stream Processing System 6
ESDS	Eventually Serializable Data Service 119
FIC	Failure Internal Completeness 126
FIFO	First In, First Out 24
GCT	Gap Curiosity Threshold 34
GD	Guaranteed Delivery (PUB/SUB routing protocol) 34
GPS	Global Positioning System

HDFS Hadoop Distributed File System 104
HTTP Hypertext Transfer Protocol
IC Internal Completeness 126
ICT Information and Communications Technology 56
IDL Interface Definition Language 90
IFP Information Flow Processing 47
IntServ Integrated Services 33
I/O Input/Output
IP Internet Protocol
JMS Java Message Service 31
JMX Java Management Extensions 100
JSON Javascript Object Notation
JSR Java Specification Request 31
JVM Java Virtual Machine 55
LAN Local Area Network 6
LNS Large Neighborhood Search 129
LAAR Load-Adaptive Active Replication 118
MANET Mobile Ad-hoc Network 39
MTC Many-Tasks Computing 45
NACK Negative Acknowledgement 34
OD-SIG Operator Definition SIG 84
O-R Offer – Request (QoS Agreement Model) 20
O-R-C Offer – Request – Confirm (QoS Agreement Model) 20
OMG Object Management Group 35
OS Operating System 52
P2P Peer to Peer 18
PaaS Platform as a Service
PE Processing Element 70
PDBMS Parallel Data Base Management System 49
PNG Portable Network Graphics
POSIX Portable Operating System Interface
PUB/SUB Publish/Subscribe 10
QDM Quasit Domain Manager 91
QOR Quasit Operator Repository 93
QoS Quality of Service 2
QRN Quasit Runtime Node 91

RBNB	Ring Buffer Network Bus 31
RBO	Ring Buffer Object 31
RFID	Radio-frequency Identification
RMI	Remote Method Invocation 12
RPC	Remote Procedure Call 23
RSVP	Resource Reservation Protocol 33
RTT	Round Trip Time 108
SIG	Streaming Information Graph 82
SLA	Service Level Agreement 56
SPC	Stream Processing Core 55
SPE	Smart Pervasive Environment 2
SPL	Stream Processing Language 69
SQL	Structured Query Language 49
SQuAL	Stream Query Algebra 67
TCB	Timely Computing Base 37
TCP	Transmission Control Protocol
TMS	Traffic Management System 7
TTL	Time To Live 25
UDF	User Defined Function 53
UDP	User Datagram Protocol
WFQ	Weighted Fair Queueing 28
WSN	Wireless Sensor Network 153

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

BIBLIOGRAPHY

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94–104, 1991.

[2] N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. Campbell, "A survey of mobile phone sensing," *Communications Magazine, IEEE*, vol. 48, no. 9, pp. 140–150, 2010.

[3] U. Lee, B. Zhou, M. Gerla, E. Magistretti, P. Bellavista, and A. Corradi, "Mobeyes: smart mobs for urban monitoring with a vehicular sensor network," *Wireless Communications, IEEE*, vol. 13, no. 5, pp. 52–57, 2006.

[4] F. Martinez, C.-K. Toh, J.-c. Cano, C. Calafate, and P. Manzoni, "Emergency services in future intelligent transportation systems based on vehicular communication networks," *Intelligent Transportation Systems Magazine, IEEE*, vol. 2, no. 2, pp. 6–20, 2010.

[5] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna, "Adaptive rate stream processing for smart grid applications on clouds," in *Proc. of the 2nd international workshop on Scientific cloud computing*, (San Jose, CA, USA), p. 33, ACM Press, 2011.

[6] S. Aman, Y. Simmhan, and V. K. Prasanna, "Improving Energy Use Forecast for Campus Micro-grids Using Indirect Indicators," in *Proc. of the IEEE 11th International Conference on Data Mining Workshops*, (Vancouver, Canada), pp. 389–397, IEEE, 2011.

[7] N. Wickramasinghe, "Pervasive computing and healthcare," in *Pervasive Health Knowledge Management* (R. Bali, I. Troshani, S. Goldberg, and N. Wickramasinghe, eds.), ch. 2, pp. 7–13, New York, NY: Springer New York, 2013.

[8] J. Norman, "Impact of pervasive computing in education," *International Journal of Education and Learning*, vol. 2, no. 2, pp. 39–48, 2013.

[9] K. Martinez, J. Hart, and R. Ong, "Environmental sensor networks," *Computer, IEEE*, vol. 37, no. 8, pp. 50–56, 2004.

[10] G. Ritzer and N. Jurgenson, "Production, consumption, prosumption: The nature of capitalism in the age of the digital 'prosumer'," *Journal of Consumer Culture*, vol. 10, no. 1, pp. 13–36, 2010.

[11] A. Stuart and E. Thorsen, eds., *Citizen Journalism: global perspectives*. Peter Lang Publishing, 2009.

[12] T. Sakaki, M. Okazaki, and Y. Matsuo, "Earthquake shakes twitter users: Real-time event detection by social sensors," in *Proc. of the 19th International Conference on World Wide Web*, (Raleigh, NC, USA), pp. 851–860, ACM, 2010.

[13] F. Wang, S. Liu, P. Liu, and Y. Bai, "Bridging physical and virtual worlds: Complex event processing for rfid data streams," in *Proc. of the of 10th International Conference on Extending Database Technology*, pp. 588–607, Munich, Germany: Springer Berlin Heidelberg, 2006.

[14] J. Candia, M. C. González, P. Wang, T. Schoenharl, G. Madey, and A.-L. Barabási, "Uncovering individual and collective human dynamics from mobile phone records," *Journal of Physics A: Mathematical and Theoretical*, vol. 41, no. 22, p. 224015, 2008.

[15] G. Cardone, L. Foschini, P. Bellavista, A. Corradi, C. Borcea, M. Talasila, and R. Curtmola, "Fostering participaction in smart cities: a geo-social crowdsensing platform," *Communications Magazine, IEEE*, vol. 51, no. 6, pp. 112–119, 2013.

[16] A. Grilo, A. Casaca, P. Pereira, L. Buttyan, J. Goncalves, and C. Fortunato, "A wireless sensor and actuator network for improving the electrical power grid dependability," in *Proc. of the 8th IEEE Conference on Next Generation Internet (NGI)*, pp. 71–78, IEEE, 2012.

[17] S. Djahel, M. Salehie, I. Tal, and P. Jamshidi, "Adaptive traffic management for secure and efficient emergency services in smart cities," in *Proc. of the IEEE International Conference on Pervasive Computing and Communication – WIP Session*, (San Diego, CA, USA), pp. 340–343, IEEE, 2013.

[18] J. W. Yoon, F. Pinelli, and F. Calabrese, "Cityride: A predictive bike sharing journey advisor," in *Proc. of the 13th IEEE International Conference on Mobile Data Management*, (Luleå, Sweden), pp. 306–311, 2012.

[19] Y. Bengio, *Learning Deep Architectures for AI*, vol. 2 of *Foundations and Trends in Machine Learning*. Now Publishing, 2009.

[20] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei, "A scalable and elastic publish/subscribe service," in *Proc. of the 25th IEEE International Parallel and Distributed Processing Symposium*, (Anchorage, AK, USA), pp. 1254–1265, IEEE, 2011.

[21] L. Barroso, J. Dean, and U. Holzle, "Web search for a planet: the google cluster architecture," *Micro, IEEE*, vol. 23, no. 2, pp. 22–28, 2003.

[22] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. of the 19th ACM Symposium on Operating Systems Principles*, (Bolton Landing, NY, USA), pp. 164–177, ACM, 2003.

[23] S. Srikantaiah, A. Kansal, and F. Zhao, "Energy aware consolidation for cloud computing," in *Proc. of the 2008 Conference on Power Aware Computing and Systems*, (San Diego, CA, USA), pp. 10–14, USENIX Association, 2008.

[24] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on COmputer Systems*, vol. 41, (Lisbon, Portugal), pp. 59–72, ACM, 2007.

[26] A. Alexandrov, M. Heimel, V. Markl, F. Hueske, E. Nijkamp, S. Ewen, O. Kao, and D. Warneke, "Massively parallel data analysis with pacts on nephele," in *Proc. of the 36th International Conference on Very Large Data Bases*, (Singapore), pp. 1625–1628, VLDB Endowment, 2010.

[27] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, 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 *Proc. of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, (Asilomar, CA, USA), CIDR Conference, 2005.

[28] L. Amini, H. Andrade, R. Bhagwan, and Frank Eskesen and Richard King and Yoonho Park and Chitra Venkatramani, "Spc: A distributed, scalable platform for data mining," in *Proc. of the 4th Workshop on Data Mining Standards, Services and Platforms*, (Philadelphia, PA, USA), pp. 27–37, ACM Press, 2006.

[29] N. Marz, "Storm project." `http://storm-project.net/`. Web Page, last visited in Dec. 2013.

[30] A. S. Foundation, "Hadoop mapreduce." `http://hadoop.apache.org/`. Web Page, last visited in Dec. 2013.

[31] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[32] P. Tang and T. Venables, "Smart homes and telecare for independent living," *J. Telemed. Telecare*, vol. 6, no. 1, pp. 8–14, 2000.

[33] C. Xiang, P. Yang, C. Tian, Y. Yan, X. Wu, and Y. Liu, "Passfit: Participatory sensing and filtering for identifying truthful urban pollution sources," *Sensors Journal, IEEE*, vol. 13, no. 10, pp. 3721–3732, 2013.

[34] P. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *Computing Surveys, ACM*, vol. 35, no. 2, pp. 114–131, 2003.

[35] Z. Wan and P. Hudak, "Functional reactive programming from first principles," *SIGPLAN Not.*, vol. 35, no. 5, pp. 242–252, 2000.

[36] J.-L. Maréchaux, "Combining service-oriented architecture and event-driven architecture using an enterprise service bus," *IBM Developer Works*, pp. 1269–1275, 2006.

[37] P. Pietzuch, D. Eyers, S. Kounev, and B. Shand, "Towards a common api for publish/subscribe," in *Proc. of the 2007 Inaugural International Conference on Distributed event-based systems*, (Toronto, Canada), pp. 152–157, ACM, 2007.

[38] I. Delamer and J. Lastra, "Loosely-coupled automation systems using device-level soa," in *Proc. of the 5th IEEE International Conference on Industrial Informatics*, vol. 2, (Vienna, Austria), pp. 743–748, 2007.

[39] C. Esposito, M. Ficco, F. Palmieri, and A. Castiglione, "Interconnecting federated clouds by using publish-subscribe service," *Cluster Computing*, pp. 1–17, 2013.

[40] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A survey of qos architectures," *Multimedia Systems*, vol. 6, no. 3, pp. 138–151, 1998.

[41] J. A. Zinky, D. E. Bakken, and R. E. Schantz, "Architectural support for quality of service for corba objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 55–73, 1997.

[42] D. Menasce, "Qos issues in web services," *Internet Computing, IEEE*, vol. 6, no. 6, pp. 72–75, 2002.

[43] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *Transactions on Software Engineering, IEEE*, vol. 30, no. 5, pp. 311–327, 2004.

[44] F. Araújo and L. Rodrigues, "On QoS-aware publish-subscribe," in *Proc. of the 22nd International Conference on Distributed Computing Systems Workshops*, (Vienna, Austria), pp. 511–515, IEEE, 2002.

[45] R. Baldoni, L. Querzoni, S. Tarkoma, and A. Virgillito, "Distributed event routing in publish/subscribe systems," in *Middleware for Network Eccentric and Mobile Applications*, ch. 10, pp. 219–244, Göteborg, Sweden: Springer, 2009.

[46] J. Martins and S. Duarte, "Routing algorithms for content-based publish/subscribe systems," *Communications Surveys & Tutorials, IEEE*, vol. 12, no. 1, pp. 39–58, 2010.

[47] R. Meier, "Taxonomy of distributed event-based programming systems," *The Computer Journal*, vol. 48, no. 5, pp. 602–626, 2005.

[48] J.-L. Poza-Luján, J.-L. Posadas-Yagüe, and J.-E. Simó-Ten, "A survey on quality of service support on middleware-based distributed messaging systems used in multi agent systems," in *Proc. of the International Symposium on Distributed Computing and Artificial Intelligence*, vol. 91, (Salamanca, Spain), pp. 77–84, Springer Berlin / Heidelberg, 2011.

[49] S. P. Mahambre, M. Kumar, and U. Bellur, "A taxonomy of qos-aware, adaptive event-dissemination middleware," *Internet Computing, IEEE*, vol. 11, no. 4, pp. 35–44, 2007.

[50] G. Mühl, L. Fiege, F. Gartner, and A. Buchmann, "Evaluating advanced routing algorithms for content-based publish/subscribe systems," in *Proc. of the 10th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, (Forth Worth, TX, USA), pp. 167–176, IEEE, 2002.

[51] A. Corsaro, L. Querzoni, S. Scipioni, S. Piergiovanni, and A. Virgillito, "Quality of service in publish/subscribe middleware," in *Global Data Management*, ch. 5, pp. 78–96, IOS Press, 2006.

[52] S. Behnel, L. Fiege, and G. Muhl, "On quality-of-service and publish-subscribe," in *Proc. of the 26th International Conference on Distributed Computing Systems Workshops*, (Lisboa, Portugal), pp. 20–20, IEEE, 2006.

[53] P. Eugster and R. Guerraoui, "Distributed programming with typed events," *Software, IEEE*, vol. 21, no. 2, pp. 56–64, 2004.

[54] P. Eugster, B. Garbinato, and A. Holzer, "Pervaho: A specialized middleware for mobile context-aware applications," *Electronic Commerce Research*, vol. 9, no. 4, pp. 245–268, 2009.

[55] OMG, "Data distribution service for real-time systems, version 1.2," specification, Object Management Group, 2007.

[56] P. R. Pietzuch and J. Bacon, "Hermes: a distributed event-based middleware architecture," in *Proc. of the 22nd International Conference on Distributed Computing Systems Workshops*, (Vienna, Austria), pp. 611–618, IEEE, 2002.

[57] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, 2001.

[58] G. Cugola, E. Di Nitto, and A. Fuggetta, "The jedi event-based infrastructure and its application to the development of the opss wfms," *Transactions on Software Engineering, IEEE*, vol. 27, no. 9, pp. 827–850, 2001.

[59] G. Cugola, A. Margara, and M. Migliavacca, "Context-aware publish-subscribe: Model, implementation, and evaluation," in *Proc. of the IEEE Symposium on Computers and Communications*, (Sousse, Tunisia), pp. 875–881, IEEE, 2009.

[60] G. Mühl, "Generic constraints for content-based publish/subscribe," in *Cooperative Information Systems*, vol. 2172 of *Lecture Notes in Computer Science*, pp. 211–225, Berlin, Heidelberg: Springer Berlin / Heidelberg, 2001.

[61] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. V. Steen, "Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks," in *Proc. of the 5th International Workshop On Peer-to-Peer Systems*, (Santa Barbara, CA, USA), 2006.

[62] OMG, "Corba notification service, version 1.1," specification, Object Management Group, 2004.

[63] G. Pardo-Castellote, "Omg data-distribution service: architectural overview," in *Proc. of the 23rd International Conference on Distributed Computing Systems Workshops*, (Providence, RI, USA), pp. 200–206, IEEE, 2003.

[64] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java message service," specification, Sun Microsystems Inc., Santa Clara, CA, 2002.

[65] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication," in *Proc. of the Inaugural International Conference on Distributed Event-Based Systems*, (Toronto, Canada), pp. 14–25, ACM, 2007.

[66] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *Networked Group Communication*, vol. 2233 of *Lecture Notes in Computer Science*, pp. 30–43, Berlin / Heidelberg, Germany: Springer Berlin / Heidelberg, 2001.

[67] S. Tilak, P. Hubbard, M. Miller, and T. Fountain, "The ring buffer network bus (rbnb) dataturbine streaming data middleware for environmental observing systems," in *Proc. of the 3rd International Conference on e-Science and Grid Computing*, (Bangalore, India), pp. 125–133, IEEE, 2007.

[68] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, "Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination," in *Proc. of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, (Port Jefferson, NY, USA), pp. 11–20, ACM, 2001.

[69] B. Wang and J. Hou, "Multicast routing and its qos extension: problems, algorithms, and protocols," *Network, IEEE*, vol. 14, no. 1, pp. 22–36, 2000.

[70] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski, "The padres distributed publish/subscribe system," in *Proc. of the 8th International Conference on Feature Interactions in Telecommuncations and Software Systems*, (Leicester, UK), pp. 12–30, IOS Press, 2005.

[71] P. Eugster, P. Felber, R. Guerraoui, and S. Handurukande, "Event systems. how to have your cake and eat it too," in *Proc. of the 22nd International Conference on Distributed Computing Systems Workshops*, (Vienna, Austria), pp. 625–630, IEEE, 2002.

[72] R. Henjes, D. Schlosser, M. Menth, V. Himmler, and A. Hubland, "Throughput performance of the activemq jms server," in *Kommunikation in Verteilten Systemen (KiVS) 2007*, Informatik aktuell, ch. 10, pp. 113–124, Springer Berlin Heidelberg, 2007.

[73] S. Pallickara and G. Fox, "Naradabrokering: a distributed middleware framework and architecture for enabling durable peer-to-peer grids," in *Proc. of the ACM/IFIP/USENIX International Conference on Middleware*, (Rio de Janeiro, Brazil), pp. 41–61, Springer-Verlag New York, Inc., 2003.

[74] P. R. Pietzuch and J. Bacon, "Peer-to-peer overlay broker networks in an event-based middleware," in *Proc. of the 2nd International Workshop on Distributed Event-Based Systems*, (San Diego, CA, USA), pp. 1–8, ACM, 2003.

[75] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes,"

Communications Surveys & Tutorials, IEEE, vol. 7, no. 2, pp. 72–93, 2005.

[76] R. Meier and V. Cahill, "On event-based middleware for location-aware mobile applications," *Transactions on Software Engineering, IEEE*, vol. 36, no. 3, pp. 409–430, 2010.

[77] TIBCO Software Inc., "Tibco rendezvous data sheet," tech. rep., 2008. `http://www.tibco.com/multimedia/ds-rendezvous_tcm8-826.pdf`. Online resource, last retrieved in Dec. 2013.

[78] A. Tanenbaum, *Computer Networks*, ch. 5, pp. 397–417. Prentice Hall, 4th ed., 2003.

[79] R. Baldoni, R. Beraldi, S. Piergiovanni, and A. Virgillito, "Measuring notification loss in publish/subscribe communication systems," in *Proc. of the 10th Pacific Rim International Symposium on Dependable Computing*, (Papeete, Tahiti), pp. 84–93, IEEE, 2004.

[80] S. Bhola, R. Strom, S. Bagchi, and J. Auerbach, "Exactly-once delivery in a content-based publish-subscribe system," in *Proc. of the 2002 International Conference on Dependable Systems and Networks*, (Washington, DC, USA), pp. 7–16, IEEE, 2002.

[81] H. Yang, M. Kim, K. Karenos, F. Ye, and H. Lei, "Message-oriented middleware with qos awareness," in *Service-Oriented Computing*, vol. 5900 of *Lecture Notes in Computer Science*, pp. 331–345, Berlin / Heidelberg, Germany: Springer Berlin / Heidelberg, 2009.

[82] M. Kim, K. Karenos, F. Ye, J. Reason, H. Lei, and K. Shagin, "Efficacy of techniques for responsiveness in a wide-area publish/-subscribe system," in *Proc. of the 11th International Middleware Conference Industrial track*, (Bangalore, India), pp. 40–45, ACM, 2010.

[83] S. Guo, K. Karenos, M. Kim, H. Lei, and J. Reason, "Delay-cognizant reliable delivery for publish/subscribe overlay networks," in *Proc. of the 31st International Conference on Distributed Computing Systems*, (Minneapolis, MN, USA), pp. 403–412, IEEE, 2011.

[84] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola, "Introducing reliability in content-based publish-subscribe through epidemic algorithms," in *Proc. of the 2nd international workshop on Distributed event-based systems*, (San Diego, CA, USA), pp. 1–8, ACM, 2003.

[85] P. Costa, M. Migliavacca, G. Picco, and G. Cugola, "Epidemic algorithms for reliable content-based publish-subscribe: an eval-

uation," in *Proc. of the 24th International Conference on Distributed Computing Systems*, (Tokyo, Japan), pp. 552–561, IEEE, 2004.

[86] T. Fountain, S. Tilak, P. Hubbard, P. Shin, and L. Freudinger, "The open source dataturbine initiative: Streaming data middleware for environmental observing systems," in *Proceedings of the 33rd International Symposium on Remote Sensing of Environment*, (Tucson, AZ, USA), pp. 1–6, ISRSE, 2009.

[87] R. Chand and P. Felber, "Xnet: a reliable content-based publish/subscribe system," in *Proc. of the 23rd IEEE International Symposium on Reliable Distributed Systems*, (Florianpolis, Brazil), pp. 264–273, IEEE, 2004.

[88] R. Kazemzadeh and H.-A. Jacobsen, "Reliable and highly available distributed publish/subscribe service," in *Proc. of the 28th IEEE International Symposium on Reliable Distributed Systems*, (Niagara Falls, NY, USA), pp. 41–50, IEEE, 2009.

[89] R. Kazemzadeh and H.-A. Jacobsen, "Partition-tolerant distributed publish/subscribe systems," in *Proc. of the 30th IEEE International Symposium on Reliable Distributed Systems*, (Madrid, Spain), pp. 101–110, IEEE, 2011.

[90] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *Computing Surveys, ACM*, vol. 36, no. 4, pp. 372–421, 2004.

[91] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[92] N. Carvalho, F. Araujo, and L. Rodrigues, "Scalable qos-based event routing in publish-subscribe systems," in *Proc. of the 4th IEEE International Symposium on Network Computing and Applications*, (Cambridge, MA, USA), pp. 101–108, IEEE, 2005.

[93] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," in *Proc. of the 19th International Conference on Distributed Computing Systems*, (Montreal, Canada), pp. 262–272, IEEE, 1999.

[94] Microsoft Corp., "Microsoft message queuing (msmq)." `http://msdn.microsoft.com/en-us/library/ms711472.aspx`. Web Page, last visited in Dec. 2013.

[95] R. Rajamani and N. Bhatt, "Oracle Database 11g: Advanced Queuing," tech. rep., Oracle Corporation, 2010. `http://www.oracle.com/technetwork/database/features/`

data-integration/oracle-aq-tech-wp11-2-191324.pdf. Online resource, last retrieved in Dec. 2013.

[96] N. Leavitt, "Will nosql databases live up to their promise?," *Computer, IEEE*, vol. 43, no. 2, pp. 12–14, 2010.

[97] A. Malekpour, A. Carzaniga, F. Pedone, and G. Toffetti Carughi, "End-to-end reliability for best-effort content-based publish/subscribe networks," in *Proc. of the 5th ACM international conference on Distributed Event-Based System*, (New York, NY, USA), pp. 207–218, ACM, 2011.

[98] PrismTech, "OpenSplice DDS." http://www.prismtech.com/opensplice/. Web Page, last visited in Dec. 2013.

[99] M. Ion, G. Russello, and B. Crispo, "Providing confidentiality in content-based publish/subscribe systems," in *Proc. of the International Conference on Security and Cryptography*, (Athens, Greece), pp. 1–6, Springer-Verlang, 2010.

[100] Java Community Process, "JSR 914: Java (TM) Message Service, Version 1.1," specification, 2002.

[101] F. Araújo and L. Rodrigues, "The IndiQoS Message Broker: an Instantiation Using RSVP," tech. rep., Department of Informatics, University of Lisbon, Lisbon, Portugal, 2002.

[102] R. Strom, G. Banavar, T. Chandra, and M. Kaplan, "Gryphon: An information flow based approach to message brokering," in *Proc. of the 9th International Symposium on Software Reliability Engineering*, (Paderborn, Germany), IEEE, 1998.

[103] R. Meier and V. Cahill, "Steam: event-based middleware for wireless ad hoc networks," in *Proc. of the 22nd International Conference on Distributed Computing Systems Workshops*, (Vienna, Austria), pp. 639–644, IEEE, 2002.

[104] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 335–371, 2004.

[105] D. Clark, R. Braden, and S. Shenker, "Integrated services in the internet architecture: an overview," RFC 1663, IETF, Network Working Group, 1994.

[106] J. Wroclawski, "The use of rsvp with ietf integrated services," RFC 2210, IETF, Network Working Group, 1997.

[107] K. An, T. Kuroda, A. Gokhale, S. Tambe, and A. Sorbini, "Model-driven generative framework for automated omg dds

performance testing in the cloud," in *Proc. of the 12th international conference on generative programming: concepts & experiences*, (Indianapolis, IN, USA), pp. 179–182, ACM Press, 2013.

[108] Real Time Innovations, "Connext DDS." `http://www.rti.com/products/dds/`. Web Page, last visited in Dec. 2013.

[109] M. Killijian, R. Cunningham, R. Meier, L. Mazare, and V. Cahill, "Towards group communication for mobile participants," in *Proc. of the ACM Workshop on Principles of Mobile Computing*, (Newport, RI, USA), pp. 75–82, ACM, 2001.

[110] R. Cunningham and V. Cahill, "Time bounded medium access control for ad hoc networks," in *Proc. of the 2nd International workshop on Principles of mobile computing*, (Tolouse, France), pp. 1–8, ACM, 2002.

[111] C.-F. Sørensen, M. Wu, T. Sivaharan, G. S. Blair, P. Okanda, A. Friday, and H. Duran-Limon, "A context-aware middleware for applications in mobile ad hoc environments," in *Proc. of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, (Toronto, Canada), pp. 107–110, ACM, 2004.

[112] P. Verissimo and A. Casimiro, "The timely computing base model and architecture," *Transactions on Computers, IEEE*, vol. 51, no. 8, pp. 916–930, 2002.

[113] A. Casimiro and P. Verissimo, "Using the timely computing base for dependable qos adaptation," in *Proc. of the 20th Symposium on Reliable Distributed Systems*, (New Orleans, LA, USA), pp. 208–217, IEEE, 2001.

[114] Y. Yoon, V. Muthusamy, and H.-A. Jacobsen, "Foundations for highly available content-based publish/subscribe overlays," in *Proc. of the 31st International Conference on Distributed Computing Systems*, (Minneapolis, MN, USA), pp. 800–811, IEEE, 2011.

[115] S. P. Mahambre and U. Bellur, "Reliable routing of event notifications over p2p overlay routing substrate in event based middleware," in *Proc. of the 21st IEEE International Parallel and Distributed Processing Symposium*, (Long Beach, CA, USA), pp. 1–8, IEEE, 2007.

[116] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[117] J. Luo, P. Eugster, and J.-P. Hubaux, "Route driven gossip: probabilistic reliable multicast in ad hoc networks," in *Proc. of the*

22nd Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 3, (San Francisco, CA, USA), pp. 2229–2239, IEEE, 2003.

[118] E. Pagani, "Providing reliable and fault tolerant broadcast delivery in mobile ad-hoc networks," *Mobile Networks and Applications*, vol. 4, no. 3, pp. 175–192, 1999.

[119] T. Pongthawornkamol, K. Nahrstedt, and G. Wang, "Probabilistic qos modeling for reliability/timeliness prediction in distributed content-based publish/subscribe systems over best-effort networks," in *Proc. of the 7th international conference on Autonomic computing*, (Washington, DC, USA), pp. 185–194, ACM, 2010.

[120] D. Laney, "3-d data management: Controlling data volume, velocity and variety," *META Group Inc. (now Gartner): Application Delivery Strategies*, 2001.

[121] S. Kaisler, F. Armour, J. A. Espinosa, and W. Money, "Big data: Issues and challenges moving forward," in *Proc. of the 46th Hawaii International Conference on System Sciences*, (Wailea, HI, USA), pp. 995–1004, IEEE, 2013.

[122] A. Katal, M. Wazid, and R. H. Goudar, "Big data: Issues, challenges, tools and good practices," in *Proc. of the 6th IEEE International Conference on Contemporary Computing (IC3)*, (Noida, India), pp. 404–409, IEEE, 2013.

[123] A. Osman, M. El-Refaey, and A. Elnaggar, "Towards real-time analytics in the cloud," in *Proc. of the 9th IEEE World Congress on Services*, (Santa Clara Marriot, CA, USA), pp. 428–435, IEEE, 2013.

[124] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *Transactions on Parallel and Distributed Systems, IEEE*, vol. 22, no. 6, pp. 931–945, 2011.

[125] I. Raicu and I. T. Foster, "Many-task computing for grids and supercomputers," in *Proc. of the 1st Workshop on Many-Task Computing on Grids and Supercomputers*, (Austin, TX, USA), pp. 1–11, IEEE, 2008.

[126] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Raid for data-intensive scalable computing," in *Proc. of the 4th Workshop on Petascale Data Storage*, (Portland, OR, USA), pp. 6–10, ACM Press, 2009.

[127] R. E. Bryant, "Data intensive scalable computing: Finding the right programming models," in *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing — Keynote Presentation*, (Chicago, IL, USA), ACM Press, 2010.

[128] D. Warneke, *Massively Parallel Data Processing on Infrastructure as a Service Platforms*. Ph.D. Thesis, Technishen Universität Berlin, 2011.

[129] M. Stonebraker, "The case for shared nothing," *Database Engineering Bulletin*, vol. 9, no. 1, pp. 4–9, 1986.

[130] H. Farhangi, "The path of the smart grid," *Power and Energy Magazine, IEEE*, vol. 8, no. 1, pp. 18–28, 2010.

[131] S. Sakr, A. Liu, D. M. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *Communications Surveys & Tutorials, IEEE*, vol. 13, no. 3, pp. 311–336, 2011.

[132] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, 2012.

[133] D. McCarthy and U. Dayal, "The architecture of an active database management system," *ACM Sigmod Record*, vol. 18, no. 2, pp. 215–224, 1989.

[134] D. C. Luckham, *The power of events: an introduction to complex event processing in distributed enterprise systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 2001.

[135] S. Geisler, "Data stream management systems," in *Data Exchange, Integration, and Streams* (P. G. Kolaitis, M. Lenzerini, and N. Schweikardt, eds.), vol. 5, ch. 5, pp. 275–304, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

[136] D. Turaga, H. Andrade, B. Gedik, C. Venkatramani, O. Verscheure, J. D. Harris, J. Cox, W. Szewczyk, and P. Jones, "Design principles for developing stream processing applications," *Softw. Pract. Exper.*, vol. 40, no. 12, pp. 1073–1104, 2010.

[137] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[138] E. Wong and R. H. Katz, "Distributing a database for parallelism," *ACM SIGMOD Rec.*, vol. 13, no. 4, p. 23, 1983.

[139] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[140] D. DeWitt, S. Ghandeharizadeh, D. Schneider, a. Bricker, H.-I. Hsiao, and R. Rasmussen, "The gamma database machine project," *Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, pp. 44–62, 1990.

[141] C. Ballinger and R. Fryer, "Born to be parallel: Why parallel origins give teradata an enduring performance edge," *Data Engineering Bullettin, IEEE*, vol. 20, no. 2, pp. 3–12, 1997.

[142] Microsoft Corp., "Sql server parallel data warehouse." `http://www.microsoft.com/en-us/sqlserver/solutions-technologies/data-warehousing/pdw.aspx`. Web Page, last visited in Dec. 2013.

[143] IBM Corp., "Ibm db2 enterprise server edition." `http://www-03.ibm.com/software/products/en/db2enterprise-server-edition/`. Web Page, last visited in Dec. 2013.

[144] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "Mapreduce and parallel dbmss: friends or foes," *Communications of the ACM*, vol. 53, no. 1, p. 64, 2010.

[145] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *Proc. of the 1st ACM symposium on Cloud computing*, (Indianapolis, IN, USA), p. 119, ACM Press, 2010.

[146] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *Proc. of the 27th International Conference on Data Engineering*, (Hannover, Germany), pp. 1151–1162, IEEE, 2011.

[147] S. Babu and J. Widom, "Continuous queries over data streams," *ACM SIGMOD Record*, vol. 30, no. 3, p. 109, 2001.

[148] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous queries over append-only databases," in *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, (San Diego, CA, USA), pp. 321–330, ACM Press, 1992.

[149] M. Sullivan, "Tribeca: A stream database manager for network traffic analysis," in *Proc. of the 22th International COnference on Very Large Data Bases*, (Bombay, India), p. 594, VLDB Endowment, 1996.

[150] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2005.

[151] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proc. of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, (Madison, WI, USA), pp. 1–16, ACM Press, 2002.

[152] L. Golab and M. T. Özsu, "Issues in data stream management," *ACM SIGMOD Record*, vol. 32, no. 2, pp. 5–14, 2003.

[153] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah, "Adaptive query processing: Technology in evolution," *Data Engineering Bullettin, IEEE*, vol. 23, no. 2, pp. 7–18, 2000.

[154] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *Proc. of the 21st ACM SIGMOD International Conference on Management of Data*, (Madison, WI, USA), p. 49, ACM Press, 2002.

[155] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, resource management, and approximation in a data stream management system," in *Proc. of thr 1st Biennial Conference on Innovative Data Systems Research*, (Asilomar, CA, USA), CIDR Conference, 2003.

[156] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," Technical Report 2004-20, Stanford InfoLab, 2004.

[157] 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 *Proc. of the 28th international conference on Very Large Data Bases*, (Hong Kong, China), VLDB Endowment, 2002.

[158] 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 International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[159] Y. Ahmad, N. Tatbul, W. Xing, Y. Xing, S. Zdonik, B. Berg, U. Çetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, and A. Rasin, "Distributed operation in the borealis stream processing engine," in *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, (Baltimore, MD, USA), pp. 882–884, ACM Press, 2005.

[160] H.-c. Yang, A. Dasdan, R. Hsiao, and D. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proc. of the 2007 ACM SIGMOD International Conference on Management of Data*, (Beijing, China), pp. 1029–1040, ACM Press, 2007.

[161] D. Alves, P. Bizarro, and P. Marques, "Flood: Elastic streaming mapreduce," in *Proc. of the 4th ACM International Conference on Distributed Event-Based Systems*, (Cambridge, UK), pp. 113–114, ACM Press, 2010.

[162] J. Horey, "A programming framework for integrating web-based spatiotemporal sensor data with mapreduce capabilities," in *Proc. of the 1st ACM SIGSPATIAL International Workshop on GeoStreaming*, (San Jose, CA, USA), pp. 51–58, ACM Press, 2010.

[163] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu, "Deduce: At the intersection of mapreduce and stream processing," in *Proc. of the 13th International Conference on Extending Database Technology*, (Lausanne, Switzerland), pp. 657–662, ACM Press, 2010.

[164] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. of the 10th IEEE International Conference on Data Mining Workshops*, (Sydney, Australia), pp. 170–177, IEEE, 2010.

[165] Apache Sofware Foundation, "Apache Samza." `http://samaza.incubator.apache.org`. Web Page, last visited in Dec. 2013.

[166] B. Gedik and H. Andrade, "A model-based framework for building extensible, high performance stream processing middleware and programming language for ibm infosphere streams," *Soft. Pract. Exper.*, vol. 42, no. 11, pp. 1363–1391, 2012.

[167] P. Bellavista, A. Corradi, and A. Reale, "The quasit model and framework for scalable data stream processing with quality of service," in *In Proc. of the 5th International Conference on Mobile Wireless Middleware, Operating Systems, and Applications*, vol. 65 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 92–107, Berlin, Germany: Springer Berlin Heidelberg, 2012.

[168] B. Gedik, H. Andrade, K.-l. Wu, P. S. Yu, and M. Doo, "Spade: the system s declarative stream processing engine," in *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data*, (Vancouver, Canada), pp. 1123–1134, ACM Press, 2008.

[169] B. Lohrmann, D. Warneke, and O. Kao, "Nephele streaming: stream processing under qos constraints at scale," *Cluster Computing*, pp. 1–18, July 2013.

[170] T. Lindholm and F. Yellin, *The Java virtual machine specification*, vol. 297. Addison-Wesley Reading, 1997.

[171] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, and J. Wolf, "Cola : Optimizing stream processing applications via graph partitioning," in *Proc. of the 10th ACM/IFIP/USENIX International Middleware Conference*, (Urbana Chamapign, IL, USA), pp. 308–327, 2009.

[172] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic load distribution in the borealis stream processor," in *Proc. of the 21st International Conference on Data Engineering*, (Tokyo, Japan), pp. 791–802, IEEE, 2005.

[173] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, and D. Rajan, "Soda : An optimizing scheduler for large-scale stream-based distributed computer systems," in *Proc. of the 9th ACM/IFIP/USENIX International Middleware Conference*, (Leuven, Belgium), pp. 306–325, Springer Berlin Heidelberg, 2008.

[174] Y. Xing, J.-h. Hwang, and S. Zdonik, "Providing resiliency to load variations in distributed stream processing," in *Proc. of the 32nd International Conference on Very Large Data Bases*, (Seoul, Korea), pp. 775–786, 2006.

[175] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Proc. of the 23rd IEEE International Parallel & Distributed Processing Symposium*, (Roma, Italy), pp. 1–12, IEEE, 2009.

[176] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proc. of the 35th SIGMOD international Conference on Management of Data*, (Providence, RI, USA), pp. 165–179, ACM Press, 2009.

[177] R. Avnur and J. M. Hellerstein, "Eddies: continuosly adaptive query processing," in *Proc. of the 2000 ACM SIGMOD international conference on Management of data*, vol. 29, (Dallas, Texas, USA), pp. 261–272, ACM Press, 2000.

[178] G. Chen, M. Li, and D. Kotz, "Data-centric middleware for context-aware pervasive computing," *Pervasive and Mob. Comput.*, vol. 4, no. 2, pp. 216–253, 2008.

[179] L. Kleinrock, *Theory, volume 1, Queueing systems*. Wiley-interscience, 1975.

[180] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," in *Proc.*

of the 39th International Conference on Very Large Data Bases, no. 11, (Riva del Garda, Trento, Italy), pp. 1–12, VLDB Endowment, 2013.

[181] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, and K.-L. Wu, "Job admission and resource allocation in distributed streaming systems," in *Proc. of the 14th Workshop on Job Scheduling Strategies for Parallel Processing* (E. Frachtenberg and U. Schwiegelshohn, eds.), (Rome, Italy), pp. 169–189, Springer Berlin / Heidelberg, 2009.

[182] V. Kumar, B. Cooper, and K. Schwan, "Distributed stream management using utility-driven self-adaptive middleware," in *Proc. of the 2nd International Conference on Autonomic Computing*, (Seattle, WA, USA), pp. 3–14, IEEE, 2005.

[183] M. Barabanov and V. Yodaiken, "Real-time linux," *Linux journal*, vol. 23, 1996.

[184] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proc. of the 7h ACM International Conference on Distributed Event-Based Systems*, (Arlington, TX, USA), pp. 1–12, ACM, 2013.

[185] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems," in *Proc. of the 26th IEEE International Conference on Distributed Computing Systems*, (Lisbon, Portugal), pp. 71–71, Ieee, 2006.

[186] N. Tatbul and S. Zdonik, "Dealing with overload in distributed stream processing systems," in *Proc. of the 22nd International Conference on Data Engineering Workshops*, (Atlanta, GA, USA), pp. 24–24, IEEE, 2006.

[187] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *Proc. of the 33rd international conference on Very large data bases*, (Vienna, Austria), pp. 159–170, VLDB Endowment, 2007.

[188] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation*, (San Diego, CA, USA), 2008.

[189] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Proc. of the 26th International Conference on Data Engineering Workshops*, (Long Beach, CA, USA), pp. 41–51, IEEE, 2010.

[190] I. Boutsis and V. Kalogeraki, "Radar: Adaptive rate allocation in distributed stream processing systems under bursty workloads," in *Proc. of the 31st IEEE Symposium on Reliable Distributed Systems*, (Irvine, CA, USA), pp. 285–290, Ieee, 2012.

[191] A. Avizienis, "Fault-tolerant systems: Concepts and terminology," *Transactions on Computers, IEEE*, vol. C-25, no. 12, pp. 1304–1312, 1976.

[192] J.-h. Hwang, M. Balazinska, A. Rasin, M. Stonebraker, and S. Zdonik, "A comparison of stream-oriented high-availability algorithms," Tech. Rep. 1, Brown University, Department of Computer Science, 2003.

[193] G. Jacques-Silva, B. Gedik, H. Andrade, and K.-L. Wu, "Language level checkpointing support for stream processing applications," in *Proc. of the 39th IEEE/IFIP International Conference on Dependable Systems & Networks*, (Lisbon, Portugal), pp. 145–154, IEEE, 2009.

[194] A. Brito, C. Fetzer, and P. Felber, "Minimizing latency in fault-tolerant distributed stream processing systems," in *Proc. of the 29th IEEE International Conference on Distributed Computing Systems*, (Montreal, Canada), pp. 173–182, IEEE, 2009.

[195] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proc. of the 21st International Conference on Data Engineering*, (Tokyo, Japan), pp. 779–790, IEEE, 2005.

[196] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," *ACM Trans. Database Syst.*, vol. 33, no. 1, pp. 1–44, 2008.

[197] A. Brito, C. Fetzer, and P. Felber, "Multithreading-enabled active replication for event stream processing operators," in *Proc. of the 28th IEEE International Symposium on Reliable Distributed Systems*, (Niagara Falls, NY, USA), pp. 22–31, IEEE, 2009.

[198] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proc. of the 2004 ACM SIGMOD international conference on Management of data*, (Paris, France), p. 827, ACM Press, 2004.

[199] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "Telegraphcq: Continuous dataflow

processing for an uncertain world," in *1st Conference on Innavative Data Systems Research*, (Asilomar, CA, USA), pp. 1–12, CIDR Conference, 2003.

[200] B. Lohrmann and O. Kao, "The nephele livescale toolkit : Real-time video stream processing at scale," in *Proc. of the 20th International Packet Video Workshop - Posters Track*, (San Jose, CA, USA), IEEE, 2013.

[201] S. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan, "The aurora and medusa projects," *Data Engineering Bullettin, IEEE*, vol. 26, no. 1, 2003.

[202] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: Operator scheduling for memory minimization in data stream systems," in *Proc. of the 2003 ACM SIGMOD international conference on on Management of data*, (San Diego, CA, USA), pp. 253–264, ACM Press, 2003.

[203] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proc. of of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, (Paris, France), p. 263, ACM Press, 2004.

[204] P. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *Transations on Knowledge and Data Engineering, IEEE*, vol. 15, no. 3, pp. 555–568, 2003.

[205] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *Proc. of the 20th International Conference on Data Engineering*, (Boston, MA, USA), pp. 350–361, IEEE Comput. Soc, 2004.

[206] S. Chandrasekaran, M. A. Shah, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, and F. Reiss, "Telegraphcq: Continuous dataflow processing," in *Proc. of the 22nd ACM international conference on on Management of data*, vol. 1, (San Diego, CA, USA), p. 668, ACM Press, 2003.

[207] M. Shah, J. Hellerstein, and M. Franklin, "Flux: an adaptive partitioning operator for continuous query systems," in *Proc. of the 19th International Conference on Data Engineering*, (Bangalore, India), pp. 25–36, IEEE, 2003.

[208] V. Raman, B. Raman, and J. Hellerstein, "Online dynamic reordering," *The VLDB Journal*, vol. 3, no. 9, pp. 247–260, 2000.

[209] B. Gedik, H. Andrade, and K.-L. Wu, "A code generation approach to optimizing high-performance distributed data stream

processing," in *Proc. of the 18th ACM conference on Information and knowledge management*, (Hong Kong, China), p. 847, ACM Press, 2009.

[210] E. I. Labs, "Stratosphere project." `http://stratosphere.eu/`. Web Page, last visited in Dec. 2013.

[211] B. Lohrmann, D. Warneke, and O. Kao, "Massively-parallel stream processing under qos constraints with nephele," in *Proc. of the 21st international symposium on High-Performance Parallel and Distributed Computing*, (Delft, The Netherlands), pp. 271–283, ACM Press, 2012.

[212] F. Reiss and J. Hellerstein, "Data triage: An adaptive architecture for load shedding in telegraphcq," in *Proc. of the 21st International Conference on Data Engineering*, (Tokyo, Japan), pp. 155–156, IEEE, 2005.

[213] A. Das, J. Gehrke, and M. Riedewald, "Semantic approximation of data stream joins," *Transactions on Knowledge and Data Engineering, IEEE*, vol. 17, no. 1, pp. 44–59, 2005.

[214] H. Feng, Z. Liu, C. H. Xia, and L. Zhang, "Load shedding and distributed resource control of stream processing networks," *Performance Evaluation*, vol. 64, no. 9-12, pp. 1102–1120, 2007.

[215] B. Gedik, K.-l. Wu, P. S. Yu, and L. Liu, "Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding," *Transactions on Knowledge and Data Engineering*, vol. 19, no. 10, pp. 1363–1380, 2007.

[216] A. Reale, "Quasit project." `http://lia.deis.unibo.it/research/quasit`. Web Page, last visited in Dec. 2013.

[217] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proc. of the 7h USENIX conference on Networked systems design and implementation*, (San Jose, CA, USA), pp. 1–15, USENIX Association, 2010.

[218] D. Logothetis and K. Yocum, "Ad-hoc data processing in the cloud," in *Proc. of the 34th International Conference on Very Large Data Bases*, vol. 1, (Auckland, New Zealand), pp. 1472–1475, VLDB Endowment, 2008.

[219] Apache Software Foundation, "Apache s4." `http://incubator.apache.org/s4/`. Web Page, last visited in Dec. 2013.

[220] OMG, "Interface definition language, version 3.5 beta 1," specification, Object Management Group, 2013.

[221] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," tech. rep., École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2004.

[222] S. Ghemawat, H. Gobioff, and S.-t. Leung, "The google file system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[223] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. of the ACM SIGCOMM 2008 conference on Data Communication*, (Seattle, WA, USA), pp. 63–74, ACM Press, 2008.

[224] The Open Source Initiative, "The bsd 3-clause license." `http://opensource.org/licenses/BSD-3-Clause`. Web Page, last visited in Dec. 2013.

[225] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," in *Proc. of the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (Vienna, Austria), p. 313, ACM, 2001.

[226] R. Hundt, "Loop recognition in c ++ / java / go / scala," in *Proc. of the 2011 Scala Days*, (Stanford, CA, USA), pp. 1–10, Typesafe Inc., 2011.

[227] Java Community Process, "JSR 255: Java (TM) Management Extensions (JMX), Version 2.0," specification, 2008.

[228] Java Community Process, "JSR 160: Java (TM) Management Extensions (JMX) Remote API, Version 1.4," specification, 2006.

[229] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[230] OMG, "Extensible and dynamic topic types for dds (ddsxtypes), version 1.0 beta 2," specification draft, Object Management Group, 2011.

[231] IEEE Austin Group, *Standard for Information Technology — Portable Operating System Interface (POSIX.1-2008, IEEE 1003.1-2008). Base Specification, Issue 7*, pp. 1523–1527. IEEE, 2008.

[232] G. A. Agha, *Actors: a model of concurrent computation in distributed systems*. Ph.D. Thesis, Massachusetts Institute of Technology – Department of Artificial Intelligence, 1985.

[233] 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.

[234] Typesafe Inc., "Akka actor framework." http://akka.io/. Web Page, last visited in Dec. 2013.

[235] D. Lea, "A Java fork/join framework," in *Proc. of the ACM Java Grande 2000 Conference*, (San Francisco, CA, USA), pp. 36–43, ACM Press, 2000.

[236] Netty Project Community, "Netty." http://netty.io/. Web Page, last visited in Dec. 2013.

[237] VMware Hyperic, "System information gatherer and reporter api." http://www.hyperic.com/products/sigar. Web Page, last visited in Dec. 2013.

[238] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, 2008.

[239] S. Sanfilippo, "Redis." http://redis.io/. Web Page, last visited in Dec. 2013.

[240] Apache Software Foundation, "Apache Maven Project." http://maven.apache.org/. Web Page, last visited in Dec. 2013.

[241] Apache Software Foundation, "Archiva: the build artifact repository manager.." http://archiva.apache.org/. Web Page, last visited in Dec. 2013.

[242] Intel Corporation, Itseez, Willow Garage , "Open source computer vision library." http://opencv.org. Web Page, last visited in Dec. 2013.

[243] Eastman Kodak, "True color kodak images (mirror)." http://r0k.us/graphics/kodak/. Web Page, last visited in Dec. 2013.

[244] B. Emir, M. Odersky, and J. Williams, "Matching objects with patterns," in *Proc. of the 21st European Conference on Object-Oriented Programming* (E. Ernst, ed.), (Berlin, Germany), pp. 273–298, Springer Berlin / Heidelberg, 2007.

[245] OMG, *Common Object Request Broker Architecture (CORBA), Version 3.2*, pp. 71–93. OMG, 2011.

[246] N. Sweet, M. Grotzke, R. Levenstein, and S. Ritchie, "Kryo — java serialization and cloning: fast, efficient, automatic." http://github.com/EsotericSoftware/kryo/. Web Page, last visited in Dec. 2013.

[247] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2009.

[248] Z. Cai, V. Kumar, B. Cooper, G. Eisenhauer, K. Schwan, and R. Strom, "Utility-driven proactive management of availability in enterprise-scale information flows," in *Proc. of the ACM/IFIP/USENIX 7h International Middleware Conference*, (Melbourne, Australia), Springer, 2006.

[249] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu, "A hybrid approach to high availability in stream processing systems," in *Proc. of the 30th IEEE International Conference on Distributed Computing Systems*, (Genoa, Italy), pp. 138–148, IEEE, 2010.

[250] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniacak, and M. Stonebraker, "Load shedding in a data stream manager," in *Proc. of the 29th International Conference on Very Large Data Bases*, (Berlin, Germany), pp. 309–320, The VLDB Endowment, 2003.

[251] Y. Zhou, B. Ooi, T. Kian-Lee, and W. Ji, "Efficient dynamic operator placement in a locally distributed continuous query system," in *Proc. of the 2006 Confederated international conference On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*, (Montpellier, France), Springer, 2006.

[252] A. Martin, C. Fetzer, and A. Brito, "Active replication at (almost) no cost," in *Proc. of the 2011 IEEE International Symposium on Reliable Distributed Systems*, (Madrid, Spain), pp. 21–30, IEEE, 2011.

[253] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer, "Scalable and low-latency data processing with stream mapreduce," in *Proc. of the 3rd International Conference on Cloud Computing Technology and Science*, (Athens, Greece), pp. 48–58, IEEE, 2011.

[254] L. Michel, A. Shvartsman, E. Sonderegger, and P. Hentenryck, "Optimal deployment of eventually-serializable data services," in *Proc. of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, (Paris, France), pp. 188–202, Springer Berlin Heidelberg, 2008.

[255] M. Lombardi and M. Milano, "Allocation and scheduling of conditional task graphs," *Artificial Intelligence*, vol. 174, no. 7–8, pp. 500 – 529, 2010.

[256] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: reliable stream computation in the cloud," in *Proc. of the ACM SIGOPS European Conference on Computer Systems*, (Prague, Czech Republic), ACM, 2013.

[257] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and unsupervised discretization of continuous features," in *Proc. of the 12th International Conference on Machine Learning*, ICML '95, (Tahoe City, CA, USA), pp. 194–202, Morgan Kaufmann, 1995.

[258] P. Shaw, "Using constraint programming and local search methods to solve vehicle routing problems," in *Proc. of the 4th International Conference on Principles and Practice of Constraint Programming*, pp. 417–431, Pisa, Italy: Springer, 1998.

[259] P. Laborie, "Ibm ilog cp optimizer for detailed scheduling illustrated on three problems," in *Proc. of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, (Pittsburgh, PA, USA), pp. 148–162, Springer Berlin Heidelberg, 2009.

[260] W. Cleveland and S. Devlin, "Locally weighted regression: an approach to regression analysis by local fitting," *Journal of the American Statistical Association*, vol. 83, no. 403, pp. 596–610, 1988.

[261] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proc. of the 1984 ACM SIGMOD international conference on Management of data*, (Boston, Massachusetts), pp. 47–57, ACM, 1984.

[262] A. Reale, P. Bellavista, A. Corradi, and M. Milano, "Evaluating cp techniques to plan dynamic resource provisioning in distributed stream processing — on-line appendix." `http://middleware.unibo.it/people/ar/laar-rap/`. Web Page, last visited in Dec. 2013.

[263] G. Jacques-Silva, B. Gedik, H. Andreade, K.-L. Wu, and R. Iyer, "Fault injection-based assessment of partial fault tolerance in stream processing applications," in *Proc. of the 5th International Conference on Distributed Event-based Systems*, (New York, NY, USA), pp. 231–242, ACM, 2011.

[264] E. Duarte-Melo and M. Liu, "Analysis of energy consumption and lifetime of heterogeneous wireless sensor networks," in *Proc. of the IEEE Global Telecommunications Conference*, (Taipei, Taiwan), pp. 21–25, IEEE, 2002.

ACKNOWLEDGEMENTS

I would like to thank Prof. Paolo Bellavista and Prof. Antonio Corradi for their precious advices and support during my Ph.D. studies. They helped me during every step of my work with unmatched competence and patience. A sincere thanks goes to Dr. Pól Mac Aonghusa and Dr. Spyros Kotoulas for giving me the chance to work with them at the IBM Dublin Research Lab: my stay in Ireland has been a unique experience for me, both professionally and personally. I also want to thank the reviewers of this thesis, Prof. Sumi Helal and Dr. Spyros Kotoulas, for their many suggestions that helped to greatly improve the quality of this work.

Thanks to all the people I met around the world: my dear friends Giuseppe and Andreas, and all the guys that made my stay in Dublin unforgettable: Piér, Simone, Stefania, Zubair, Marco, Spyros, Vanessa, Giusy, and Ernesto. Thanks to all the LIA — Carlo, Giuseppe, Federico, Daniela, and Stefano — and to all the other friends I met during my Ph.D. studies at the University of Bologna — Mario, Primiano, Ghedo, and Alessio — for all the memorable launch, afternoon, and (sometimes) dinner breaks. An important mention goes to the nameless woman that "cleans" our lab: thanks to you, nothing can kill me now. A special acknowledgment goes also to all my friends that shared with me many important moments during my time in Bologna; thanks to Mauro, Marco, Bif, Cristian, Fabio, Antonio, Demo, Andrea, Savio, Daniele, Salvo, Nicola, il Sardo, Françoise, Vale, Giuseppe, Gigi, Claudia, Daniele, Antonella, and Alessandra. I cannot forget to be grateful to all my closest friends from Campobasso — Valerio, Ianna, Jack and, of course, the *girls*, Rossana, Antonella, Valeria, and Nidia: I am sad that we keep spending less and less time together, but I sincerely hope that we will improve on that very soon. And thanks to Bologna, which I now consider my second home, for nourishing me and for providing the perfect background for many life-changing experiences.

Finally, the most important acknowledgements are for Minola and for my family. Minola for being always by my side with her sweet and endless support, encouragement, and love, even in the many occasions when any other mentally-sane person would have run away; my parents, Anna and Francesco, and my brothers, Antonello and Gianni, for being always present whenever I needed them and because they will always represent for me a safe and solid shelter.

fin.