UNIVERSITY OF ROME TOR VERGATA
DEPARTMENT OF CIVIL ENGINEERING AND COMPUTER SCIENCE ENGINEERING

Doctorate in Computer Science, Control and Geoinformation
Cycle XXX

QoS-aware Deployment and Adaptation of Data Stream Processing Applications in Geo-distributed Environments

Matteo Nardelli

Advisor
Valeria Cardellini

PhD Coordinator
Giuseppe F. Italiano

ACADEMIC YEAR 2016/17

APRIL 2018

WEBSITE
http://www.matteonardelli.it
E-MAIL
nardelli@ing.uniroma2.it

Acknowledgments

There are many people without whom this thesis would not have been possible, and I would like to thank all of them.

First, I would like to thank Prof. Valeria Cardellini, my advisor, for her guidance, inspiration, patience, and friendship. Valeria is a great advisor and I am very thankful for her constant support and help during these years. I am grateful to Prof. Francesco Lo Presti and Prof. Vincenzo Grassi, who, together with Valeria, encouraged me towards the PhD studies. They created a very pleasurable and genuine environment, that initiated and fostered my passion in doing research. Valeria, Francesco, and Vincenzo continuously helped me in improving my scientific, technical, and writing skills. All my achievements would not have been possible if not supported by them.

I am also thankful to Prof. Schahram Dustdar and Prof. Stefan Schulte for hosting my visit to TU Wien, Austria. They gave me the great opportunity to work in their research group. A special thank to Prof. Schulte, his technical advice has been valuable to me. Besides resulting in a fruitful collaboration, I enjoyed the possibility of meeting very interesting people.

I am grateful to all my colleagues with whom I shared and discussed ideas. In particular, I would like to thank Gabriele Russo Russo, Michael Borkowski, Christoph Hochreiner, Olena Skarlat, and Philipp Waibel.

Most importantly, I would like to thank my parents, my brother Vincenzo, and Fabiana for their encouragement and patience throughout these years. This thesis is dedicated to them.

Abstract

Exploiting on-the-fly computation, Data Stream Processing (DSP) applications are widely used to extract valuable information in a near real-time fashion, thus enabling the development of new pervasive services. Nevertheless, running DSP applications is challenging, because they are subject to a varying workload, require long provisioning time, and express strict QoS requirements. Moreover, since data sources are, in general, geographically distributed (e.g., in Internet-of-Things scenarios), recently we have also witnessed a paradigm shift with the deployment and execution of DSP applications over distributed Cloud and Fog computing resources. This computing environment allows to move applications closer to the data sources and consumers, thus reducing their expected response time. This diffused computing infrastructure also promises to reduce the stress upon the Internet infrastructure, by reducing the movement of large data sets, and to improve the scalability of DSP systems, by better exploiting the ever increasing amount of resources at the network periphery. Nevertheless, such geo-distributed infrastructures pose new challenges to deal with, including the heterogeneity of computing and networking resources and the non-negligible network latencies.

In this thesis, we study the challenges of executing DSP applications over geo-distributed environments. A DSP application is represented as a directed graph, with data sources, operators, and final consumers as vertices, and streams as edges. For the execution, we need to solve the operator placement problem, which consists in determining the computing nodes that should host and execute each operator of a DSP application. Moreover, when the DSP application should efficiently process huge amount of incoming load, we also need to solve the operator replication problem. It consists in determining the number of parallel instances (or replicas) for the operators, so that each instance can process a subset of the incoming data flow in parallel (i.e., data parallelism). We first present a taxonomy to classify the existing deployment and runtime adaptation approaches for DSP systems. Starting from the literature review, we provide several contributions to the initial deployment and runtime adaptation of DSP applications over heterogeneous resources. We propose a general unified formulation of

the operator placement problem, which also provides a benchmark against which placement heuristics can be evaluated. Then, we present several new heuristics for efficiently solving the operator placement problem in a feasible amount of time. Differently from existing research efforts, the heuristics are also evaluated in terms of quality of the computed placement solution. Afterwards, we study the operator replication problem and propose a general formulation that jointly optimizes the replication and placement of the DSP operators, while considering the application QoS requirements. To preserve the application performance in highly changing execution environments, the deployment of DSP applications should be accordingly reconfigured at runtime. Hence, we formulate the elastic replication and placement problem, which determines whether the application should be more conveniently redeployed by explicitly considering the adaptation costs (i.e., state migration, application downtime). To efficiently deal with runtime adaptation over large scale and geo-distributed infrastructures, we present a hierarchical approach for the autonomous control of elastic DSP applications. To evaluate the devised deployment and adaptation solutions on real DSP systems, we design and implement two extensions of Apache Storm, namely Distributed Storm and Elastic Storm, which are available as open source projects.

Our thesis work demonstrates the importance of models, prototypes, and empirical experiments to deeply understand and overcome the challenges of running DSP applications over geo-distributed environments. Indeed, a suitable representation of applications, computing and network resources, which explicitly considers their relevant QoS attributes, allows to improve the application performance, while efficiently managing geo-distributed infrastructures.

Contents

Chapter 1

Introduction

The amount of daily generated data is quickly growing with a never seen before pace. This tendency is supported and accelerated by many recent developments, which are leading us towards a data-centric society. In the last few years, the reduced cost of sensing devices and smartphones, together with the (almost) ubiquitous Internet connectivity, has fostered the wide diffusion of new pervasive services and devices (e.g., social networks, smart watches, wearable devices). As a result, nowadays we are witnessing to the development of the so-called Internet of Things (IoT), where devices (or things) exploit interconnectivity to cooperate and expose high value services. As the world becomes more connected, there is a deluge of data coming from disseminated sensors in the form of continuous streams.

The emerging data-intensive panorama presents old and new challenges. In 2001, Gartner introduced the term *Big Data* to collectively identify data with specific properties[1], which stress the traditional processing systems and, ultimately, require to redesign the processing paradigms. These properties are well depicted by the IBM definition of Big Data[2], which spans along the 4Vs dimensions: volume, variety, velocity, and veracity. Volume refers to challenges of storing and processing large data set, which might not fit within the memory of a single machine. These data sets can easily reach the size of petabyte (PB) or exabyte (EB), respectively 10^6 GB and 10^9 GB, thus requiring the adoption of distributed and efficient processing solutions. Variety refers to the need of managing and integrating structured, unstructured, and multimedia data. Velocity refers to the high production and consumption rate, which requires to quickly analyze streams of data. Veracity is a property that emphasizes the uncertainty of the data, meaning that the presence of noisy data can compromise the quality of extracted information.

[1] `http://www.gartner.com/newsroom/id/1731916`

[2] `https://www.ibm.com/analytics/us/en/big-data/`

Accessing to raw data is not interesting by itself. Instead, we are interested in extracting high level information (e.g., aggregated insights), which enables the development of new intelligent and pervasive services aiming to improve our everyday life. Decision-making processes can benefit from data-driven evidences[3]. Similarly, service providers can exploit real-time data to optimize their offerings and services. For example, a public transportation company can exploit information regarding user mobility and presence of events within the city, to plain (and adjust) in real-time the number of buses within the city as well as their route. The ACM DEBS 2015 Grand Challange presented a similar idea: considering taxis moving in New York City, the challenge asks to find the top most frequent routes and to identify the most profitable area for taxi drivers in real-time [99]. We can readily observe that these kind of applications have a great economical value, because they can take advantage of informed decisions (i.e., data driven decisions) to develop products and services and optimize their provisioning[4]. The real-world use-case from Dublin city has shown how an urban monitoring system can identify traffic congestions and (pro-actively) change traffic light priorities and speed limits, so to reduce ripple effects [12]. Many other examples range from manufacturing [155], health-care [187], energy and utilities management [98], to financial markets [155].

Two different processing modes are commonly used to elaborate data over distributed computing resources: batch processing and stream processing. The *batch processing* approaches store all the data, usually on a distributed file system, and then operate on them on the basis of different programming models, among which the well-known MapReduce [49]. *Stream processing* approaches operate on data on-the-fly, i.e., without storing them, so they can produce results in a near real-time fashion. The idea behind DSP applications is not new, but results from almost 50 years [143] of evolution in terms of methodologies, architectures, and technologies. Nevertheless, the advent of the Big Data era and the diffusion of the Cloud computing paradigm have renewed the interest in DSP applications [78]. We can identify two stream processing models: the *once-at-a-time* model, where each tuple is sent individually, and the *micro-batched* model, where some tuples are grouped before being sent [141]. Thanks to their properties,

[3] In recent years, many different research communities have investigated how to extract insightful information from raw data, i.e., how to perform data analytics. This lead to the definition of a new form of science, the so called *data science*. Data science concerns using models, hypotheses, and empirical data to analyze phenomena, so to estimates an outcome with a certain degree of uncertainty. Differently from the classical science, which is theory-driven and leads to incontrovertible facts, data science shows data-driven evidences that are deduced from the available data. This means that data science does not necessarily lead to incontrovertible facts, but may help to narrow the field of investigation.

[4] `https://www.economist.com/news/leaders/21721656-data-economy-demands-new-approach-antitrust-rules-worlds-most-valuable-resource`

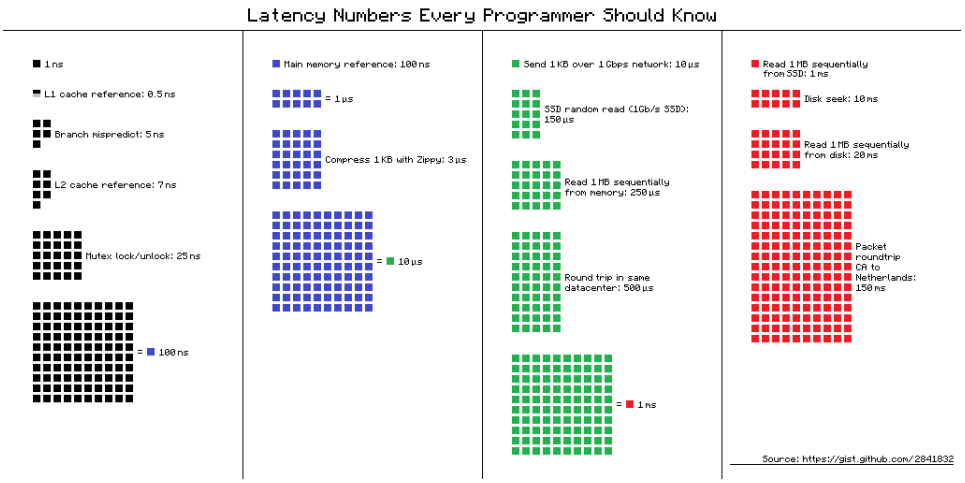


Figure 1.1: Latency comparison in performing different local and network operations. Source: `https://gist.github.com/jboner/2841832`

Data Stream Processing (DSP) applications are widely used to collect and process unbounded streams of data, aiming to extract valuable information in a near real-time fashion. Keeping data only on main memory, DSP applications exploit the technological advantage of this kind of memory, which is characterized by a reduced access latency with respect to the persistence on stable storage (see Figure 1.1). On the other hand, developing algorithms that process data on the fly (i.e., operate on-line) might not be trivial [4, 148]. DSP applications can be used to solve different problems, ranging from simple monitoring and pattern detection, to more complex data processing tasks that, e.g., perform reasoning, learning, or in general distill new information in an automated fashion. As a consequence, they enable the development of new intelligent and pervasive services that can improve our everyday life in several domains (e.g., health-care, energy management, logistic, and transportation). For example, a DSP application can be used to perform sentiment analysis on multiple tweet streams from Twitter, to create user profile (as Yahoo! does), or to trace trends evolution (as Google does).

Since DSP applications produce results whose utility decreases quickly over time, they are required to satisfy *Quality of Service* (QoS) requirements, which describe the expected behavior with respect to a set of performance metrics (e.g., availability, response time, throughput, cost). QoS requirements play a key role in nowadays DSP applications that require an efficient utilization of the computing infrastructure. This requirement is especially true when DSP applications run over heterogeneous and/or geographically distributed infrastructures, collect data from multiple data sources, and forward results to multiple consumers. The enforcement of

QoS requirements translates to the presence of mechanisms that, by controlling the allocation of computing resources to different part of an application, allow to meet the performance objectives at runtime. Using the available resources adaptively and according to the actual needs of DSP applications allows to scale to larger and larger scenarios while efficiently managing resources.

The emerging scenario pushes DSP systems to a whole new performance level. Strict quality requirements, great volumes of data, and high production rate exacerbate the need of an efficient usage of the underlying infrastructure. To date, DSP applications are typically deployed on large-scale and centralized (Cloud) data centers that are often distant from data sources. However, as data increase in size, pushing them towards the Internet core could cause excessive stress for the network infrastructure and also introduce excessive delays. A solution to improve scalability and reduce network latency lies in taking advantage of the ever increasing presence of near-edge/Fog Cloud computing resources [22, 23]. The recent idea behind Fog computing is to make the Cloud descending to the network edges by moving computational resources from few large data centers, located in the network core, to diffused cloudlets or micro Clouds, that serve as a second-class data center with soft state [176]. The use of a diffused infrastructure allows to decentralize the execution of DSP applications, by moving the computation to the edges of the network, close to data sources and information consumers [217]. Nevertheless, this infrastructure poses new challenges that include network and system heterogeneity, geographic distribution as well as non-negligible network latencies among distinct nodes that process parts of a DSP application. This latter issue can have a strong negative impact for DSP applications running in latency-sensitive domains (e.g., [24]).

Recently, Cisco provided an analysis of Cisco's global IP traffic and an estimation of the near future trends [37], which show the importance of scalability and decentralization for the near future Internet-based systems. Cisco reports that, in 2016, the annual run rate for global IP traffic was 1.2 ZB per year (i.e., 1.2×10^{12} GB per year, which corresponds to about 3.3×10^9 GB per day). Moreover, by 2021 PCs will account for only 25% of traffic, whereas smartphones will generate about the 33% of total IP traffic. The emerging of a strongly decentralized environment is also supported by the prevision that, by 2021, Content Delivery Networks will carry 71% of Internet traffic and 35% of end-user Internet traffic will be delivered within a metro network (i.e., by resources closer to the edge of the network). Overall, these results show the ever increasing importance of exploiting near-edge resources to alleviate the Internet stress and reduce communication latencies. This trend eventually calls for the development of efficient approaches to decentralize the execution of distributed applications.

In this context, a very interesting problem regards the efficient exploitation of the available computing resources, with the aim of executing DSP applications with high performance. This problem is challenging not only because of the infrastructure characteristics (i.e., computing and network resources are heterogeneous and possibly geo-distributed), but also because of the characteristics of DSP applications. Indeed, the latter expose a computing demand that is not usually known in advance and, most importantly, can change at runtime (DSP applications are usually long running and subject to varying incoming workloads). Therefore, we investigate the application deployment problem, which consists in determining the computing resources that should host and execute each part a DSP application, aiming to optimize the application QoS attributes. Due to dynamic and long-running nature of DSP applications, the computed initial application deployment might not guarantee satisfying performance throughout the whole application lifetime. As a consequence, we often need to reconfigure (or adjust) the application deployment at runtime, so to preserve performance. Basically, reconfigurations can be regarded as small adjustment steps that update and correct the application deployment so to keep adherence with QoS requirements. Differently from the initial deployment, runtime reconfigurations usually do not propose a fresh new deployment, because this can prohibitively penalize the application performance (through high reconfiguration costs). Hence, even though computing the initial application deployment and its runtime adaptation seem to be two similar problems, they are slightly different in terms of assumptions and optimization goals.

1.1 Data Stream Processing

In this section, we introduce the basic features of DSP applications. We introduce the concepts of stream, tuples, and the main properties of DSP operators, including the presence of internal state and the concept of windowing.

A DSP application can be represented at different levels of abstraction; we distinguish between a user-defined *abstract model* and an *execution model*, which is used to run the application.

The DSP *abstract model* defines the streams and their characteristics, along with the type, role, and granularity of the stream processing elements. At this level, the DSP application is represented as a directed graph, with data sources, operators, and final consumers as vertices, and streams as edges. Note that, even though application graphs can be cyclic, most systems only support directed acyclic graphs (DAG). The application graph is also known as topology or stream graph. A stream is an unbounded

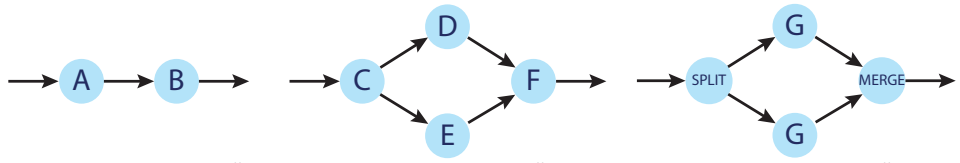


Figure 1.2: Form of parallelism: pipeline, task parallelism, data parallelism.

sequence of data items (e.g., tuple, event, file chunk)[5]. In a DSP application, a data item is the unit of communication exchanged between DSP operators. An operator is a self-contained processing element that carries out a specific operation (e.g., filtering, aggregation, merging) or something more complex (e.g., POS-tagging). Each operator can be seen as a black-box processing element, that continuously receives incoming streams, applies a transformation, and generates new outgoing streams. A data source generates one or more streams that feed the DSP application; differently from operators, data sources have no incoming streams. A final consumer (or sink) is a final receiver of the application streams; it can push data on a message queue, forward information to a persistent storage, or — more generally — trigger the execution of external services. Differently from operators, sinks have no outgoing streams.

Running on a distributed infrastructure, the design of DSP applications tries to conveniently exploit different forms of parallelism among the operators, namely pipeline, task parallelism, and data parallelism [86]. Pipeline parallelism is the concurrent execution of a producer with a consumer of the stream. Task parallelism is the concurrent execution of different operators that do not constitute a pipeline on the same or different data. Data parallelism is the concurrent execution of multiple instances of the same operator on different portions of same data. Figure 1.2 shows the different forms of parallelism.

Moreover, we distinguish between *stateless* and *stateful* operator whether the operator computes the output data using only the incoming data or also some internal state information, respectively. A stateless operator processes data items independently one another, independently of prior history, or even from their order of arrival. As such, this kind of operators are easily parallelized and can be restarted upon failures without the need of any recovery procedure. A stateful operator maintains information across different data items to detect complex patterns or compute aggregate statistics (e.g., item summarization, detection of fraudulent financial transactions). In general, stateful operators are not easily parallelized, except when

[5]Even though a stream is possibly infinite sequence of items, operators queues only contain a finite sequence of in-flight data items at any point in time.

the state is partitioned. A partitioned state maintains independent state information for each partition, which is usually identified by a key; different partitions do not interact one another. In case of partitioned state, multiple parallel instances of an operator can be executed, where each instance is tied to one or more state partitions, and can operate on them independently from the other operator instances.

Windowed operators are a special kind of operators that buffers the incoming data items (so to retain previously received data) before running the operator logic on the buffered data items. This kind of operators can be used to realize lightweight stateful operators, whose state is completely defined by the buffered data items [7]. A windowing is characterized by two parameters: size and slide period. The window size determines the amount of data that should be buffered before triggering the operator execution; two main policies have be adopted so far: time-based, which fills the buffer for a specific amount of time, and count-based, which fills the buffer with a specific number of data items. The slide period determines how the window moves forward, and it can rely on time-based or count-based policies. By combining accordingly the window size and the slide period, different kinds of windowing patterns can be realized. The most common ones are: sliding windows, tumbling windows, and session windows. Sliding windows are characterized by a static window size and a slide period having value different from the window size. Tumbling windows (or fixed windows) are a special case of sliding window, where the slide period is equal to the window size (i.e., they span the incoming stream without overlapping). Session windows aim to capture some period of activity over a subset of the data; as such, they are characterized by a dynamic size, which depends on a timeout gap: any events that occur within a span of time less than the timeout are grouped together as a session. Furthermore, windows can be either aligned or unaligned, if they are applied across all data or only on specific subsets of the data (e.g., per key) for the given window of time, respectively [7].

The DSP *execution model* is obtained from the abstract model by replacing each operator with the current number of operator replicas, that is operator instances each of which processes a subset of the incoming data flow (i.e., data parallelism). By partitioning the stream over multiple replicas, running on one or more computing nodes, the load of each replica is reduced, which, in turn, yields lower operator (and overall application) latency. Since the load can vary over time, the number of replicas in the execution model can change accordingly as to optimize some non functional requirements, e.g., response time.

1.2 Deployment and Runtime Adaptation Problem

For the execution, a DSP application needs to be deployed on computing resources, which will execute the application operators. At runtime, the allocation of resources can be conveniently adapted so to address changes of the execution environment.

Deployment problem. Determining the application deployment requires to perform several tasks, among which solving the operator replication problem and the operator placement problem. The *operator replication* problem consists in determining the number of instances that should be executed for each operator of a DSP application; in other words, it computes the parallelism degree of DSP operators. The *operator placement problem* (or scheduling problem) consists in determining, within a set of available distributed computing resources, the ones that should host and execute each operator (or operator replicas) of a DSP application. Although many feasible replication and placement solutions can be found, we are usually interested in determining the one that optimizes the application QoS attributes.

Runtime adaptation. Since DSP applications are usually long-running, the operators can experience changing working conditions (e.g., fluctuations of the incoming workload, variations in the execution environment). To preserve the application performance within acceptable bounds, their deployment should be adapted at runtime, through migration and scaling operations. A *migration* moves an operator replica to another computing resource, so to balance resource utilization or consolidate replicas on fewer computing nodes. A *scaling* operation changes the replication degree of an operator: a scale-out decision increases the number of replicas when the operator needs more computing resources, whereas a scale-in decreases the number of replicas when the operator under-uses its resources. If the operator is stateless, a reconfiguration involves only starting, stopping, adding, or removing fresh operator replicas. Conversely, if the operator is stateful, a reconfiguration involves also the migration of the operator state or its redistribution among the operator replicas. The presence of an internal state increases the complexity of performing reconfigurations, which should preserve the application integrity. Therefore, the main drawback of reconfigurations is that they cause application downtime and, if applied too often, they can negatively impact the application performance.

1.3 Research Methodology

We are interested in deploying DSP applications over an infrastructure that comprises heterogeneous and (possibly geographically) distributed computing resources. This task is challenging, because it requires:

- to understand the needs of DSP applications and the challenges of running them over geo-distributed environments;
- to identify relevant QoS attributes of DSP applications, computing and network resources;
- to develop suitable benchmarks that enable to identify the best deployment and adaptation strategies;
- to develop efficient and effective resolution approaches that can operate in DSP systems at runtime;
- to design a reference system architecture that can conveniently work in geo-distributed environments.

In this thesis, we analyze all of these key issues adopting an engineering-oriented perspective, which goes through the following steps: problem identification and modeling, design of resolution approaches, prototype development, and evaluation. Moreover, to rule the complexity of the problem under investigation, we use a stepwise approach. First, we investigate the initial deployment of DSP applications; then, we focus on their runtime adaptation. Indeed, by referring to different stages of the applications life-cycle, the initial deployment and its runtime adaptation are characterized by different requirements.

In general, for both these problems (i.e., initial deployment and runtime adaptation), we proceed as follows. First, we thoroughly analyze the existing literature, so to find suitable approaches that can operate in the environments under investigation. Moreover, by exploring the latest research contributions, we can identify and understand the most relevant features of DSP applications and of their execution. Second, we identify a suitable representation of DSP applications and system resources (both computing and networking resources) with the aim of modeling the operator placement problem, the operator replication problem, and the runtime adaptation problem. Leveraging on the optimization theory, we formalize them as Integer Linear Programming (ILP) problems. They provide a general framework against which existing (centralized and decentralized) heuristics can be evaluated in order to assess their quality. Third, we design and develop efficient heuristics to solve the initial deployment and runtime adaptation problems. Differently from the existing solutions, the deployment and adaptation models help to steer the heuristics design and to quantitatively analyze their performance, in terms of resolution time and quality of the computed solution. Finally, we validate and evaluate our policies and architectures, relying on empirical experiments run on system prototypes. Therefore, we have developed new DSP frameworks that can integrate centralized and decentralized QoS-aware placement policies as well as support the elastic deployment adaptation. We do not usually rely on simulations, because DSP systems might show high complexity, so they

are difficult to be simulated; furthermore, there is a lack of DSP workload characterization studies.

1.4 Thesis Contribution

The goal of this thesis is to analyze, model, and develop solutions for the initial deployment and runtime adaptation of DSP applications over distributed infrastructures. The latter accounts for distributed Cloud and Fog computing environments, where computing resources are interconnected with not negligible delays. This emerging environment is appealing, because it promises to improve scalability and reduce latency of modern DSP systems by moving the computation to the edges of the network, close to data sources and consumers. Nevertheless, if we want to select one among the existing deployment policies that can efficiently operate in this newly emerging environment, we have to face an interesting challenge: most existing solutions rely on different modeling assumption and optimization objectives, thus resulting in a wide pool of heuristics or best-effort approaches not easily comparable one another. We advance the state of the art by providing general formulations of the deployment problem and runtime adaptation problem, by developing new heuristics, and by designing new open-source DSP frameworks.

The main contributions of this thesis are the following.

- We analyze and classify the existing deployment and adaptation policies for DSP systems. To this end, we develop a general taxonomy that summarizes the main design choices of current solutions along the *five Ws one H* concept.
- We design two DSP frameworks and develop them as extensions of Apache Storm, an open-source DSP system. The first extension is Distributed Storm; it supports the execution of decentralized, QoS-aware and self-adaptive placement policies on a distributed and heterogeneous infrastructure. The second extension is Elastic Storm; it introduces in Storm two mechanisms that support the runtime adaptation of DSP applications (i.e., elasticity and stateful migration).
- We propose Optimal DSP Placement (for short, ODP), a unified general formulation of the operator placement problem for distributed and networked DSP applications. ODP takes into account QoS attributes of applications and computing and network resources. Moreover, ODP also represents a general benchmark against which existing heuristics can be compared.
- We design several new heuristics for solving the operator placement problem and assess their quality relying on ODP. We propose two main

classes of heuristics: the model-based approaches, which execute ODP on a conveniently selected solution subspace, and the model free approaches, which customize the well-known meta-heuristics greedy first-fit, local search, and tabu search.

- We propose Optimal DSP Replication and Placement (for short, ODRP), an extension of ODP, that — differently from existing solutions — jointly optimizes the replication and placement of the DSP operators, while maximizing the QoS attributes of the application. Similarly to ODP, ODRP provides a general framework for QoS optimization and evaluation of existing heuristics.
- We investigate the challenges of adapting at runtime the application deployment and show the importance of adaptation costs. Specifically, we present Elastic DSP Replication and Placement (for short, EDRP), an extension of ODRP, that adapts at runtime the replication and placement of DSP operators while explicitly accounting for reconfiguration costs. As such, EDRP can conveniently determine whether adaptation actions should be enacted.
- We propose a hierarchical distributed control approach for adapting at runtime the deployment of elastic DSP applications. Differently from existing solutions (which are either centralized or decentralized), our proposal revolves around a two layered approach with separation of concerns and time scale between layers. Here, higher-level MAPE components oversee the overall application deployment and control subordinate MAPE components, which are in charge of adapting single DSP operators.

1.5 Thesis Outline

The remainder of this thesis is organized as follows.

In Chapter 2, we analyze and classify the main deployment policies that have been proposed so far to compute the initial application deployment and to determine its runtime adaptation. We devise a general taxonomy leveraging on the *five Ws one H concept*. The taxonomy aims to easily present the key design choices proposed by the existing research efforts in literature.

In Chapter 3, we start to investigate the operator placement problem. Specifically, we present the design and implementation of Distributed Storm, our extension of Apache Storm with distributed placement management capabilities, and evaluate two decentralized placement policies. In this thesis, we will resort on Distributed Storm so to prototype the designed deployment policies and assess their behavior on a real testbed. As fur-

ther outcome of the chapter, we show the difficulty of developing fully decentralized placement heuristics. Indeed, we show that, for complex DSP applications, fully decentralized heuristics might suffer from lack of coordination, thus leading to too frequent reconfigurations which can be detrimental for the application performance.

In Chapter 4, we present our general unified formulation of the Optimal DSP Placement problem. Besides presenting the system model and the problem formulation, we conduct a thoroughly evaluation aimed to investigate the flexibility and scalability of the proposed model. Moreover, we demonstrate how our formulation represents a general framework by comparing the performance achieved by some centralized and decentralized placement heuristics.

In Chapter 5, we present several heuristics for efficiently solving the operator placement problem. Some of the heuristics rely on ODP, aiming to compute high quality placement solutions. Other heuristics customize to the problem at hand several well-known approaches (i.e., greedy first-fit, local search, and tabu search). Using ODP as benchmark, we show that the heuristics find different trade-offs between computation time and quality of the computed solution.

In Chapter 6, we extend ODP to jointly optimize the operator replication and placement problem, thus obtaining ODRP. We show the benefits of a joint optimization of operators replication and placement on the application performance and how ODRP can contextually optimize several QoS metrics. Thanks to its flexibility, ODRP represents a general benchmark against which to compare existing heuristics.

From Chapter 7 onwards, we consider the runtime adaptation problem and, first of all, we present Elastic Storm, our extension of Storm that supports elasticity and stateful migrations. This chapter is devoted to the description of the extended Storm architecture and of the designed elasticity and migration policies. We also show how elasticity can improve resource utilization and enable to properly process the incoming varying workload.

By analyzing the current literature, we realize that current solutions for runtime adaptation often neglect reconfiguration costs, propose best-effort solutions, and hardly optimize the deployment in geo-distributed environment. Therefore, in Chapter 8, we investigate the impact of adaptation costs on the application performance. We propose a new framework for the optimization of QoS metrics, named EDRP, that considers the impact of adaptation cost, while determining whether to apply a reconfiguration. Our results show the importance of taking into account reconfiguration costs and how EDRP can find interesting trade-offs among performance metrics.

In Chapter 9, we start to explore a hierarchical distributed control approach for managing elastic DSP applications. This approach represents

a new class of heuristics with respect to those existing in literature and promises to address scalability and stability of existing solutions. Our heuristic uses a two layered architecture: a lower level per-operator component issues reconfiguration requests, whereas a higher level per-application component coordinates reconfigurations exploiting a global view on the application performance. The experimental results are promising and encourage the development of new hierarchical heuristics.

Finally, in Chapter 10, we summarize results and contributions of this work and indicate some directions for future research on the basis of the results presented in this thesis.

1.6 Publications

Part of the work in this thesis has previously appeared in international journal (J) and conference proceedings (C) papers. For our contributions, we received the *best poster award*, at the ACM DEBS 2015 conference, and the *honorable mention paper award*, at the ACM DEBS 2016 conference.

Chapter 3: Distributed Storm

C1. V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, "Distributed QoS-aware scheduling in Storm", In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS 2015)*, pp. 344–347, Oslo, Norway, July 2015. doi: 10.1145/2675743.2776766. **Best Poster Award**

C2. V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, "On QoS-aware scheduling of data stream applications over fog computing infrastructures", *5th International Workshop on Management of Cloud and Smart City Systems 2015 (MOCS 2015)*, in conjunction with ISCC 2015. Published in *Proceedings of the 2015 IEEE Symposium on Computers and Communication (ISCC 2015)*, pp. 271–276, Larnaca, Cyprus, July 2015. doi: 10.1109/ISCC.2015.7405527.

C3. M. Nardelli, "A Framework for Data Stream Applications in a Distributed Cloud", In *Proceedings of the 8th ZEUS Workshop 2016 (ZEUS 2016)*, pp. 56–63, Vienna, Austria, January 2016. CEUR-WS.org/Vol-1562, online: `http://ceur-ws.org/Vol-1562/paper9.pdf`.

Chapter 4: Optimal Operator Placement

C4. V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, "Optimal Operator Placement for Distributed Stream Processing Applications", In *Proceedings of the 10th ACM International Conference on Distributed and Event-*

Based Systems (DEBS 2016), pp. 69–80, Irvine, CA, USA, June 2016. doi: 10.1145/2933267.2933312. **Honorable Mention Paper Award**

Chapter 6: Optimal Operator Replication and Placement

J1. V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, "Optimal Operator Replication and Placement for Distributed Stream Processing Systems", *ACM SIGMETRICS Performance Evaluation Review*, Vol. 44, No. 4, pp. 11–22, May 2017. doi: 10.1145/3092819.3092823.

C5. V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, "Joint Operator Replication and Placement Optimization for Distributed Streaming Applications", In *Proceedings of InfQ 2016 - New Frontiers in Quantitative Methods in Informatics (in conjunction with ValueTools 2016)*, pp. 263–270, Taormina, Italy, October 2016. doi: 10.4108/eai.25-10-2016.2266628.

Chapter 7: Elastic Storm

C6. V. Cardellini, M. Nardelli, D. Luzi, "Elastic Stateful Stream Processing in Storm", In *Proceedings of the 2016 International Conference on High Performance Computing & Simulation (HPCS 2016)*, pp. 583–590, Innsbruck, Austria, July 2016. doi: 10.1109/HPCSim.2016.7568388. **Outstanding Paper Award Nomination**

Chapter 8: Elastic Operator Placement and Replication

J2. V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo, "Optimal Operator Deployment and Replication for Elastic Distributed Data Stream Processing", *Concurrency and Computation: Practice and Experience*, 25 pages, accepted for publication in 2017. doi: 10.1002/cpe.4334.

Chapter 9: Hierarchical Autonomous Control for Elastic DSP

C7. V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo, "Towards Hierarchical Autonomous Control for Elastic Data Stream Processing in the Fog", *International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP 2017)*, in conjunction with Euro-Par 2017. Published in *Euro-Par 2017: Parallel Processing Workshops*, Lecture Notes in Computer Science Vol. 10659, pp. 106–117, 2018. doi: 10.1007/978-3-319-75178-8_9.

Other publications. In the following, a list of partially related publications that the author published during his doctorate activity, including international journal (J) and conference proceedings (C) papers and book chapters (BC).

J3. M. Nardelli, S. Nastic, S. Dustdar, M. Villari, R. Ranjan, "Osmotic Flow: Osmotic Computing + IoT Workflow", *IEEE Cloud Computing*, vol. 4, no. 2, pp. 68–75, 2017. doi: 10.1109/MCC.2017.22.

J4. O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, P. Leitner, "Optimized IoT Service Placement in the Fog", *Service Oriented Computing and Applications*, vol. 11, no. 4, pp. 427–443, 2017. doi: 10.1007/s11761-017-0219-8.

J5. M. Borkowski, W. Fdhila, M. Nardelli, S. Rinderle-Ma, S. Schulte, "Event-based Failure Prediction in Distributed Business Processes", *Information Systems*, 2017. doi: 10.1016/j.is.2017.12.005.

BC1. V. Cardellini, T. G. Grbac, A. Kassler, A. Marotta, P. Kathiravelu, F. Lo Presti, M. Nardelli, L. Veiga, "Integrating SDN and NFV with QoS-aware Service Composition", *Autonomous Control for a Reliable Internet of Services: Methods, Models, Approaches, Techniques, Algorithms and Tools*, Lecture Notes in Computer Science, Springer. Accepted for publication in 2017.

BC2. V. Cardellini, T. G. Grbac, M. Nardelli, N. Tanković, H. L. Truong, "QoS-based Elasticity for Service Chains in Distributed Edge Cloud Environments", *Autonomous Control for a Reliable Internet of Services: Methods, Models, Approaches, Techniques, Algorithms and Tools*, Lecture Notes in Computer Science, Springer. Accepted for publication in 2017.

C8. M. Nardelli, "QoS-aware Deployment of Data Streaming Applications over Distributed Infrastructures", In *Proceedings of the 39th IEEE International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2016)*, pp. 736–741, Opatija, Croatia, May 2016. doi: 10.1109/MIPRO.2016.7522238.

C9. G. Marciani, M. Piu, M. Porretta, M. Nardelli, V. Cardellini, "Grand Challenge: Real-time Analysis of Social Networks Leveraging the Flink Framework", In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems (DEBS 2016)*, pp. 386–389, Irvine, CA, USA, June 2016. doi: 10.1145/2933267.2933517.

C10. M. Nardelli, "Doctoral Symposium: Placement of Distributed Stream Processing over Heterogeneous Infrastructures", In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems (DEBS 2016)*, pp. 422–425, Irvine, CA, USA, June 2016. doi: 10.1145/2933267.2933432.

C11. M. Nardelli, "Elastic Allocation of Docker Containers in Cloud Environments", In *Proceedings of the 9th ZEUS Workshop 2017 (ZEUS 2017)*, pp. 59–66, Lugano, Switzerland, February 2017. CEUR-WS.org/Vol-1826, online: `http://ceur-ws.org/Vol-1826/paper10.pdf`.

C12. M. Nardelli, C. Hochreiner, S. Schulte, "Elastic Provisioning of Virtual Machines for Container Deployment", *1st International Workshop on Autonomous Control for Performance and Reliability Trade-offs in Internet of Services (ACPROSS 2017)*, in conjunction with ACM/SPEC ICPE 2017. Published in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE '17 Companion)*, pp. 5–10, L'Aquila, Italy, April 2017. doi: 10.1145/3053600.3053602.

C13. O. Skarlat, M. Nardelli, S. Schulte, S. Dustdar, "Towards QoS-aware Fog Service Placement", In *Proceedings of the 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC 2017)*, pp. 89–96, Madrid, Spain, May 2017. doi: 10.1109/ICFEC.2017.12.

C14. G. Marciani, M. Porretta, M. Nardelli, G. F. Italiano, "A Data Streaming Approach to Link Mining in Criminal Networks", In *Proceedings of the 5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW 2017)*, pp. 138–143, Prague, Czech Republic, August 2017. doi: 10.1109/FiCloudW.2017.88.

C15. V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo, "Auto-scaling in Data Stream Processing Applications: A Model Based Reinforcement Learning Approach", To appear in *Proceedings of InfQ 2017 - New Frontiers in Quantitative Methods in Informatics*, in conjunction with ValueTools 2017, Venice, Italy, December 2017.

Chapter 2

DSP Application Deployment

> The existing solutions for determining the application deployment differ in terms of optimization goals, modeling assumptions, and resolution approach. We first develop a taxonomy based on the *five Ws one H* concept and then we use it to classify some of the existing research results and identify their main design choices.

The deployment of a DSP application over a large-scale distributed computing infrastructure is a key task that has a great impact on the application performance. In literature, we can find many different approaches, each tailored for optimizing a specific objective, such as the application throughput, its response time, or the efficient utilization of the available computing resources. As shown in [29], determining the deployment of DSP applications is an NP-hard problem, therefore several heuristics have been proposed in literature. Nevertheless, they often rely on different modeling assumptions and optimization goals [117].

In this chapter, we devise a taxonomy based on the *five Ws one H* concept to describe the research results and development efforts related to the deployment of DSP applications, including the approaches devoted to the runtime deployment adaptation.

2.1 Taxonomy: A General Perspective

To summarize and organize the most relevant approaches that deal with the deployment of DSP applications, we devise a general taxonomy, built on the six questions: *why*, *what*, *who*, *when*, *where*, and *how*. These questions help to identify the key features of the existing deployment solutions, enabling to quickly pinpoint their commonalities and differences.

Focusing on the deployment of DSP applications and on their runtime

adaptation, we conjugate the six questions as follows.

- **Why**: This question investigates the motivations behind the design and development of new approaches to the application deployment. Indeed, each deployment solution has a specific and well-defined optimization goal, which should be pursued or met by conveniently assigning computing resources to DSP operators (or replicas). The *why* question pinpoints the utility function and the quantitative metrics that the deployment solution aims to optimize or satisfy. For example, many existing solutions minimize the application response time or maximize the application throughput, whereas, in case of runtime reconfigurations, they try to minimize the adaptation costs.
- **What**: This question identifies what entities can be managed by the deployment policy, in terms of span and granularity of control. The span of control identifies how many applications the policy controls at the same time; specifically, the policy can oversee single applications independently one another or multiple applications that compete or not for obtaining resources. The granularity of control identifies the main entity that is controlled by the policy so to achieve the optimization goals; the managed entities can be single tuples, operators, or the whole application topology.
- **Who**: This question aims to identify the authority in charge of computing the application deployment and its runtime adaptation. In a large scale DSP system, the control authority represents a key element, because it influences the scalability, the optimality of the deployment solution, and the management overhead.
- **When**: This question investigates the temporal aspects of the deployment solutions, exploring when the deployment optimizations and actions should are executed. As usually happens in complex systems, timing plays an important role to efficiently perform management operations. Besides dealing with the initial deployment, since DSP applications are long-running, many solutions are specifically tailored for addressing challenges and needs of runtime adaptation. In fact, the latter enables to preserve high performance in face of changing working conditions.
- **Where**: This question is concerned with the characterization of the computing infrastructure managed by the deployment solution. Each computing infrastructure exposes different features and challenges, including resource heterogeneity and distribution, which ultimately impact on the application performance (e.g., response time). Moreover, when the scheduler plans a deployment reconfiguration at runtime, the *where* question identifies the architectural level where the adaptation actions should take place (i.e., application-level, infrastructure-lever).

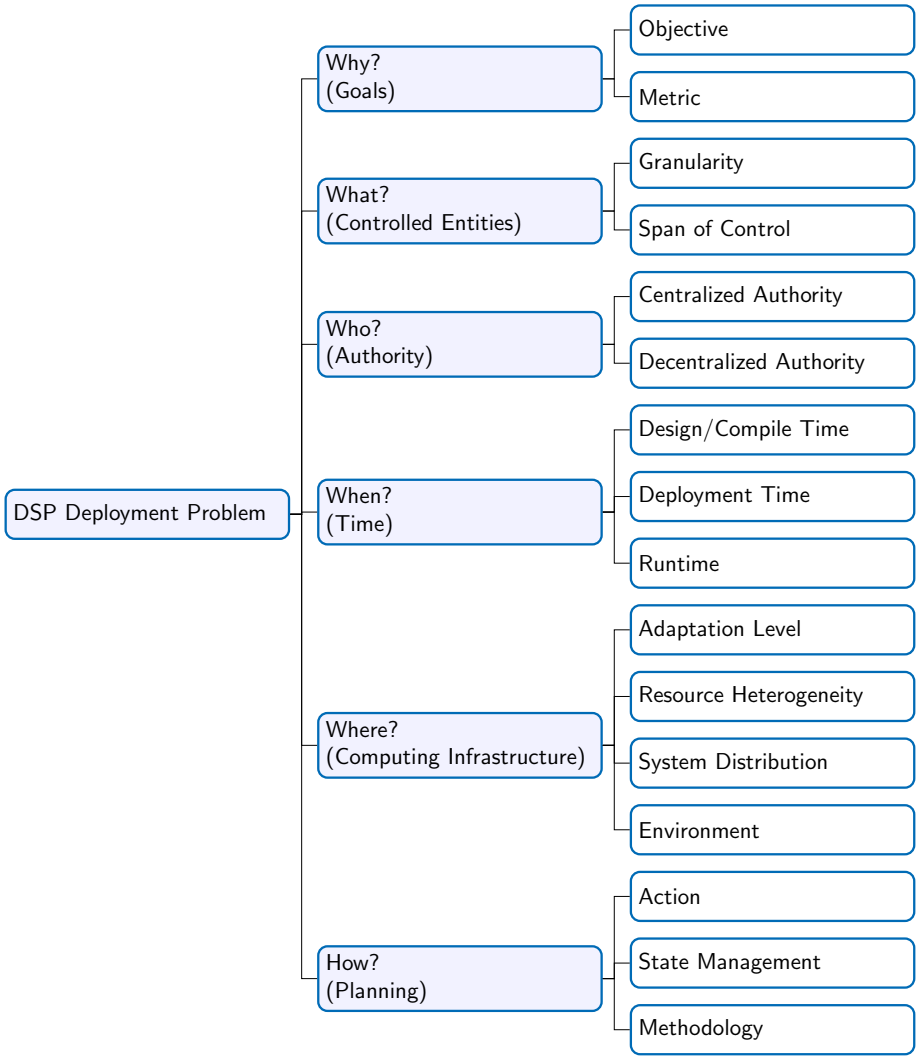


Figure 2.1: Taxonomy of deployment solutions for DSP applications. A high-level perspective.

- **How**: The *how* question identifies the set of actions and methodologies that can be used for determining or changing the application deployment. Examples of actions are the operator placement, its replication, or the application topology transformation. A methodology allows to determine when and how the specific actions should be performed in order to achieve the deployment objectives. Examples of methodologies are the greedy heuristics, which are commonly used for determining the operators placement, or the threshold based policies, which are commonly used for changing the operator replication degree at runtime. Furthermore, this dimension explores how the runtime adaptation solutions deal with the operator internal state (i.e., whether they preserve the operator state during reconfigurations).

Under the guidance of the six questions, we develop a taxonomy that classifies the most relevant solutions proposed so far in literature. Figure 2.1 presents a high-level overview of the taxonomy. To better present the key design choices of the different solutions, we separate them into two main sections that cover two different dimensions along the *when* question.

The efficient deployment of a DSP application may be a complex task that requires to apply different operations (e.g., topology optimization, operator replication, operator placement, efficient load distribution). Each of these operations can be executed in different stages of the application lifetime; we identify the following stages.

- **Design and compile time.** The design and compile time cover all the operations performed before the application submission to the DSP system for execution. At design time, the user creates the DSP application. At compile time, the application is compiled and packed in an archive, executable, or package, which is (possible optimized and) ready for submission. The operations performed at design and compile time are concerned with static transformations of the application topology. Although applied once during the application lifetime, these transformations enable to perform more sophisticated operations at runtime. Examples of design and compile time optimizations are the operator reordering, which moves more selective operators upstream to filter data early, and the operator separation, which splits operators into smaller computational steps to better exploit pipelining[1]. These operations can be manually executed by the application designer or can be automatically applied by a pre-processing engine that optimizes the application topology before its submission to a DSP system for execution.
- **Deployment time.** The deployment time spans between the application submission to the DSP system and the beginning of its execution. In this stage, the DSP system needs to compute the initial replication

[1] An extensive catalog of optimizations for DSP applications can be found in [86].

degree of each application operator and define the placement of each operator replica on the available computing resources. At this stage, the system can only rely on a-priori knowledge about the application requirements, because execution data are not yet available.

- **Runtime.** At runtime, DSP applications can be subject to changing working conditions, e.g., in terms of incoming workload and resource availability. Since a varying workload changes the demand of computing resources, to retain acceptable application performance, the application deployment should be smoothly reconfigured at runtime. Nonetheless, together with long term benefits, adapting the application deployment also introduces some adaptation costs, usually expressed in terms of downtime, that penalize the application performance in the short period. Such adaptation costs take into account the time needed to relocate the operator state, so to preserve the application integrity. Because of these costs, reconfigurations cannot be applied too frequently. Therefore, a key challenge is to wisely select the most profitable adaptation actions to enact.

In the following sections, we present the solutions proposed so far in literature and classify their key design choices relying on the proposed taxonomy. First, in Section 2.2, we describe the challenges and approaches that deal with the initial deployment of DSP applications. In this section, we cover the solutions that can be adopted at design, compile, and deployment time. Then, in Section 2.3, we describe the approaches that enable to meet desirable application performance even in face of dynamic working conditions. In this section, we cover the solutions that adapt the application deployment at runtime.

2.2 Initial Deployment

The taxonomy presented in Section 2.1 is here extended to discuss how the different questions have been addressed by the existing literature that investigates the initial deployment of DSP applications. Figure 2.2 shows the extended taxonomy, tailored for the initial deployment.

Relying on this taxonomy, we classify and describe the most relevant works proposed so far in literature. To this end, we augment the key nodes of the taxonomy with a special label, wrapped in squared bracket (e.g., [S]). Table 2.1 positions the most relevant works with respect to our taxonomy, whereas Table 2.2 reports, for sake of clarity, the list of all the labels used to classify the existing solutions.

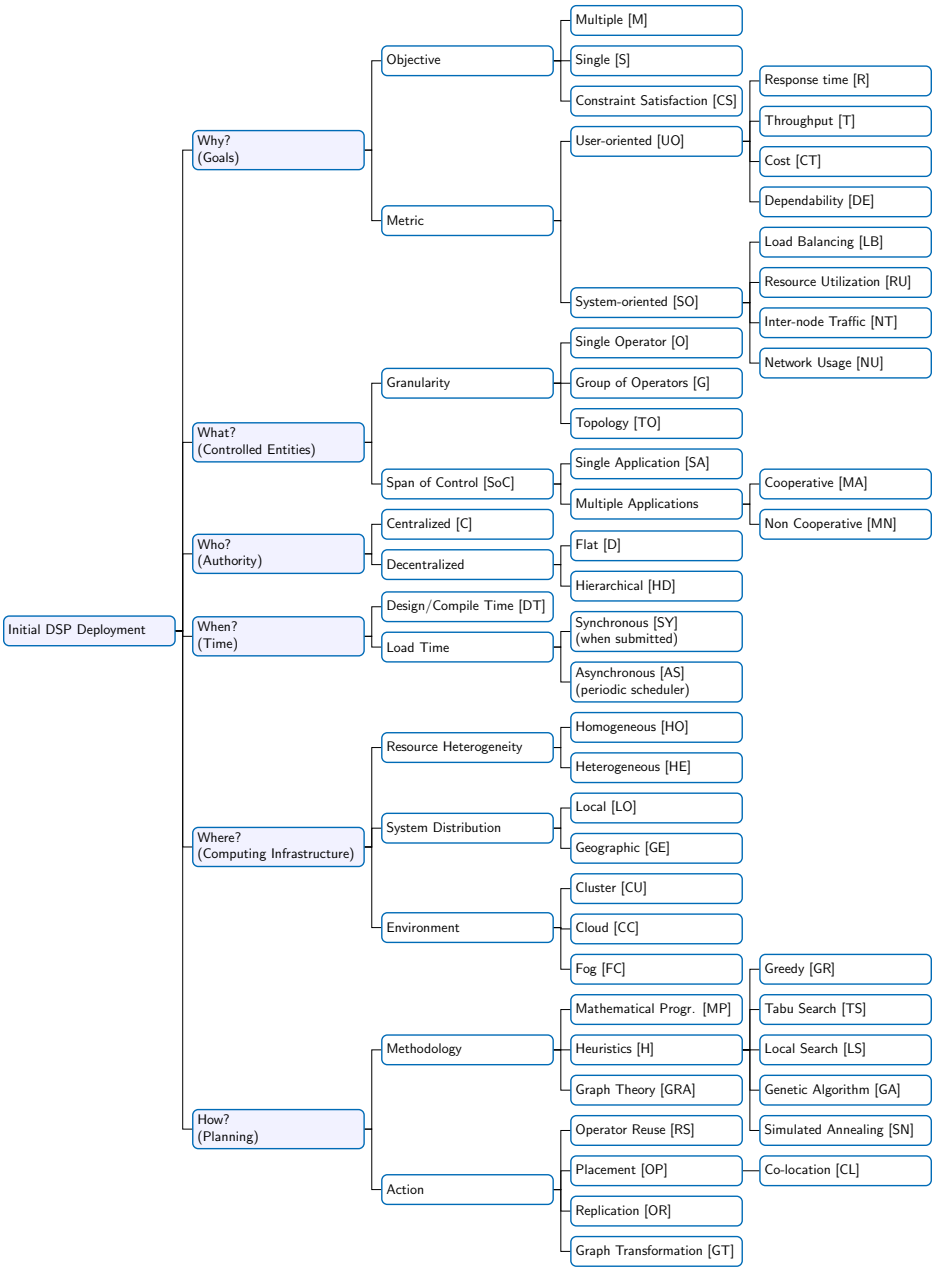


Figure 2.2: Taxonomy of the initial deployment for DSP applications.

Table 2.1: Solutions for the initial deployment of DSP applications.

	WHY		**WHAT**		**WHO**	**WHEN**	**WHERE**			**HOW**	
	Obj.	Metric	SoC	Gra.	Aut.	Time	Heter.	Distr.	Env.	Act.	Method.
Ahmad et al. [5]	S	NT, NU	SA	O	D	SY	HO	GE	CU	OP	H
Arkian et al. [11]	S	SO	SA	O	C	SY	HE	GE	FC	OP	MP
Backman et al. [14]	S	R	SA	O	C, D	SY	HE	GE	CC	OP, OR	GR
Eidenbenz et al. [53]	S	UO	SA	O	C	SY	HO	(GE)	-	OP	MP
Eskandari et al. [54]	S	NT	SA	O	C	AS	HO	LO	CU	OP	GRA
Fischer et al. [57]	S, CS	NT, LB	SA	O	C	SY	HO	(GE)	CU	OP, CL	GRA
Fischer et al. [58]	S, CS	NT, LB	SA	O	C	AS	HO	(GE)	CU	OP, CL	GRA
Gedik et al. [64]	S	T	-	G	C	SY	HE	LO	CU	OR	GR
Ghaderi et al. [66]	S	NT	MA	G	C	SY	HO	(GE)	CU	OP	H
Gu et al. [67]	S	NT	SA	O	C	SY	HE	GE	CU	OP	MP
Gu et al. [68]	S	LB	MA	O	D	SY	HE	GE	CU	OP	LS
Gu et al. [69]	S	UO	SA	O	C	SY	HE	GE	CU	OP	H
Hwang et al. [96]	CS	R	MA	O	C	SY	HE	GE	CU	OR	H
Khandekar et al. [107]	S, CS	NT, LB	SA	G	C	SY	HE	(GE)	CU	CL	GRA
Lakshmanan et al. [118]	S	R	SA	O	C	AS	HO	LO	CU	OR	LS
Li et al. [121]	S	DE	SA	O	C	SY	HO	LO	CU	OP	H
Li et al. [122]	S	NU	MA	O	C	AS	HE	GE	CU	OP, RS	GRA
Li et al. [124]	S	R	SA	O	C	AS	HE	GE	CU	OP	GR, LS
Peng et al. [156]	S	T	SA	O	C	AS	HE	LO	CU	OP	H
Pollner et al [159]	S	LB	SA	G	C	DT, SY	HE	LO	CU	OP, OR	H
Pollner et al [160]	S	UO	SA	O, TO	C	SY	HE	GE	-	OP, GT	H
Rychly et al [174]	S	R	SA	O	C	AS	HE	LO	CU	OP	GR
Sajjad et al. [175]	S	UO	SA	O	C	AS	HO	GE	FC	OP	H
Schneider et al. [180, 181]	S	RU	-	G	C	DT, SY	HO	LO	CU	OR	GR
Schultz-Møller et al. [183]	S	NT, LB	MA	O, TO	C	SY	HO	LO	CU	OP, RS, GT, CL	GR
Shukla et al. [185]	CS	T	MA	O	C	SY	HO	LO	CC	OR	H
Smirnov et al. [186]	S	T	SA	O	C	AS	HO	LO	CU	OP	GA

(continued on next page)

(continued from previous page)

	WHY		WHAT		WHO	WHEN	WHERE			HOW	
	Obj.	Metric	SoC	Gra.	Aut.	Time	Heter.	Distr.	Env.	Act.	Method.
Stanoi et al. [188]	S	T	SA	O, TO	C	SY	HE	LO	CU	OP, GT	MP, GR, TS, SN
Thoma et al. [196]	CS	-	SA	O	C	SY	HO	(GE)	CU	OP	MP
Tian et al. [197]	S	SO	MA	O	D	AS	HO	LO	CU	OP	H
Xing et al. [211]	CS[2]	LB	SA	O	C	SY	HE	LO	CU	OP	H
Zhu et al. [226]	S	R	SA	O	C	SY	HE	GE	CU	OP	GRA

[2]This solution identifies an operator placement plan that is *resilient* to unpredictable incoming load variations.

Table 2.2: Acronyms used within our taxonomies.

Acronym	Meaning	Question
A	Application-level adaptation	Where
AC	Optimized metric: Adaptation cost	Why
AS	Solution executed asynchronously at load time	When
C	Centralized authority	Who
CC	Cloud computing environment	Where
CL	Planning action: Operators co-location	How
CS	Constraint satisfaction problem	Why
CT	Optimized metric: cost	Why
CTR	Planning methodology: control theory	How
CU	Cluster computing environment	Where
D	Flat decentralized authority	Who
DBS	Planning action: Dynamic batch sizing	How
DT	Solution working at design or compile time	When
E	Event-based optimization	When
F	State management supported: stateful operators	How
FC	Fog computing environment	Where
FTM	Planning action: Fault-tolerance mechanisms	How
G	Granularity of control: Groups of operators	What
GA	Planning methodology: Genetic algorithm	How
GE	Geographically distributed system	Where
GR	Planning methodology: Greedy approach	How
GAM	Planning methodology: Game theory	How
GRA	Planning methodology: Graph theory	How
GT	Planning action: Graph transformation	How
H	Planning methodology: Heuristic	How
HD	Hierarchical decentralized authority	Who
HE	Heterogeneous computing resources	Where
HO	Homogeneous computing resources	Where
I	Infrastructure-level adaptation	Where
L	State management not supported: stateless operators	How
LB	Optimized metric: Load balancing	Why
LD	Planning action: Load distribution	How
LO	Locally distributed system	Where
LS	Planning methodology: Local search	How
M	Multi-objective optimization function	Why
MA	Control of multiple cooperative applications	What
ML	Planning methodology: Machine learning	How
MN	Control of multiple non-cooperative applications	What
MP	Planning methodology: Mathematical programming	How
NT	Optimized metric: Inter-node traffic	Why
NU	Optimized metric: Network usage	Why
O	Granularity of control: Single operator	What
OP	Planning action: Operator placement	How
OR	Planning action: Operator replication	How
OS	Planning action: Operator scaling	How

(continued on next page)

(continued from previous page)

Acronym	Meaning	Question
P	Periodic optimization	When
PR	Time mode: proactive	When
QT	Planning methodology: Queuing theory	How
R	Optimized metric: Response time	Why
RE	Time mode: reactive	When
RS	Planning action: Operator reuse	How
RU	Optimized metric: Resource utilization	Why
S	Single-objective optimization function	Why
SA	Control of a single application	What
SH	Planning action: Load shedding	How
SN	Planning methodology: Simulated annealing	How
SO	System-oriented metric optimized	Why
SoC	Span of Control	What
SY	Solution executed synchronously at load time	When
T	Optimized metric: Throughput	Why
TB	Planning methodology: Threshold-based	How
TO	Granularity of control: Topology	What
TP	Planning action: Tuple scheduling	How
TS	Planning methodology: Tabu search	How
TU	Granularity of control: Single tuple or batch of tuples	What
UO	User-oriented metric optimized	Why

2.2.1 Why: Deployment Goals

The DSP placement problem [117] has been widely investigated in literature under different modeling assumptions and optimization goals, e.g., [29, 53, 196]. We distinguish three main optimization objectives: constraint satisfaction, single-objective, and multi-objective.

In a *constraint satisfaction* problem (referred as [CS] in Table 2.1), the scheduler identifies a deployment solution among all the feasible ones that satisfy requirements expressed in terms of computing resources, application performance, or deployment constraints (e.g., co-location, isolation). Relying on a-priori knowledge of the application performance, Shukla and Simmhan [185] determine the initial deployment (in terms of operators replication degree) that allows to process incoming workload characterized by specific data rate. Conversely, Thoma et al. [196] propose an approach to restrict the set of feasible deployment by improving the expressiveness of constraints, including co-location, upstream/downstream, isolation, and tag-based constraints.

Since not all feasible assignments result in desirable application performance, most of the existing solutions optimize (i.e., minimize or maximize) a single-objective function (referred as [S] in Table 2.1) or a multiple objective function (referred as [M]). A single-objective function can target

a specific, well-defined QoS metric (e.g., response time, throughput, network usage) or a generic cost function. Among the QoS metrics, we distinguish between user-oriented (referred as [UO]) and system-oriented ones (referred as [SO]). We detail more on these metrics later in this section.

A multi-objective function could be aimed at optimizing a diversity of possibly conflicting QoS attributes or optimization goals. For example, a deployment solution might seek to minimize the application response time, while maximizing the application availability[3]. In such cases, the deployment policy has to deal with a multi-objective optimization problem, where the importance of each optimization goal (e.g., application response time, availability) strongly depends on the utilization scenario. A well-established approach to easily deal with a multi-objective problem is to transform it into a single-objective problem, using the Simple Additive Weighting (SAW) technique [218].

User-oriented QoS metrics. A *user-oriented metric* models a specific aspect of the application performance, as can be perceived by the user; examples of this kind of metric are the application response time, throughput, cost, and dependability[4] [119]. DSP applications are usually employed in latency-sensitive domains (e.g., [12, 98, 99, 149]), therefore many solutions try to minimize the application *response time* [R], which results from the operators deployment. For a DSP application, with data flowing from several sources to several destinations, there is no unique definition of response time. Hence, most of the solutions [14, 118, 124, 174, 226] recur on the minimization of the critical path average delay. The critical path of a DSP application is defined as the set of nodes and edges, forming a path from a data source to a sink, for which the sum of the operator computational latency and network delays is maximal.

Other approaches try to maximize the application *throughput* [T]; in the state of the art we can find two different definitions of throughput, namely output and input throughput. The *output throughput* is defined as the number of tuples that can be produced by the DSP application in a given unit of time. This is the most largely optimized QoS metric [64, 156, 185, 186]. The *input throughput* is defined as the number of input tuples that can be processed by the system in a given unit of time [188]. The existing approaches that maximize throughput are usually designed to work in a locally distributed cluster environment, where network latencies can be neglected, because they introduce a lightweight overhead on each transmitted tuple.

System-oriented QoS metrics. A *system-oriented metric* aims to quantify a specific aspect of the system, following the service provider's standpoint

[3]We define as application availability the probability that each application operator is up and running.

[4]We consider dependability as a general concept that includes many attributes such as reliability, availability, safety, integrity, and so on [13].

who wants to efficiently use the available resources. Besides the classic metrics that account for the utilization of computing resources (e.g., load unbalance, CPU utilization, and number of active machines), in geographically distributed systems network-related metrics are of key importance, because transmitting data through the network is slower than transmitting them between processes running on the same node. As a consequence, network transmissions can easily became the system bottleneck. Examples of this kind of metric are the inter-node traffic and network usage.

Improving *resource utilization* [RU] aims to more efficiently use the infrastructure so to process higher workloads. To this end, most of the existing solutions run multiple replicas of the same DSP operator so to exploit data parallelism, as in [159, 180, 181]. Following this approach, multiple threads run on each CPU core in parallel. Another solution explicitly tailored to improve resource utilization is to discard useless data as close as possible to the application data sources. Following this idea, Schultz-Møller et al. [183] have proposed a system that automatically rewrites the application topology, by combining and reordering its operators, so to obtain an equivalent application with lower CPU cost. Several policies for the initial deployment of applications also optimize the (initial) *load balancing* [LB] among computing nodes. This metric investigates a stronger property than resource utilization, representing whether the incoming load is evenly distributed among resources. Often, load balancing is seen as a secondary goal by the placement algorithm: indeed, the latter uses it in combination with the optimization of other QoS metrics, such as the minimization of inter-node traffic as in [58, 183, 57].

The *inter-node traffic* [NT] is the overall amount of data exchanged per time unit between operators placed on different nodes. In a distributed environment (both locally [LO] and geographically [GE] distributed), inter-node communication has a higher impact on response time rather than intra-node (i.e., inter-process) communication, which can be performed very efficiently through in-memory operations. This is expecially true in geographically distributed environment, where the network imposes not negligible communication latencies [5, 57, 66]. Interestingly, Xing et al. [211] investigate approaches to identify operator distributions that are resilient to unpredictable load variations. Informally, a resilient distribution does not easily become overloaded in presence of bursty and fluctuating input rates. In such a way, the system will be able to better withstand short input bursts. Observe that a static resilient placement is not in conflict with a dynamic deployment adaptation, and a DSP system can benefit from the combination of these two optimizations.

In geographically distributed environment, an alternative QoS metric is usually adopted, representing better the impact of network latencies when DSP operators are placed on distinct nodes. The *network usage* [NU] mea-

sures the amount of data that traverse the network at a given time; formally, it is defined as $NU = \sum_{l \in L} DR(l) \cdot Lat(l)$, where L is the set of links the stream uses, $DR(l)$ is the data rate over link l, and $Lat(l)$ is the latency of l. Basically, this metric considers the data rate exchanged on network links weighted by the latency of the link itself. The idea is that we would like to send data with higher production rate on faster links, whereas — at the same time — we might accept to send some few data on links with higher latency. The deployment solutions designed to work in geographic environment, such as the ones by Ahmad and Çetintemel [5] and Li et al. [122], very often minimize the network usage.

Generic cost functions. A few other works recur to the definition of a generic cost function to be minimized (or utility function to be maximized) that accounts for different QoS metrics (both user-oriented [53, 160, 175] and system-oriented [11, 197]). Some examples from literature show how the generic cost function can model energy consumption [104, 122] or processing and transmission cost [53, 160]. Alternatively, Tian and Chandy [197] propose a utility function that represents the net economic value generated by a fixed set of resources, that should be maximized by efficiently placing DSP applications.

2.2.2 What: Controlled Entities

Span of Control. The vast majority of the existing solutions control the deployment of each single application independently from the other ones already running. As such, the deployment policy neglects the interaction among multiple applications concurrently running on the same resources (which can interfere one another). In this case, we say that the scheduler controls the deployment of a *single application* [SA] at a time. Most of the solutions existing in literature work in this setting.

When the scheduler controls *multiple applications* at the same time, to properly identify and address the existing challenges, we need to distinguish between a cooperative and a non-cooperative setting (referred as [MA] and [MN], respectively).

In a cooperative setting, the scheduler has the ability to control the deployment of every application and might rearrange them so to achieve the optimization goal. For example, the deployment of some applications can be updated so to host a new application or to release computing nodes. This is usually the case of a single provider that manages multiple applications. In literature, a few solutions deal with the management of multiple applications in a cooperative environment. Ghaderi et al. [66] consider that applications can arrive and depart over time, therefore computing resources could be continuously re-assigned so to efficiently manage the varying load and minimize the resulting overall network traffic. Tian

and Chandy [197] and Shukla and Simmhan [185] investigate a more static setting, where multiple applications run together and have the same existence interval. Other research efforts have been proposed in the field of Complex Event Processing (CEP), a special case of DSP that allows to discover hidden and complex patterns from multiple streams of events. In this setting, multiple queries can share well-defined operators and streams of complex events, therefore the deployment policies try to reuse the already deployed operators, aiming to reduce resource utilization by avoiding to duplicate the computation [122, 183].

In a non-cooperative environment, multiple schedulers (or agents) have a set of application to execute, therefore they compete for obtaining computing and network resources. In nowadays systems this kind of environment is worthy of further investigation, also for implementing efficient strategies that work upon resource management tools (e.g., Mesos [84, 85]); these tools allows to dynamically share resources among different kinds of data analytics applications. To the best of our knowledge, so far there are no known solutions that determine the application initial deployment in a setting with multiple non-cooperative applications.

Granularity of Control. Another important property of the *what* dimension is the granularity of control. The vast majority of the existing approaches control the application deployment at the granularity of a *single operator* (referred as [O]).

Other approaches exploit the presence of "similarities" between operators, so to consider them as a unique black-box to be deployed; these solutions work at the granularity of *groups of operators* [G]. Specifically, Schneider et al. [180, 181], Gedik et al. [64], and Pollner et al. [159] (who build on [180]) divide the application into regions, that represent groups of operators with similar characteristics in terms of data partitions to be processed. These regions allow to automatically exploit both data parallelism and pipeline parallelism without compromising the application integrity. Another approach has been proposed by Ghaderi et al. [66], who consider deployment templates, which are unique ways of partitioning a graph and allocating each partition to a computing resource.

Finally, there is a class of solutions [160, 183, 188] that work at the level of *topology* [TO]; they apply several graph transformation techniques, which allow to further improve the application performance. These transformations include the operator reordering, fusion, separation, and algorithm selection.

2.2.3 Who: Management Authority

Along the *who* dimension, we characterize the existing approaches with respect to the management authority distribution, which can be centralized

or decentralized.

A *centralized authority* (referred as [C]) has access to the entire system current state, including resource availability, operators working conditions, and workload information. Relying on this broad knowledge, the centralized authority can potentially find a globally optimal deployment. At the same time, this centralized point of coordination might represent the system bottleneck that limits scalability. The vast majority of the existing approaches for the initial DSP deployment consider systems managed by a single centralized authority.

Decentralized authorities [D] usually have a local view of the system, therefore they take decisions based on a limited knowledge of the system state. If on the one hand this approach allows to overcome scalability issues, on the other hand it might not guarantee to find a globally optimal solution (the decentralized agents can get stuck in local optimum configuration, thus missing the globally optimum one [5, 14, 197]). To coordinate the decentralized authorities, a *hierarchical distributed architecture* [HD] can be used, where a centralized coordinator oversees the distributed agents with the aim to improve the optimality of the deployment solution. The management responsibilities of the centralized coordinator depends on the specific implementation of the hierarchical architecture. At present, solutions that exploit a hierarchical architecture for determining the initial application deployment are, to the best of our knowledge, still largely unexplored. We further discuss about the hierarchical distributed architectures in Section 2.3, where we detail more on reconfiguring at runtime the application deployment.

2.2.4 When: Timing

We distinguish between three main phases of the application lifetime, where deployment operations and optimizations can be automatically applied: design and compile time, deployment time, and runtime.

At design time, the user is fully responsible for designing the application topology. To achieve scalability, the user usually structures the complex data analytics application as a directed acyclic graph of multiple operators, each performing elementary tasks [190]. At *compile time* [DT], the application topology can be automatically optimized by the DSP system, which applies graph transformations [GT], aiming to improve performance. This approach is pursued by Schneider et al. [180, 181] and Pollner et al. [159], who determine groups of operators (named regions) that can be replicated as a whole thing. Stratosphere [8] also exploits graph transformations: it includes a query optimizer that automatically reorders and parallelizes operators in an application DAG, so to better exploit computing and network resources. We will discuss more about graph transformations while inves-

tigating the *how* dimension of our taxonomy in Section 2.2.6.

At deployment time, the DSP system executes the deployment policy to determine replication and placement of DSP operators. A good set of solutions from the state of the art tackles these problems. When a DSP system receives a new application, it can execute the deployment policy in a synchronous or in an asynchronous manner. In a *synchronous* [S] approach, the scheduler is activated as soon as a new application arrives to the system. This approach works very well in theory and it is the most commonly followed one when conceptual policies are designed. Other solutions rely on an *asynchronous* [AS] approach, where the scheduler runs periodically. At each round, it collects the submitted applications and computes their deployment [58, 118, 122, 124, 156, 174, 175, 186, 197]. Note that among two subsequent executions of the scheduler, multiple applications, that request resources for execution, can be submitted to the system. In this case, the deployment solution can consider them either independently (if the span of control is [SA]) or in conjunction (if the span of control is [MA]).

We describe the solutions dealing with the application runtime in Section 2.3.

2.2.5 Where: Computing Infrastructure

The taxonomy allows us to describe with a fine granularity the resource infrastructure considered by the solutions existing in literature. Three main properties describe the computing infrastructure: resource heterogeneity, resource distribution, and computing environment.

Resource heterogeneity ([HE] in Table 2.1) refers to the ability, of the deployment algorithm, to consider specific features of computing and network resources, such as processing capacity, available resources, or network delay (e.g., [11, 14, 64]). Other solutions consider resources as *homogeneous* [HO], i.e., as they all have same features in terms of (usually) capacity. Although this homogeneous view of the infrastructure might seem a simplification, it is reasonable when the deployment solution is designed to work in a clustered environment (e.g., [118, 180, 183, 186, 197]).

The resource distribution property summarizes the ability of the approaches to work with resources disseminated on different scale. We only distinguish between *local distribution* [LO] and *geographic distribution* [GE]. Locally distributed resources are located within the same data center, where they are inter-connected with high speed communication links. Therefore, network delays have a light impact on the application performance and can be neglected. Conversely, geographically distributed resources can span multiple data centers and, although interconnected with high speed links, they experience a communication delay that cannot be neglected. This is usually the case of distributed Cloud and Fog computing environments. In

Table 2.1, we also consider "(GE)" as resource distribution, meaning that the deployment solutions only implicitly consider geographically distributed resources (e.g., by minimizing inter-node traffic) [53, 57, 58, 66].

The computing environment property assumes one of the following values: *cluster* environment [CU], *Cloud computing* environment [CC], *Fog computing* environment [FC]. Each computing environment is characterized by specific features. A cluster environment is usually characterized by a static pool of computing machines. In the Cloud, the pool of resources is dynamic, meaning that virtual machines can be acquired and released as needed with a very low provisioning time. Moreover, this environment can offer virtual machines with homogeneous or heterogeneous capacity or resources. In case of heterogeneous resources, the deployment solution has to wisely pick the most suitable configuration that satisfies the application needs. The Fog computing environment is characterized by highly heterogeneous computing resources, because the environment encompasses high performance resources and small entry resources located at the edge of the network [22, 23]. Due to geographic distribution, network latencies can have a strong impact on the application performance.

The vast majority of existing solutions, which deal with the initial deployment of DSP applications, have been designed to work in a locally distributed clustered environment. Here, solutions consider both homogeneous [118, 180, 181, 183, 186, 197] and heterogeneous resources [64, 156, 159, 174, 188]. Solutions for geographically distributed environments usually consider resource heterogeneity so to model network delays [122, 124, 226]. In this setting, several other approaches use a coarse definition of the network[5], being only interested in reducing the amount of inter-node traffic [5, 57, 58, 66]. For example, Backman et al. [14] investigate the initial placement in a geographically distributed Cloud environment. Nowadays the Fog computing environment is attracting a lot of interest, promising to improve system scalability and reduce application response time by decentralizing the computation (i.e., by moving operators close to data sources and final information consumers). Arkian et al. [11] and Sajjad et al. [175] propose deployment solutions explicitly designed for this environment.

2.2.6 How: Actions and Methodologies

Actions. The deployment actions represent the set of mechanisms that can be used to achieve the deployment goals. Several mechanisms have been proposed in literature and, in our taxonomy, we identify five main deploy-

[5]The coarse definition of the network does not explicitly model network latencies, but only defines an ordering relation among possible placement configurations. At most, they distinguish whether the communication is performed within the same node, same rack, same data center, or between different data centers.

ment actions: graph transformation, operator reuse, operator replication, operator placement, and operator co-location.

Graph transformations [GT] rearrange the application topology with the aim to improve performance [86, 160]. The most popular transformations are: operator reordering, which moves more selective operators upstream to filter data early [8, 133, 188]; *operator reuse* [RS] (or redundancy elimination), which avoids redundant computations by reusing already deployed operators [122, 183]; operator separation, which splits operators into smaller computational steps [160]; and algorithm selection, which automatically (and safely) selects a faster algorithm for implementing an operator [166]. Among graph transformations, we also include the pre-processing of the application graph so to identify regions of operators (i.e., groups) that can be replicated homogeneously, as a whole thing [64, 159, 180, 181]. Schultz-Møller et al. [183] propose a system that automatically rewrites the application topology, by combining and reordering its operators, so to obtain an equivalent application with lower CPU cost. Stratosphere [8] also includes a query optimizer that automatically transforms and allocates the application graph, so to minimize a cost function that captures network traffic and CPU load; this component manages also User Defined Functions (UDFs).

Operator replication [OR] (also known as *operator fission* or *data parallelism*) consists in determining the operator replication degree (which is also referred to as parallelization degree) so to efficiently process the incoming workload. Indeed, by partitioning the stream over multiple replicas, running on one or more computing nodes, the load of each replica is reduced, which, in turn, yields better operator (and overall application) performance. Determining the operator replication degree by hand is possible, but cumbersome. Besides identifying the bottleneck operators that can truly benefit from replication, developers should determine the best parallelism degree avoiding resource over-utilization or wastage; most importantly, they should verify whether applying data parallelism preserves the application semantics. To this end, developers may have to solve some issues on their own (e.g., data ordering). All of these tasks are tedious and error-prone, especially when the application size and the number of relations among operators grow. Moreover, since the load can vary over time, the number of replicas should be changed accordingly as to optimize some non functional requirements. In this section, we consider only the deployment solutions that compute the initial replication degree and postpone to Section 2.3 the analysis of the approaches that adapt the replication degree at runtime (i.e., elasticity). To determine the initial number of operator replicas, the deployment solutions usually exploit performance information so to identify and replicate the application bottlenecks (e.g., [14, 64, 185]). Other approaches try to use all the available resources by allocating replicas to each available processing unit (e.g., [180]). Gedik

et al. [64] repeatedly locate the group of bottleneck operators and greedily parallelize them until no performance improvements can be achieved. Similarly, Lakshmanan et al. [118] propose a local search algorithm that, assuming (estimated or measured) knowledge on the system dynamics, replicates stateless operators. Shukla and Simmhan [185] suggest to build the operators performance model by relying on micro-benchmarks to be executed a-priori. The research efforts by Schneider et al. [180, 181] focus on preserving safety when stateful and stateless operators are replicated. A transformation is safe if it does not change the observable behavior of the application (in terms of integrity, stateful operations, and ordering of produced results).

Operator placement [OP] consists in determining, within a set of available distributed computing resources, the nodes that should host and execute each operator of a DSP application. Most of the existing works focus on solving this problem (e.g., [14, 53, 174, 183, 196]). The approaches that aim to minimize inter-node traffic (that are usually adopted in geographically distributed environment) also solve the *operator co-location* [CL] problem, which is special case of the operator placement problem. It deals with identifying a group of operators that can be placed on the same computing node as to avoid the overhead of data serialization and transport [8, 57, 58, 107, 133, 183]. Indeed, sending data through the network introduces a not negligible latency that can be detrimental for the application performance. The key challenge comes from the finite capacity of computing nodes, which requires to wisely identify groups of operators to be allocated on distributed nodes so that the overall amount of data exchanged using the network is minimized.

Methodology. The *methodology* property of the *how* dimension investigates the class of algorithms used to determine *what* deployment actions should be used and *how*, aiming to achieve the deployment goals. We classify the methodologies proposed so far in three main categories: mathematical programming, approaches based on graph theory, and heuristics. The latter is further specialized in subclasses that cover the most commonly adopted approaches, namely greedy, local search, tabu search, genetic algorithm, and simulated annealing.

Mathematical programming [MP] approaches focus mainly on the operator placement problem, which is formulated and solved using tools from operational research. Arkian et al. [11] formulate the placement over Fog computing resources as a non-linear integer programming problem, which is then approximated and more efficiently solved through linearization (thus obtaining an Mixed-Integer Linear Programming (MILP) problem — which nevertheless does not scale well as the problem size increases [29]). Eidenbenz and Locher [53] analyze the placement problem for a special kind of DSP application topologies, i.e., serial-parallel decomposable graphs. This

allows them to exploit strong theoretical foundations and propose an approximation algorithm, which, however, can allocate operators only on resources with uniform capacity. A non-linear programming model is formulated by Stanoi et al. [188], who also propose several heuristics to efficiently cope with the problem resolution. The heuristics implement some of the well-known meta-heuristics: greedy first-fit augmented with a local search, tabu search, and simulated annealing. Aiming to satisfy requirements (and not to optimize an utility function), Thoma et al. [196] recur to a Constraint Satisfaction Problem (CSP), which can be more efficiently solved with respect to equivalent optimization problems. This happens because, in a CSP, it suffices to find one solution among all the feasible ones (i.e., not the best one). In general, mathematical programming approaches allow to find the optimal deployment solution; nevertheless, their great limitation is scalability. Indeed, the deployment problem is NP-hard and its resolution time can prohibitively grow for large problem instances. However, we will show in Chapters 4 and 5 how these models can be used to develop efficient model-based approaches that compute the application placement without strongly compromising the solution quality.

The approaches based on *graph theory* [GRA] usually aim to partition the application acyclic graph in a such a way that inter-node traffic is minimized (this condition corresponds to minimize the sum of edge weights cut by the partitioning). Li et al. [122] show that this approach finds the optimal solution if the application can be represented as a tree. Fisher et al. [57, 58] consider generic graphs and propose a solution that, together with the minimization of inter-node traffic, aims at evenly distributing load across computing nodes. Zhu and Agrawal [226] also exploit graph theory; they model the placement problem as a graph isomorphism problem, where the application graph has to be mapped on the graph of resources that minimizes the application response time. Although very elegant, this approach assumes that a resource node can host at most a single operator (which might be a non-realistic hypothesis in today's DSP systems). COLA [107] is a operator co-location optimizer, used in System S, that works with heterogeneous computing nodes and allows the user to specify a number of location constraints. Starting with all operators fused together into a single group, the COLA algorithm iteratively splits large groups into separate groups by solving a specially formulated graph partitioning scheme. Eskandari et al. [54] solve the operator placement problem for Apache Storm, which needs to assign operators to worker nodes, and then, for each worker node, operators to Java processes (i.e., worker processes). The authors solve both the problems with a graph partitioning algorithm that aims to minimize the inter-node and inter-process traffic. From the state of the art, we can observe that the approaches based on graph theory can quickly find a solution that minimizes some inter-node cost (e.g., traffic). Nevertheless,

they might not be easily adopted to optimize other QoS metrics (e.g., network usage) and multi-objective functions.

As shown in [29], the deployment problem is NP-hard, therefore many research efforts propose heuristics to solve the problem in a feasible amount of time. The most common meta-heuristic, which is then customized to address the problem at hand, is the *greedy* [GR] heuristic (e.g., first-fit, best-fit); it has been adopted in [14, 64, 124, 174, 180, 181, 183, 188]. This approach explores the solution space and accepts the first configuration which satisfies (or optimizes) a given utility function. Backman et al. [14] model the placement as a bin-packing problem, which is then solved with different greedy approaches that differ in the amount of required system knowledge. The solution by Rychly et al. [174], which works in heterogeneous clusters, first benchmarks each pair of operator-computing node and then greedily selects the configuration of nodes that maximizes the application throughput. A greedy approach is also pursued by Schneider et al. [180, 181] to identify groups of operators that can be replicated together: their heuristic creates regions of operators from left-to-right (i.e., from data sources to sinks), by adding operators to the same region until all the safety conditions are satisfied. Greedy approaches are usually very fast, because they terminate as soon as a local optimum is found. However, if the objective function admits many local optimum points, other approaches should be used to further explore the solution space so as to determine the global optimum.

Approaches based on *local search* [LS] move from solution to solution by greedily applying local changes [64, 124, 188]. For example, the heuristic by Lakshmanan et al. [118], which focuses on parallelizing stateless operators, replicates every (predicted) bottleneck operator until no further performance improvements can be achieved. Li et al. [124] use a regression model to estimate the application performance resulting from the operators placement; then, starting from an initial configuration, the heuristic keeps improving it by adjusting the placement solution according to the predicted performance.

The drawback of methods with local improvements (e.g., greedy, local search) is that they might find only a local optimum and miss the global optimum configuration. *Tabu search* [TS] and *simulated annealing* [SN] increase the chances to find a global optimum by moving, if needed, through non-improving placement configurations [188]. Specifically, starting from an initial configuration and a set of neighbor configurations, tabu search only accepts improving configurations, till finding a local optimum. Then, it continues to explore the search space by selecting the best non-improving configuration found in the neighborhood of the local optimum. To avoid accepting the already visited local optimum, tabu search uses a tabu list of configurations that cannot be accepted anymore. When a stopping condi-

tion is reached, it returns the best local optimum configuration. Simulated annealing is another meta-heuristic conceived to escape from local minimum configurations. Differently from tabu search, simulated annealing first aims to find the region with the global optimum configuration and then move with small steps to the optimum. First, it has the flexibility of taking steps in random directions; then, as time passes, it reduces the probability to accept configurations that do not improve the objective function. Usually these meta-heuristics find high-quality deployment configurations, even though they might take considerably longer time to compute the best configuration. To the best of our knowledge, so far there are only a few research works exploring these approaches to compute the deployment of DSP applications (e.g., [3, 188]).

An approach based on a *genetic algorithm* [GA] has been proposed by Smirnov [186] to compute the operator placement. Initially, the genetic algorithm generates a random population of chromosomes, which represent deployment configurations. Then, it performs genetic operations such as crossover and mutations to obtain successive generations of these chromosomes. A crossover operator takes a pair of parent chromosomes and generates an offspring chromosome by crossing over individual genes from each parent. A mutation operator randomly alters some parts of a given chromosome so to avoid to get stuck in a local optimum. Afterwards, the genetic algorithm picks the best chromosomes from the entire population based on their fitness values and eliminates the rest. This process is repeated until a stopping criterion is met. Smirnov et al. [186] use the minimization of the application throughput as fitness function; to compute its value during the initial deployment, the fitness function relies on performance models built on statistical data. A genetic algorithm works by randomly moving through configurations that can improve the deployment objectives. To the best of our understanding, this approach works well when the number of possible deployment choices for a DSP operator is limited, even though the total number of configurations (i.e., ways of deploying the whole application) is high.

The taxonomy uses the *heuristic* [H] class to cover all the other approaches that adopt custom solutions to solve the operator replication and placement problem. Most of the existing research efforts consider these two problems as orthogonal, so they focus on one of them [5, 156, 175, 185, 197] or solve them in two stages [180]. In [5], a DHT drives the operator placement and load balancing among resources. Peng et al. [156] place computing nodes on a resource space; then, DSP operators are assigned to nodes by minimizing a distance function in the resource space. The distance function considers the operators' resource requirements and the available resources. Shukla and Simmhan [185] propose a model-driven approach for computing the replication degree of DSP operators. Their approach first

computes a performance model relying on micro-benchmarks and then determines the replication degree that supports a given input data rate. To maximize the economic value generated by a fixed set of resources, Tian and Chandy [197] treat the DSP placement problem as a commodities market problem. Ghaderi et al. [66] propose a placement solution that revolves around the idea of deployment templates, i.e., unique way of partitioning a graph and allocating each partition on a computing resource. Differently, Pollner et al. [159] jointly compute the initial replication degree of operators and their placement. First, they group operators in regions using [180]. Then, greedy heuristics are proposed to assign each parallelizable region to computing resources, by exploiting the alternative (network) routes between processing nodes. Gu et al. [69] investigate the operator placement problem for infrastructures having heterogeneous computing and network resources. After providing a problem formulation, the authors also present two heuristics. The first one minimizes the application response time by recursively improving the placement of the operators in the application critical path (which is expressed in terms of latency). The second one maximizes the application throughput; it relies on a dynamic programming procedure that identifies and optimizes the placement of the bottleneck operators. Li et al. [121] focus on the time to recovery the application execution after a failure. Indeed, in case of upstream backup, when a computing node fails, the application experiences a recovery time to restore a previously stored application state and to reprocess new data (see Section 2.3.6). The authors observe that, when dependent operators are located on the same processor, failing together, they may introduce a longer recovery time. This is due to the need of reprocessing lost data under the constraint of sharing computing resources. Therefore, the authors design an operator placement heuristic that limits the expected recovery time in case of failures.

2.2.7 Wrap-up

In this section, we wrap-up the related works regarding the initial deployment of DSP applications and summarize the most common solutions considered along the *five Ws one H* dimensions.

Most solutions consider a single centralized authority which computes the initial deployment of single applications at a time. As such, it does not usually change the deployment of running applications to accommodate new ones. A common deployment goal is the optimization of a single-objective utility function, which accounts for a user-oriented metric (i.e., inter-node traffic, response time, or throughput). As regards the computing infrastructure, existing policies usually consider a cluster environment where resources are locally distributed. In this setting, the most common

deployment problem which has been investigated is the operator placement problem. To solve this problem, heuristics are by far the most adopted methodology, with a special preference for greedy approaches that consider the operator placement problem as a bin-packing problem. Here, operators (or groups of operators), possibly sorted according to a specific criteria (e.g., exchanged data rate), are assigned to the set of computing resources in a first-fit or best-fit manner. These strategies can be easily used to increase the operator co-location, which, in turn, reduces the application latency (or increases the application throughput).

2.2.8 Thesis Contribution

The analysis included in this section has shown that the deployment solutions existing in literature are characterized by different assumptions and optimization goals (as also pointed out in [117]). Moreover, since there is no general formulation of the deployment problem, it is not easy to analyze and compare these solutions one another, e.g., to select the most suitable one for the use case at hand. Today's DSP applications are usually high demanding, and a solution to improve scalability and reduce network latency lies in taking advantage of the ever increasing presence of near-edge/Fog Cloud computing resources [23]. Nevertheless, the use of a diffused infrastructure poses new challenges that include network and system heterogeneity, geographic distribution, and non-negligible network latencies among distinct nodes processing different parts of a DSP application. Considering this setting, we provide three main contributions that deal with the initial application deployment.

First, in Chapter 4, we propose ODP, a unified general formulation of the operator placement problem for distributed and networked DSP applications, which takes into account the heterogeneity of application requirements and infrastructural resources. Differently from [19, 53], we model the placement of DSP applications that can be represented by a directed acyclic graph, therefore we do not limit the formulation to special topologies. Unlike all the approaches presented in this section (e.g., [53, 94, 196]), we model the placement problem for DSP applications with a holistic vision of both computing and networking resources (i.e., we explicitly model the impact of the network on application performance). Furthermore, ODP considers QoS attributes of applications and resources, and is flexible enough to accommodate new QoS metrics. Thanks to the adjustment of suitable knobs, it can adapt the meaning of "optimal placement" according to the application context. As such, ODP also represents a general framework for QoS optimization and comparison of different approaches.

Second, in Chapter 5, we build on ODP to design several new heuristics. We divide them into two main groups. In the first one, we have

model-based heuristics, which run ODP on a properly restricted solution space. These heuristics propose different approaches for reducing the solution space so to enable a faster resolution of ODP. In the second group, we have the model-free heuristics, which customize some well-known meta-heuristics, namely greedy first-fit, local search, and the tabu search. All of them optimize a multi-objective utility function that accounts for user-oriented and system-oriented metrics. They consider heterogeneous resource, whose distribution is geographic. Differently from the approaches proposed so far, we use ODP to thoroughly evaluate the designed heuristics, aiming to assess their quality along two dimensions: processing speedup and quality of the computed solution. These two dimensions allow to evaluate the heuristics behavior and identify those that achieve a good performance trade-off (i.e., that do not sacrifice one dimension in favor of the other). Chapter 5 will show that different trade-offs between reduced resolution time and high solution quality can be achieved. These trade-offs depend on the deployment configurations, expressed in terms of application type, infrastructure size, and deployment objectives. On average, the local search heuristic shows the best trade-off between the two considered dimensions.

These first two contributions address only the operator placement problem. Chapter 6 presents our third contribution, which considers the problem of determining the replication degree of the DSP operators and their placement over a set of distributed computing nodes. We name the resulting model as ODRP; differently from the existing works (e.g., [53, 132, 145]), it provides a unified formulation of the replication and placement problem. ODRP allows us to jointly optimize the placement and replication of the DSP operators, while maximizing the QoS attributes of the application. The proposed formulation considers several QoS attributes of applications and resources, and is flexible enough to accommodate new QoS metrics. Similarly to ODP, ODRP provides a general framework for QoS optimization, that can be used as a benchmark against which existing approaches that deal with operator replication and placement can be compared.

With respect to the taxonomy of Figure 2.2, our contributions address the following region of the problem space:

- Why: optimization of a multi-objective function, which takes into account both user-oriented and system-oriented QoS metrics. As user-oriented metrics, we model response time and availability; as system-oriented metrics, we model several network-related metrics that have been widely used in the literature (i.e., network usage, inter-node traffic, and the so called elastic energy).
- What: control of a single application at a time, with granularity of a single operator.

- Who: since we compute the optimal deployment solution, which requires the global view of the system, we rely on a single centralized authority.
- When: ODP, the heuristics, and ODRP operate at deployment time; they consider a synchronous approach for determining the application deployment.
- Where: our contributions model a cluster of heterogeneous computing and network resources, whose distribution is geographic.
- How: ODP and the heuristics determine the operator placement, whereas ODRP jointly optimizes the operator replication and replica placement. As regards the methodologies, ODP and ODRP rely on a mathematical programming approach, defining the deployment problem as an Integer Linear Programming (ILP) problem.

2.3 Runtime Deployment Adaptation

The computational requirements of DSP applications are usually unknown a-priori and, most importantly, they can change continuously at runtime. This, together with the long-running nature of DSP applications, requires DSP systems to monitor the application at runtime and adapt its deployment in a proactive or reactive manner.

In this section, we present the existing approaches that deal with the deployment adaptation of DSP applications at runtime. As we will see, the different research efforts address a wide range of challenges that arise when applications with stringent QoS requirements run in a dynamic environment. To provide a good overview of the existing solutions for runtime adaptation, we extend the taxonomy of Section 2.1 and discuss how the existing literature addresses the *six questions*. Figure 2.3 shows the resulting taxonomy, whereas Table 2.3 positions the most relevant works with respect to our taxonomy. The list of all the labels used to classify the existing solutions is reported in Table 2.2.

2.3.1 Why: Deployment Goals

Similarly to the initial deployment of DSP applications (Section 2.2), their runtime adaptation can be driven by optimization goals which can be distinguished into three main categories: *constraint satisfaction* [CS], *single-objective* [S] optimization, and *multi-objective* [M] optimization. Moreover, since DSP applications usually run under dynamic working conditions (e.g., changing incoming workload), several works optimize a single- or multi-objective function while meeting requirements on another QoS metric. In Table 2.3, we mark these solutions as [S; CS] (or [M; CS]) and indicate the QoS metric to be satisfied in the *Why: Metric* column after the semicolon. The existing solutions rely on QoS metrics that can be categorized in *user-oriented* [UO] and *system-oriented* [SO] metrics.

User-oriented QoS metrics. Many solutions consider the same metrics optimized for the initial deployment, such as *response time* [R] and *throughput* [T]. As regards response time, most of the existing approaches model the end-to-end application response time (e.g., [78, 127, 132, 167, 172]), whereas other approaches consider only the response time of a single DSP operator [61, 142, 178, 200]. As regards throughput, all the approaches consider the output throughput, i.e., the number of tuples that can be processed by the DSP application per unit of time [60, 65, 109, 111, 133, 47, 48, 179, 214, 215], whereas no one currently optimizes the input throughput.

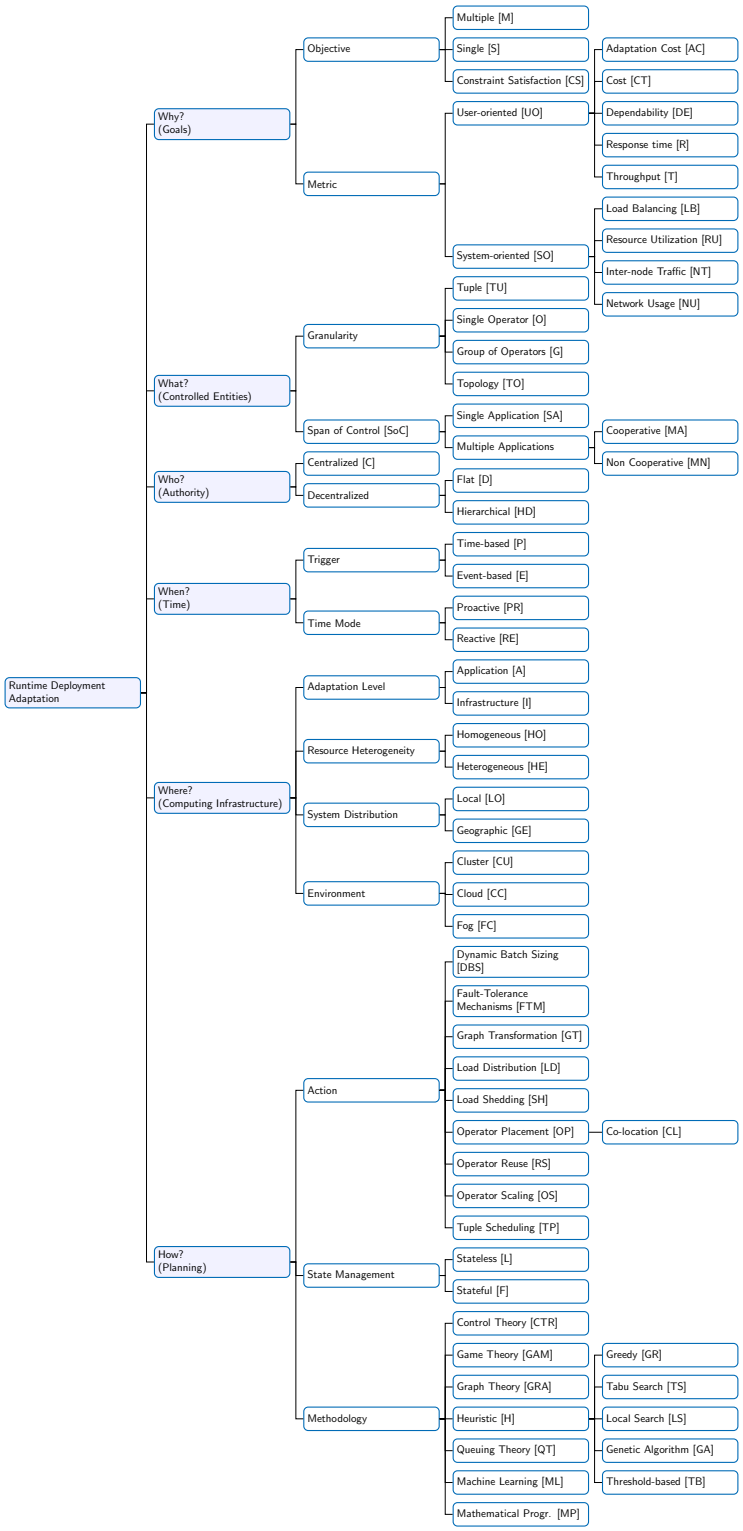


Figure 2.3: Taxonomy of self-adaptation approaches for DSP applications.

Table 2.3: State-of-the-art solutions for adapting the DSP application deployment at runtime.

	WHY		**WHAT**		**WHO**	**WHEN**		**WHERE**				**HOW**		
	Obj.	Metric	SoC	Gra.	Aut.	Trig.	Mode	Lev.	Heter.	Distr.	Env.	Act.	State	Method.
Aniello et al. [9]	S	NT	SA	O	C	P	RE	A	HE	LO	CU	OP, CL	L	GR
Bellavista et al. [18]	S; CS	CT; UO	SA	O	C	P	RE	A	HO	LO	CU	FTM	L	MP, H
Caneill et al. [25]	M	NU, LB	SA	O	C	P	RE	A	HO	GE	CU	OP, CL	F	GRA
Castro Fernandez et al. [31]	-	DE	SA	O	C	P	RE	A, I	HO	LO	CC	OS	F	TB
Cheng et al. [35]	S	T	SA	O	C	P	RE	A	HO	LO	CU	OS	L	RL
Cerviño et al. [32]	S	RU	MA	-	C	P	RE	I	HE	GE	CC	LD	L	H
Chatzistergiou et al. [34]	S; CS	NT; RU	SA	G	C	P	RE	A	HE	GE	CU	OP, CL	L	H
Das et al. [44]	S	R	SA	TO	C	P	RE	A	HO	LO	CU	DBS	L	CTR
De Matteis et al. [47, 48]	M	UO, CT, AC	SA	O	C	P	PR	A	HO	LO	CU	OS, LD	F	CTR
Delimitrou et al. [50]	S	RU	MA	O	D	P	RE	A	HE	LO	CU	OP	L	LS
Floratou et al. [60]	S	LB	SA	O	C	P	RE	A	HO	LO	CC	OS, LD	L	H
Floratou et al. [60]	S; CS	LB; T	SA	O	C	P	RE	A	HO	LO	CC	OS, LD	L	H
Fu et al. [61]	S	R	SA	O	C	P	RE	A	HE	GE	CC	OP, OS	L	QT
Gedik et al. [65]	S	T	SA	G	C	P	RE	A	HO	LO	CC	OS	F	TB
Gulisano et al. [70, 71]	S	RU	SA	G	C	P	RE	A, I	HO	LO	CC	OS	F	TB
Han et al. [73]	S	RU	SA	O	C	P	RE	A	HE	LO	CU	OP	L	GR
Heintz et al. [75]	M	NT, R	SA	O	D	P	RE	A	HO	GE	CU	DBS	L	H
Heinze et al. [78]	M	RU, R, AC	SA	O	C	P	RE	A, I	HE	LO	CC	OS	F	TB[6]
Heinze et al. [79]	S	RU, (AC)	MA	O	C	E	RE	A, I	HE	LO	CC	OP	F	GR
Heinze et al. [80]	S	RU	SA	O	C	P	RE	A, I	HO	LO	CC	OP	L	TB, RL
Heinze et al. [82]	M	RU, DE, AC	SA	O	C	P	RE	A	HO	LO	CC	FTM	L	TB
Hidalgo et al. [83]	S	RU	SA	O	C	P	PR, RE	A	HO	LO	CC	OS	L	TB
Hochreiner et al. [87]	S, CS	CT, UO	SA	O	D	P	RE	A, I	HE	LO	CC	OS	F	TB
Hochreiner et al. [88]	S	CT	SA	O	D	P	RE	A, I	HE	LO	CC	OS	F	TB

(continued on next page)

[6]When a migration should be performed, this policy considers only the subset of migrations that introduces a latency spike smaller than a given threshold.

(continued from previous page)

	WHY		WHAT		WHO	WHEN		WHERE				HOW		
	Obj.	Metric	SoC	Gra.	Aut.	Trig.	Mode	Lev.	Heter.	Distr.	Env.	Act.	State	Method.
Hoseiny Farahabady et al. [90]	M; CS	RU; UO	MA	O	C	P	PR	A	HO	LO	CU	OP	L	CTR, H
Hoseiny Farahabady et al. [91]	CS	UO	SA	O	C	P	PR	A	HO	LO	CU	OP	L	LS
Huang et al. [94]	S; CS	NU; UO	SA	O	C	E	RE	A	HE	GE	CU	OP	L	GR
Hummer et al. [95]	M	NT, RU, AC	MA	O	C	P	RE	A, I	HO	LO	CC	OP, RS	L	MP, TB
Ishii et al. [97]	S	CT	SA	O	C	P	PR	I	HE	LO	CU+CC	OP	L	MP
Jiang et al. [100]	S	NT	MA	O	C	P	RE	A	HE	LO	CU	OP	L	GRA, GR
Kalyvianaki et al. [101]	S	SO	MA	O	D	P	RE	A	-	-	-	SH	-	CTR
Kalyvianaki et al. [102]	S	SO	MA	O	D	P	RE	A	-	-	-	SH	-	H
Kalyvianaki et al. [103]	S	RU	MA	O	C	P	RE	A	HE	LO	CU	OP, RS	L	MP
Khorlin et al. [108]	S	UO	SA	O	C	P	RE	A	HO	LO	CU	OP	L	QT
Khorlin et al. [108]	S	UO	SA	O	D	P	RE	A	HO	LO	CU	OP	L	CTR
Kleiminger et al. [109]	M	T, CT	SA	O	C	P	RE	A, I	HO	LO	CU+CC	LD	L	TB
Koliousis et al. [111]	S	T	MA	O	C	P	RE	A	HE	LO	CU	LD	L	H
Kombi et al. [112]	S	T	SA	O	C	P	PR	A	HO	LO	CU	OS	-	H
Kumar et al. [114]	S	UO	SA	O	D	E	RE	A	HO	GE	CU	OP	L	H, TB
Kumbhare et al. [115]	S	LB	-	O	D	E	RE	A, I	HE	LO	CC	LD	F	H, TB
Kumbhare et al. [116]	M	UO	SA	O	C	P	RE	A, I	HE	GE	CC	OP	L	GA, GR
Li et al. [123]	S	NT, (AC)	SA	O	C	P	RE	A, I	HO	LO	CC	OS	F	TB
Li et al. [125]	S	R	SA	TO	C	P	RE	A	HO	GE	CU	DBS	L	MP
Liu et al. [127]	S; CS	NT, RU; R	SA	O	C	P	RE	A, I	HO	LO	CC	OP, CL, OS	L	GR
Loesing et al. [131]	S	LB	MA	O	C	P	RE	A, I	HE	LO	CC	OS	L	TB
Lohrmann et al. [132]	S; CS	RU; R	SA	O	C	P	RE	A	HO	LO	CC	OS	L	MP, QT
Lohrmann et al. [133]	S	T	SA	O	C	P	RE	A	HO	LO	CU	DBS, CL	-	H
Lombardi et al. [134]	S	RU, (AC)	MA	O	C	P	PR, RE	A, I	HO	LO	CC	OS	F	TB, H
Madsen et al. [136, 139]	S	LB, NT, (AC)	SA	O,TO	C	-	RE	A	HO	LO	CU	GT, OP, CL	F	H
Madsen et al. [137]	S	AC	SA	O	C	P	PR, RE	A	HO	LO	CU	-	F	H
Madsen et al. [138]	M; CS	LB, NT; AC	SA	O	C	P	RE	A	HO	LO	CU	OP, CL	F	H, MP
Mayer et al. [142]	CS	R	SA	O	D	P	PR	A	HE	LO	CU	LD	L	GR

(continued on next page)

(continued from previous page)

	WHY		WHAT		WHO	WHEN		WHERE				HOW		
	Obj.	Metric	SoC	Gra.	Aut.	Trig.	Mode	Lev.	Heter.	Distr.	Env.	Act.	State	Method.
Mencagli [145]	M	T, CT	MN	O	D	P	RE	A	HO	LO	CC	OS	F	GAM
Mencagli et al. [146]	S	T	SA	O	HD	P	RE	A	HO	LO	CU	LD, OS	L	CTR, H
Pietzuch et al. [158]	S	NU	SA	O	D	P	RE	A	HE	GE	CU	OP	L	H
Pundir et al. [162]	S	LB, (AC)	SA	O	C	P	RE	A	HO	LO	CU	OS	F	H
Qian et al. [163]	S	LB	MA	O	C	P	RE	A	HO	LO	CU	OP	L	GR
Ravindra et al. [167]	CS	R	SA	O	C	P	RE	I	HO	LO	CU+CC	OP	L	TB
Repantis et al. [168]	S; CS	LB, R	MA	O	D	P	RE	A	HE	GE	CU	OP, RS	L	H
Rizou et al. [170, 172]	CS	R	SA	O	D	P	RE	A	HE	GE	CU	OP	L	H
Rizou et al. [171]	S	NU	SA	O	D	P	RE	A	HE	GE	CU	OP	L	MP
Saurez et al. [178]	CS	R	MA	O	D	P	PR, RE	A	HE	GE	FC	-	F	TB
Schneider et al. [179]	S	T	SA	O	C	P	RE	A	HO	LO	CU	OS	L	H
Schneider et al. [182]	S	SO	SA	G	D	P	PR	A	HE	LO	CU	LD	F	H
Sun et al. [191]	CS	R	SA	O	C	P	RE	A	HE	LO	CU	OP	L	TB
Sun et al. [192]	S	R	MA	O	C	E	RE	A	HE	LO	CU	OP, OS	L	H
Tudoran et al. [200]	S	R	SA	O	C	P	RE	A	HE	GE	CC	OP, CL, DBS	L	LS
Van der Veen et al. [202]	M	RU, LB	SA	O	C	P	RE	A, I	HO	LO	CC	OP, OS	L	TB
Wang et al. [205]	S; CS	RU; UO	SA	O	C	P	RE	A	HE	LO	CU	OP, OS	L	ML
Wolf et al. [209]	S	SO	MA	O	C	P	RE	A	HE	LO	CU	OP	L	MP, H
Xing et al. [212]	S	LB	MA	O	C	P	RE	A	HO	LO	CU	OP	L	GR
Xu et al. [213]	S	NT	SA	O	C	P	RE	A	HE	LO	CU	OP	L	GR
Xu et al. [214]	S	T	SA	O	C	P	RE	A	HO	LO	CU	OS	L	GR
Yang et al. [215]	S	T	MA	O	C	P	RE	A	HO	LO	CC	OP	L	GA
Zacheilas et al. [219]	M	UO, AC	SA	O	C	P	PR	A	HO	LO	CU	OS	L	MP
Zhang et al. [222]	M	NT, LB	SA	O	C	P	RE	A	HE	LO	CU	OP	L	H
Zhang et al. [223]	S	R	SA	O	C	P	RE	A	HO	LO	CU	DBS	F	H
Zhou et al. [224]	S	R[7], LB	SA	O	C	P	RE	A	HO	LO	CU	OP	L	LS

(continued on next page)

[7]This solution optimizes response time only while computing the initial application deployment.

(continued from previous page)

	WHY		**WHAT**		**WHO**	**WHEN**		**WHERE**				**HOW**		
	Obj.	Metric	SoC	Gra.	Aut.	Trig.	Mode	Lev.	Heter.	Distr.	Env.	Act.	State	Method.
Zhou et al. [225]	S	UO	SA	TP	C	P	RE	A	HO	LO	CU	TP	L	H

Many solutions for runtime adaptation also consider other user-oriented metrics, which aim to improve the application dependability, reduce the execution cost or the adaptation cost. The long running nature of DSP applications stresses the importance of fault tolerance, which allows to preserve the working conditions even when failures occur in the execution environment. To tackle this problem and propose effective solutions, the existing approaches improve a set of metrics (i.e., availability, reliability, safety) that collectively determine the application *dependability* [DE]. In general, the higher the dependability, the higher the probability that the application performs the required function at any randomly chosen point in time [18, 31, 82]. Dependability plays a key role in the approach by Heinze et al. [82], which dynamically combines fault tolerance mechanisms, characterized by different performance, with the aim to reduce costs while minimizing the number of violations of a user-defined recovery time threshold. Interestingly, Bellavista et al. [18] consider a slightly different perspective; they observe that, for a number of applications, perfect fault tolerance is not always needed, therefore it can be sacrificed for effectively managing temporary load variations.

Cloud computing enables to quickly acquire and release computing resources when needed, and to pay only for the leased resources (i.e., a pay-per-use pricing model is applied). These features perfectly fit with the dynamic nature of DSP applications. Exploiting them, several policies have been developed to efficiently change the number of computing resources used by the DSP application at runtime so to successfully handle varying incoming workloads, while optimizing the *cost* [CT] of execution [18, 88, 97, 109]. Indeed, if on the one hand acquiring more resources allows to better exploit data parallelism and quickly process huge incoming workloads, on the other hand, it results in higher costs. Achieving the optimal trade-off is not trivial. Even though cost is not explicitly considered in [32], Cerviño et al. propose a solution that improves resource utilization in the Cloud (which yields to a reduced execution cost). This solution autonomously scales the number of computing resources in response to workload variations, aiming to achieve low application response time, while using resources at their maximum processing capacity.

Reconfiguring the application at runtime involves the execution of management operations that apply the deployment changes while preserving the application integrity. The latter is a critical task especially when the application includes stateful operators. Indeed, when a stateful operator is relocated, besides distributing the operator code, the DSP system should efficiently migrate its internal state before resuming the computation. Similarly, when the replication degree of a stateful operators is changed, the system should efficiently redistribute its internal state among all the active replicas of the operator. Performing these operations can temporar-

ily degrade the application performance, because, e.g., most existing DSP frameworks require to restart the application to adapt its deployment (e.g., Apache Storm [199]), thus introducing a downtime. We refer to these penalties as *adaptation costs* [AC] or reconfiguration costs. As regards adaptation costs, we can identify three main groups of works. The research efforts belonging to the first group assume that the application contains only stateless operators or, more generally, neglect the management overhead to handle the operators internal state during reconfigurations, i.e., they do not consider reconfiguration costs (e.g., [9, 34, 83, 132]). In this first group, there are also those approaches that deal with the state management but do not explicitly take into account the adaptation costs (e.g., [25, 65, 70, 145, 182, 223]). Zhou et al. [224] adopt a different perspective; they explicitly neglect adaptation costs, because they value more the long term benefits of (any kind of) reconfigurations. The second group indirectly considers the reconfiguration overhead. A few solutions (marked with (AD) in Table 2.3) indirectly minimize adaptation costs by limiting the set of possible reconfigurations [123], by reducing the amount of state to relocate during migrations [136, 139, 162], or by limiting the events that trigger adaptation [79]. Specifically, Heinze et al. [79] allow to apply reconfigurations only when an application is removed (so to consolidate and turn off virtual machines) and when a new application is added (so to reduce resource fragmentation) to the set of computing resources. Differently, Madsen et al. [136, 139] exploit semantic information about the DSP application to better support the management of window-based operators; this enables to efficiently avoid (or reduce) the overhead of state migration at runtime. The approaches belonging to the third group explicitly model the adaptation cost (marked with AD in Table 2.3). Here, some works only consider the number of deployment changes (i.e., migrations, scaling operations) [48, 47], whereas others predict the performance penalties for enacting the changes (e.g., application downtime, latency spikes), with the aim to enact only the less expensive reconfigurations [78, 95, 219] or to enforce constraints on it [82, 138].

System-oriented QoS metrics. As regards the system-oriented metrics, the solutions working at runtime optimize the metrics already introduced for the initial deployment, namely resource utilization, load balancing, inter-node traffic, and network usage.

Many research works consider the maximization of *resource utilization* [RU], which allows to better exploit the available computing resources and more efficiently run DSP operators [34, 50, 73, 103]. This metric is often used to steer the elastic replication of DSP operators [83, 132, 205] or the elastic allocation of computing resources [70, 71, 79, 80, 78, 82, 127, 202]. As a consequence, since these approaches indirectly optimize costs, they are usually employed in Cloud computing environments (e.g., [70, 78, 202]).

Differently from the above cited works, Han et al. [73] and Kalyvianaki et al. [103] do not consider replication, but rely on resource utilization to determine an efficient placement of the application operators. The same idea is exploited in [95], which, together with the operator placement, also aims to maximize the operator reuse among multiple applications. Chatzistergiou and Viglas [34] use the resource utilization metric to trigger reconfigurations of the application deployment and keep resource utilization below a given critical value at runtime.

A good number of approaches also optimize *load balancing* (LB) among computing resources [131, 138, 163, 168, 202, 224, 162, 222]. This metric is very important at runtime, because an even distribution of load among resources allows to better accommodate incoming load fluctuations (basically, a balanced load reduces the presence of few highly loaded computing nodes). Observe that most of the approaches that enforce load balancing have been designed to work in clustered environment (all except [131, 202]). Interestingly, Xing et al. [212] propose a solution that not only balances the average load among computing resources, but also minimizes the load variance among these resources. This deployment goal results in a placement configuration more resilient to load variations and traffic bursts. Some research results optimize load balancing among multiple replicas of the *same* operator [25, 60, 115]; this is a key property to effectively take advantage of data parallelism and let data experience homogeneous performance (e.g., waiting time) independently from the operator replica where they are processed.

The *inter-node traffic* [NT] measures the overall amount of data exchanged using the network per time unit. Since it does not explicitly account for network latencies, it is usually adopted by those approaches designed to work on a locally distributed pool of resources (as in [9, 95, 100, 123, 127, 138, 213, 222]). Chatzistergiou and Viglas [34] use this metric in settings with significant inter-node transfer latencies, even though they (implicitly) assume that network latencies are homogeneous among pairs of nodes. Differently, Heintz et al. [75, 76] propose to use this metric in combination with the minimization of response time, so to control network latencies in the geographically distributed setting. In these settings, many approaches are in favor of minimizing *network usage* [NT], which explicitly accounts for latencies of network links involved in the transmission of data streams [21, 25, 94, 158, 171].

Generic cost functions. A few other works recur to the definition of a generic cost function to be minimized (or utility function to be maximized) that can account for different QoS metrics (both user-oriented [91, 108, 116, 47, 145, 205, 219, 225] and system-oriented [102, 182, 209]). For example, Khorlin and Chandy [108] and Mencagli [145] rely on a generic, high-level formulation of a cost function to be minimized. A similar approach is also

adopted in [91, 116, 205], where however the cost function is expected to model application-related QoS metrics (i.e., throughput, response time). Interestingly, De Matteis and Mencagli [47, 48] propose an energy-aware deployment strategy tailored for multicore CPUs systems with frequency scaling support; together with user-oriented metrics, the deployment strategy considers energy consumption of computing resources. To conclude, it is worth mentioning the work by Zhou et al. [225], which considers DSP applications running on a single machine, where a deadline can be associated to each single tuple and all the operators may not run in parallel. The devised approach allocates tuples to each operator aiming to maximize the number of tuples that meet the deadline.

Deployment stages and deployment goals. Most of the existing works consider the runtime deployment adaptation as tightly correlated with the initial deployment of DSP applications. As such, these approaches optimize the same objective function while computing the initial application deployment and its runtime adaptation (e.g., [61, 75, 88, 158, 170, 215]). Although this is by large the most commonly adopted approach, a small set of solutions differentiates the goals of the two deployment tasks, thus recurring to different policies and optimization functions (e.g., [213, 224]). For example, the solution by Zhou et al. [224] computes the initial application placement by minimizing the communication latency between data producers and consumers, whereas, at runtime, it aims to maintain a good load balance despite the changes in the execution environment. Xu et al. [213] use a simple round robin strategy for determining the initial placement of DSP operators, whereas the developed (greedy) heuristic relocates operators at runtime, so to minimize inter-node traffic.

2.3.2 What: Controlled Entities

Span of Control. The vast majority of the existing solutions control the deployment of each single application independently from the other already running. In this case, the scheduler controls the deployment of a *single application* [SA] at a time. Most of the solutions existing in literature follow this design choice (e.g., [9, 61, 83, 97, 133, 167, 170, 202]).

Differently from the initial application deployment, here several solutions consider *multiple applications* in a cooperative [MA] or non-cooperative [MN] environment, while computing the deployment. In this case, the policies consider possible interactions among the applications that concurrently run on the same computing infrastructure. As regards the cooperative environment, some solutions consider the possibility of sharing operators (or data) among multiple applications [95, 103, 168, 215], whereas some others consider the possibility to control the deployment of multiple independent applications [79, 100, 131, 163, 209]. Even though not designed

for DSP applications, it is worth mentioning the approach by Delimitrou et al. [50] that estimates interferences among applications by running a set of micro-benchmarks before computing the application placement. Recently, also Hoseiny Farahabady et al. [90] have proposed a resource allocation strategy that explicitly quantifies the slowdown rate caused by multiple DSP operators running on the same host.

Differently from the above cited works, Mencagli [145] investigates how to implement elasticity when multiple applications run in a non cooperative environment. Here, applications are managed by distributed agents, where each agent independently optimizes the deployment goals of the managed application.

Granularity of Control. The vast majority of existing works adapt the application deployment at runtime by considering a *single operator* [O] (e.g., [9, 65, 88, 138, 172]) or *groups of operators* [G] (as in [34, 65, 70, 71, 182]) as a whole entity. Zhou et al. [225] propose a solution that works at the level of *tuples* [TP]; specifically, for each execution round, it schedules input tuples to operators aiming to avoid violations of application deadlines. Nevertheless, to reduce the scheduler overhead, the authors allow the scheduler to operate on batches of tuples. Among the approaches that consider self-adaptation, the ones by Madsen et al. [136, 139] require to perform some tasks before the application execution (i.e., at compile time). Specifically, exploiting the application semantics, it transforms stateful operators in a combination of a stateless component and a stateful one, so to more efficiently deal with migrations. To perform these operations, it works at the *topology* level [TO].

2.3.3 Who: Management Authority

The *who* question helps us to characterize the existing approaches with respect to the distribution of the management authority, i.e., the scheduler that adapts the application deployment. We distinguish between centralized and decentralized authorities.

A *centralized authority* [C] has access to the entire system current state, can potentially find a globally optimal deployment solution, but it may suffer from scalability issues. The vast majority of the existing approaches for runtime adaptation rely on a single centralized authority (e.g., [9, 82, 83, 138, 209, 223]).

Decentralized authorities [D] usually have a partial (or local) view of the system, can overcome scalability issues, but may not guarantee globally optimal solutions. Many (flat) decentralized solutions have been proposed so far [88, 108, 115, 142, 145, 158, 168, 170, 171, 172, 182]. For example, Kumbhare et al. [115] propose a solution with multiple infrastructure managers, which are organized in a peer-ring. Exploiting this structure, these man-

agers compute the operator deployment (i.e., placement and replication) exploiting the concept of consistent hashing. Distributed authorities also determine the placement in the work by Pietzuch et al. [158] and Rizou et al. [170, 171, 172]. Interestingly, Rizou et al. [171] show how their approach, although decentralized, can converge to a global optimum solution exploiting some mathematical properties of the objective function (i.e., the convexity of the network usage function). As such, if each operator iteratively finds its local optimal placement, the optimal placement for the application is achieved. Dealing with multiple applications in a non cooperative environment, Mencagli [145] has designed a decentralized elasticity policy, where agents manage applications and determine the best deployment relying on a game-theoretic approach. Differently from the above solutions, Hochreiner et al. [87, 88] propose to assign a manager to each DSP operator, which is in charge of performing the operator reconfigurations (in this case, horizontal scaling decisions). Kumar et al. [114] design a distributed operator placement algorithm that considers the infrastructure under different levels of abstractions. In such a way, the algorithm can manage large clusters while limiting its resolution time. First, it recursively aggregates the computing resources in high level groups; then, it determines the application deployment by proceeding from the highest abstraction level down to the lowest one that comprises the computing resources.

To the best of our knowledge, only the research work in [146] uses an architecture with *hierarchical decentralized authorities* [HD] for determining the runtime adaptation of DSP applications. In general, a hierarchical decentralized architecture can achieve a suitable trade-off between limitations and strengths of centralized and fully decentralized architectures. Within the hierarchy, the management authorities can be organized with separation of concerns and time scale. In [146], Mencagli et al. study the problem of parallelizing a special kind of windowed operators. By observing that workload variabilities occur at different time-scales, the authors design a two-level adaptation solution that controls load balancing across the operator replicas (at the lower level) and resource allocation through vertical elasticity (at the higher level). Since the proposed solution works in a single multi-core computing node, the vertical elasticity deals with changing the number of operator replicas at runtime.

Finally, we observe that some distributed deployment solutions rely on a centralized authority, which uses a centralized version of the deployment policy, so to determine the initial application deployment. This approach is useful to determine a good initial placement, relying on a global view of the system state [158, 170, 171, 172].

2.3.4 When: Adaptation Triggers and Time Strategies

In this section, we describe when the deployment operations and optimizations, proposed by the existing approaches, are applied. We use the *when* question to identify two main properties: triggering mode and time strategy.

Trigger. The adaptation trigger launches the deployment policy, which determines whether a reconfiguration is needed and, if so, plans the reconfiguration actions. To control adaptation, many solutions implicitly or explicitly rely on the MAPE loop [106], which represents a prominent and well-know reference model for organizing the autonomous control of a software system. In a MAPE loop, four components (Monitor, Analyze, Plan, and Execute) are responsible for the primary functions of self-adaptation. The Monitor component retrieves information about the application execution, the system state, and relevant environmental changes. By evaluating these data, the Analyze component is in charge of triggering the adaptation of the DSP application by running the Plan component. The latter hosts the deployment policy that computes an adaptation plan (e.g., [127, 202]). Finally, the Execute component accordingly runs the reconfiguration actions, so to enact the adaptation.

In general, the adaptation trigger can be either time-based or event-based. A *time-based* [P] trigger periodically executes the deployment policy. This strategy is simpler to be implemented than the event-based one, so it is by far the most diffused approach for executing the deployment adaptation policy (e.g., [9, 34, 31, 60, 88, 138, 167, 170, 219]). Nonetheless, determining the time interval between two consecutive executions of the deployment policy is critical. On the one hand, executing the policy with high frequency allows to quickly respond to system status changes; on the other hand, it may introduce a considerable management overhead and it might hide reconfiguration effects (this happens if the policy is faster than the system transients due to the reconfiguration enactment). The existing solutions consider a periodicity in the order of seconds or minutes. Most of these approaches avoid to recompute the deployment if it is not necessary. Specifically, in each round, they first check whether some QoS metrics have been violated and, in positive case, run the deployment policy (see for example [34, 61, 65, 71, 78, 82, 94, 115, 127, 133]). To increase stability after the enactment of a reconfiguration, some solutions recur to a grace period (or cool-down period), where the updated DSP operator cannot be further reconfigured (e.g., [80]).

An *event-based* [E] trigger executes the deployment policy as soon as an event is generated within the DSP system [79, 167]. Implementing this approach is harder, so many solutions fall back to a time-based approach, which periodically evaluates the generated events — as such, we classify

these works as time-based [P] solutions. A pure event-based approach is proposed by Heinze et al. [79], which recompute the applications deployment only when an application is added to or removed from the computing infrastructure. Similarly, Ravindra et al. [167] propose to perform scale-in operations to release virtual machines only when their billing time unit ends.

Time Strategy. The vast majority of the approaches are *reactive* [RE], i.e., they determine how the application deployment should be updated relying on the current system state (e.g., resource utilization, incoming data rate). This is usually the case of DSP systems running in an environment with unforeseeable incoming data rates, which should run deployment policies that can quickly react to changes (e.g., [65, 71, 131, 171, 179, 224]).

A second set of solutions uses a *proactive* [PR] approach, i.e., they determine the reconfiguration actions relying on a prediction of the system evolution. For example, these approaches try to exploit recurring patterns on the incoming data rate or, in general, to predict the system evolution in the near future [47, 48] (e.g., in term of incoming data rate [91, 97, 219], response time [142, 219], or network congestion [182]). For example, Farahabady et al. [91] propose a scheduler based on a Model Predictive Controller that allocates resources so to minimize QoS violation with respect to the predicted arriving data rate and resource utilization in each worker node. De Matteis and Mencagli [47, 48] rely on a control-theoretic method that takes into account the system behavior over a future time horizon in order to decide the best reconfiguration to execute (in terms of operators replication degree). Madsen et al. [137] have designed a checkpointing mechanism that allows fast state migration when changing the operator replication degree. To speed up the migration, they allocate the checkpoints so to maximize their probability to be used without relocation (i.e., they predict where computation will be located in the next future). Finally, Schneider et al. [182] consider the problem of distributing load among multiple operator replicas; to avoid congestion, they minimize the predicted blocking rate per each TCP connection among communicating operators that reside on different computing nodes.

Interestingly, some few works propose an approach that combine a reactive strategy and a proactive one [83, 178]. The elasticity control algorithm by Hidalgo et al. [83] includes a reactive and a proactive algorithm, which work at two different time scales. The reactive short-term algorithm evaluates the incoming data rate and changes the operator parallelism with a fine granularity. The proactive mid-term algorithm uses a Markov chain to predict the operator load in the next time window, so it can accordingly adjust the operator parallelism with a coarse granularity.

2.3.5 Where: Computing Infrastructure

Relying on the taxonomy of Figure 2.3, we can describe in detail the computing infrastructure used by the existing solutions in literature. Together with the three main properties that describe the infrastructure (i.e., computing environment, resource heterogeneity, and resource distribution), as introduced for the initial deployment solutions, here we add a fourth property, namely the adaptation level.

Adaptation Level. The adaptation level identifies whether the deployment solution operates at the application level, at the infrastructure level, or uses a combination thereof.

A policy that computes *application-level* adaptations (referred as [A] in Table 2.3) manipulates only the application (and its operators), but does not change the set of available computing resources. In other words, it considers the pool of computing nodes to be statically defined (i.e., resources cannot be acquired and released at runtime). Usually, the approaches designed for clustered environments investigate only application-level adaptations. For example, they can change the operator replication degree (e.g., [48, 179, 205]), the replica placement (e.g., [9, 34, 100]), load distribution (e.g., [25, 111, 222]), or fault-tolerance mechanisms [18] (as we will describe in Section 2.3.6). We observe that, although several solutions investigate scaling operations (i.e., how to change the operators replication degree), they work at the application level and do not acquire or release computing nodes (e.g., [60, 61, 65, 83, 132, 145, 162, 214, 219]).

A policy determines *infrastructure-level* adaptations (referred as [I]) when it dynamically changes (e.g., resizes) the set of computing resources. Pure infrastructure-level solutions do not necessarily change the application deployment; indeed, they can postpone the utilization of new resources for accommodating new upcoming applications. Investigating this kind of solutions is out of the scope of this thesis work, however we mention a couple of representative approaches. Cerviño et al. [32] proposed a solution that autonomously resizes the number of virtual machines in response to variations of the input streams rates. Their approach aims to improve resource utilization while running low latency applications. As soon as a new virtual machine is acquired or released, the incoming workload is equally balanced across the active virtual machines. Other solutions combine a local cluster with a remote Cloud; the latter is used only when the incoming load exceeds the cluster capacity. In such a case, the application deployment is recomputed [97, 167].

A policy can also determine adaptation both at the application and at the infrastructure level, thus changing the application deployment (e.g., operator replication, placement) on an elastic pool of computing resources. We mark these solutions with the label [A, I]. Many existing solutions con-

sider this setting and adapt the operators replication degree, while expanding or shrinking the pool of computing resources so to accommodate the incoming workload (e.g., [31, 71, 79, 80, 78, 88, 109, 131, 202]). These approaches are well suited to work in a Cloud computing environment.

Computing Infrastructure. We now analyze how the different research efforts consider the computing infrastructures, in terms of heterogeneity, distribution, and execution environment.

As regards resource heterogeneity, the currently existing approaches are equally distributed among those that consider *heterogeneous resources* [HE] (e.g., [61, 88, 111]) and those that consider *homogeneous resources* [HO] (e.g., [25, 214, 202]). Observe that the approaches that deal with elasticity, which horizontally scale DSP operators, usually work with homogeneous resources (as we will see in the next section — e.g., [60, 65, 71, 83, 219]).

The vast majority of research works consider a *locally distributed* [LO] infrastructure, where computing resources are co-located in the same data center and are inter-connected with high-speed communication links (e.g., [123, 132, 65, 82]). As such, these approaches usually neglect the transmission overhead (e.g., latency) introduced by the network. Conversely, few works consider a *geographically distributed* [GE] infrastructure, thus explicitly modeling networks costs in terms of latency or bandwidth (e.g., [34, 61, 200, 158, 171]). For example, Saurez et al. [178] present Foglets, a framework designed to work in a Fog computing environment, which takes into account the heterogeneity of computing and network resources. We point out that the work by Heintz et al. [75] considers homogeneous resources, albeit it runs in a geographically distributed environment. Indeed, it focuses on the optimization of windowed aggregations with the goal of minimizing the exchanged traffic and response time (more precisely, the staleness).

We consider three main categories of computing environment: *cluster* [CU], *Cloud computing* [CC], and *Fog computing* [FC]. Each of them has specific features in terms of resource capabilities, dynamism, and elasticity. Many solutions have been designed for cluster environments, even though recently an ever increasing number of research efforts are tailored to exploit the elasticity of Cloud computing (e.g., [31, 60, 78, 83, 88]). At present, solutions that efficiently work in Fog computing (or, in general, distributed Clouds) environment are, to the best of our knowledge, still largely unexplored.

The vast majority of existing solutions designed for a Cloud computing environment consider locally distributed infrastructures (with both homogeneous resources, e.g., [65, 71, 83, 123, 132, 145, 215] and heterogeneous resources [61, 79, 88, 131, 115, 116]). In this setting, Hochreiner et al. [88] investigate the elasticity of DSP operators, whereas Floratou et

al. [60] define a load balancing approach for distributing the computation among replicas of the same operator. Differently, Heinze et al. [82] combine multiple approaches for fault tolerance (i.e., active replication, upstream backup), while meeting requirements on recovery time. Relatively few works for locally distributed Cloud computing environments investigate the operator placement problem and the operator co-location problem, e.g., [127, 215]. In this setting, the majority of the solutions investigate elasticity. Conversely, we observe that approaches designed for geographically distributed Clouds mainly focus on the operator placement problem and explicitly model resource heterogeneity (as in [61, 116, 200]). In this case, few works consider elasticity.

As regards the clustered environment, the approaches mainly consider locally distributed infrastructures with both homogeneous (e.g., [48, 91, 108, 139, 163, 225]) and heterogeneous resources (e.g., [9, 50, 111, 142, 213, 222]). The approaches dealing with geographic distribution often consider resource heterogeneity; this allows to efficiently utilize the available computing and, most importantly, network resources while computing the application placement (e.g., [34, 158, 170, 171, 172]).

Finally, some few approaches consider a hybrid architecture, where the public Cloud extends a local cluster when the incoming load demands capacity is higher than the available cluster resources [97, 109, 167]. Most the solutions propose a view of the Cloud as a set of infinite, homogeneous resources interconnected to the cluster with negligible network latencies.

2.3.6 How: Actions and Methodologies

Actions. Actions represent the tool-set that can be used to change the application deployment. The set of actions used at runtime extends those adopted during the initial application deployment (presented in Section 2.2), namely graph transformation, operator reuse, operator placement, and operator co-location. Furthermore, at runtime, several research efforts use the following additional actions: scaling the operator replication degree (elasticity), adaptation of fault-tolerance mechanisms, dynamic batch sizing when transferring data, load balancing, and load shedding.

Graph transformations [GT] are only considered in Enorm by Madsen et al. [136, 139]. Enorm improves the runtime adaptation of window-based stateful operators by exploiting instant and parallel track migrations. To enable these enhanced techniques, at compile time, Enorm exploits the application semantics and replaces the stateful operators with a combination of stateless components and, if needed, lightweight stateful components.

Operator reuse [RS] (or redundancy elimination) avoids redundant computations by reusing already deployed operators [95, 103, 168]; this optimization can be applied when multiple applications use a predefined set of

operators rather than UDFs. At runtime, if the current deployment cannot satisfy the application requirements, a new instance of the operator can be deployed on a new computing node. On the other hand, if many operator instances lead to resource wastage, their number can be reduced, increasing reuse by multiple applications. For example, Kalyvianaki et al. [103] propose a query planner for CEP applications that allocates operators with reuse. The planner relies on an optimization problem, tuned to maximize the number of satisfied queries while minimizing resource utilization.

The *operator placement* [OP] is by large the most common action exploited by the existing solutions (especially for cluster environments, e.g., [9, 108, 138, 158, 171]). Changing the operator placement at runtime allows to better exploit the available resources, e.g., by balancing load among computing nodes or by moving operators to other nodes in order to consolidate resources or avoid their overload. The operator migration is the mechanism that allows to change the operator placement at runtime. Relocating a stateless operator requires only to start a new operator instance on the new location. Conversely, when a stateful operator should be relocated, the DSP system should also efficiently migrate the operator internal state. Performing these migration operations may introduce adaptation costs that can be expressed, e.g., in terms of application downtime [47, 48, 79, 136, 138, 139]. We further discuss about stateful operations later in this section. Similarly to the initial deployment, several approaches solve the *operator co-location* [CL] problem so to minimize the amount of data exchanged using the network [9, 25, 34, 127, 133, 136, 139, 138, 200]. At runtime, the operators forming a co-location group can be changed in order to keep satisfying the deployment goals. Therefore, some approaches try to make groups (statistically) more resilient to execution condition variations, i.e., they aim to reduce the number of member reassignments to groups over time [34].

The *operator scaling* [OS] allows to increase or reduce the amount of computing resources to execute an operator; we can distinguish two main categories: horizontal and vertical scaling. The most popular approach relies on horizontal scaling, which changes the number of operator replicas at runtime, thus enabling to increase or reduce the amount of data that can be processed in parallel on distributed nodes. Specifically, a *scale out* operation increases the number of operator replicas, whereas a *scale in* operation reduces the number of replicas. When the application receives increasing workloads, a higher number of operator replicas can exploit computing resources available on different nodes — thus increasing the amoung of data processed in parallel (e.g., [48, 61, 65, 71, 83, 179, 219]). Although a high parallelism can appear to be always preferable, we observe that it may introduce a management overhead, e.g., to preserve tuple ordering [65], and result in higher execution cost, e.g., [88, 132, 145]. Vertical scaling keeps the number of operator replicas constant, but adds resources to or removes

resources from the operator (*scale up* and *scale down* actions, respectively). Performing vertical scaling operations requires to control the amount of resources assigned to each operator; e.g., this can be done using threads, cgroups[8], or other virtualization techniques. To the best of our knowledge, only few works explore vertical scaling operations [140, 179, 210]; they leverage on a thread pool that can be scaled up/down at runtime.

The decisions about placing the operators of a DSP application and determining their replication degree can be either taken as independent and orthogonal decisions or can be jointly considered. Most works in literature consider DSP operator placement and replication as independent and orthogonal decisions, where the placement is first carried out without determining the optimal number of replicas for each operator. Then, in response to some performance deterioration, the operators to be replicated and their new replication degree are identified (e.g., [78, 179]). This two-stage approach requires to reschedule the DSP application in order to enact the new application configuration (thus possibly introducing downtime). To the best of our knowledge, no work considers a joint optimization of operator replication and placement (i.e., in a single stage). Nevertheless, an interesting solution has been proposed by Madsen et al. [138]; in the same optimization round, their approach first fixes co-location groups, and then determines replication and load balancing among these co-location groups.

Another action is the on-line adaptation of *fault-tolerance mechanisms* [FTM]. Two main mechanisms have been adopted so far: active replication and upstream backup. With active replication[9], the DSP system deploys two (or more) identical instances of the same operator on different computing nodes, which process the same input in parallel and generate the same results. In case of failure, the additional replica ensures an immediate take over of the processing. Active replication can be expensive, because it requires multiple resources to perform the same computation. Liu et al. [129] present mechanisms to preserve the operator state in the case of failures for Apache Storm; these mechanisms implement an active replication scheme. After a failure, to restore the number of active replicas with a consistent operator state, the authors propose an asynchronous recovery protocol, which allows to reconstruct the replica state in parallel and in background. With the idea that perfect fault tolerance is not always required (by most applications) while it is more important to effectively manage temporary load variations, Bellavita et al. [18] present a method that sacrifices fault tolerance (through active replication) for increased capacity during load spikes. With

[8] `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`

[9] We observe that the terms *replication* and *replica* are ambiguously used in literature. They can refer to data parallelism, when each operator instance processes a subset of the incoming data, but they can also refer to replication for fault tolerance, when multiple identical instances of the same operator process the same data and produce the same results. Throughout this thesis, we use "replication" and "replica" meaning data parallelism.

upstream backup, the DSP system periodically creates a checkpoint of the current operator state. In case of failure, the checkpoint is used to restore the operator and resume computation (spending a time interval known as recovery time). Heinze et al. [82] consider a replication scheme that combines active replication and upstream backup, with a twofold objective: to minimize the number of violations of a user-defined recovery time threshold and reduce resource utilization with respect to active replication. Castro Fernandez et al. [31] also use an upstream backup approach; their solution supports the explicit state management of stateful operators, allowing the DSP system to checkpoint, backup, restore, and partition it as needed. ChronoStream, by Wu et al. [210], introduces a distributed checkpoints protocol that works with reduced size slices (i.e., checkpoint segments), which allow to perform data recovery, as well as stateful migrations, in parallel.

The remaining actions (i.e., dynamic batch sizing, load balancing, load shedding) deal with the efficient management of data, so to reduce the communication overhead, change data distribution, and discard data in case of overload.

Dynamic batch sizing [DBS] allows to reduce the communication overhead by sending multiple tuples as a single batch. The approaches that investigate dynamic batch sizing deal with changing the window size at runtime, so to maximize the application throughput [75, 133, 200, 223]. As observed by Zhang et al. [223], determining the batch size needs to address the following principle: the processing time of a batch increases with the batch size, whereas the communication overhead decreases with the batch size. The solution by Zhang et al. [223] finds a trade off between batch size and operator replication so to optimize the application response time. Similarly, JetStream [200] achieves high throughput using adaptive streams, which dynamically change the batch size and the number of parallel communication connections. Das et al. [44] investigate how to dynamically adapt the batch size in micro-batched stream processing systems (e.g., in Spark Streaming). Indeed, these systems require that the batch processing time must not exceed the batch interval (i.e., each batch should be completely processed by the time next batch arrives). The same problem has been also investigated in [204].

Load distribution [LD] actions enable to evenly distribute load among multiple replicas of the same operator [60, 142, 150, 182], among heterogeneous resources [111], or among different computing environments [109]. When an operator is instantiated with multiple replicas, their upstream operators need to accordingly route data towards the replicas. For stateless operator, this routing can be accomplished in a round-robin fashion without compromising the application semantics. Observe that more advanced shuffle routing techniques also exist that dynamically adjust load distribution, with the aim to reduce the application response time (e.g., [169]). In

case of stateful operators, only partitioned stateful operators are amenable to data parallelism: each replica keeps a state on a sub-stream basis, where each sub-stream is identified by a partitioning key. Importantly, for partitioned stateful operators, data of the same sub-stream must be handled by the same operator replica; therefore, the routing cannot be performed using a round-robin policy. To preserve the application integrity, usually a consistent hash function on the partitioning key is used to route data to the replica in charge of its sub-stream [181]. Dhalion [60] includes a policy that evaluates the average number of packets waiting for processing, with the aim to improve load distribution among multiple replicas of the same operator. To this end, the policy updates a hash-based routing function. Nasir et al. [150] propose partial key grouping, an elegant solution relying on two hash functions where a key can be sent to two different replicas instead of one. Although widely used in distributed systems, hash functions may not distribute the workload evenly, due to the presence of heavy keys, i.e., keys that appear with high frequency. Gedik [63] proposes to resolve the stateful load balance problem with a small routing table. To preserve load balancing, this routing table is updated at runtime. A similar idea is also exploited by Fang et al. [56]. They use a routing table of limited size, which only maps highly frequent keys, together with a hash function, which is applied to route all the other keys. To deal with short-term workload variations, the routing table is updated at runtime. In some applications, we also need to preserve data ordering also in face of parallelization [181]; therefore, the application can experience aggregation costs which result from the effort of combining results after a parallelized operator. Katsipoulakis et al. [105] demonstrate the need to incorporate these aggregation costs in the partitioning model, together with even load distribution. Differently, Mayer et al. [142] propose to assign as many subsequent data windows as possible to the same operator replica, until its operational latency reaches a threshold; then, the next operator replicas (selected with a round robin strategy) is considered. Considering the distribution among heterogeneous resources, SABER [111] runs window-based streaming SQL queries on servers with heterogeneous CPU and GPGPU processors. It assigns tasks to the heterogeneous processor that, based on past behavior, achieves the highest throughput. To ensure ordering among batches processed by the different processors, some coordinator operators are introduced. A combination of a local cluster and the Cloud is considered by Kleiminger et al. [109], who propose a solution to adaptively balance the DSP application workload between the two environments.

When the DSP system is overloaded and no new computing resource can be acquired, *load shedding* [SH] helps to provide a best-effort service by dropping a fraction of tuples from the input stream. Kalyvianaki et al. [102] design a load shedder that randomly discard tuples so as to control the av-

erage application response time in periods of resource overload. The same research group also proposes, in [102], a load shedder that tries to fairly preserve processing quality under overload conditions. The proposed solution quantifies the perceived quality contribution of a tuple for each managed application, so that, during overloading periods, the load shedder can discard tuples by fairly degrading quality of all managed applications (i.e., without compromising the quality of only few applications).

State Management. Changing the operators deployment at runtime requires to consider whether the operator is stateful, that is it maintains an internal state to properly emit results. In such a case, when the deployment changes, the operator internal state should be relocated (or redistributed) accordingly. For example, while scaling or migrating stateless operators can be achieved by just turning off/on or moving operator replicas, elasticity of stateful operators requires state migration and repartitioning among the replicas, because the system needs to preserve the consistency of the operations [65]. State management in DSP systems is nicely surveyed by To et al. [198].

Operator state migration is a challenging task, because it should be application-transparent and with a minimal footprint (i.e., amount of migrated state). The most common solutions are the *pause-and-resume* approach and the *parallel track* approach [78]. In the pause-and-resume approach, the current state is extracted from the old operator instance, which is paused to ensure a semantically correct migration; then, the state is moved to the new instance and the buffered tuples are rerouted and replayed within the new instance. Its drawback is a peak in the application latency during the migration. To identify the portion of state to migrate, Castro Fernandez et al. [31] expose an API to let the user manually manage the state, whereas Gedik et al. [65] automatically determine, on the basis of a partitioning key, the optimal number of state partitions to be used and to migrate. ChronoStream [210] natively supports stateful migrations and uses a lightweight protocol that leverages on distributed checkpoints to minimize the amount of state relocated during a migration. To improve the efficiency of the pause-and-resume approach, some works [31, 137, 210] exploit the checkpoints that are already backed up for failure recovery. A checkpoint allocation problem is defined by Madsen et al. [137] with the goal to maximize the checkpoints reuse for migration.

To migrate in a more gradual fashion, in the parallel track approach, the old and the new operator instances run concurrently until the state of both is synchronized and the new instance can safely take over. While this approach does not entail a latency peak, it requires enhanced mechanisms [31], e.g., to avoid incorrect results, which can increase the cost of state migration. Furthermore, this approach is more suitable for window-based operators (e.g., [139, 157]). This latter work also models the time cost

of the two approaches for partitioned stateful operators.

To avoid the overhead of stateful migrations, some solutions rely on an externalized state (e.g., [87, 88]). In this case, the operator state is stored in a shared memory that is accessed by the operator replicas to perform the state updates. Nevertheless, this approach is not viable in geo-distributed environments.

Methodology. The *methodology* identifies the class of algorithms used to plan how the deployment should be changed so to achieve the deployment goals. With respect to the initial deployment, here solutions rely on a broader set of methodologies, that we classify in the following categories: mathematical programming, control theory, game theory, graph theory, queuing theory, machine learning, and heuristics. The latter is further specialized into classes of heuristics with similar properties, namely greedy (e.g., first-fit, best-fit), local search, tabu search, genetic algorithm, and threshold-based policies.

The *mathematical programming* [MP] approaches exploit tools from operational research in order to compute the operator placement [95, 97, 103, 171, 209], to change the operator parallelism [132, 138], and to decide which fault tolerance mechanisms should be adopted [18]. For example, Ishii and Suzumura [97] recur to an ILP formulation to combine a local cluster with a remote Cloud with the aim to reduce execution costs. Hummer et al. [95] formalize the placement of CEP operators as a multi-objective optimization problem that balances load distribution, minimizes inter-node traffic, and maximizes operator reuse. To speedup the computation, they also propose a heuristic based on a neighborhood search on the solution space. Similarly, Kalyvianaki et al. [103] have developed a planner for CEP applications, modeled as an ILP problem, that also performs admission control; in this case, the deployment objective is to maximize the number of admitted applications. At runtime, it can re-allocate the applications whose resource consumption deviates from initial estimation; nevertheless, we observe that the reallocation is performed in a simple — and possibly not efficient — manner, i.e., by removing and then re-adding the application to the cluster of resources (which results in a new computation of its deployment). An interesting solution has been proposed by Rizou et al. [171] for the operator placement problem. They exploit the mathematical properties of the network usage function (i.e., convexity) in order to find the global optimum solution in a completely decentralized manner: if each operator iteratively finds its local optimal placement, the optimal placement for the application is achieved. As regards the runtime operator replication, Lohrmann et al. [132] propose a strategy that aims to satisfy requirements on the application response time while minimizing resource consumption. The proposed approach first predicts operator response time relying on a queuing model and then finds the replication degree relying on a gradient descent method

with variable step size. Nevertheless, this solution manages only stateless DSP applications. The state migration overhead is considered by Madsen et al. [138], who formulate a MILP optimization problem to control load balancing and horizontal scaling, with requirements on the maximum admissible downtime for migrations. The optimization model works in combination with a heuristic that computes co-location (so to reduce inter-node traffic). This approach does not consider network delays among computing nodes. Bellavista et al. [18] present an optimization model that trades fault tolerance for increased capacity during load spikes. To this end, it can dynamically deactivate/activate redundant replicas of DSP operators in order to claim/release resources and accommodate temporary load variations. In this setting, the optimization problem minimizes execution costs while meeting fault-tolerance requirements. The main drawback of these approaches, that rely on integer or non linear formulations, is scalability. Indeed, as we have shown [29], the deployment problem is NP-hard and resolving the exact formulation may require prohibitive time when the problem size grows.

Few research works rely on *control theory* [CTR] to adjust the operator placement [108] or the operator replication degree [47, 48]. In this case, the policy usually identifies three main entities: disturbance, decision variables, system configuration. According to the existing solutions, the disturbances represent the events that cannot be controlled; nevertheless, their future value can be predicted (at least in the short term), e.g., incoming data rate, load distribution, and processing time. The decision variables identify the placement or replication of each operator. By combining the decision variables, alternative configurations of the application deployment can be obtained, which result in different performances, e.g., in terms of application latency or throughput. Das et al. [44] rely on a control algorithm for dynamically adapting the batch size in micro-batched stream processing systems. The control algorithm aims to maintain queuing delay low in face of changing working conditions by reducing the batch size; in such a way, the system improves its stable at high data rate. In micro-batched processing systems, the stability condition requires that the batch processing time must not exceed the batch interval. Khorlin and Chandy [108] propose a decentralized policy that exploits concept of distributed feedback control to adapt the operator placement at runtime, with the aim to maximize an utility function (whose definition is generic). Differently, De Matteis and Mencagli [47, 48] propose a proactive strategy for realizing elastic DSP applications. Their control-theoretic method takes into account the system behavior over a limited future time horizon in order to choose the reconfigurations. A similar approach is also proposed by Hoseiny Farahabady et al. [90], who devised a resource allocator for DSP systems. Their solution is based on model predictive controller (from control theory) that aims

to achieve a good utilization of computing resources and a reduced average response times, while satisfying the QoS application requirements. Recently, Mencagli et al. [146] have proposed a hierarchical approach for parallelizing windowed operators so as to efficiently handle changing workloads. The lower level policy handles load burstiness by dynamically adjusting load balancing across the available CPU cores. It relies on a control theoretic approach that operates on a short time scale. The higher level policy handles slow workload variabilities by changing the parallelism degree of the windowed operator. This policy uses the Fuzzy Logic Control paradigm [120], which allows to design model-free controllers suitable for systems with complex dynamics. The critical point of the control-theoretic approaches is that, to be efficiently adopted, they require a good model of the system, which nevertheless can be difficult to be formulated (e.g., when the decision variables inter-play in a complex manner). The literature review by Shevtsov et al. [184] shows that, although research on control-theoretical software adaptation is still in a preliminary stage, in recent years, an ever increasing number of solutions apply control theory to realize self-adaptive software systems.

Mencagli [145] proposes a *game-theoretic approach* [GAM] for changing the operator parallelism degree, when multiple applications work in a non cooperative environment (i.e., they compete for resources). As such, distributed agents perform local control strategy so to pursuit their own interests, but have to wisely interact so to maximize the social welfare. The agreement among agents follows the concept of Nash equilibrium, which aims to fairly maximize the utility function of every agent. Game theoretic approaches represent a nice solution for modeling settings with non cooperative applications that ask for resources. Nevertheless, properly designing a distributed policy that exploits results from game theory and converges in a reasonable way is not an easy task and, so far, this methodology is largely unexplored.

The approaches based on *graph theory* [GRA] are usually adopted to determine the operators placement. Specifically, they partition the application topology in groups of operators to be allocated on different computing nodes, with the aim of reducing the inter-node traffic while balancing load among nodes [25, 100]. An example of this solution has been proposed by Jiang et al. [100]; it periodically recomputes the topology partitions and reassigns operators if needed. Nonetheless, it considers only stateless operators. Differently, Caneill et al. [25] improve stream locality for stateful DSP applications: their solution uncovers correlations between the keys used in successive routing operations and assigns these operators to the same node (thus reducing inter-node traffic). As observed in Section 2.2.6, this methodology can help in determining the operator placement, but it cannot be easily employed when other actions should be determined (e.g.,

replication).

Queuing theory [QT] is often used to predict the response time of an operator with respect to its replication degree [61, 132] or its placement [108]. The key idea is to model the operator as a queuing system with inter-arrival times and service times having general statistical distributions (this assumption might require to approximate the system behavior). The mostly used queuing models are M/M/1 [195], M/M/k [61], and G/G/1 [48]. For example, Fu et al. [61] model the relation between the application response time and the provisioned resources (and, in turn, the operator replication degree) as an M/M/k queuing system, relying on the theory of Jackson open queuing networks. After having defined the operators replication degree with the queuing model, the policy assigns processors to replicas so to minimize the application response time. Lohrmann et al. [132] use a mathematical programming approach to elastically scale the operator replication degree, which relies on a queuing model to predict the operator response time (with respect to its parallelism degree). Moreover, Tesauro et al. [195], who investigate the resource allocation problem, have proposed a solution that combines on-line reinforcement learning and queuing models in a hybrid approach. While the learner exploits experience to improve the adaptation policy, the queuing model is used to determine the initial allocation as well as to drive exploration by estimating the system performance. In general, queuing theory is well suited to determine the replication degree of DSP operators or predict operator performance. Nevertheless, it often requires to approximate the system behavior so to apply models from the established theory (e.g., in terms of incoming workload distribution, serving rate). Therefore, when the system is very complex or not stationary, also the queuing theory becomes complex, discouraging its adoption.

OrientStream, by Wang et al. [205, 206], is a framework for dynamic resource allocation that relies on *machine learning* [ML]. Firstly, it uses different workloads as training set to predict the operator demand for resources; then it determines the best application deployment while taking into account the application requirements. In this work, the best deployment configuration is the one that minimizes CPU and memory utilization while meeting requirements on the application throughput and response time. Although conceptually easy to design, machine learning techniques suffer from two main drawbacks. First, they require reasonably big training sets. Second, they cannot easily and rapidly address unforeseen configurations.

Reinforcement Learning [RL] is a special method belonging to the branch of machine learning. It refers to a collection of trial-and-error methods by which an agent can learn to make good decisions through a sequence of interactions with a system or environment [80, 195]. In the context investigated by this thesis, reinforcement learning techniques deal with the elastic replication of DSP operators. These techniques learn from experience the

adaptation policy, i.e., they learn the best scaling action to take with respect to the system state through a trial-and-error process. The system state can consider the amount of incoming workload, the current application deployment, its performance, or a combination thereof. After executing an action, the policy gets a response or reward from the system (e.g. performance improvement), which indicates how good that action was. One of the challenges that arise in reinforcement learning is the trade-off between *exploration* and *exploitation*. To maximize the obtained reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward (exploitation). However, in order to discover such actions, it has to try actions that it has not selected before (exploration). The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best [193]. To the best of our knowledge, only a couple of works [35, 80] have so far exploited reinforcement learning techniques to drive adaptation decisions in DSP systems. Heinze et al. [80] propose a simple reinforcement learning approach that learns from experience when to acquire and release computing nodes so to efficiently process the incoming workload. It populates a lookup table that associates the node utilization with the best action to perform (i.e., scale in, scale out, or do nothing). The learner objective is to keep the system utilization within a specific range. Similarly, Cheng et al. [35] propose an adaptive scheduler for Spark Streaming, which dynamically changes the execution parallelism of concurrent applications (i.e., jobs). A larger number of works has exploited reinforcement learning techniques to drive elasticity in the Cloud computing context, as surveyed in [135]. Tesauro et al. [195] observe that reinforcement learning approaches can suffer from poor scalability in systems with a large state space, because the lookup table has to store a separate value for every possible state-action pair. Moreover, the performance obtained during on-line training may be unacceptably poor, due to the absence of domain knowledge or good heuristics. To overcome these issues, they combine reinforcement learning with a model of the system, defined using queuing theory, which computes the initial deployment decisions and drives the exploration actions. Reinforcement learning exposes a very interesting approach that is capable of learning the best deployment actions, even when a precise model of the system is not known (this property is very powerful). The main limitation of this solution is the slow convergence time that is required to determine acceptabily good decision policies [135].

A couple of solutions rely on *genetic algorithms* [GA] to compute the operator placement and to switch the operators' logic at runtime. Yang et al. [215] consider mobile DSP applications and investigate the problem of placing a subset of operators to the Cloud in a such a way that the resulting

application throughput is maximized. In this solution, chromosomes represent the different ways of partitioning the application, whereas the resulting application throughput is the fitness function. Interestingly, Kumbhare et al. [116] propose to define multiple alternative implementations for each operator, so that each implementation has different performance characteristics, e.g., in terms of resource requirements and result quality. At runtime, only a single implementation can be up and running. In this solution, chromosomes represent the deployment configurations in terms of operator implementation, type and number of active virtual machines, and mapping of operators to these virtual machines. Here, the fitness function depends on whether the throughput requirement is violated and on the number of deployment changes. We discussed in Section 2.2.6 about the critical points of this approach and [116] nicely shows a suitable use case for it. We also observe that this methodology cannot easily be used to decide about other deployment actions (e.g., replication).

Most of the existing approaches are *greedy* [GR] heuristics that compute the operator placement [9, 73, 79, 94, 100, 127, 163, 213] or distribute load among replicas or resources [142]. For example, Aniello et al. [9] sort pairs of communicating operators in decreasing order of exchanged data rate and then assign each pair on the set of computing resources relying on a greedy first-fit heuristic. Xu et al. [213] build on [9] by considering a single operator at a time. Heinze et al. [79] consider the placement problem as a bin-packing problem, therefore they rely on a greedy first-fit approach whose goal is to minimize resource utilization. Jiang et al. [100] assign groups of operators (i.e., topology partitions) on computing nodes with the aim to reduce the resulting inter-node traffic. Huang et al. [94] first model the relationship between the operator execution time and the amount of residual computing capacity on a node, and then propose a best-fit heuristic that aims at minimizing the network usage. At runtime, these approaches recompute the placement from scratch and, if a new deployment is determined, it is enacted. A greedy approach is also followed in S-Storm [163], which periodically moves groups of operators from overloaded to underloaded nodes, with the aim to balance load among cluster resources. As regards load distribution, Mayer et al. [142] observe that processing overlapping windows on multiple operator replicas increases computation and communication overhead. Therefore, they propose to greedily assign as many subsequent windows as possible to the same operator instance until the operational latency reaches a critical value. Afterwards, windows are routed to the next operator replica. Stela [214] supports scaling operations when users request them. When the user requires a scale-out with a given number of new machines, Stela identifies and replicas the bottleneck operator that leads to the highest throughput improvement. Similarly, when a scale-in operation is requested, Stela removes the ma-

chine that hosts operator replicas which contribute less to the application throughput (replicas are then re-assigned to the remaining machines in a round robin fashion).

A few solutions rely on a *local search* [LS] on a subset of possible deployment configurations [50, 91, 224]. For example, after having defined an initial placement that minimizes communication cost between data sources and consumers, Zhou et al. [224] focus on preserving load-balancing at runtime. The latter is performed using a receiver-initiated strategy: periodically, a node greedily identifies the neighbor node with highest load and, if needed, generates a workload distribution request so to balance load. Interestingly, JetStream [200] combines dynamic batch sizing with another optimization practice, which exploits multiple and parallel communication links to transfer data among data centers. Since different combinations of window size and parallel communication links yield different performance, the proposed policy greedily explores the solution space by fixing a dimension and exploring the other, until no improvement can be found.

Many solutions exploit best-effort *threshold-based* [TB] policies to change the operator replication degree (e.g., [65, 83, 88, 109]) or to recompute the operator placement at runtime (e.g., [167, 191]). The main idea is to increase (or reduce) the operator parallelism degree or to change the operator placement as soon as a QoS metric is above (or below) a critical value. Several works use as QoS metric the utilization of either the system nodes [115, 95, 167] or the operator replicas [31, 70, 71, 88, 123, 131, 202]. Gedik et al. [65] use the throughput and network congestion. Hidalgo et al. [83] rely on a load metric, defined as the ratio between the number of incoming events to all the operator replicas and the amount of events the replicas can theoretically process. The policy by Heinze et al. [78] performs scaling operations only if they result in a latency spike below a predefined threshold. In [82], the same authors have designed a policy that combines two fault tolerance mechanisms: the policy switches from upstream backup to active replication so to not violate a user-defined recovery time threshold; this approach aims to reduce resource utilization with respect to active replication. Kleiminger et al. [109] activate Cloud computing resources and maximize the application throughput as soon as the incoming queue length of an operator reaches a critical value. The same metric is used by Li et al. [123] to scale out an operator. Hochreiner et al. [88] combine queue length and resource utilization so to elastically allocate resources to operators. We also observe that different approaches can be identified for the threshold definition. A single statically-defined threshold is used, e.g., in [71] to limit load unbalance among computing nodes. Multiple statically-defined thresholds are used, e.g., in [88] to customize the behavior of each individual DSP operator. A dynamically set threshold improves the system adaptivity, and Heinze et al. [80] have shown how a reinforcement learn-

ing approach can be used to dynamically adapt the thresholds. Threshold-based policies are very popular for reassigning and scaling in/out DSP operators at runtime. They can be easily implemented and very often they work sufficiently well. Most of the Cloud service providers, e.g., Amazon, Google, use this approach for driving elasticity. However, this is a best-effort approach that provides no guarantees about the reconfiguration optimality. Furthermore, it moves complexity from determining the reconfiguration strategy to the selection of critical values that act as thresholds.

With *heuristic* [H], we cover all the other approaches that adopt custom solutions to solve the deployment adaptation problem at runtime. Several solutions adapt the operator placement [138, 136, 139, 158, 168, 170, 172, 200, 222], elastically scale the operator replication degree [179, 60], and deal with the incoming load (performing load distribution [60, 111, 115, 182, 225], dynamic batch sizing [75, 133, 200, 223], or load shedding [102]). For example, as regards the operator placement, Pietzuch et al. [158] represent the application as an equivalent system of springs: operators are massless bodies tied together by springs that represent the exchanged streams. The stream data rate and network latency determine the stretching of the springs. The network usage is indirectly minimized by finding the assignment that minimizes the overall elastic energy of this equivalent system. Zhang et al. [222] propose a scheduling policy for Apache Storm that minimizes inter-node traffic and maximizes load balancing among worker nodes. It proceeds in two steps: first, it uses the application topology and the exchanged inter-node traffic to assign the operators instances in slots, so to minimize the inter-slot traffic; then, it assigns slots to worker nodes, starting from the lowest loaded ones. Pundir et al. [162] study elasticity from another perspective: they investigate which operators should be migrated when the user requests a scaling operation. The basic idea is to organize computing resources on a peer-ring and allocate operators using a hash-based partitioning strategy; then, when resources are scaled in/out, the policy determines which server within the peer-ring should be removed/added so mitigate load imbalance. Enorm [136, 139] extends Storm with the support for window management. Therefore, Enorm can perform adaptations using an instant migration technique, which does not involve state relocation, or a parallel track migration technique. Schneider et al. [179] are among the first to propose an approach for elastically scaling the operator replication degree. Their approach periodically increases the number of threads associated to an operator, based on runtime performance: specifically, the policy increases parallelism as long as there are significant performance improvement. Liu et al. [128] propose a profiling approach that benchmarks the application performance on a given computing infrastructure; this solution allows to determine the operator replication and placement while avoiding performance bottlenecks and re-

source wastage. As regards load distribution, SABER [111] employs an adaptive lookahead scheduling strategy: it assigns each operator instance to the heterogeneous processor (either CPU or GPGPU) that, based on past behavior, achieves the highest throughput for that task. Differently, Schneider et al. [182] detect network congestion by analyzing the blocking rate of TCP connections. Therefore, they propose a dynamic load balance among computing nodes that minimizes the predicted blocking rate; considering each TCP connection at a time, it does not require knowledge on the global system state. The dynamic batch sizing allows to reduce the overhead of transmission, but might introduce undesirable delay due to the time needed to populate the batch. Focusing on geo-distributed DSP applications, Heintz et al. [75] model the problem of determining the batch size as a caching problem where the cache size varies over time. Then, the authors present different strategies for flushing the caches (i.e., transmitting the batch), ranging from eager to lazy solutions.

When system is overloaded and no new computing resources can be acquired (usually in a cluster environment), *load shedding* [SH] helps to provide a best-effort service by reducing the resource requirements of application by dropping a fraction of tuples from the upcoming data streams. Kalyvianaki et al. [102] propose a distributed load shedder for federated DSP systems, that aims to achieve a globally fair processing quality under overload conditions. When system is overloaded, it discards tuples so to balance the quality degradation per application, which depends on the amount of dropped tuples.

2.3.7 Wrap-up

We now summarize the most popular design choices used to adapt the application deployment at runtime.

Similarly to the approaches for the initial deployment, also in this case most solutions consist of a single centralized authority, which considers applications as single and independent entities. Nevertheless, we observe that, at runtime, a greater number of proposals deals with the concurrent deployment of multiple applications. As regards the deployment goals, many solutions optimize a single-objective utility function and/or aim to satisfy some QoS requirements at runtime. Indeed, constraint satisfaction is very important, because DSP applications are often exposed to changing working conditions. So far, the proposed approaches evenly consider user-oriented metrics (e.g., response time) and system-oriented metrics (e.g., inter-node traffic, resource utilization, load balance) while defining the deployment goals. Interestingly, since reconfiguring the application deployment introduces penalties in performance, several solutions explicitly account for adaptation costs, with the aim of controlling or minimizing them.

Along the time dimension, the most popular approach is that the DSP system executes periodically the adaptation policy which is reactive: it reconfigures the application deployment on the basis of the current system status. As regards the computing infrastructure, many solutions have been designed to work within a cluster, although more recent research efforts explore and exploit the interesting features of Cloud computing (e.g., elasticity). Either way, resources are often considered to be locally distributed; we also observe that there is not a strong preference on how to model resource heterogeneity: an even number of solution uses homogeneous and heterogeneous resources. The adaptation policy mainly plays with two different types of reconfigurations: the elastic replication of operators and their placement adaptation. As discussed, these are often considered as two orthogonal problems that are solved using a two-stages approach. Best-effort threshold-based policies are usually adopted to change the operator replication degree: when the utilization of system nodes or operator replicas exceeds predefined a critical value, the number of replicas is changed accordingly. The relocation of operator (or of their replicas) is usually performed using a greedy heuristic that, similarly to the initial deployment, models this task as the resolution of a bin-packing problem. This approach usually works sufficiently well also because most solutions work in a locally distributed environment.

2.3.8 Thesis Contribution

The analysis included in this section has shown that there are several interesting issues regarding the runtime adaptations of DSP applications. Some of them have been widely investigated, such as, e.g., performing adaptation in locally distributed infrastructure: many solutions propose to scale DSP operators using a threshold-based policy or to place DSP operators using a greedy first-fit heuristic that neglects network latencies. Nevertheless, several other branches of the taxonomy in Figure 2.3 are still largely unexplored. For example, the jointly optimization of runtime operator placement and replication should be further investigated, so to efficiently operate in distributed environments. Similarly, we deem adaptation costs to be very important and approaches that avoid greedy reconfigurations (which can degrade the application performance) should be further investigated. This need is especially true for geo-distributed settings, where relocating parts of DSP application can introduce significant downtime. Another interesting direction is represented by the integration of different policies. So far, few solutions combine reactive and proactive adaptation policies, or exploit the presence of heterogeneous computing resources (e.g., CPU/GPGPU). These solutions might help to understand and address the key challenges of the emerging near-edge environment, which

is characterized by heterogeneity and dynamism.

With respect to runtime adaptation of DSP applications, in this thesis we provide the following two main contributions. First, we present Elastic DSP Replication and Placement (for short, EDRP), a unified general formulation of the elastic operator replication and placement problem (Chapter 8). Second, we present a preliminary approach for managing elastic DSP applications relying on a hierarchical distributed control (Chapter 9).

By taking into account the heterogeneity of infrastructural resources and the QoS application requirements, EDRP determines the number of replicas for each operator and where to deploy them on the geo-distributed computing infrastructure. Moreover, EDRP models the reconfiguration costs that arise when a migration or a scaling operation should be performed, so to determine whether the application can be more conveniently redeployed. Differently from most works in literature [95, 103, 132, 138, 145, 219], EDRP can jointly determine the operator replication and placement, while optimizing the QoS attributes of the DSP application. Unlike the solution by Madsen et al. [138], which is the most closely related to ours, we explicitly model the impact of network latencies on the application performance and on reconfigurations. With respect to the taxonomy of Figure 2.3, EDRP populates the six dimensions as follows:

- Why: optimization of a multi-objective function, which takes into account the application response time, execution cost, and adaptation costs (i.e., user-oriented QoS metrics).
- What: control of a single application at a time, with granularity of a single operator.
- Who: being interested in computing the optimal deployment solution, which requires a global view of the system, we rely on a single centralized authority.
- When: EDRP computes the initial application deployment and then periodically evaluates it, so to reactively adapt to changes of the working environment.
- Where: our contributions model a cluster of heterogeneous computing and network resources, whose distribution is geographic.
- How: EDRP formulates the operator replication and replica placement as an ILP problem (i.e., it uses a mathematical programming approach to jointly optimize operator replication and placement).

By analyzing the outcomes of our contributions and of most existing works, we easily realize that scalability represents a key issue for determining the application deployment and, most importantly, for computing its runtime adaptation. Scalability can be achieved by means of efficient adaptation heuristics, but also by properly organizing the control authori-

ties that monitor and oversee the application execution. Also in this case, by looking at the existing literature, we can see that there are several largely uncovered regions of the taxonomy. Specifically, there is only a single hierarchical distributed control solution. However, this architecture is worthy of further investigations, because it might overcome the limitation of fully centralized approaches (e.g., scalability — as we will show in Section 4.6) and fully decentralized approaches (e.g., lack of coordination — as we will show in Section 3.5).

In Chapter 9, we present our hierarchical distributed approach for controlling elastic DSP applications. The proposed solution, named Elastic and Distributed DSP Framework (for short, EDF), organizes the control leveraging on hierarchical decentralized MAPE feedback loops. Specifically, EDF includes a high-level centralized MAPE-based Application Manager, which coordinates the runtime adaptation of subordinated MAPE-based Operators Managers, which, in turn, locally control the adaptation of single DSP operators. Differently from the existing solutions (e.g., [88, 158, 171]), we add a centralized coordinator to steer the adaptation actions taken by decentralized agents. The latter can perform scaling operations and stateful migrations, whereas the centralized Application Manager identifies the most effective reconfigurations so to control their number and the resulting adaptation costs. With respect to the taxonomy, EDF populates the six dimensions as follows:

- Why: minimization of adaptation costs, while meeting requirements on application response time.
- What: control of a single application at a time, with granularity of a single operator.
- Who: hierarchical decentralized control authorities.
- When: EDF periodically evaluates the application deployment and reactively adapts it, if needed.
- Where: our contributions model a Fog-based environment, with heterogeneous and distributed computing resources.
- How: EDF uses a simple heuristics to control adaptation: the Operator Manager migrates and scales operators using a threshold-based policy; the Application Manager grants reconfiguration permissions using a token-based policy.

2.4 DSP Frameworks

A number of DSP frameworks has been developed in academic, open source, and industry communities, thus showing that, albeit challenging, DSP is of key importance for data intensive applications [45, 77].

De Matteis [45] and Cugola and Margara [41] nicely describe the historical evolution of stream processing frameworks. *Data Stream Management Systems* (DSMSs) were born as an evolution of traditional Database Management Systems (DBMSs). Indeed, while DBMSs are built around a persistent storage and optimize user-triggered queries, DSMSs natively work with transient data and can execute continuous queries that provide updated answers as soon as new data arrives. Moreover, DSMSs offer an SQL-like declarative language to programmers as to define continuous queries. Heinze et al. [77] classify these framework as the first generation of stream processing systems, because they are built as stand-alone prototypes or as extensions of existing database engines. Specifically, they offer a very limited support for operator types and functionalities (e.g., often UDFs are not supported). Among the frameworks belonging to this category, we can find TelegraphCQ [33], STREAM [10], Gigascope [39], Aurora [1], and Borealis [2]. For example, Aurora adopts a visual programming language (i.e., Stream Query Language — also known as SQuAl), which defines operators able of processing single-tuples or windows of tuples at once; these operators create an extended relational algebra. Borealis [2] has completely redesigned Aurora so as to obtain a commercial stream processing product. Differently from Aurora, Borealis works on distributed resources and provides advanced capabilities, such as dynamic query rewriting, load shedding, and the runtime deployment adaption.

A second category of solutions is known as *Complex Event Processing* (CEP) systems. These systems emerged as an evolution of Publish/Subscribe systems [41]. Here, data items are notification of events, which announce a system status change. CEP systems allow to realize applications that combine primitive events coming from (possibly) multiple data sources, with the aim to detect complex patterns (in time and space). As soon as a pattern is detected, the system notifies the interested parties, who previously subscribed to the system. According to Heinze et al. [77], we can find CEP systems belonging to the second and third generation of stream processing engines. Second generation systems include advanced features such as fault tolerance [2], adaptive query processing [173], and enhanced operator expressiveness [16]. Third generation systems are designed to be elastic and highly scalable, exploiting Cloud computing resources (e.g., SEEP [147]). An emerging trend also explores the utilization of CEP systems in mobile environments, where data sources and consumers, such as IoT devices, vehicles, and citizens, interact with the system while moving around (e.g., [89, 153, 189]). Among the great number of existing CEP systems, we just name a few of them: T-Rex [40], Oracle CEP[10], and Esper[11].

[10] `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html`

[11] `http://www.espertech.com/esper/`

CEP systems are incredibly good in detecting patterns; nevertheless, they cannot modify incoming data and do not fully support complex data processing (e.g., they cannot easily run machine learning tasks).

Data Stream Processing systems are often employed to overcome the above limitations. Specifically, DSP systems work with generic data (i.e., not only with events) and support the definition of complex application logic. As such, they often represent a more flexible solution with respect to CEP systems. A great variety of DSP frameworks has been proposed so far; however, aside the specific functionalities, the most popular DSP frameworks (e.g., Storm, Spark Streaming, Flink, and Heron) use directed graphs to model DSP applications. They provide an abstraction layer where to execute DSP applications and allow their users to focus solely on the application logic, being the tasks related to the application placement, distribution, and execution managed by the frameworks themselves. Heinze et al. [77] classify most of DSP systems in the third generation of stream processing engines. In most cases these frameworks require their users to manually tune the number of replica per operator. Since the user might over-/underestimate the expected load, this approach can lead to a sub-optimal provisioning of resources. Furthermore, these frameworks are equipped with elasticity mechanisms in an embryonic stage; indeed, they dynamically scale the application in a disruptive manner, because they enact reconfigurations by killing and restarting the whole application, thus introducing a significant downtime.

Apache Storm [199] is one of the most popular open-source DSP framework. Several research efforts have used Storm to either evaluate new operator placement algorithms in a real environment or to propose some architectural improvements (e.g., [9, 54, 72, 78, 123, 156, 199, 213]). Leveraging on Trident[12], Storm supports stateful applications. Differently from most solution, Trident does not manage an operator-related state; Trident can persist a state which is obtained by applying a sequence of Trident transformations on the input data. However, this approach requires to play the stream as a sequence of micro-batches, processed in a commit-like fashion, thus causing a constant latency overhead.

It is worth observing that may research prototypes extend Storm so to implement placement algorithms (e.g., [9, 25, 52, 61, 61, 73, 156, 186, 213, 222]) or introduce architectural changes and new features (e.g., [123, 137, 136, 139, 166, 175, 194, 214, 216]). For example, Yang ang Ma [216] propose different strategies for relocating stateless executors, achieving a reduction of the application downtime. Qin and Eichelberger [166] introduce a runtime switching protocol that changes the operator logic with a reduced adaptation cost (to this end, they exploit a parallel track strategy

[12] `http://storm.apache.org/releases/current/Trident-API-Overview.html`

which runs the active and target algorithms in parallel for a limited transient). Li et al. [123] propose an approach to support elastic scaling of DSP applications; their solution reduces the interruption due to scaling operations by keeping the application running while scaling, instead of shutting down the application operators and restarting them. However, their improved version of Storm has not been released publicly. The recently presented SpanEdge [175] considers the execution of Storm in decentralized data centers and, similarly to our works, places the application operators so to minimize network latencies. Nevertheless, it does not support operator migrations. Although for a different application scenario than the one investigated in this thesis, we mention the Storm extension by Basanta-Val et al. [17]. In this system, streams are treated as real-time entities (i.e., with explicit data generation rate and deadlines), so that they can be scheduled using existing solutions from real-time scheduling theory.

Developed by Twitter as the successor of Storm, Heron [113] preserves its abstraction layer while introducing some improvements and a multi-layer architecture. Dhalion [60] is a newly presented framework on top of Heron that provides elastic capabilities to the underlying streaming system. It also addresses the overhead related to restarting the topology by allowing its updating. However, at the time of writing Dhalion has not yet been open-sourced and the elasticity policy simply adjusts the replication degree of an operator so to satisfy its throughput; anyway, the investigation of reinforcement learning techniques in Dhalion is considered as an exciting area for future research [59].

Apache S4 (Simple Scalable Streaming System) [151] is a DSP framework designed to operate in large-scale clusters that are built with commodity hardware. To achieve scalability, S4 uses a decentralized architecture, where all the processing nodes share the same functionalities and responsibilities (i.e., there is no central node with special functionalities).

Apache Samza [152] is a distributed system that supports stateful processing of real-time streams, along with the fast reprocessing of entire data streams. It uses Apache Kafka for messaging and Apache YARN [203] for resource management [110]. To provide fault-tolerance and reduced recovery time, Samza uses a changelog capturing changes to the state in the background. Maintaining the changelog is lightweight than periodically performing checkpoints and, moreover, allows to manage very large state in spite of limited memory.

Apache Spark [220] is a general-purpose framework for large-scale processing. Spark Streaming [221] is a module of Apache Spark that enables data stream processing. It is throughput-oriented, whereas Storm can minimize the application latency and can thus be preferable in latency-sensitive scenarios. From version 2.0, Spark Streaming supports elastic scaling using the dynamic allocation feature. It uses a simple heuristic where the number

of executors is scaled up when there are pending tasks and is scaled down when executors have been idle for a specified time. Building on Spark, Drizzle [204] enhances the Spark architecture along three main directions. First, it reduces the control overhead in the micro-batched streaming model (i.e., Spark Streaming); specifically, it performs the control operations between groups of processing tasks (i.e., not after every processing task). Second, it removes the centralized coordination point, enabling the direct communication among processing tasks. Third, it provides self-adaptation capabilities, improving fault-tolerance and introducing elasticity.

Another emerging framework is Apache Flink [27], which traces its origin back to the Stratosphere project [8]. Flink provides a unified solution for batch and stream processing (like also AJIRA does [201]). To this end, it uses the idea of processing pipelines, i.e., series of data-centric transformations expressed in a functional programming API — as in the Google's Dataflow Model [7]. Flink allows to declare the application state, so that it can be autonomously managed and persisted. Using a distributed checkpointing procedure, Flink also guarantees consistency in case of failures and reconfigurations (e.g., to change the execution parallelism). Flink does not autonomously adapt the application deployment [26]; conversely, it aims to adopt a stable and long-running allocation of tasks. However, it supports the reconfiguration of pipelines on demand so as to change the execution parallelism and re-allocate the application state.

IBM Infosphere Streams [15] is a commercial system by IBM, which has been developed as a successor of IBM System S. This product includes a data flow composition language (known as Streams Processing Language or SPL), which allows to define and compose DSP operators. IBM Infosphere Streams has been mainly designed to work on a cluster of resources and, to the best of our knowledge, does not yet support elasticity.

StreamScope [126] (or StreamS) is a system by Microsoft that has been designed for business-critical applications. It processes data with exactly-once guarantees, despite server failures and message losses. To this end, StreamScope explicitly manages the operators internal state and supports different failure recovery strategies, which are based on checkpoints (i.e., upstream backup), stream replay, and active replication.

Many other solutions have been developed especially from academia (e.g. [74, 92, 164, 177]). Each one optimizes specific aspects of the processing engine, although all of them share the same key conceptual abstractions regarding the definition of DSP applications. It is worth mentioning that Foglets, by Saurez et al. [178], proposes a programming model specifically designed for the Fog computing environment. Moreover, Foglets includes algorithms for controlling the execution of application operators, simplifying communication among them, and supporting their stateful migration. Several solutions exploit features of Cloud computing to realize

elastic and fault-tolerant systems (e.g., StreamCloud [70, 71], VISP [88], Timestream [165], and Chronostream [210]). For example, Hochreiner et al. [88] have developed VISP, a framework for distributed DSP applications that natively supports elasticity. Differently from the above solutions, VISP relies on decentralized controllers, one per each operator, that collectively are in charge of managing the application deployment.

We observe that, despite the recent efforts towards elasticity in some frameworks, most of those cited are not designed to efficiently operate in a geo-distributed environment.

DSP systems are also offered as Cloud services. Google Cloud Dataflow[13] provides a unified programming model to process batch and streaming data [7]. It comes with an interesting processing model, which supports different types of windows, distinguishing between event time and processing time. The Dataflow processing model has been implemented on top of MillWheel [6], a streaming processing system by Google explicitly designed to be fault-tolerant and support Internet-scale processing. Amazon offers Kinesis Streams[14] to process near real-time streams of data. Both of them abstract the underlying infrastructure and support dynamic scaling of the computing resources. However, it appears that they execute in a single data center, conversely to the geo-distributed environment we investigate in this thesis work. Microsoft offers Azure Stream Analytics[15] (ASA), which allows to run queries over streams of data. This service offers a SQL-like query language for performing transformations and computations over continuous streams of events.

Thesis Contribution. Most of the existing DSP frameworks have been designed to run in a centralized cluster environment. Therefore, to evaluate existing decentralized heuristics and develop new ones, we have developed a prototype DSP framework that can execute both centralized and decentralized placement and adaptation strategies on a distributed and heterogeneous infrastructure. The developed framework is named Distributed Storm and it has been designed as an extension of Apache Storm. In Chapter 3, we provide details on its architecture and we show how it can be used to evaluate different policies. Specifically, we use it to show the critical issues of fully decentralized heuristics that should be taken into account while designing new deployment policies.

Being interested in adapted at runtime the application deployment, we also need a framework that can be used as testbed for evaluating elasticity policies. Looking at the literature, we see that the most popular open-source DSP frameworks, i.e., Storm, Spark Streaming, and Flink, do not fully support elasticity. Therefore, we have developed Elastic Storm, a

[13] `https://cloud.google.com/dataflow/`

[14] `https://aws.amazon.com/kinesis/`

[15] `https://azure.microsoft.com/en-us/services/stream-analytics/`

second extension of Storm, which will be described in Chapter 7. Elastic Storm introduces in Storm two mechanisms that support the runtime adaptation of DSP applications: elasticity and stateful migration. An approach to support elastic scaling of DSP applications in Storm has been also presented, at the same time, in [123]; interestingly, their proposal reduces the interruption due to scaling operations by keeping the application running while scaling, instead of shutting down the application operators and restarting them. However, they considered a clustered architecture and their improved version of Storm has not been released publicly.

Chapter 3

Distributed Storm

Storm is a distributed stream processing system that has recently gained increasing interest. We extend Storm to make it suitable to operate in a geographically distributed and dynamic environment, such as the one envisioned by the convergence of Fog computing, Cloud computing, and Internet of Things.

With the advent of the Big Data era, DSP systems have received a renewed and increasing interest. To simplify the execution of DSP applications, several DSP systems (or DSP frameworks) have been proposed. Besides the specific functionalities, they provide an abstraction layer where DSP applications can be executed. Moreover, they oversee the applications runtime by taking care of (at least) the following functionalities: computing resource management, placement and replication of the application DSP operators (computed using a *scheduler* component), code distribution, data transfer, fault tolerance, and management interface. As presented in [77] and briefly reviewed in Section 2.4, we can distinguish at least three generations of DSP systems, where the current one is driven by the trend towards Cloud computing, where elasticity and fault tolerance are key architectural features. Despite this, most DSP systems are still designed to run in local clusters, where the often homogeneous nodes are interconnected with negligible latencies (e.g., [151, 199, 201]). As such, these systems may not perform well when executed on the emerging infrastructures that comprise near-edge/Fog and Cloud computing resources.

Apache Storm [199] is an open source, scalable, and fault-tolerant DSP system designed for locally distributed clusters. Its default operator placement policy evenly distributes the processing elements on the computational nodes, aiming at load sharing in the cluster. Being among the first open source solutions of modern DSP systems [77], Storm has attracted increasing industrial and academic interests: several research efforts use Storm to evaluate operator placement algorithms, self-adaptation policies,

and architectural improvements (e.g., [9, 123, 156, 199, 213]). We are also interested in using Storm as a testbed upon which we can implement and evaluate new DSP deployment policies. Nevertheless, the current architecture of Storm as well as the above cited Storm-based works, which adopt a centralized scheduler, are not designed to process data streams in a distributed environment, such as the one envisioned by the convergence of Fog computing, Cloud computing, and IoT. In this setting, DSP systems should exploit both distant and proximate computing resources; therefore, they are requested to scale on a wide-spread infrastructure with many running applications and distributed sources. To address these challenges, we propose Distributed Storm, an extension of Storm that allows to run self-adaptive distributed scheduling algorithms, making it suitable to operate in geo-distributed environments. In the following chapters, we will use Distributed Storm so to prototype and evaluate, on a real environment, new deployment solutions for DSP applications.

In this chapter, we present the design and implementation of a distributed, QoS-aware, and self-adaptive scheduler for Storm by adding a few key modules to the standard Storm architecture. Our scheduler can scale as the number of applications and network resources increase since it does not require a global knowledge of the system. Moreover, the self-adaptive capability allows to automatically reconfigure the operator placement in a distributed fashion when unpredictable environmental changes occur. To demonstrate the effectiveness of the extended Storm, we implemented a distributed scheduling algorithm that is aware of QoS attributes. To this end, we adapt and implement in Storm the network-aware scheduling algorithm by Pietzuch et al. [158]. The main contributions of our work are as follows.

- We extend the Strom architecture by designing and implementing the support for distributed QoS-aware scheduling and runtime adaptivity.
- To show the flexibility of the proposed extension, we implement and evaluate the distributed network-aware scheduling algorithm proposed by Pietzuch et al. [158].
- We present a thorough experimental evaluation of the proposed solution using a good variety of DSP applications, coming with different processing requirements.

The rest of this chapter is organized as follows. We position our work with respect to the state of the art in Section 3.1. In Section 3.2, we briefly introduce the official release of the Storm framework and its DSP model. Then, in Section 3.3 we present the design of our Storm extension that introduces the distributed QoS-aware scheduler. Section 3.4 describes the adopted proof-of-concept distributed scheduling policy. Then, we analyze a wide set of experiments run on our Storm prototype in Section 3.5 and

conclude with Section 3.6.

3.1 Related Work

Exploiting on-the-fly computation and several forms of parallelism, DSP applications can constantly emit new results. As such, these applications are usually adopted in scenarios with strict QoS requirements in terms of throughput or response time. The literature analysis presented in Chapter 2 has shown that, to achieve high performance, DSP systems have to quickly compute the application deployment and efficiently exploit features of the computing infrastructure. Most DSP frameworks have been designed to run in centralized data centers (see Section 2.4), where (cluster or Cloud) computing resources are interconnected with high-speed communication channel. In these settings, communication delays are negligible. Nevertheless, this property does not hold true in geographically distributed environments, where most of existing DSP frameworks may not experience satisfying performance. As discussed by Heinze et al. [77], the next generation of DSP systems should explicitly take into account the features of the computing infrastructure, so to improve performance and efficiently satisfy the application requirements. Therefore, the existing DSP frameworks natively designed for clustered environments should be conveniently enhanced to efficiently work in the emerging geo-distributed environment.

Storm has recently gained interest in academic research, where it is used to evaluate operator placement algorithms (e.g., [9, 58, 156, 163, 213]) and architectural improvements (e.g., [123, 199, 216]). For example, Fischer et al. [58] use existing graph partitioning algorithms to optimize the amount of data sent between nodes. Aniello et al. [9] design two Storm schedulers which reduce the inter-node traffic during the application execution; one of them uses online information to improve system and application performances. The same idea is also exploited by Xu et al. [213], who propose a traffic-aware scheduler for dynamically assigning operators and enabling fine-grained control over worker node consolidation. Differently from [9], their solution, as ours, is completely transparent to Storm users. Besides minimizing inter-node traffic (as in [9, 213, 222]), the placement solutions implemented in Storm aim to improve resource utilization (e.g., [73, 156, 202]), maximize application throughput (e.g., [186]), or minimize response time (e.g., [52, 194]). All these Storm-based works rely on a centralized scheduler, which does not consider the network-related characteristics of the computing infrastructure (e.g., network latency) and, moreover, can suffer from scalability issues. Being Storm open source and very popular in the research community, we decide to extend this framework thus obtaining Distributed Storm, which makes Storm suitable to operate

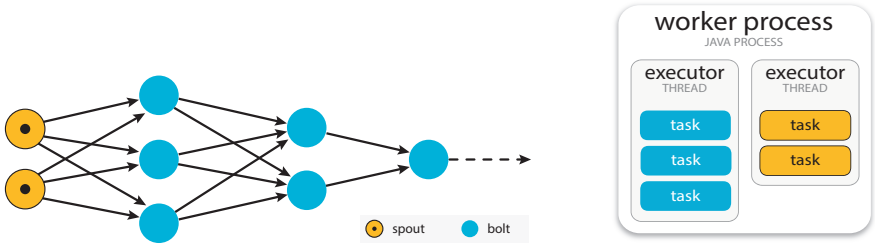
in geo-distributed environments. Moreover, Distributed Storm provides a platform where decentralized and self-adaptive operator placement solutions can be implemented and evaluated. We release the developed code as open source[1], so to provide the community with a new tool that can be be used to develop new deployment policies.

Nowadays[2], several DSP frameworks can oversee resources belonging to infrastructures disseminated among multiple data centers, i.e., Fog-like [175, 178]. Foglets [178] proposes a programming model specifically designed for Fog computing environments. It includes algorithms for controlling the execution of application operators, simplifying communication among them, and supporting their stateful migration. SpanEdge [175] is a Storm extension that, similarly to our extension, considers network latencies. It allows programmers to specify where operators should be allocated (e.g., close to data sources). Differently from our solution, their scheduler is centralized and does not support the execution of decentralized policies.

To validate the functionalities introduced by Distributed Storm, we select, from the literature, a proof-of-concept placement policy that can be adapted for running in Storm. The placement policy should be decentralized, self-adaptive, and should be infrastructure-aware. Exploiting geo-distributed resources, it should determine the operator placement with the aim of reducing the amount of data exchanged using the network (which, in turn, improves the application response time). Chatzistergiou et al. [34] propose a scalable centralized solution, which efficiently co-locates operators using a group-based job representation and adapts to execution environment changes. Zhou et al. in [224] investigate a fully decentralized algorithm, which reduces the inter-node traffic while balancing the load among the nodes. However, these research efforts neglect to address latency issues because they assume a cluster-based deployment. A latency-aware scheduler is presented by Backman et al. [14], whose centralized scheduler parallelizes and executes workflows of stream operators to meet latency objectives. Pietzuch et al. [158] propose an elegant and fully decentralized placement algorithm that allocates the application operators with the aim of minimizing network usage. Their approach exploits a latency space as a search space to find the best placement solution. A similar idea has been pursued by Rizou et al. [171]. Moreover, they exploit the mathematical properties of the network usage function so to find the global optimum solution in a completely decentralized manner. Besides these latter two works (i.e., [158, 171]) that we exploit in this chapter, other network-aware operator placement algorithms have been proposed in literature (see Chapter 2). Among them, SAND [5] leverages knowledge of network character-

[1]Distributed Storm on GitHub: `http://bit.ly/extstorm`

[2]At the best of our knowledge, when we developed Distributed Storm in 2015, it was the first Storm extension to introduce distributed monitoring and scheduling capabilities.



(a) Topology of a DSP application (b) Execution model

Figure 3.1: Storm abstractions

istics and can be implemented in a distributed manner using a DHT overlay. XFlow [154] is an Internet-scale monitoring framework. Similarly to our approach, it relies on a cost model to drive the operators' placement, which is performed by an adaptive and fully distributed scheduler. However, XFlow focuses on monitoring applications, whereas our extended version of Storm allows the execution of generic DSP applications.

Our proposal differs from the previous works because we extend Storm to run self-adaptive distributed scheduling algorithms, preserving the transparency of the framework to its users. Furthermore, we exploit a distributed QoS-aware scheduling algorithm that takes into account node state information (i.e., node availability and utilization), besides the network awareness considered in [5, 14, 158].

3.2 Apache Storm

Storm is an open source, real-time, and scalable DSP system maintained by the Apache Software Foundation. It provides an abstraction layer where DSP applications can be executed over a set of worker nodes interconnected in an overlay network. A *worker node* is a generic computing resource (i.e., physical or virtual machine), whereas the overlay network comprises the logical links among these nodes. In Storm, we can distinguish between an abstract application model and an execution application model. In the abstract model, an application is represented by its *topology* (see Figure 3.1a), which is a DAG with spouts and bolts as vertices and streams as edges. A *spout* is a data source that feeds data into the system through one or more streams. A *bolt* is either a processing element, which generates new outgoing streams, or a final information consumer. A *stream* is an unbounded sequence of *tuples*, which are key-value pairs. We refer to spouts and bolts as operators. Figure 3.1a shows an example of a DSP application.

In the execution model, Storm transforms the topology by replacing

each operator with its tasks. A *task* is an instance of an application operator (i.e., spout or bolt), and it is in charge of a share of the incoming operator stream. Therefore, if the operator has some internal state (i.e., it is a stateful operator), a task handles a partition of it. In Storm, the number of tasks for an operator is statically defined. Each task processes incoming data with an at-least-once semantic, which can be turned into an exactly-once semantic relying on the Trident API (not used in this work because of its processing overhead). For the execution, one or more tasks of the *same* operator are grouped into executors. An *executor* is the smallest schedulable unit; Storm can process large data volumes in parallel by launching multiple executors for each operator. The number of executors of an operator must always be less than or equal to the number of tasks of the same operator. From an operational perspective, Storm implements the executors as threads and also introduces the *worker process*, that is a Java process, to run a subset of executors of the *same* topology. As represented in Figure 3.1b, there is an evident hierarchy among the Storm entities: a group of tasks runs sequentially in the executor, which is a thread within the worker process, that serves as container on the worker node. The latter is a node of the Storm cluster that offers the computing resources.

Besides the worker nodes, the architecture of Storm includes two additional components: Nimbus and ZooKeeper. *Nimbus* is a centralized component in charge of coordinating the topology execution; it uses its *scheduler* to define the placement of the application operators on the pool of available worker nodes. The assignment plan determined by the scheduler is communicated to the worker nodes through *ZooKeeper*[3], that is a shared in-memory service for managing configuration information and enabling distributed coordination. Since each worker node can execute one or more worker processes, a *Supervisor* component, running on each node, starts or terminates worker processes according to the Nimbus assignments. A worker node can concurrently run a limited number of worker processes, based on the number of available *worker slots*.

3.3 Distributed Scheduling in Storm

We extend the Storm architecture to run distributed QoS-aware scheduling algorithms and enhance the system with adaptation capability. The newly introduced components, illustrated in orange in Figure 3.2, are: the *AdaptiveScheduler*, the *QoSMonitor*, and the *WorkerMonitor*. They realize a fully decentralized MAPE loop [106, 208], where: the QoSMonitor and the WorkerMonitor implement the distributed Monitor component; the Adap-

[3] `http://zookeeper.apache.org/`

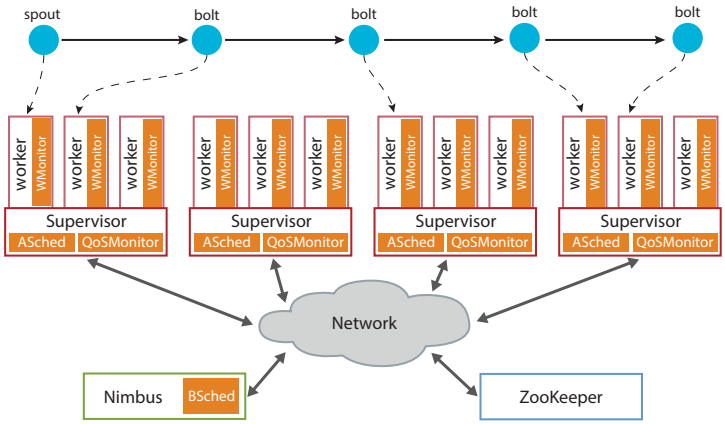


Figure 3.2: Extended Storm architecture (new components in orange): AdaptiveScheduler is abbreviated as ASched, WorkerMonitor as WMonitor, and BootstrapScheduler as BSched

tiveScheduler implements the Analyze and Plan components; and the Supervisor (natively present in Storm) implements the Execute component.

The *QoSMonitor* estimates the network latency among the system nodes and monitors the QoS attributes of the worker node, namely the node availability and its resource utilization. A *WorkerMonitor* is executed for each Storm worker process; it computes the data rate exchanged among the application operators. The information collected by these monitors can be used by the QoS-aware scheduling algorithm. The *AdaptiveScheduler* is located on each worker node and executes the distributed scheduling policy. Besides introducing the distributed scheduler on each worker node, we also maintain a centralized scheduler, called *BootstrapScheduler*. It is executed by Nimbus and is in charge of defining the initial placement of the application operators and monitoring their execution. Furthermore, it can reschedule the application when a worker process fails. Our extension is completely user-transparent, meaning that applications can be executed using the new scheduling system without requiring any change. The source code of our extension is available at `http://bit.ly/extstorm`.

3.3.1 Monitoring Components

QoSMonitor. The QoSMonitor provides the QoS awareness to each distributed scheduler, thus it is responsible of obtaining intra-node (i.e., utilization and availability) and inter-node (i.e., network) node information. For the latter, we resort to a network coordinates (NC) system that provides an accurate estimate of the round-trip latency between any two network locations, without the need of an exhaustive probing. The NC system is

maintained through the Vivaldi algorithm [43], a decentralized algorithm with linear complexity with respect to the number of network locations. To make each node be informed of all the QoS attributes of the other nodes, we rely on a gossip-based dissemination protocol.

WorkerMonitor. The *WorkerMonitor* runs on a worker process. It is responsible of obtaining the incoming and outgoing data rate for each executor running on the worker process. This information is stored in a local database to be subsequently used by the distributed scheduler.

3.3.2 AdaptiveScheduler

The *AdaptiveScheduler* executes the Analyze and Plan components of the MAPE loop on every worker node so as to adapt the application deployment at runtime. The behavior of these MAPE components can be customized according to a distributed scheduling policy. In principle, any policy can be implemented; in this chapter, as a proof of concept, we implemented a known use case, relying on the Pietzuch et al. algorithm presented in Section 3.4. Since the architecture of our extended Storm is modular and low-coupled, the implementation of this algorithm is straightforward. The algorithm has been adjusted to account for the specific Storm application model, where a processing operator can be instantiated in one or more executors and pinned operators are not modeled. Only the executors assigned to the worker node can be managed by this scheduler, which can reassign them to improve the application performance.

Analyze. During the *Analyze* phase the AdaptiveScheduler acquires the information collected by the monitoring components and identifies the set of local executors that could be moved. An executor is movable if it is not pinned (i.e., without a fixed physical location) and not directly connected to an operator which is going to be reassigned. For each movable executor, the AdaptiveScheduler determines whether it will be effectively moved to another position in a decentralized fashion. To this end, it resorts to the elegant approach proposed by Pietzuch et al. [158] named as Virtual Placement Algorithm, which will be described in Section 3.4.

Plan. If the Analyze phase finds that an executor e_i needs to be moved to a new position, it triggers the *Plan* phase. The Plan phase is responsible of executing the second step of the placement algorithm, which, relying on a cost space, determines the worker node that will host and execute the executor e_i. If no worker node is found, the MAPE iteration for executor e_i terminates; otherwise, the Plan phase performs two additional tests and the executor e_i is moved only if both of them are passed.

The first test is called *relative distance*: its goal is to avoid moving executors to a new candidate worker node if such change of position does not improve the application performance. Indeed, moving an operator has a

not negligible cost (i.e., downtime, network and computational cost, loss of state information), which can negatively affect the application performance. Formally, let d_{cand} and d_{cur} be respectively the distance of the candidate worker node ws_{cand} and of the current worker node ws_{cur} from the operator position $\vec{P}_i$ in the cost space. The test is passed if the relative distance $D_{rel}(d_{cur}, d_{cand})$ exceeds a given migration threshold thr_{migr}:

$$D_{rel}(d_{cur}, d_{cand}) = \frac{|d_{cur} - d_{cand}|}{d_{cur} + d_{cand}} > thr_{migr}$$

where $D_{rel}, thr_{migr} \in [0, 1]$. The second test is called *look-ahead* and it aims to reduce placement oscillations: it simulates the migration of a local executor to the candidate worker node and checks if a next reassignment will take the executor back to the current worker node. For example, consider two pinned components $\{c^p_\alpha, c^p_\beta\}$, an unpinned component $\{c^u_\gamma\}$ and two nodes $\{A, B\}$; c^p_α and c^u_γ run on A, while $\{c^p_\beta\}$ runs on B. By evaluating the network traffic between c^u_γ and c^p_β, the decentralized placement policy may decide to move c^u_γ on B. Nevertheless, the new configuration is similar to that preceding the reassignment; hence, the decentralized policy may decide to reassign c^u_γ to A once again. This ping-pong effect must be avoided since it is useless and may impacts negatively on the performance; so the test is passed only if the simulated "next move" does not take back the executor. This simple but effective test allows to solve the described problem without storing additional operator-related information. After having selected the worker node, the last step is to determine which worker slot on the node will be used. The algorithm tries to reuse an existent slot, if there is someone that already runs another executor of the same topology. Anyway, a worker slot can be reused only if it holds less than a given number of executors ($\#eps_{max}$). If no reusable slot is found, a new one is used.

The AdaptiveScheduler moves only one executor at a time in order to reduce the effect of multiple relocations, which can negatively affect the application performance.

Execute. Finally, in the *Execute* phase, if a new assignment must take place, the executor e_i is moved to the new candidate node. The new assignment decision is shared with the involved worker nodes through ZooKeeper. To enact the placement, Storm stops the executor on the previous worker node and starts it on the new one[4].

Thanks to the adaptation cycle and the multi-dimensional cost space, the AdaptiveScheduler can manage changes that may occur both in the infrastructure layer (e.g., new worker nodes that appear or existing ones that fail) and the application layer (e.g., increase in the source data rate).

[4]The initial version of Distributed Storm, presented in this chapter, did not support stateful migration. However, we introduced this feature later in time (see Chapters 7 and 8).

3.3.3 BootstrapScheduler

Nimbus runs a centralized scheduler, that we called *BootstrapScheduler*, which defines the initial assignment of the application, monitors its execution, and restarts failed executors, when, for example, a worker node disappears. The BootstrapScheduler is a centralized version of the QoS-aware scheduling algorithm, which assigns executor pools (i.e., groups of executors), instead of single executors, in order to efficiently use the system resources.

To retrieve the system state, the BootstrapScheduler uses Nimbus that, in turn, communicates with the worker nodes using ZooKeeper as message broker. Therefore, we should observe that ZooKeeper represents a bottleneck in scaling Storm and should be more conveniently replaced with a more sophisticated system, such as PaceMaker [36]. PaceMaker provides a new heartbeat and metrics service that will be fully integrated in Storm since version 2.0. Distributed Storm can also benefit from this new and scalable component; therefore, we plan its integration as future work.

3.4 A QoS-aware Heuristic

We implement a distributed scheduling algorithm that is aware of QoS attributes, specifically latency, node utilization, and availability. To this end, we adapt to Storm the network-aware scheduling algorithm proposed by Pietzuch et al. [158]. It comprises a *cost space*, which models the placement problem by transforming the performance metrics of interest into distances in this space, and an *operator placement algorithm*, which places operators in this cost space. It achieves good application performance by minimizing the amount of data that transits the network at a given instant, blending together network delay and network resource consumption: the smaller the link delays, the better the overall application delay; but, at the same time, the larger the data rate between two operators, the closer they should be in the network (possibly co-located in the same physical node). Along with network usage, we consider two other metrics which capture the nodes performance: utilization and availability. The former captures the node processing latency, which is function of the node utilization level. The latter represents the fraction of time a node is up and able to execute operators code.

3.4.1 Cost Space

A cost space is a metric space where distance between two points *estimates* the cost of routing and processing data between two nodes placed in those two points. We adopt a four-dimension cost space, where two dimensions

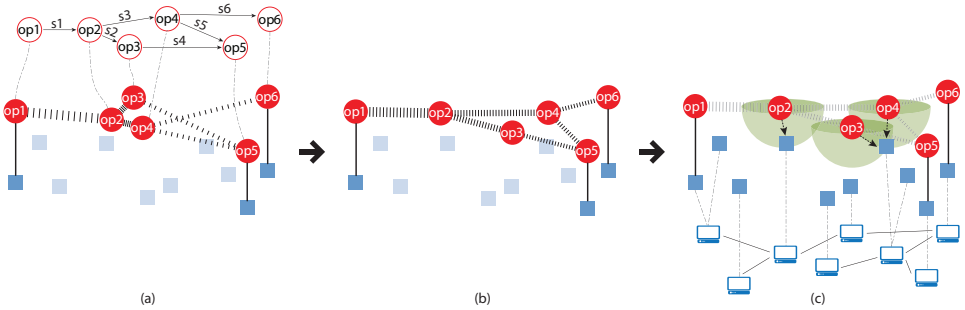


Figure 3.3: High-level architecture of our solution: (a) pinned and unpinned operators; (b) Virtual Operator Placement; (c) Physical Operator Placement

refer to the latency attribute, while the other two refer to node availability and utilization, respectively. The two latency dimensions of the cost space form a latency sub-space, where the distance between two nodes is an estimate of their network latency. The cost space dynamically adapts to changing network and node conditions as nodes continuously adjust their coordinates through measurements.

3.4.2 Placement Algorithm

The placement policy comprises two phases: the virtual operator placement and the physical operator mapping. The former determines the operators placement in the cost space; the latter maps its decision back to physical nodes (see Figure 3.3). To compute the placement, we need to distinguish between two kind of operators: a *pinned* operator has a fixed physical location (e.g., *producers*, *consumers*), whereas an *unpinned* one can be conveniently instantiated on any node of the network. The placement algorithm is fully decentralized and is periodically executed, so to adapt the unpinned operators placement in face of changing working conditions.

Virtual Placement Algorithm. The idea behind the placement algorithm is to regard the system of operators and links as a collection of massless bodies connected by springs. In this mechanical analogy, the rest position of the springs represents the minimum potential energy configuration; the system naturally converges to it, by simply letting each operator op_i *moves* as the results of the force $\vec{F}_i$ applied to it by the systems of springs. As observed in [158], by setting the spring extension equal to the latency, $s_l = \text{Lat}(l)$, and the spring constant to the data rate over that link, $k_l = \text{DR}(l)$, the minimum energy configuration of the spring system corresponds to the minimum network usage configuration of the operators.

Physical Placement Algorithm. Once the Virtual Placement Algorithm

terminates, the operator op_i has associated a coordinate $\vec{P_i}$ in the latency space. Since the virtual operator placement uses only the latency dimension, $\vec{P_i}$ has the coordinate associated to availability and utilization equal to 1 and 0, respectively, i.e., $\vec{P_i} = (p_{l1i}, p_{l2i}, 1, 0)$. In the second stage of the placement algorithm, we map the operator op_i to an actual physical node ws_j. Ideally, the best candidate node is the one closest to the operator coordinate $\vec{P_i}$; thus, we choose the node ws_j with coordinate $\vec{P_j}$ with minimal Euclidean distance to $\vec{P_i}$. Differently from the original work [158], we formalize how the metrics interact among them. First, being the distance a trade-off among different dimensions, we need to normalize each coordinate. Availability and utilization are both in the range $[0, 1]$, thus we scale the latency coordinates to the same interval, dividing them by the largest observed delay. Second, since distinct applications can be differently affected by the performance indices, we introduce the weights w_l, w_a and w_u, for the latency, availability and utilization coordinates, respectively, to gauge the relative importance of the different performance indices. The distance between $\vec{P_i} = (P_{l1i}, P_{l2i}, P_{ai}, P_{ui})$ and $\vec{P_j} = (P_{l1j}, P_{l2j}, P_{aj}, P_{uj})$ is computed as follows:

$$d(\vec{P_i}, \vec{P_j}) = \sqrt{w_l^2 \left[\frac{(P_{l1i} - P_{l1j})^2 + (P_{l2i} - P_{l2j})^2}{Lat_{max}^2}\right] + w_a^2(P_{ai} - P_{aj})^2 + w_u^2(P_{ui} - P_{uj})^2}$$

3.5 Experimental Results

In this section, we present four sets of experiments aimed to investigate the potentialities and critical points of the distributed scheduling approach. First, we show how the distributed scheduler can optimize different QoS metrics (Section 3.5.1). We show the impact of QoS-awareness in improving performance of two well known applications, namely Log Processing and Word Count (Section 3.5.2). Then, we investigate how the QoS-aware scheduler reacts to changes and can adapt the placement of applications, when the application has different requirements in terms of resource demand and replication degree (Section 3.5.3). Leveraging on the most challenging configuration of Section 3.5.3, we investigate the distribute scheduler behavior when it is equipped with another decentralized policy (Section 3.5.4).

All the experiments have been performed on a Storm cluster composed of 8 worker nodes, each with 2 worker slots, and 2 further nodes for Nimbus and ZooKeeper. Each node is a virtual machine with Ubuntu 14.04 LTS configured with one vCPU on an Intel Xeon E5504 Quad-code and 2 GB of RAM. We emulated wide-area network latencies among the Storm

Table 3.1: Parameters of the Gaussian distributions used by netem (values are in ms)

Storm node	Mean μ	Standard deviation σ
1	15	1
2	23	2
3	19	3
4	25	1
5	12	3
6	17	2
7	32	1
8	27	2

Table 3.2: Parameters of the QoS-aware scheduler

Parameter	Description	Value
T_{as}	Time interval between two executions of the AdaptiveScheduler	30 s
T_{qm}	Time interval between two executions of the QoSMonitor	30 s
K_{cld}	Number of executions of the AdaptiveScheduler to be skipped for a cooling down topology	5
F_t	Force threshold of the VirtualOperatorPlacement algorithm	1.0
γ	Damping factor of the VirtualOperatorPlacement algorithm	0.1
$\#eps_{max}$	Maximum number of executors per worker slot	4
thr_{migr}	Migration threshold (relative distance)	0.15

nodes using *netem*[5], which applies to outgoing packets a Gaussian delay with mean and standard deviation reported in Table 3.1. Table 3.2 summarizes the parameters' setting of the distributed QoS-aware scheduler. The values of γ and F_t correspond to those used in [158].

The reference application, named tag-and-count, tags and counts sentences produced by a data source; its topology is represented in Figure 3.4 and is composed by a source, which generates 10 tuples/s, followed by a sequence of 5 operators before reaching the final consumer[6]. The source and the consumer are the pinned operators. The source can resend tuples at most once.

[5]Network Emulator: `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`

[6]Observe that, in our experimental setup, the presence of network delays and the *once-at-a-time* stream processing model of Storm require to limit the source data rate; the network is the system bottleneck.

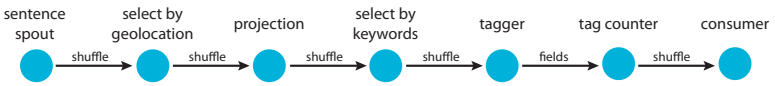


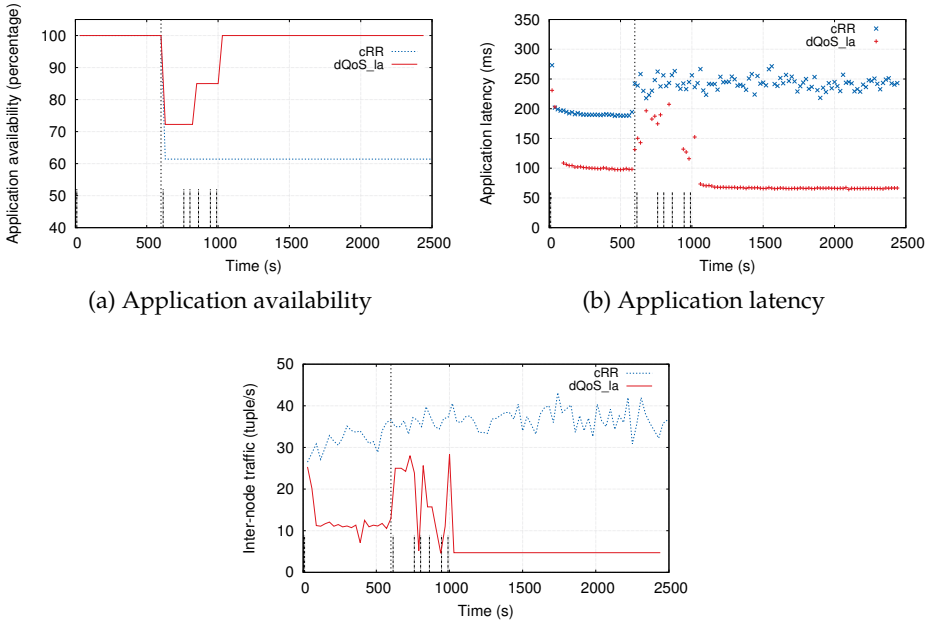
Figure 3.4: Tag-and-count topology

3.5.1 Optimizing QoS Metrics

In this section, we present two experiments that show the self-adaptation capabilities of the distributed QoS-aware scheduler (**dQoS_la** and **dQoS_lu**) and the default EvenScheduler of Storm (**cRR**) during the application execution. The former places operators exploiting QoS attributes as described in Section 3.4, whereas the latter uses a centralized round-robin policy.

Exploiting Availability Dimension. The first experiment investigates dQoS_la, which uses network latency and node availability as QoS attributes. We give more weight to availability by setting $w_a = 10$ and $w_l = 1$. Different metrics for this experiment are reported in Figure 3.5. We start the application with all nodes having 99.999% availability. After 600 s, the availability of an active node suddenly decreases to 85%; this event is represented with a vertical dotted line in Figure 3.5. Figure 3.5a shows the overall application availability, which is computed as $\prod_{i \in WN^q} a_i$ being the application q a sequence of operators, where WN^q is the set of worker nodes involved with the execution of q, and a_i is the availability of node i. Since the default scheduler is blind to this metric, when the node availability decreases, the overall availability decreases to 61.41%. As a consequence, the application end-to-end latency increases, because tuples spend more time in the upstream buffers and some of them are resent from the source (see Figure 3.5b). On the contrary, our distributed scheduler reassigns the operators to nodes with better availability; a new runtime reassignment performed by some distributed scheduler is indicated with vertical dot-dash lines in Figure 3.5. We can see that, after 1000 s, the application runs with an overall availability of 99.993%, which reduces the median of the application latency of about 72% with respect to that achieved by cRR. We can see that after 600 s, with dQoS_la there is a transient period of about 400 s (clearly visible in Figure 3.5c), where some distributed schedulers are executed to improve the application availability and reduce the network usage (Figure 3.5a). We also observe that during this experiment, the application scheduled with dQoS_la resent 2.53% of the tuples, while such percentage increases to 20.23% when scheduled with cRR.

Exploiting Utilization Dimension. The second experiment investigates dQoS_lu, which uses network latency and node utilization as QoS attributes. To make the placement decision resilient to minor fluctuations in the nodes' utilization, we weigh twice the latency with respect to the utilization ($w_l = 1; w_u = 0.5$). We start the application and, after 1200 s, we artificially in-



(a) Application availability

(b) Application latency

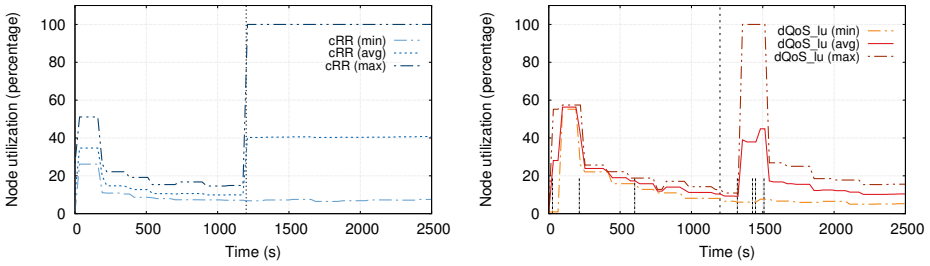
(c) Inter-node traffic

Figure 3.5: Comparison of dQoS_la with the default Storm scheduler (cRR) when the nodes' availability changes

crease the load on a subset of three nodes using the Linux tool *stress*. This subset is composed by one worker node running three application executors and two free worker nodes. Figure 3.6 shows the minimum, average, and maximum utilization of the subset of worker nodes that run the application, while the beginning of the stress event is represented with a vertical dotted line. Since dQoS_lu is aware of the execution environment, after a transient period which ends at 1500 s, it moves all the application operators on lightly loaded nodes (i.e., load balancing). Each new placement decision of a dQoS_lu scheduler is represented with a vertical dot-dash line in Figure 3.6b. On the contrary, as shown in Figure 3.6a, the default Storm scheduler cannot react and change its scheduling decision.

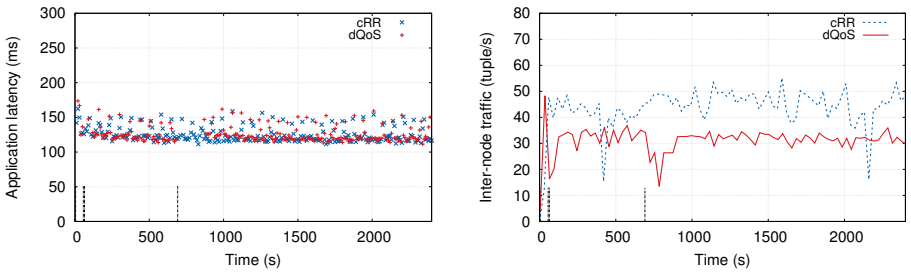
3.5.2 Performance with Well-known Applications

In this section, we aim to show the importance of QoS-awareness, especially when computing resources are interconnected with not negligible network latencies (see Table 3.1). We compare the performance of our dQoS scheduler, which uses network latency and node utilization as QoS attributes, against the centralized Storm scheduler (referred as cRR), which assigns operators in a round robin fashion. Similarly to the previous exper-



(a) Node utilization for cRR (b) Node utilization for dQoS_lu

Figure 3.6: Comparison of dQoS_lu with the default Storm scheduler (cRR) when the the nodes' utilization changes



(a) Application latency (b) Inter-node traffic

Figure 3.7: Performance of the Word Count topology

iment, dQoS weighs twice the latency with respect to the utilization (i.e., $w_l = 1; w_u = 0.5$). For this experiment, we consider two well-known DSP applications, namely Word Count and Log Processing (both as streaming applications).

Word Count. The Word Count topology is composed by a sequence of a source generating 1 tuple/s, two operators, and a consumer. The first operator splits the sentence into words and feeds the next one, which counts the occurrence of each word; each update of the counters is notified to the consumer. Source and consumer are pinned. We assign two executors to the source and three executors to each other operator. dQoS and cRR schedule the application on the same two worker nodes. As detailed in Section 3.4, the QoS-aware scheduler relies on the pinned operators to drive the placement of the executor pools. However, for the current topology the scheduler is constrained by the presence of at least a pinned operator for each executor pool. Therefore, the application latency shown in Figure 3.7a is comparable for the two scheduling strategies. Anyway, the dQoS scheduler

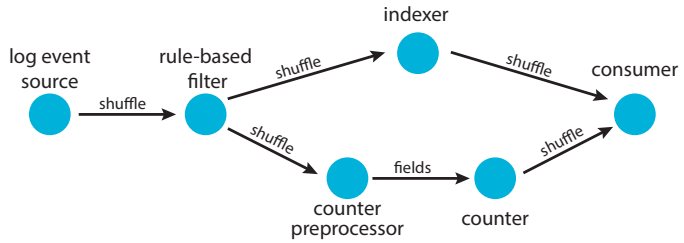
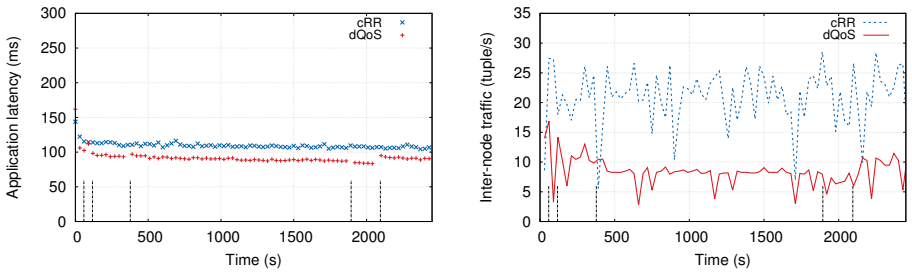


Figure 3.8: Log Processing topology

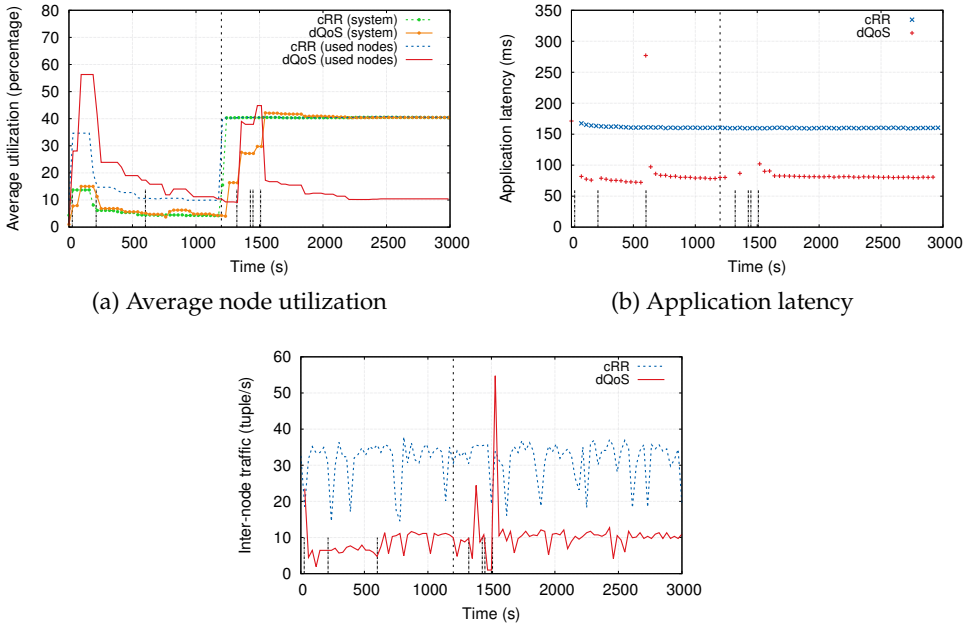


(a) Application latency (b) Inter-node traffic

Figure 3.9: Performance of the Log Processing topology

minimizes the network usage by re-arranging the executors on the worker nodes; the inter-node traffic is reduced of about 31% (see Figure 3.7b).

Log Processing. We have developed a log processing application based on `log-topology`[7], which relies on Redis as external caching service and whose topology is illustrated in Figure 3.8. The data source emits 10 log events/s that are filtered and sent to two different branches of the topology. The first branch is made of one operator, which indexes log events on Redis, while the latter is made of a sequence of two operators, which collect statistics. Subsequently, both the branches are merged together on the final consumer. The source and the consumer are pinned into the network. We assign one executor to each pinned operator and two executors for unpinned ones. From Figure 3.9 we observe that the QoS-aware placement improves the application performances, while reducing the network usage: the average application latency and the inter-node traffic are reduced by about 18% and 63% respectively.



(a) Average node utilization

(b) Application latency

(c) Inter-node traffic

Figure 3.10: Performance of the tag-and-count topology when the nodes' utilization changes

3.5.3 On Adaptation Capabilities

In this section, we evaluate how the distributed QoS-aware scheduler reacts to runtime changes when applications with different requirements are considered. Specifically, we rely on the tag-and-count application and propose a baseline setting, a configuration where operators have a higher CPU demand, and a configuration where operators are replicated. We compare the two schedulers when the load experienced by the worker nodes changes during the application execution due to same external noise event.

Baseline Case. We use the tag-and-count topology with a single executor for each operator. Figure 3.10 shows the evolution of the observed metrics; vertical dot-dash lines indicate a new runtime reassignment performed by some dQoS scheduler, while, on the contrary, the cRR scheduler does not intervene during the application execution. We start the application on an idle cluster. As soon as the application metrics are collected by the QoS-Monitor components, i.e., the exchanged data rate between operators and the node utilization are available, the dQoS scheduler performs some ad-

[7] `https://bitbucket.org/qanderson/log-topology/`

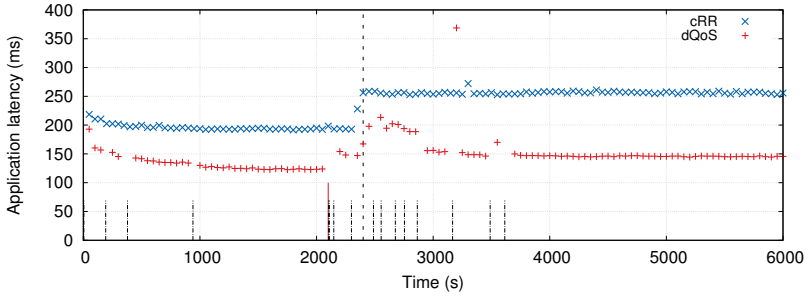
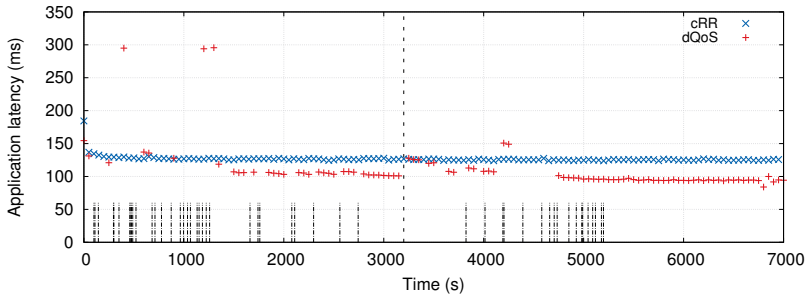


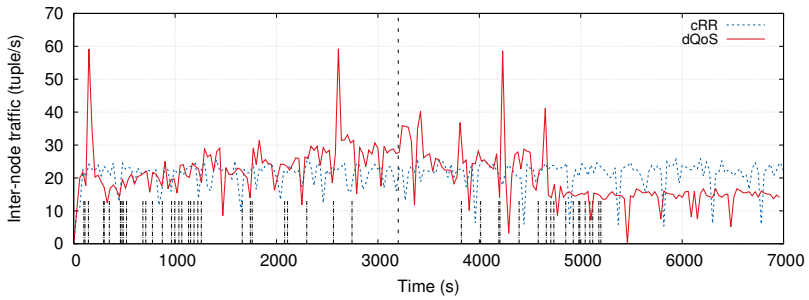
Figure 3.11: Average latency of the "heavy" version of the tag-and-count topology when the nodes' utilization changes

justments to the initial placement decision. After 1200 s, we artificially increase the load on a subset of three nodes using the Linux tool *stress*. This subset is composed by one worker node running two application executors and two free worker nodes. This event is represented with a vertical dotted line in Figure 3.10. Figure 3.10a shows the average utilization of the subset of worker nodes that run the application as well as the average utilization of the overall system. Soon after the event, the dQoS scheduler moves the application operators on lightly loaded nodes. The effectiveness of the decision is clearly visible in Figure 3.10a. Figure 3.10b shows that the presence of a stressed node does not degrade the application latency; this happens because the processing time of each operator is one order of magnitude lower than the network latency between worker nodes. Overall, the placement strategy of the dQoS scheduler reduces the application latency of about 49% with respect to that achieved by cRR. Furthermore, the inter-node traffic, reported in Figure 3.10c, shows a transient period for the dQoS scheduler after 1200 s which lasts for about 300 s. This period length depends on the number of reassigned operators and on the time interval each distributed scheduler must wait between two consecutive placement decisions (being $K_{cld} = 5$, this time interval is equal to 150 s).

Heavy Application. This second experiment investigates how the scheduler performs when a "heavy" application is submitted to the system. We modify the unpinned operators of the tag-and-count topology in order to waste some CPU time so that each operator completes its execution in about 14 ms, which is one order of magnitude larger than the baseline case. The load stress event is launched at 2450 s. Similarly to the previous experiment, the dQoS scheduler reacts to the load surge by reassigning operators, which impacts positively on the utilization (not shown for space reasons). However, in this case (see Figure 3.11) the degradation of the application latency for cRR is clearly visible, because the processing time is negatively influenced by the load stress event. On the other hand, dQoS allows to



(a) Application latency



(b) Inter-node traffic

Figure 3.12: Performance of the tag-and-count topology with replicated unpinned operators when the nodes' utilization changes

keep the latency low after a transient period. Between 1000 s and 2000 s, the average application latency experienced with dQoS is reduced by 35%, while between 4000 s and 5000 s it is reduced by 43%. We also observe that the BootstrapScheduler intervenes at 2100 s, because a worker process is erroneously terminated by the system.

Replicated Operators. This third experiment, reported in Figure 3.12, investigates the adaptation capabilities of the system when the unpinned operators of the tag-and-count application are replicated (i.e., two executors are assigned to each unpinned operator). The stress event is launched at 3200 s, and the distributed scheduler reacts to it improving the observed metrics: after the final placement, the application latency is reduced by about 25%. However, this experiment provides us a different insight on the distributed scheduler. First, we observe that the two transient periods (the initial one after the start and that after the load stress event) increase their length, because dQoS must deal with a larger number of operators, which, in turn, are more connected among them. Indeed, the number of logical links between pairs of operators grows from 6 to 20. A decentralized reassignment decision independently taken by one distributed scheduler pro-

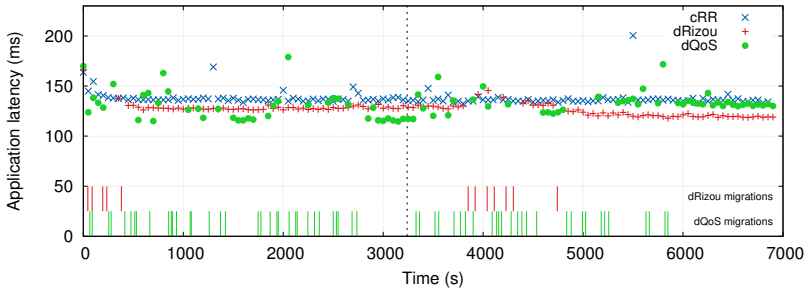


Figure 3.13: Performance of the tag-and-count topology when the nodes' utilization changes

vokes chain reactions in the other schedulers and the lack of coordination among them can thus originate some instability in the placement decisions. Even if the final application placement can improve the performance in terms of the observed metrics, the more frequent and larger number of operator reassignments and related stop-and-replay of the involved operators can negatively impact on the application availability during the transient periods. Second, the application latency obtained by the cRR scheduler is lower than that measured in the previous experiments. By comparing Figures 3.10c and 3.12b we can understand the reason: when the number of executors increases while keeping the number of worker process constant, the probability that the executors of two consecutive operators are on the same worker process increases as well. Therefore, data spend on average less time to traverse the topology.

3.5.4 Comparison with Another Distributed Policy

Starting from the outcomes of the previous section, here we evaluate a second distributed QoS-aware scheduling policy, which has been proposed by Rizou et al. [171]. Specifically, we investigate the adaptation capabilities of our distributed scheduler when it is equipped with the Rizou's placement policy (named as dRizou). We compare **dRizou** against the centralized Storm scheduler (named as cRR) and dQoS, that is our distributed scheduler equipped with the Pietzuch's algorithm. dRizou and dQoS assign operators to computing nodes by exploiting QoS attributes (i.e., latency, utilization), whereas cRR uses a round-robin policy.

The DSP application is tag-and-count, where the unpinned operators are replicated twice (as in the previous section). Figure 3.13 shows the evolution of the application end-to-end latency; on its bottom, we indicate the runtime reassignments performed by the distributed schedulers (cRR does not intervene during the execution). We start the application and, after 3240 s, we artificially increase the load on a subset of three nodes using

the Linux tool `stress`. The subset is composed by one worker node running some application executors and two free worker nodes. This event is represented in Figure 3.13 with a vertical dotted line. As the distributed scheduler (both dQoS and dRizou) perceives the change, it moves the application operators on lightly loaded nodes. dRizou reduces the application latency with respect to cRR of about $12.6\ \%$ (measured between 5000 s and the end of the experiment). Furthermore, differently from dQoS, dRizou converges with a lower number of reassignments, increasing the application availability.

3.6 Summary

We designed and implemented a distributed QoS-aware scheduler for DSP systems based on Storm, which is able of operating in a geographically distributed and dynamic environment. The experimental results provide many interesting information. First, we have shown that, when the computing environment comprises nodes with not negligible delays, our QoS-aware scheduler outperforms the default Storm scheduler (i.e., a centralized round-robin solution). Second, our extension enhances Storm with adaptation capabilities, which allow to react to changes in a distributed fashion. Moreover, our investigation also pointed out a weakness of fully decentralized scheduling algorithms: since each placement decision is taken in a reactive, distributed, and independent manner, for complex topologies involving many operators, they can determine some instability that affects negatively the application availability. This behavior fosters the design of lightweight coordination mechanisms that can improve performance of decentralized schedulers (as we will see in Chapter 9).

Since in recent years Storm has been widely adopted, we believe that Distributed Storm may represent a useful tool for the DSP-related community. Therefore, we have publicly released its code, which is available as an open source project on GitHub (`http://bit.ly/extstorm`). In the next chapters of this thesis, we will prototype our deployment solutions by leveraging on Distributed Storm.

Chapter 4

Optimal Operator Placement

> Several operator placement policies have been proposed in the literature, but they are based on different assumptions and optimization goals and, as such, they are not completely comparable with one another. We provide a general formulation of the optimal DSP placement, which takes explicitly into account the heterogeneity of computing and networking resources.

The operator placement problem consists in determining, within a set of available distributed computing nodes, the nodes that should host and execute each operator of a DSP application, with the goal of optimizing the QoS attributes of the application. From the previous chapters, we have seen that existing placement solutions are characterized by different modeling assumptions and optimization goals (e.g., [9, 158, 171, 213, 224]). As such, they are not completely comparable with one another. Moreover, there is no general formulation of the placement problem, which makes difficult to analyze and compare the different solutions. As a consequence, even though the state of the art proposes a wide set of solutions (see Chapter 2), we cannot easily select the most suitable one to be used in the emerging near-edge/Fog computing environment. Indeed, in a geographically distributed environment where network latencies are not negligible, the operator placement problem should explicitly take into account the heterogeneity of computing and network resources.

In this chapter, we propose *Optimal DSP Placement* (for short, ODP), a unified general formulation of the operator placement problem for distributed and networked DSP applications, which takes into account the heterogeneity of application requirements and infrastructural resources. Differently from prior approaches (e.g., [53, 94, 196, 226]), ODP provides a general modeling framework that can be easily extended to include new constraints and QoS attributes. At the same time, ODP provides a benchmark

against which other centralized and decentralized placement algorithms can be compared. The main contributions of this chapter are as follows.

- We model the ODP problem as an ILP problem (Sections 4.2 and 4.3), which can be used to optimize different QoS metrics. We first propose a basic formulation that considers only user-oriented metrics, i.e., the application end-to-end latency and availability. Then, to show how easily ODP can be extended, we include network-related QoS metrics, considering the exchanged data rate between the application operators as additional metric (Section 4.4). Similarly, other metrics (e.g., monetary cost, memory) can be considered.
- Leveraging on Distributed Storm (presented in Chapter 3), we develop a prototype scheduler that allocates the DSP application operators according to the ODP solution (Section 4.5).
- Using Storm, we show how ODP can optimize different user-oriented QoS metrics (Section 4.6.1) and compare the performance achieved by some centralized and decentralized placement policies (i.e., [158, 171, 213]) to that obtained by ODP (Section 4.6.2). To this end, we have also implemented into Storm the centralized placement policy in [213] and the decentralized one in [171].
- We extensively evaluate the computational cost of solving ODP, under different configurations of application requirements and resource capabilities, and examine two simple strategies to reduce the resolution time (Section 4.6.3).

4.1 Related Work

As extensively discussed in Chapter 2, several works provide a formulation of the operator placement problem; however, differently from our work, they consider homogeneous nodes, neglect network latencies, or focus on topologies with special properties.

Among the first proposals, Zhu et al. [226] study a placement problem which accounts for computational and communicational delays. However, they assume that a resource node can host at most a single operator; we consider this hypothesis not realistic in today's DSP systems, therefore our formulation enables the co-location of operators on a resource node, according to the node computational capacity. Eidenbenz et al. [53] analyze the placement problem for a subset of DSP application topologies, i.e., serial-parallel decomposable graphs. This allows them to exploit strong theoretical foundations and propose an approximation algorithm, which, however, can allocate operators only on resources with uniform capacity. Similarly to our approach, Benoit et al. [19] use an ILP problem to rep-

resent the placement problem for in-network stream-processing applications, which are a slightly different kind of applications, with the topology restricted to a binary tree of operators.

Thoma et al. [196] model the relationship between operators and resources, improving the expressiveness of the user constraints to include co-location, upstream/downstream, and attribute or tag-based constraints. We focus only on global QoS metrics and do not investigate this issue, however the related user constraints can be easily integrated within our placement model. It is worth mentioning the work by Huang et al. [93], where the authors elegantly rearrange the Multi-Constrained Optimal Path problem to model and solve the service composition problem. Nevertheless, their model allows only to express constraints on communication links but not on computing resources.

Stanoi et al. [188] focus on maximizing the rate of the input streams that the DSP system can support, acting on both the order of operators and the placement on the resource nodes. We do not consider operator re-ordering as possible.

Unlike all the above approaches, we provide a general formulation of the placement problem. On the one hand, it models composite and networked applications that, as DSP applications, can be represented by a directed acyclic graph. On the other hand, it models the heterogeneity of both computing and networking resources. Therefore, the proposed formulation can find the optimal placement by taking into account the QoS attributes of applications and resources, and is flexible enough to accommodate new QoS metrics. Thanks to the adjustment of suitable knobs, the meaning of "optimal placement" can change according to the application context. As a consequence, ODP provides a general framework for QoS optimization and comparison of different placement heuristics.

4.2 System Model and Problem Statement

Devising an optimal application deployment strongly depends on the assumptions made about the domain it will be applied to. A suitable model grasps these relevant assumptions, taking into account the heterogeneity of represented entities, i.e., application requirements and resource capabilities. Therefore, before going into the details of our placement model, we provide a formal description of the involved entities. For the sake of clarity, in Table 4.1 we summarize the notation used throughout the chapter.

Table 4.1: Main notation adopted in this chapter.

Symbol	Description
G_{dsp}	Graph representing a DSP application
V_{dsp}	Set of vertices (operators) of G_{dsp}
E_{dsp}	Set of edges (streams) of G_{dsp}
C_i	Resources required to execute $i \in V_{dsp}$
R_i	Execution time per data unit of $i \in V_{dsp}$ on a reference processor
G_{res}	Graph representing computing and network resources
V_{res}	Set of vertices (computing nodes) of G_{res}
E_{res}	Set of edges (logical links) of G_{res}
C_u	Amount of resources available on $u \in V_{res}$
S_u	Processing speed-up of $u \in V_{res}$
A_u	Availability of node $u \in V_{res}$
$d_{(u,v)}$	Network delay on $(u, v) \in E_{res}$
$A_{(u,v)}$	Availability of $(u, v) \in E_{res}$
$V^i_{res} \subseteq V_{res}$	Subset of nodes where $i \in V_{dsp}$ can be placed
$x_{i,u}$	Placement of $i \in V_{dsp}$ on $u \in V^i_{res}$
$y_{(i,j),(u,v)}$	Placement of $(i, j) \in E_{dsp}$ on $(u, v) \in E_{res}$

4.2.1 DSP Model

From an operational perspective, a DSP application is made of a network of operators connected by streams. An operator is a self-contained processing element that can execute a generic user-defined code, whether it is a predefined operation (e.g., filtering, aggregation, merging) or something more complex (e.g., POS-tagging), whereas a stream is an unbounded sequence of data (e.g., packet, tuple, file chunk).

A DSP application can be represented as a labeled directed acyclic graph (DAG) $G_{dsp} = (V_{dsp}, E_{dsp})$, where the nodes in V_{dsp} represent the application operators as well as the data stream sources (i.e., nodes with no incoming links) and sinks (i.e., nodes with no outgoing link), and the links in E_{dsp} represent the streams, i.e., data flows, between nodes. Due to the difficulties of formalizing a generic relationship between the operator code and its non-functional attributes, we consider each operator as a black-box component, assuming that its attributes, if not known a-priori, can be obtained thanks to empirical measurements or benchmarks. Each node $i \in V_{dsp}$ has the following attributes: C_i, the amount of resources required for its execution; and R_i, its execution time per unit of data on a reference processor. We assume, without loss of generality, that C_i is a scalar value, but our placement model can be easily extended to consider C_i as a vector of re-

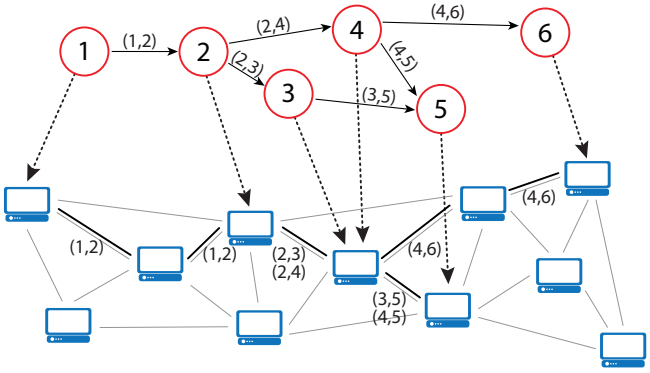


Figure 4.1: Placement of the application operators on the computing and network resources

quired resources. Furthermore, to avoid binding together applications and resources, we require R_i to be measured on a reference processor.

4.2.2 Resource Model

Computing and network resources can be represented as a labeled fully connected directed graph $G_{res} = (V_{res}, E_{res})$, where the set of nodes V_{res} represents the distributed computing resources, and the set of links E_{res} represents the *logical connectivity* between nodes. Observe that, at this level, links represent the logical link across the networks which results by the underlying physical network paths and routing strategies. The label associated with each node $u \in V_{res}$ specifies the following QoS attributes: C_u, the amount of resources available at node u; S_u, the processing speed-up on a reference processor; and A_u, its availability, i.e., the probability that u is up and running. The label associated with each link $(u, v) \in E_{res}$, with $u, v \in V_{res}$, specifies: $d_{(u,v)}$, the network delay between node u and v; and $A_{(u,v)}$, the link availability, i.e., the probability that the link between u and v is active. This model considers also loop links, i.e., edge of the type (u, u); they capture network connectivity between operators placed in the same node u, and are considered as perfect links, i.e., always active with no network delay. We assume that the considered QoS attributes can be obtained by means of either active/passive measurements or with some network support (e.g., Software Defined Networking — SDN).

4.2.3 Operator Placement Problem

The DSP placement problem consists in determining a suitable mapping between the DSP graph G_{dsp} and the resource graph G_{res} in a such a way

that all constraints are fulfilled. Figure 4.1 represents a simple instance of the problem. Observe that a DSP operator cannot be usually placed on every node in V_{res}, because of physical (i.e., *pinned* operator) or other motivations (e.g., security, political). This observation allows us to consider for each operator $i \in V_{dsp}$ a subset of candidate resources $V_{res}^i \subseteq V_{res}$ where it can be deployed. For example, if sources and sinks ($I \subset V_{dsp}$) are external applications, their placement is fixed, that is $\forall i \in I, |V_{res}^i| = 1$.

We can conveniently model the DSP placement with binary variables $x_{i,u}$, $i \in V_{dsp}$, $u \in V_{res}^i$: $x_{i,u} = 1$ if operator i is deployed on node u and $x_{i,u} = 0$ otherwise. A correct placement must deploy an operator on one and only one computing node; this condition can be guaranteed requiring that $\sum_u x_{i,u} = 1$, with $u \in V_{res}^i$, $i \in V_{dsp}$. We also find convenient to consider binary variables associated to links, namely $y_{(i,j),(u,v)}$, $(i,j) \in E_{dsp}$, $(u,v) \in E_{res}$, which denotes whether the data stream flowing from operator i to operator j traverses the network path from node u to node v. By definition, we have $y_{(i,j),(u,v)} = x_{i,u} \wedge x_{j,v} = x_{i,u} \cdot x_{j,v}$. For short, in the following we denote by $\boldsymbol{x}$ and $\boldsymbol{y}$ the placement vectors for nodes and edges, respectively, where $\boldsymbol{x} = \langle x_{i,u} \rangle$, $\forall i \in V_{dsp}$, $\forall u \in V_{res}^i$ and $\boldsymbol{y} = \langle y_{(i,j),(u,v)} \rangle$, $\forall x_{i,u}, x_{j,v} \in \boldsymbol{x}$.

4.3 Optimal Placement Model

There are several strategies to determine the deployment of a DSP application on a set of computing resources, as reviewed in Chapter 2. Each strategy has been developed following a specific objective, which, ultimately, can be generalized in the optimization of a specific utility function, such as the exchanged traffic, user-perceived latency, and resource utilization. In this section, exploiting tools provided by the optimization theory, we propose a model for the optimal operator placement problem that can be adjusted to satisfy different utility functions. First, we present a simple version of the model, which allows to better argument the design choices. Then, in Section 4.4, we illustrate the challenges of modeling other QoS constraints and attributes.

4.3.1 QoS Metrics

We first consider the application response time and availability, which are primarily user-oriented rather than system-oriented QoS metrics.

Response Time For a DSP application, with data flowing from several sources to several destinations, there is no unique definition of response

time. For any placement vector $\boldsymbol{x}$ (and resulting $\boldsymbol{y}$), we consider as response time $R(\boldsymbol{x}, \boldsymbol{y})$ the critical path average delay. We define the critical path of the DSP application as the set of nodes and edges, forming a path from a data source to a sink, for which the sum of the operator computational latency and network delays is maximal. Therefore, the critical path average delay is the expected traversal time of the critical path. Given this definition, we have that:

$$R(\boldsymbol{x}, \boldsymbol{y}) = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y}) \tag{4.1}$$

where $R_\pi(\boldsymbol{x}, \boldsymbol{y})$ is the end-to-end delay along path π and Π_{dsp} the set of all source-sink paths in G_{dsp}. For any path $\pi = (i_1, i_2, \ldots, i_{n_\pi}) \in \Pi_{dsp}$, where i_p and n_π denote the p^{th} operator and the number of operators in the path π, respectively, we obtain:

$$R_\pi(\boldsymbol{x}, \boldsymbol{y}) = \sum_{p=1}^{n_\pi} R_{i_p}(\boldsymbol{x}) + \sum_{p=1}^{n_\pi - 1} D_{(i_p, i_{p+1})}(\boldsymbol{y}) \tag{4.2}$$

where for any $i \in V_{dsp}$ and $(i, j) \in E_{dsp}$

$$R_i(\boldsymbol{x}) = \sum_{u \in V_{res}^i} \frac{R_i}{S_u} x_{i,u} \tag{4.3}$$

$$D_{(i,j)}(\boldsymbol{y}) = \sum_{(u,v) \in V_{res}^i \times V_{res}^j} d_{(u,v)} y_{(i,j),(u,v)} \tag{4.4}$$

denote respectively the execution time of operator i when mapped on node u and the network delay for transferring data from i to j when mapped on the path from u to v, where $i, j \in V_{dsp}$ and $u, v \in V_{res}$.

Availability We define the application availability A as the availability of all the nodes and paths involved in the processing and transmission of the application data streams. For the sake of simplicity, we assume the availability of the different components to be independent. We acknowledge that independence does not hold true in general and that a more detailed model is needed to capture the possibly complex dependency relationship among logical components sharing physical nodes and networks links. This will be subject of future work.

With the independence assumption, we readily have:

$$A(\boldsymbol{x}, \boldsymbol{y}) = \prod_{i \in V_{dsp}} A_i(\boldsymbol{x}) \cdot \prod_{(i,j) \in E_{dsp}} A_{(i,j)}(\boldsymbol{y}) \tag{4.5}$$

where

$$A_i(\boldsymbol{x}) = \sum_{u \in V_{res}^i} A_u x_{i,u} \tag{4.6}$$

$$A_{(i,j)}(\boldsymbol{y}) = \sum_{(u,v) \in V_{res}^i \times V_{res}^j} A_{(u,v)} y_{(i,j),(u,v)} \tag{4.7}$$

denote respectively the availability of the operator $i \in V_{dsp}$ and of the data stream from i to j, $(i,j) \in E_{dsp}$. To avoid dealing with multiplication, we consider the logarithm of the availability, obtaining:

$$\begin{aligned} \log A(\boldsymbol{x}, \boldsymbol{y}) = & \sum_{i \in V_{dsp}} \sum_{u \in V_{res}^i} a_u x_{i,u} + \\ & + \sum_{(i,j) \in E_{dsp}} \sum_{(u,v) \in V_{res}^i \times V_{res}^j} a_{(u,v)} y_{(i,j),(u,v)} \end{aligned} \tag{4.8}$$

where $a_u = \log A_u$ and $a_{(u,v)} = \log A_{(u,v)}$. Expression (4.8) deserves some comments. Let us focus on the first term and observe that the logarithm of the first factor of $A(\boldsymbol{x}, \boldsymbol{y})$, that is $\prod_{i \in V_{dsp}} A_i(\boldsymbol{x}) = \prod_{i \in V_{dsp}} \left(\sum_{u \in V_{res}^i} A_u x_{i,u} \right)$, is actually $\sum_{i \in V_{dsp}} \log \left(\sum_{u \in V_{res}^i} A_u x_{i,u} \right)$. In general, $\log \left(\sum_{u \in V_{res}^i} A_u x_{i,u} \right) \neq \sum_{u \in V_{res}^i} (\log A_u) \, x_{i,u}$; however, we observe that only one term of the sum in the expression $\log \left(\sum_{u \in V_{res}^i} A_u x_{i,u} \right)$ can be different from zero, because an operator is assigned to exactly one node (which implies that only variable in the set $\{x_{i,u}\}_{u \in V_{res}^i}$ is equal to 1). It follows that for any application placement $\boldsymbol{x}$, $\log \left(\sum_{u \in V_{res}^i} A_u x_{i,u} \right) = \sum_{u \in V_{res}^i} (\log A_u) \, x_{i,u}$, from which the first term in (4.8) directly follows. Similar arguments apply to the second term.

4.3.2 Optimal Placement Formulation

In a feasible assignment $\boldsymbol{x}$ (and associated $\boldsymbol{y}$), the computing resources $u \in V_{res}$ execute the application operators $i \in V_{dsp}$ with respect to their capabilities (i.e., C_u). Nevertheless, not all feasible assignments produce desirable application performances, thus we introduce a utility function that analytically defines an order relationship among all feasible solutions. The optimal placement results from the maximization of a given utility function. Depending on the utilization scenario, the utility function could be aimed at optimizing specific QoS attributes. These different optimization goals could be possibly conflicting, thus leading to a multi-objective optimization problem, which can be transformed into a single objective problem, using the SAW technique [218]. Therefore, we define the utility function $F(\boldsymbol{x}, \boldsymbol{y})$ as a weighted sum of the normalized QoS attributes of the

application, as follows:

$$F(\boldsymbol{x},\boldsymbol{y}) = w_r \frac{R_{\max} - R(\boldsymbol{x},\boldsymbol{y})}{R_{\max} - R_{\min}} + w_a \frac{\log A(\boldsymbol{x},\boldsymbol{y}) - \log A_{\min}}{\log A_{\max} - \log A_{\min}} \tag{4.9}$$

where $w_r,\ w_a \geq 0$, $w_r + w_a = 1$, are weights for the different QoS attributes. $R_{\max}$ ($R_{\min}$) and $A_{\max}$ ($A_{\min}$) denote, respectively, the maximum (minimum) value for the overall expected response time and availability. Observe that after normalization, each metric ranges in the interval $[0, 1]$, where the value 1 corresponding to the best possible case, that is $R(\boldsymbol{x},\boldsymbol{y}) = R_{\min}$, $\log A(\boldsymbol{x},\boldsymbol{y}) = \log A_{\max}$, and 0 to the worst case, $R(\boldsymbol{x},\boldsymbol{y}) = R_{\max}$, $\log A(\boldsymbol{x},\boldsymbol{y}) = \log A_{\min}$.

The Optimal DSP Placement (ODP) can be formulated as an ILP model as follows:

$$\max_{\boldsymbol{x},\boldsymbol{y},r} F'(\boldsymbol{x},\boldsymbol{y},r)$$

where

$$F'(\boldsymbol{x},\boldsymbol{y},r) = w_r \frac{R_{\max} - r}{R_{\max} - R_{\min}} + w_a \frac{\log A(\boldsymbol{x},\boldsymbol{y}) - \log A_{\min}}{\log A_{\max} - \log A_{\min}}$$

subject to:

$$r \geq \sum_{p=1}^{n_\pi} \sum_{u \in V_{res}^{i_p}} \frac{R_{i_p}}{S_u} x_{i_p,u} + \sum_{p=1}^{n_\pi - 1} \sum_{(u,v) \in V_{res}^{i_p} \times V_{res}^{i_{p+1}}} d_{(u,v)} y_{(i_p,i_{p+1}),(u,v)} \quad \forall \pi \in \Pi_{dsp} \tag{4.10}$$

$$\sum_{i \in V_{dsp}} C_i x_{i,u} \leq C_u \quad \forall u \in V_{res} \tag{4.11}$$

$$\sum_{u \in V_{res}^i} x_{i,u} = 1 \quad \forall i \in V_{dsp} \tag{4.12}$$

$$x_{i,u} = \sum_{v \in V_{res}^j} y_{(i,j),(u,v)}\ \forall (i,j) \in E_{dsp},\ u \in V_{res}^i \tag{4.13}$$

$$x_{j,v} = \sum_{u \in V_{res}^i} y_{(i,j),(u,v)}\ \ \forall (i,j) \in E_{dsp}, v \in V_{res}^j \tag{4.14}$$

$$x_{i,u} \in \{0,1\} \quad \forall i \in V_{dsp}, u \in V_{res}^i \tag{4.15}$$

$$y_{(i,j),(u,v)} \in \{0,1\}\ \ \forall (i,j) \in E_{dsp}, (u,v) \in V_{res}^i \times V_{res}^j \tag{4.16}$$

In the problem formulation we replaced the utility function $F(\boldsymbol{x},\boldsymbol{y})$ with the objective function $F'(\boldsymbol{x},\boldsymbol{y},r)$ which is obtained from $F(\boldsymbol{x},\boldsymbol{y})$ by replacing $R(\boldsymbol{x},\boldsymbol{y})$ with the auxiliary variable r, which represents the application response time in the optimization problem, in order to obtain a linear objective function. Observe that, indeed, while F is not linear in $\boldsymbol{x}$, $\boldsymbol{y}$ since

$R(\boldsymbol{x}, \boldsymbol{y}) = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y})$ is a not linear term, F' is linear in r as well as in $\boldsymbol{x}$ and $\boldsymbol{y}$.

Equation (4.10) follows from (4.1)–(4.4). Since r must be larger or equal than the response time of any path and, at the optimum, r is minimized, $r = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y}) = R(\boldsymbol{x}, \boldsymbol{y})$. The constraint (4.11) limits the placement of operators on a node $u \in V_{res}$ according to its available resources; Equation (4.12) guarantees that each operator $i \in V_{dsp}$ is placed on one and only one node $u \in V_{res}^i$. Finally, constraints (4.13)–(4.14) model the logical AND between the placement variables, that is, $y_{(i,j),(u,v)} = x_{i,u} \wedge x_{j,v}$. It is worth observing that the proposed AND constraints formulation is motivated by recent work in [55] which, albeit in a different context, provides empirical evidence that this specific AND formulation leads to reduced computational time with respect to alternative formulations[1]. For the sake of comparison, we compared this formulation to the alternatives (also taken from [55]) and, interestingly, we experienced - on average - a tenfold speed-up.

Theorem 4.1. *The Optimal DSP Placement problem is an NP-hard problem.*

Proof. In order to verify the NP-hardness of the Optimal DSP Placement problem, it suffices to prove that the corresponding decision problem is NP-hard. The decision problem can be formulated as follows: For a given DSP application and a set of computing and network resources, is there a feasible placement of the application? To prove the NP-hardness of this decision problem, let us consider the special case where: the resources required by the DSP application C_{i_k} are integers, where $k = \{1, \ldots, n\}$ and n is the number of operators; the network has only two nodes, $V_{res} = \{u, v\}$, with capacity $C_u = C_v = (\sum_{k=1}^{n} C_{i_k})/2$; there are no network delays, that is $d_{u,v} = d_{v,u} = 0$, which allows us to ignore the variables $\boldsymbol{y}$; and operators can be placed on both nodes, $V_{res}^{i_k} = V_{res}$, $k = \{1, \ldots, n\}$. It is easy to realize that the resulting problem is the well known Partition problem [62] which is known to be NP-hard. Since this special case is NP-hard, the general decision problem is NP-hard as well. And since the original optimization problem is at least as hard as the decision problem, it follows that Optimal DSP Placement problem is NP-hard as well. □

4.4 Network-related Extension and QoS Metrics

The ODP model can be easily extended to account for other constraints and user- and/or system-oriented QoS metrics, such as energy consump-

[1] In [55], the authors provide experimental evidence that the constraints (4.13)–(4.14) yields *better* lower bounds with respect to alternative constraints formulations. This results in more aggressive bounding in the *bound* phase of the *branch and bound* algorithm of ILP solvers and thus less iterations and faster convergence to the optimal solution.

tion, bandwidth constraints, monetary cost, privacy. In this section, without lack of generality, we include network-related QoS metrics that have been widely used in the literature to assess the quality of the DSP placement algorithms, such as network usage [171], inter-node traffic [9, 213], and the so called elastic energy [158].

Bandwidth constraint. In the ODP formulation, we made the implicit assumption that the data streams traffic between operators does not saturate the logical link bandwidth. Since this assumptions might not hold true in practice, we can explicitly account for the limited logical link capacity in ODP as follows. For each pair of communicating operators i and j, $(i,j) \in E_{dsp}$, let $\lambda_{(i,j)}$ denote the average data traffic rate and, for each logical link $(u,v) \in E_{res}$, let $B_{(u,v)}$ denote available bandwidth. Then, similarly to the node computing resource constraints (4.11), we have the following network link available bandwidth constraints:

$$\sum_{(i,j)\in E_{dsp}} \lambda_{(i,j)} y_{(i,j),(u,v)} \leq B_{(u,v)} \quad \forall u,v \in V_{res} \tag{4.17}$$

It is important to observe that, while the amount of available node computing capacity C_u in (4.11) is a known quantity, the amount of available bandwidth $B_{(u,v)}$ in (4.17) needs to be estimated since it depends on the (typically unknown) network traffic which traverses the physical links comprising the network path between node u and v[2]. Unfortunately, bandwidth estimation is known to be a non straightforward process as it requires active — and rather traffic intensive — end-to-end measurements techniques [161]. Nevertheless, the situation dramatically changes if we consider networks with advanced capabilities, e.g., SDN. In these networks, it becomes possible to have access to network information and/or allocate resources so as to reserve the desired amount of bandwidth on each logical link [42].

Network-related QoS metrics. Following the network-aware DSP placement policies in the literature [9, 158, 171, 213], we define the following metrics: the inter-node traffic T, the network usage N, and an approximation of the elastic energy EE. Let $Z(\boldsymbol{y})$, $Z = T|N|EE$, denote the QoS attribute of the DSP application under the DSP placement policy $\boldsymbol{y}$, we have:

$$Z(\boldsymbol{y}) = \sum_{(i,j)\in E_{dsp}} Z_{(i,j)}(\boldsymbol{y}) \tag{4.18}$$

where $Z_{(i,j)}(\boldsymbol{y})$ is defined as follows.

[2]Actually, since two different logical links could share some physical links, the constraints (4.17) should be expressed in terms of the physical and not logical links capacity. This is, in general, not feasible since it would require complete knowledge of: 1) the physical network topology, 2) the links characteristics, and 3) the routing tables.

The *inter-node traffic* T is the overall amount of data exchanged per time unit between operators placed on different nodes. Therefore, using the placement policy $\boldsymbol{y}$, the stream $(i,j) \in E_{dsp}$ generates an inter-node traffic equals to:

$$T_{(i,j)}(\boldsymbol{y}) = \sum_{(u,v) \in V_{res}^i \times V_{res}^j : u \neq v} \lambda_{(i,j)} y_{(i,j),(u,v)} \tag{4.19}$$

where $\lambda_{(i,j)}$ is the data rate of the stream.

The *network usage* N is the amount of data that traverses the network at a given time; therefore, the stream $(i,j) \in E_{dsp}$ imposes a load expressed by:

$$N_{(i,j)}(\boldsymbol{y}) = \sum_{(u,v) \in V_{res}^i \times V_{res}^j : u \neq v} \lambda_{(i,j)} d_{(u,v)} y_{(i,j),(u,v)} \tag{4.20}$$

where $d_{(u,v)}$ is the network delay among nodes $u, v \in V_{res}$, with $u \neq v$.

In their paper [158], Pietzuch et al. indirectly minimize the network usage through the minimization of the elastic energy, which results from the equivalent system of springs that represents the application. Basically, their solution minimizes the amount of data that traverses each link weighted by the latency of the link itself. Hence, the stream $(i,j) \in E_{dsp}$ contributes to the elastic energy of the system with:

$$EE_{(i,j)}(\boldsymbol{y}) = \sum_{(u,v) \in V_{res}^i \times V_{res}^j : u \neq v} \lambda_{(i,j)} d_{(u,v)}^2 y_{(i,j),(u,v)} \tag{4.21}$$

Observe that, in all cases, $Z(\boldsymbol{y})$ is a linear function of $\boldsymbol{y}$.

Network-related utility function. The utility function $F(\boldsymbol{x}, \boldsymbol{y})$, previously defined in Equation (4.9), can be readily re-written to include the network-related QoS metric as follows:

$$F_{net}(\boldsymbol{x}, \boldsymbol{y}) = F(\boldsymbol{x}, \boldsymbol{y}) + w_z \frac{Z_{\max} - Z(\boldsymbol{y})}{Z_{\max} - Z_{\min}} \tag{4.22}$$

where $w_z \geq 0$ weighs the network-related attribute, $w_r + w_a + w_z = 1$, $Z = T|N|EE$, and $Z_{\max}$ and $Z_{\min}$ denote respectively the maximum and the minimum value for the network term.

For sake of clarity, we report the new formulation of the placement problem that considers the bandwidth constrains and the network-related metrics:

$$\max_{\boldsymbol{x}, \boldsymbol{y}, r} F'_{net}(\boldsymbol{x}, \boldsymbol{y}, r)$$

where

$$F'_{net}(\boldsymbol{x}, \boldsymbol{y}, r) = F'(\boldsymbol{x}, \boldsymbol{y}, r) + w_z \frac{Z_{\max} - Z(\boldsymbol{y})}{Z_{\max} - Z_{\min}}$$

subject to:

$$
\begin{aligned}
r \geq & \sum_{p=1}^{n_\pi} \sum_{u \in V_{res}^{i_p}} \frac{R_{i_p}}{S_u} x_{i_p,u} + \\
& \sum_{p=1}^{n_\pi - 1} \sum_{(u,v) \in V_{res}^{i_p} \times V_{res}^{i_{p+1}}} d_{(u,v)} y_{(i_p,i_{p+1}),(u,v)} \quad \forall \pi \in \Pi_{dsp} \\
B_{(u,v)} \geq & \sum_{(i,j) \in E_{dsp}} \lambda_{(i,j)} y_{(i,j),(u,v)} \quad \forall u \in V_{res}, v \in V_{res} \qquad (4.23) \\
\sum_{i \in V_{dsp}} & C_i x_{i,u} \leq C_u \quad \forall u \in V_{res} \\
\sum_{u \in V_{res}^i} & x_{i,u} = 1 \quad \forall i \in V_{dsp} \\
x_{i,u} = & \sum_{v \in V_{res}^j} y_{(i,j),(u,v)} \quad \forall (i,j) \in E_{dsp},\ u \in V_{res}^i \\
x_{j,v} = & \sum_{u \in V_{res}^i} y_{(i,j),(u,v)} \quad \forall (i,j) \in E_{dsp},\ v \in V_{res}^j \\
x_{i,u} \in & \{0,1\} \quad \forall i \in V_{dsp}, u \in V_{res}^i \\
y_{(i,j),(u,v)} \in & \{0,1\} \quad \forall (i,j) \in E_{dsp}, (u,v) \in V_{res}^i \times V_{res}^j
\end{aligned}
$$

where Equation (4.23) limits the amount of data that flows on the logical link $(u, v) \in E_{res}$, according to its available bandwidth $B_{(u,v)}$.

4.5 Storm Integration: S-ODP

We develop a new scheduler for Storm, named **S-ODP**, whose core is the model presented in Section 4.4, where we consider the data rate exchanged among the application operators. We refer the reader to Section 3.2 for a description of the Storm architecture and of its execution model. In order to design S-ODP, we have to address two issues: (1) how to adapt the ODP formulation to the specific execution entities of Storm, and (2) how to instantiate the ODP model with the proper QoS information about computing and networking resources.

As regards the first issue, we have to model the fact that the Storm scheduler places the application executors on the available worker slots, considering that at most $EPS_{\max}$ executors can be co-located on the same slot. Hence, S-ODP defines $G_{dsp} = (V_{dsp}, E_{dsp})$, with V_{dsp} as the set of executors and E_{dsp} as the set of streams exchanged between the executors. Since in Storm an executor is considered as a black box element, we conveniently assume unitary its attributes, i.e., $C_i = 1$ and $R_i = 1$,

$\forall i \in V_{dsp}$. The resource model $G_{res} = (V_{res}, E_{res})$ must take into account that a worker node $u \in V_{res}$ offers some worker slots $WS(u)$, and each worker slot can host at most $EPS_{\max}$ executors. For simplicity, S-ODP considers the amount of available resources C_u on a worker node $u \in V_{res}$ equals to the maximum number of executors it can host, i.e., $C_u = WS(u) \times EPS_{\max}$.

As regards the second issue, Storm allows to easily develop new centralized schedulers with the pluggable scheduler APIs. However, Storm is not aware of the QoS attributes of its networking and computing resources, except for the number of available worker slots. Since we need to know these QoS attributes to apply the ODP model, we rely on the Storm extension we presented in Chapter 3, that enables the QoS awareness of the scheduling system by providing intra-node (i.e., availability) and inter-node (i.e., network delay and exchanged data rate) information. This extension estimates network latencies using a network coordinate system, which is built through the Vivaldi algorithm [43], a decentralized algorithm having linear complexity with respect to the number of network locations. S-ODP retrieves, from the monitoring components of the extended Storm, the information needed to parametrize the nodes and edges in G_{dsp} and G_{res}. Specifically, it considers: the average data rate exchanged between communicating executors (i.e., $\lambda_{(i,j)}, \forall (i,j) \in E_{dsp}$), the node availability ($A_u, u \in V_{res}$), and the network latencies ($d_{(u,v)}, \forall u, v \in V_{res}$). Once built the ODP model, S-ODP relies on CPLEX© (version 12.6.2) for solving the placement problem.

From an operative prospective, Nimbus uses S-ODP to compute the optimal placement when a new application is submitted to Storm and when a failure of the worker process compromises the application execution. In the latter case, S-ODP invalidates the existing assignment and computes the new optimal placement. When information on the exchanged data rate is unknown (e.g., first run), S-ODP defines an early assignment and monitors the application execution to harvest the needed information. Then, S-ODP reassigns the application, solving the updated ODP model with the network-related QoS attributes.

4.6 Results

We analyze an extensive set of experimental results that aim at demonstrating the flexibility of the ODP formulation and investigating its scalability. Specifically, using S-ODP, we first analyze in Section 4.6.1 how the ODP formulation allows us to consider the optimization of user-oriented QoS metrics, such as response time and availability. Then, in Section 4.6.2 using S-ODP we demonstrate how our formulation represents a general frame-

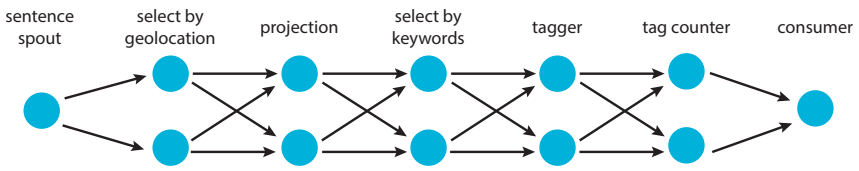


Figure 4.2: Tag-and-Count application

work and a benchmark for DSP placement optimization. To this end, we compare the performance achieved by some centralized and decentralized placement heuristics to that obtained by their corresponding optimal formulation based on ODP. Finally, since ODP is formulated as an ILP problem, we investigate through numerical experiments its scalability, analyzing the relationship between the solver resolution time and some model parameters, such as the application size and the number of resources. We close this section evaluating two simple approaches that can reduce the solver resolution time with a trade-off on the quality of the computed solution.

4.6.1 Optimizing User-oriented QoS Metrics

The ODP model we presented in Section 4.3 allows us to define the placement by optimizing different QoS attributes, whose importance depends on the utilization scenario. In this experiment, we use the S-ODP prototype to optimize different user-oriented QoS metrics, namely application response time and availability.

We perform the experiments using Apache Storm 0.9.3 on a cluster of 8 worker nodes, each with 2 worker slots, and 2 further nodes for Nimbus and ZooKeeper. A worker slot can host at most 4 executors, i.e., $EPS_{\max} = 4$. Each node is a virtual machine with one vCPU and 2 GB of RAM. We emulate wide-area network latencies among the Storm nodes using *netem*, which applies to outgoing packets a Gaussian delay with mean and standard deviation in the ranges $[12, 32]$ ms and $[1, 3]$ ms, respectively. Furthermore, half of the worker nodes has an availability of 99 % and the other of 100 %, whereas the links are always available. As test-case application we use Tag-and-Count, which tags and counts the sentences produced by a data source. Its topology is represented in Figure 4.2 and is composed by a source, which generates 10 tuples/s, followed by a sequence of 5 operators before reaching the final consumer. We also used this application in Chapter 3; however, here we consider a higher replication degree for its operators. More precisely, the source and the consumer, which are pinned operators, are instantiated with a single executor each, whereas all the others are instantiated with two executors each.

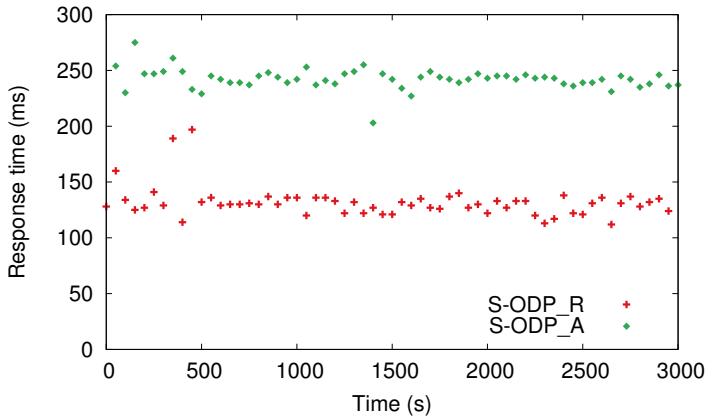
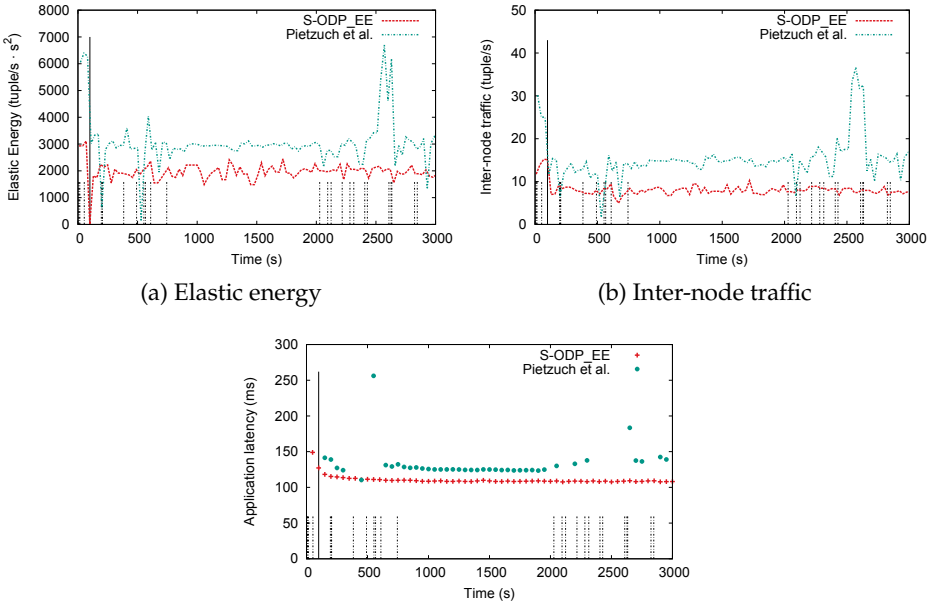


Figure 4.3: Application performance when ODP optimizes different user-oriented QoS metrics: response time (S-ODP_R) and availability (S-ODP_A)

We evaluate the effects on the application performance of two different configurations of S-ODP, namely S-ODP_R and S-ODP_A. **S-ODP_R** computes the placement by optimizing, as QoS metric, the response time R (we recall that R is the worst end-to-end delay on the source-sink path), i.e., it solves the ODP model with the utility function parametrized with weights $w_r = 1$ and $w_a = w_z = 0$. **S-ODP_A** maximizes the application availability A, by solving ODP with weights $w_a = 1$ and $w_r = w_z = 0$. Figure 4.3 shows the effects of the two utility functions on the application performance, expressed in terms of response time R. S-ODP_A places all the executors on the most available nodes, therefore the resulting application availability is 100%. However, since the scheduler does not consider the network latencies, the application experiences a response time that is, on average, 1.8 times higher than that achieved with S-ODP_R. The latter obtains the lowest response time, but, on the other hand, places 8 of 12 executors on worker nodes with 99% of availability. The resulting application availability is 92.27%, which is perceived with an increment of the tuple loss rate (3.37% during the experiment).

4.6.2 Evaluating Placement Heuristics

In this set of experiments we use the S-ODP prototype to evaluate some placement heuristics proposed in literature, namely the Pietzuch's, Rizou's, and Xu's algorithms (i.e., [158, 171, 213]). Since each solution has its own optimization goal, we conveniently adapt the ODP model as presented in Section 4.4. We use the same execution environment and application described in Section 4.6.1, except for 100% availability of all the nodes and



(a) Elastic energy

(b) Inter-node traffic

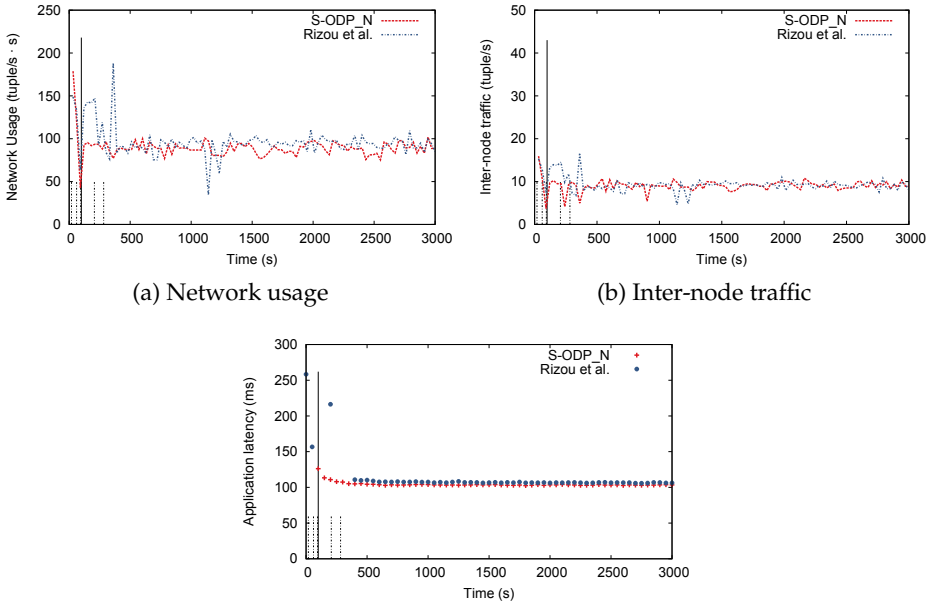
(c) Application latency

Figure 4.4: Comparison of placement decisions: Pietzuch's solution and optimal assignment (S-ODP_EE)

links. Furthermore, since network links have high available bandwidth, S-ODP omits the bandwidth constraints expressed in Equation (4.17). This setting allows for a fair comparison with the other placement solutions, here investigated, which do not take these QoS attributes into account.

Since the objective function of S-ODP needs the exchanged data rate information known only at runtime, S-ODP computes the placement in two steps, as described in Section 4.5. First, the scheduler defines the placement using weights $w_r = w_z = 0$ and $w_a = 1$. Then, it solves again the ODP model with weights $w_r = w_a = 0$ and $w_z = 1$, and reassigns the application. This rescheduling event happens at 100 s, and is represented with a vertical full line in the following figures. Also the schedulers under investigation can perform runtime reassignments; these events are represented with vertical dot-dash lines.

In this first experiment, we compare the performance of the decentralized Pietzuch's algorithm [158] with respect to the optimal placement computed by **S-ODP_EE**. The latter is the centralized S-ODP scheduler set up so to minimize the elastic energy of the spring system that represents the application, see Equation (4.21). Figure 4.4 reports the optimized metric (i.e., elastic energy), the average inter-node traffic, and the average application latency (i.e., the end-to-end delay of the tuples). We compute the



(a) Network usage

(b) Inter-node traffic

(c) Application latency

Figure 4.5: Comparison of placement decisions: Rizou's solution and optimal assignment (S-ODP_N)

elastic energy (Figure 4.4a) relying on real network latencies, because this allows to evaluate the behaviour of the Pietzuch's solution, neglecting the noise introduced by the network latency estimation system. The Pietzuch's algorithm finds a sub-optimal solution: considering the interval from 800 s to 2000 s, the inter-node traffic is on average 1.84 times higher than that achieved with the optimal placement, and the application latency is 1.15 times higher. The fluctuations introduced by the network latency estimation system are detrimental for this scheduling solution, which is very susceptible to this metric. As a consequence, even if no real environmental or application changes occur, the Pietzuch's algorithm can trigger some reassignments; see, for example, the behaviour of the system after 2000 s. The large number of reassignments required to reach a stable configuration (16 in this experiment) and the related stop-and-replay of the involved operators can negatively impact the application. Indeed, during the transient periods the application experiences unavailability, tuple loss, and peaks in traffic (Figure 4.4b) and latency (Figure 4.4c).

With the second experiment, we evaluate the decentralized Rizou's algorithm [171] with respect to the optimal placement determined by **S-ODP_N**, that is S-ODP configured to minimize the network usage, see Equation (4.20). Rizou et al. evaluated their proposal only through simulation. We have im-

plemented their algorithm in Storm in order to effectively compare its performance against the optimal placement achieved by S-ODP_N. Figure 4.5 reports the optimized metric (i.e., network usage), the inter-node traffic, and the overall application latency. Differently from the Pietzuch's solution, the Rizou's one directly minimizes the network usage. Comparing the Rizou's scheduling policy with the optimal one, we make the following observations. First, Rizou's shows good convergence, better than the Pietzuch's solution: a stable assignment is found in less than 300 s, after only 5 reassignments, thus confirming the simulation results presented by Rizou et al. in [171]. Second, latency weighs less on the scheduler's reassignment decisions, which makes the system less susceptible to fluctuations in its estimation. Lastly, in this specific, although simple, experimental scenario, the Rizou's algorithm finds the optimal placement for the application, effectively minimizing the network usage.

In the third experiment we evaluate the centralized scheduling policy proposed by Xu et al. [213] with respect to **S-ODP_T**, i.e., S-ODP that minimizes the inter-node traffic T, see Equation (4.19). In this case, no latency information is needed. Although the authors developed their solution for Storm, its code is not publicly available, so we re-implemented it adhering to the description in [213]. Differently from the previous heuristics, the Xu's scheduler is a centralized solution with a complete knowledge of the network, thus it can determine the placement relying on global information. Similarly to S-ODP, this scheduler needs a preliminary run of the application in order to extract bandwidth-related information. Hence, it initially places the application using a round-robin strategy; then, as soon as data rate information is available, it reassigns the operators using a best-fit heuristic. The latter tries to co-locate operators in decreasing order of exchanged data rate. Figure 4.6 compares the inter-node traffic achieved by the Xu's scheduler and our S-ODP_T. The Xu's algorithm performs a first reassignment at 100 s, which produces a sub-optimal configuration. In particular, there is a sequence of 3 executors that run on two nodes and exchange data using two network links instead of one. However, the Xu's scheduler detects the problem and fixes the application placement at 750 s, achieving the optimal solution computed by S-ODP_T.

In all these experiments, S-ODP computes the optimal solution in less than 420 ms.

4.6.3 Scalability Analysis

We now focus on the scalability analysis of ODP. To evaluate its computational cost, we solve the ILP problem on randomly generated problem instances, using an Amazon EC2 virtual machine (c4.xlarge with 4 vCPU and 7.5 Gb RAM). As computational cost metric, we use the resolution time

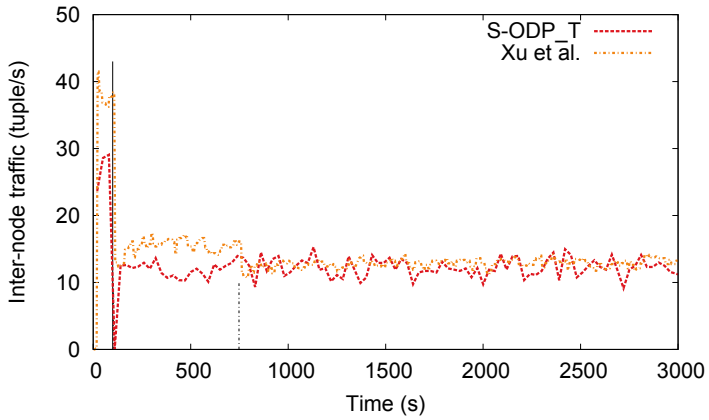


Figure 4.6: Comparison of placement decisions: Xu's solution and optimal assignment (S-ODP_T)

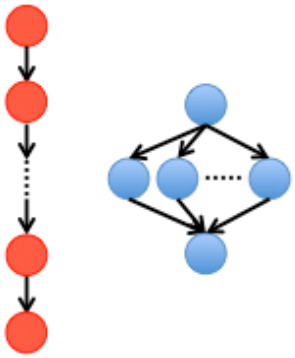


Figure 4.7: Sequential (left) and fat (right) applications

experienced by CPLEX. To avoid interferences among the model parameters, we use a baseline scenario and change a single factor at a time. The baseline scenario defines applications, computing and network resources with homogeneous characteristics. This represents the worst-case scenario for the CPLEX solver that, using a branch-and-cut resolution strategy, has to explore the whole solution space in order to find the optimum. If not otherwise specified, applications and resources are parametrized as reported in Table 4.2.

We consider two alternatives of layered topology for G_{dsp}, representing *sequential* and *fat* DSP applications, where each layer has one or more operators. The first and last layer contain the sources and sinks of the application, respectively. The DAGs of sequential (also known as pipelined) and fat applications are shown in Figure 4.7.

In the following experiments, G_{res} models a geographically distributed

Table 4.2: Parameters of application and resource models.

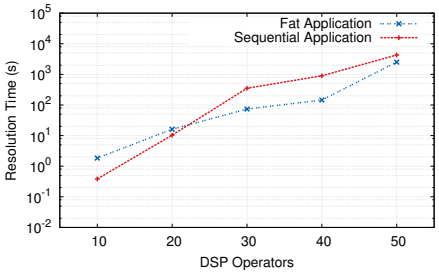
Model	Parameter	Value
Application	size of V_{dsp}	20
	R_i	1.0 s
	C_i	1
Resource	size of V_{res}	20
	A_u	100%
	C_u	4
	S_u	1.0
	$d_{(u,v)}$	$\max\{x \in \text{Gaussian } \mathcal{N}(22,5), 1\}$ ms
	$A_{(u,v)}$	100%

infrastructure, where computing nodes are interconnected with not-negligible network delays. We also assume that G_{res} is a fully connected graph, i.e., there always exists a logical link $(u, v) \in E_{res}$ between any two computing resources $u, v \in V_{res}$. We evaluate the ODP resolution time in relation to: *a)* the number of application operators, *b)* the number of computing resources, *c)* the percentage of available resources required by the application, and *d)* the capacity of each computing node.

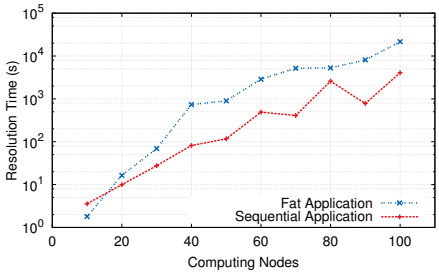
In the first experiment, we consider the ODP resolution time when the number of operators in the application increases from 10 to 50. Figure 4.8a shows the results. Note that, with 50 operators, the placement vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ contain 1000 and 19600 or 38400 variables for the sequential and fat applications, respectively.

The second experiment, depicted in Figure 4.8b, evaluates the resolution time when the number of computing resources ranges from 10 to 100 and the number of application operators is fixed to 20. With 100 computing resources, the placement vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ contain 2000 and 190K or 360K variables for the sequential and fat applications, respectively. It is worth observing that the ODP resolution time is more sensible to the increment in the number of application operators rather than the number of computing resources. For example, consider the scenarios $Sc_1 = \{V_{res} = 20, V_{dsp} = 50\}$ and $Sc_2 = \{V_{res} = 50, V_{dsp} = 20\}$. Both of them generate a placement vector $\boldsymbol{x}$ with 1000 variables; however, in Sc_2 (where the number of computing resources is increased), the resolution time decreases from 2 to 36 times for sequential and fat applications respectively, although the placement vector $\boldsymbol{y}$ is 2.38 times larger. The complexity of CPLEX does not easily reveal the motivations behind this behaviour.

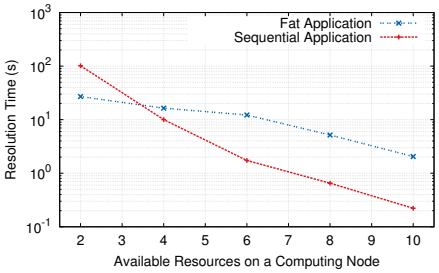
In the last experiment we investigate the impact of the amount of resources available on each computing node (C_u). We execute the applications with 20 operators on a network of 20 computing nodes, where the number of available resources C_u for each node $u \in V_{res}$ increases from 2



(a) Number of application operators $|V_{dsp}|$



(b) Number of computing nodes $|V_{res}|$



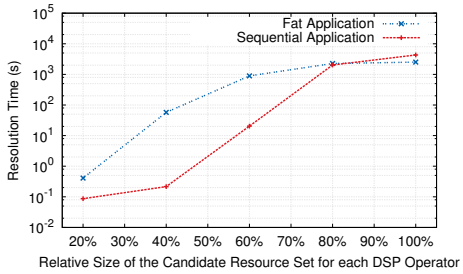
(c) Number of available resources C_u on each computing node

Figure 4.8: Computational cost of solving the ODP model in different settings

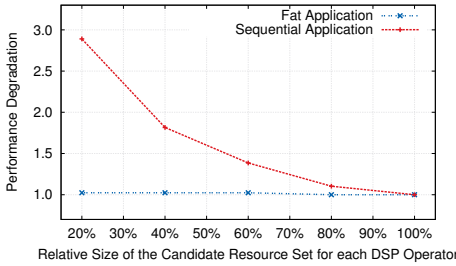
to 10. Figure 4.8c shows the results. The decreasing trend is readily motivated observing the objective function in Equation (4.9): since it penalizes network delays, the optimal placement is trivially determined when there is enough residual capacity on an already selected node, because operators are co-located on the same node. In an additional experiment not reported for the sake of space, we found that the load measured in terms of resources required by the application does not affect the resolution time.

As expected, the scalability experiments show that, when the problem cardinality increases significantly, solving analytically the ODP model is not feasible, even if CPLEX is very efficient. Even if efficient heuristics are required to solve large scale problems, we can consider two simple strategies to reduce the model resolution time: *a)* sampling the set of candidate computing nodes, and *b)* fixing an upper bound on the resolution time.

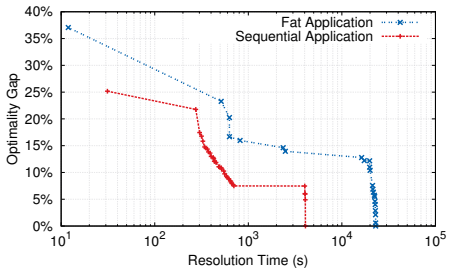
The benefits and drawbacks of reducing through sampling the set of candidate computing resources V_{res}^i where each operator i can be placed, are illustrated in Figures 4.9a and 4.9b, respectively. We define as *performance degradation* (Figure 4.9b) the ratio between the value of the objective function when the placement occurs only on a subset of candidate resources and the optimal value when all the resources can be used. We use



(a) Resolution time using a randomly sampled set of resources



(b) Performance degradation using a randomly sampled set of resources



(c) Feasible solutions and advances on the optimality gap

Figure 4.9: Effects of simple heuristic approaches on the ODP resolution time

a simple random sampling to identify the candidate resource subset for each operator. We run the experiment with 20 computing nodes and 50 operators for both the types of application topology. We observe from Figure 4.9a that the resolution time decreases tremendously (i.e., contracting up to about 5 orders of magnitude), when each operator has a subset of candidates that is less than 40% of all the available resources. Figure 4.9b points out that the solution quality strictly depends on the application topology: with only 20% of computing nodes, the fat application experiences a performance degradation only of 1.02, whereas the resolution time of the sequential application is about 3 times higher than the optimal one.

When we define an upper bound for the resolution time, CPLEX tries to find the optimal solution until the time interval is not exceeded. Then, the solver returns the best known feasible and sub-optimal solution. An appropriate upper bound strictly depends on the problem instance, but in this experiment we want to show how CPLEX closes the optimality gap during the resolution time. The *optimality gap* is a metric used by CPLEX to represent the relative gap between the objective function of the best integer solution and the best feasible solution that results from the linearization of the model (branch-and-cut strategy). We use the configuration that

reported the longest resolution time in the previous experiments: 100 computing nodes and 20 application operators. Figure 4.9c shows the trend of the optimality gap. Depending on the application topology, the solver behaves differently. For the sequential application, a near-optimal solution (i.e., with an optimality gap lower than 10%) is found in the first 500 s of computation. For the fat application, this convergence takes place just before the execution conclusion, thus it is harder to find a suitable upper bound for the resolution time. On the other hand, as shown in Figure 4.9b, the sampling technique seems to be a more effective strategy for the fat application.

4.7 Summary

In this chapter, we have presented a general formulation of the optimal placement problem (ODP) for data stream processing applications, which takes into account the heterogeneity of computing and network resources. ODP can optimize different QoS requirements of the applications and we have considered the end-to-end latency, availability, and exchanged data-rate among operators. The optimal placement provided by the ODP solution can be used as a benchmark framework against which to compare other centralized and decentralized placement algorithms. To this end, we have developed an ODP-based prototype scheduler for Storm and have compared some well-known placement solutions proposed in the literature. As the placement problem is NP-hard, a heuristic approach is needed to solve it in a feasible amount of time. However, the extensive scalability analysis of ODP has shown that, with particular application topologies, simple strategies can be successfully used. We will discuss extensively about heuristics in the next chapter.

Chapter 5

Heuristics for DSP Operator Placement

To solve the placement problem in a feasible amount of time, efficient heuristics are needed. To reduce the resolution time, the latter usually sacrifice the quality of the computed placement solution, thus leading to sub-optimal application performance. Nevertheless, when the application exposes stringent QoS requirements, the quality of the placement solution is of key importance. We design several new heuristics and show how they can achieve different trade-offs between resolution time and solution quality.

In the previous chapter, we have modeled the optimal placement for DSP applications, obtaining ODP. Then, we have shown that, being the placement problem NP-hard, ODP cannot be directly adopted in large-scale DSP systems, because it does not scale well as the problem size increases. However, it provides useful insights on the application placement and can be used to evaluate centralized and decentralized placement heuristics as well as to develop new placement solutions.

Due to the problem complexity, DSP systems usually adopt heuristics to determine the DSP application placement in a feasible amount of time. We are interested in considering geographically distributed environments, where the ever increasing presence of near-edge/Fog computing resources can be exploited so to improve scalability and reduce latency of DSP applications. Nevertheless, in this environment, we need to explicitly take into account the heterogeneity of computing and network resources, which, e.g., introduce not negligible communication latencies (see Chapter 1). As discussed in Chapter 2, several approaches have been already proposed in literature (e.g., [9, 14, 124, 158, 171, 226]). However, most of them have been designed to work in a clustered environment with negligible network

latencies (e.g., [64, 118, 174]), whereas others lack of flexibility and cannot easily optimize new placement goals (e.g., [158, 171]). Furthermore, a systematic analysis of the existing heuristics' performance, under different deployment configurations and with respect to the (theoretic) optimal solution, is almost always missing.

In this chapter, we present several heuristics aimed to solve the operator placement problem while considering the application QoS requirements as well as the heterogeneity of computing and network resources. While developing the heuristics, we have adopted guidelines focusing on the following three main aspects: *flexibility*, *optimal model awareness*, and *quality* of the computed placement solution. As regards *flexibility*, we have observed that many solutions are specifically crafted for optiming specific QoS metrics and cannot be easily customized or extended to account for new metrics (e.g., [34, 122, 158, 171]). Conversely, we aim to provide a general framework that can be easily tuned to optimize different QoS metrics (e.g., response time, availability, network usage) or a combination thereof. As regards *optimal model awareness*, we want to explore the possibility of using the ODP model in an efficient manner, aiming to determine high-quality placement solutions (as also proposed in [11, 188]). As regards *quality*, most of the existing heuristics usually determine best-effort solutions, meaning that they do not provide guarantees, quantitative, or qualitative information regarding their ability to compute near-optimal solutions. For example, many placement approaches rely on greedy strategies (e.g., [9, 14, 64, 174, 124, 213]) that, by moving through local improvements, can get stuck in locally optimal configurations, thus missing the identification of globally optimum ones. Together with the reduced resolution time, we aim to qualitatively evaluate the heuristics' ability to compute placement solutions which are as close as possible to the theoretically optimal ones. This is relevant when applications with stringent requirements run over heterogeneous and distributed infrastructures, where the inefficient utilization of the available resources can strongly penalize the application performance.

We design several heuristics for solving the operator placement problem and assess their quality using the optimal DSP placement model (i.e., ODP). The main contributions of this chapter are as follows.

- We design several heuristics, divided in two main groups: model-based and model-free. The *model-based* heuristics, presented in Section 5.4, revolve around the ODP model. They employ different strategies for identifying a set of candidate computing resources, which is then used to solve the placement problem using ODP. The very idea is to execute ODP on a suitable but limited set of computing nodes, thus reducing the problem size and, consequently, the resolution time. The *model-free* heuristics, presented in Section 5.5, implement the greedy first-fit, lo-

cal search, and tabu search approaches for the problem at hand. All the proposed heuristics rely on a penalty function that captures the cost of using any given resource of the computing infrastructure with respect to an ideal resource, characterized by infinity capacity, infinity computing speed and no network delay. Such function helps in driving the heuristics behavior towards the penalty minimization and allows to easily deal with single an multi-dimensional QoS attributes.

- We evaluate, through an extensive set of numerical experiments, the proposed heuristics under different configurations of infrastructure size, application topology, and optimization objective (Section 5.6). For each configuration, we use ODP as benchmark in order to investigate the heuristic performance in terms of resolution time and degradation of the computed solution. Unfortunately, there is not a *one-size-fits-all* heuristic, therefore we discuss how the deployment configuration changes the problem complexity and, in turn, the heuristic performance. Furthermore, by using empirical evidences, we identify the heuristic that, in general, achieves the best trade-off between resolution time and quality of the computed placement solution.

The ODP model plays an important role in this chapter. It drives the design of new heuristics and, most importantly, allows to assess the heuristic quality by providing, as benchmark, the optimal placement solution as well as the time needed to its computation.

5.1 Related Work

As extensively discussed in Chapter 2, the DSP placement problem has been widely investigated in literature under different modeling assumptions and optimization goals. The existing solutions aim at optimizing a diversity of utility functions, such as to minimize the application response time (e.g., [14, 34, 118, 174]), the inter-node traffic (e.g., [9, 66, 213, 224]), the network usage (e.g., [158, 171]), or a generic cost function that can comprise different QoS metrics (e.g., [11, 53, 160, 175, 197]). Inheriting the flexibility of ODP, our heuristics can optimize several QoS metrics, such as application response time, availability, network usage, or a combination thereof.

Furthermore, most of the existing solutions have been designed to work in a clustered environment, where network latencies are almost zero (e.g. [64, 118, 174]). Although interesting, these approaches might not be suitable for geo-distributed environments, where network latencies are not negligible and have a negative impact on the application performance. Some other works, although do not explicitly model the network, indirectly take into account the network contribution by minimizing the amount of data exchanged using the network (e.g., [9, 53, 58, 100, 213]). For example, Ei-

denbenz et al. [53] consider a special type of DSP application topologies (i.e., serial-parallel decomposable graphs) and propose a heuristic that minimizes a processing and a transfer cost. Nevertheless, the proposed solution works only on resources with uniform capacity. Fischer et al. [58] use a graph partitioning technique to optimize the amount of data sent between nodes. Relying on the best-fit meta-heuristic, Aniello et al. [9] and Xu et al. [213] propose centralized algorithms that assign operators with the goal of reducing the inter-node traffic exchanged on the network during the application execution. Specifically, the solution in [9] co-locates on the same node the pairs of operators that communicate with high data rate, whereas the proposal in [213] assigns each operator in descending order of incoming and outgoing traffic, minimizing the inter-node traffic. The same QoS metric is considered by the decentralized solution presented by Zhou et al. [224], which finds the placement while balancing the load among computing nodes.

Other works, e.g., [14, 34, 94, 158, 171], explicitly account for network latencies, thus representing more suitable solutions to operate in a geo-distributed DSP system. Pietzuch et al. [158] and Rizou et al. [171] minimize the network usage, that is the amount of data that traverses the network at a given instant. Both the solutions propose a decentralized placement algorithm that exploits a latency space as a search space to find the best placement solution in a completely distributed way (see Chapters 2 and 3 for further details). Backman et al. [14] and Chatzistergiou et al. [34] propose two different heuristic approaches to partition and assign group of operators while minimizing the application latency. Huang et al. [94] model the relationship between the operator execution time and the amount of residual computing capacity on a resource node and propose a best-fit heuristic that aims at minimizing the network usage. Also the solution by Zhu et al. [226] explicitly accounts for computational and communicational delays; however, they assume that a resource node can host at most a single operator. We consider this hypothesis not realistic in today's DSP systems, therefore our formulation enables the co-location of operators on a resource node, according to the node computational capacity. Similarly to this last group of works, we explicitly model the impact of network heterogeneity, in terms of latency, but also in terms of availability. Relying on ODP, our heuristics can easily account for other QoS metrics of computing and network resources, such as cost, bandwidth, or energy capacity.

So far, only few research efforts propose solutions specifically designed for Fog computing environments. SpanEdge [175] allows to specify which operators should be placed close to the data sources; exploiting this information, a heuristic defines the placement so to minimize a distance function. Differently, Arkian et al. [11] propose a non-linear formulation for de-

termining the placement of IoT applications over Fog computing resources, aiming to minimize a network-related cost function while guaranteeing the required applications response time. To reduce the resolution time, the authors recur to the problem linearization; nevertheless, in the previous chapter, we have shown that also linear formulations suffer from scalability issues. In this chapter, we provide new heuristics to solve the problem as so to efficiently deal with large instances of the placement problem. Using ODP as a benchmark, we also compare the resolution time of the heuristics against the one to solve the linear integer formulation.

As we discussed in Chapter 5, most of the existing approaches for solving the operator placement problem rely on mathematical programming (e.g. [11, 53, 171, 188]), graph theoretic (e.g., [58, 122, 226]), or greedy approaches (e.g., [9, 14, 64, 174, 124, 213]) as well as custom heuristics (e.g., [34, 158, 175]). We propose different approaches whose core is the optimal placement problem formulation, i.e., ODP. These approaches explore different strategies for selecting a suitable subset of computing resources, so to speed-up the resolution time of ODP. Moreover, we also implement several other approaches based on the most popular heuristics, namely greedy first-fit, local search, and tabu search. The work most closely related to ours has been proposed by Stanoi et al. [188]. Their solution focuses on maximizing the rate of the input streams that the DSP system can support, acting on both the order of operators and the placement on the resource nodes. We do not consider operator re-ordering as possible. Together with the (nonlinear) problem formulation, the authors propose different heuristics, which rely on local search, tabu search, and simulated annealing. Differently from all the above mentioned works, we use the optimal placement model ODP as a benchmark against which we can compute the heuristics performances in terms of resolution time and quality of the computed solution. Therefore, we thoroughly evaluate the proposed heuristics under different configurations of application type, network configuration, and optimization objective.

A final remark regards the runtime adaptation of the application placement. In this chapter, we explicitly target the initial deployment. A common and simple approach to perform runtime adaptation relies on periodically solving the placement problem, so to accordingly update the application deployment (e.g., [103, 213]). Nevertheless, this approach does not consider the adaptation costs, which can be detrimental for the application performance (as shown in Chapter 3). We postpone the study of the runtime adaptation problem in Chapters 7–9.

5.2 Heuristics: Overview

As demonstrated in Chapter 4, ODP is NP-hard. For supporting online operations, we develop several new heuristics for solving the operator placement problem. We present them as belonging to two main groups: model-based and model-free heuristics.

All of them aim to solve the ODP problem while minimizing the objective function F, defined as follows[1]:

$$F(\boldsymbol{x},\boldsymbol{y}) = w_r \frac{R(\boldsymbol{x},\boldsymbol{y}) - R_{\min}}{R_{\max} - R_{\min}} + w_a \frac{\log A_{\max} - \log A(\boldsymbol{x},\boldsymbol{y})}{\log A_{\max} - \log A_{\min}} + w_z \frac{Z(\boldsymbol{y}) - Z_{\min}}{Z_{\max} - Z_{\min}}$$

The terms $R(\boldsymbol{x},\boldsymbol{y})$, $\log A(\boldsymbol{x},\boldsymbol{y})$, and $Z(\boldsymbol{y})$ represent the application response time (4.1), its availability (4.8), and the network-related QoS metric (4.18), respectively. The terms $w_r,\ w_a,\ w_z \geq 0$, with $w_r + w_a + w_z = 1$, are weights for the different QoS attributes. The minimization of this objective function can be achieved by conveniently selecting the computing resources that will host and execute the DSP operators. To this end, the heuristics use a special *penalty function*, that defines an order relationship among resources, with respect to their ability in minimizing the objective function F.

The model-based heuristics are named Hierarchical ODP, ODP-PS, and RES-ODP. They try to restrict the set of candidate computing resources, before solving the ODP problem. *Hierarchical ODP* represents the computing infrastructure as organized in a hierarchy of virtual data centers (VDCs). Then, starting from the hierarchy root, this strategy recursively explores subsets of VDCs, until identifying the computing resources that will execute the application operators. Instead of aggregating resources in VDCs, *ODP on a Pruned Space* (ODP-PS) works directly with computing nodes. First, it runs a pruning algorithm so to identify a reduced subset of computing nodes, i.e., the best candidates for hosting the DSP operators. Then, it solves ODP by considering only this set of candidate computing resources. Differently, *Relax, Expand and Solve ODP* (RES-ODP) exploits the linear relaxation of ODP so as to identify a first set of candidates computing resources, which is then augmented by including some neighbor nodes of the candidates. Ultimately, RES-ODP solves the placement problem by considering only the set of candidate computing nodes. Sections from 5.4.1 to 5.4.3 present in detail the model-based heuristics.

The model-free heuristics implement some of the well-known meta-heuristics to solve the ODP problem. *Greedy First-fit* is one of the most popular approaches used to solve the bin-packing problem. Our implementation considers the DSP operators as elements to be (greedily) allocated in bins of finite capacity, which represent the computing resources.

[1] Observe that, differently from Chapter 4, we here define ODP as a minimization problem; therefore, the objective function F slightly differs from (4.22).

Local Search is an algorithm that, starting from an initial placement, greedily moves through the configurations that reduce the objective function F, until a stopping criterion is met (e.g., no further improvement can be achieved). Since it only accepts local improvements, it can get stuck in local optima, missing the identification of a global optimum solution. *Tabu Search* uses a simple strategy to escape from local optima and further explore the solution space, thus improving the probability of finding a globally optimal solution. Starting from an initial placement, through a set of iterations, it finds a local optimum; then, it explores the search space by selecting the best non-improving placement configuration, which can be found in the neighborhood of the local optimum. To avoid cycles back to an already visited configuration, the procedure uses a limited *tabu list* of previous moves that cannot be further explored. Sections from 5.5.1 to 5.5.3 present in detail the model-free heuristics.

The optimal placement depends on the location of data sources and sinks; all these heuristics assume that their placement is fixed a priori. If their location is not defined, we can conveniently pin them before solving the heuristics.

5.3 Resource Penalty Function

The heuristics involve, at different stages, the selection of suitable nodes and/or links to guide the placement decisions. To this purpose, we need to adopt a metric which captures the cost (in terms of objective function F and application QoS metrics) of using any given node/link resources for the application deployment and which can be used to efficiently compare different alternatives.

The problem would be trivial in an ideal setting where we could have access to a node with infinite capacity, infinite computing speed, and 100% availability, to which also the data sources ad data sink could be pinned: in such a case, we would just place all the operators on this single node. However, in real use cases, because of the limited capacity of a single node and the data sources and sinks distribution, we need to possibly deploy the application operators on several computing nodes. This placement introduces network delays and network traffic, and can also suffer from non-ideal node/link availability. To this end, it becomes natural to associate to any node/link resource a metric which captures the relative measure of degradation with respect to an ideal node/link resource, characterized by infinity capacity, infinite computing speed, and no network delay: the lower the penalty, the better the resource.

For our heuristics, it is sufficient to associate a metric to links. For each link $(u, v) \in E_{res}$ we introduce a link penalty function $\delta(u, v) \in [0, 1]$, which

assigns to a link $(u, v) \in E_{res}$ a *penalty* as a measure of degradation with respect to the ideal performance. Given a network link $(u, v) \in E_{res}$, we define the link penalty function $\delta(u, v)$ as the weighted combination of the QoS attributes of the link (u, v) and the associated upstream and downstream nodes $u, v \in V_{res}$, respectively. We have:

$$\delta(u, v) = w_r \delta_R(u, v) + w_a \delta_A(u, v) + w_z \delta_Z(u, v)$$

where $w_r, w_a, w_z \in [0, 1]$ are the same weights used in F. The terms $\delta_R(u, v)$, $\delta_A(u, v)$, and $\delta_Z(u, v)$ model the penalty with respect to the single QoS metrics, i.e., application response time, availability, and network usage, respectively. These terms are defined as follows:

$$\delta_R(u, v) = \frac{\tilde{R}(u, v) - \tilde{R}_{\min}}{\tilde{R}_{\max} - \tilde{R}_{\min}}$$
$$\delta_A(u, v) = \frac{\log \tilde{A}_{\max} - \log \tilde{A}(u, v)}{\log \tilde{A}_{\max} - \log \tilde{A}_{\min}}$$
$$\delta_Z(u, v) = \frac{\tilde{Z}(u, v) - \tilde{Z}_{\min}}{\tilde{Z}_{\max} - \tilde{Z}_{\min}}$$

where $\tilde{R}(u, v)$, $\log \tilde{A}(u, v)$, and $\tilde{Z}(u, v)$ capture the effects of using the link (u, v) on the application placement, in terms of the specific QoS metric. We compute these terms by considering the placement of two reference operators on u and v, respectively; the reference operators allow to neglect the application-specific contributions. The maximum and minimum values of $\tilde{R}(u, v)$, $\tilde{A}(u, v)$, and $\tilde{Z}(u, v)$ are respectively $\tilde{M}_{\max}$ and $\tilde{M}_{\min}$, with $\tilde{M} = \tilde{R}|\tilde{A}|\tilde{Z}$.

The response time $\tilde{R}(u, v)$ on the link (u, v) depends on the network delay $d_{(u,v)}$ and on the execution time of the reference operators on u and v, respectively. We have:

$$\tilde{R}(u, v) = \begin{cases} d_{(u,v)} + \frac{R_{\mathbb{1}}}{S_u} + \frac{R_{\mathbb{1}}}{S_v} & \text{if } u \neq v \\ 2\frac{R_{\mathbb{1}}}{S_u} & \text{if } u = v \end{cases}$$

where $R_{\mathbb{1}}$ is the (unitary) execution time of the reference operators and S_u and S_v are the processing speed-up of u and v, respectively. The term $\log \tilde{A}(u, v)$ gauges the probability that the link (u, v) and the nodes u and v are up and running; it is computed as:

$$\log \tilde{A}(u, v) = \begin{cases} \log A_{(u,v)} + \log A_u + \log A_v & \text{if } u \neq v \\ \log A_u & \text{if } u = v \end{cases}$$

where $A_{(u,v)}$, A_u, and A_v are the availability of (u, v), u, and v, respectively. The network usage $\tilde{Z}(u, v)$ models the amount of data that traverses the

network at a given time. Given the unitary data rate $\lambda_{\mathbb{1}}$ exchanged between the reference operators, we define $\tilde{Z}(u, v)$ as follows:

$$\tilde{Z}(u,v) = \begin{cases} \lambda_{\mathbb{1}} d_{(u,v)} & \text{if } u \neq v \\ 0 & \text{if } u = v \end{cases}$$

5.4 Model-based Heuristics

In this section, we present in detail the model-based heuristics, namely Hierarchical ODP, ODP-PS, and RES-ODP.

5.4.1 Hierarchical ODP

Hierarchical ODP represents the underlying infrastructure as organized in a limited number of entities, named *virtual data centers*[2] (VDCs). A VDC abstracts a group of computing nodes and related network links, which are exposed as an aggregated and more powerful computing element. If the computing infrastructure contains a very large number of resources, grouping them in VDCs may also result in a large number of VDCs. To further reduce the number of entities that represent the infrastructure, after having created a first level of VDCs, the heuristic can further aggregate the VDCs in a higher level of VDCs. This process results in a hierarchical representation of the computing infrastructure, where the number of resources decreases from bottom up. Hierarchical ODP exploits this structural property to iteratively solves ODP and filter out the VDCs (i.e., groups of resources) not suitable for running the DSP operators.

Specifically, Hierarchical ODP determines the application placement as presented in Algorithm 1 (see the SCHEDULE function). First, it represents the computing infrastructure as a hierarchy of VDCs, and stores each level of the hierarchy in H_{res}, an array of resource graphs (line 5). Soon after, it explores the infrastructure by navigating the hierarchy from the root down to the leaves: $H_{res}[l]$, with l from 0 to length(H_{res}). At level l, the heuristic determines the application placement on the available VDCs. If the placement solution S contains only computing resources in G_{res} (i.e., it does not contain any VDCs), the placement solution has been found (line 13). If the placement solution S includes any VDC, the heuristic updates the graph of resources at $H_{res}[l+1]$ by discarding every resource which is not included in any VDCs of $H_{res}[l]$ belonging to S (line 11). This operation filters out the computing resources that are not good candidates for hosting the ap-

[2]We use the term *virtual data center* without any correlation with the concept of physical data centers managed by service providers.

Algorithm 1 Hierarchical ODP

```
1: function HIERARCHICALODP(G_dsp, G_res, g)
2:     Input: G_dsp, DSP application graph
3:     Input: G_res, computing resource graph
4:     Input: g, grouping factor
5:     H_res ← CREATEHIERARCHY(G_res, g)
6:     for l in [0, ..., length(H_res)] do
7:         S ← solve ODP (G_dsp, H_res[l])
8:         if the placement S is on VDCs then
9:             remove from H_res[l] the VDCs not in S
10:            T_res ← inner resources of VDCs in H_res[l]
11:            remove from H_res[l + 1] nodes not in T_res
12:        else
13:            return S                          ▷ The placement S is on G_res
14:        end if
15:    end for
16: end function
17: function CREATEHIERARCHY(G_res, g)
18:    l ← ⌊log_g(|G_res|)⌉ − 1
19:    H_res[l] ← G_res                          ▷ H_res is an array of graphs
20:    while l ≥ 0 do
21:        k ← g^l
22:        C ← createClustersUsingKMeans(k, H_res[l])
23:        l ← l − 1
24:        H_res[l] ← createVDCfromClusters(C);
25:    end while
26:    return H_res
27: end function
```

plication operators. Afterwards, the heuristic solves ODP on the resource graph in $H_{res}[l+1]$; this process is repeated until line 13 is reached.

Observe that the hierarchical representation of the computing infrastructure makes the exploration of the solution space faster, thus improving the heuristic scalability. Indeed, although ODP is solved multiple times, each problem instance deals with a limited number of computing resources (either VDCs or resources in G_{res}).

On Resource Aggregation. CREATEHIERARCHY creates the hierarchical representation of the infrastructure (see Algorithm 1). First of all, it identifies the number of hierarchical levels l to be created, by considering the number of computing resources $|G_{res}|$ and a *groupingFactor* g: see line 18, where we use $\lfloor a \rceil$ to indicate the rounding of $a \in \mathbb{R}$ to the closest integer in $\mathbb{N}$. Then, the heuristic creates the hierarchical structure by proceeding in a bottom-up manner (lines 20–25). At level l, it uses k-Means [130] to determine $k = g^l$ groups of nodes, so as to minimize the average link penalty $\delta(u, v)$ between every pair of nodes u and v belonging to the same

group. Each group of resources will create a new VDC of the l-th level of the hierarchy (line 24). Each VDC is characterized by non-functional attributes that result from the inner computing and network resources. Let $\mathcal{C}_\alpha$ be the set of computing resources grouped around the same centroid α by k-Means, and let $\mathcal{U}$ be the VDC to be created. The availability of $\mathcal{U}$, $A_\mathcal{U}$, and the processing speed-up of $\mathcal{U}$, $S_\mathcal{U}$, are defined as the average availability and speed-up of the computing resources in $\mathcal{C}_\alpha$, respectively. Conversely, the amount of available resources, $Res_\mathcal{U}$, is the overall number of resources available in $\mathcal{C}_\alpha$. We have:

$$A_\mathcal{U} = \frac{\sum_{u \in \mathcal{C}_\alpha} A_u}{|\mathcal{C}_\alpha|}$$
$$S_\mathcal{U} = \frac{\sum_{u \in \mathcal{C}_\alpha} S_u}{|\mathcal{C}_\alpha|}$$
$$Res_\mathcal{U} = \sum_{u \in \mathcal{C}_\alpha} Res_u$$

At level l, the VDCs are interconnected by logical links that result from the connectivity between the computing resources at level $(l-1)$ of the hierarchy. We define $\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta$ as the set of links that connect an element in $\mathcal{C}_\alpha$ to an element in $\mathcal{C}_\beta$, where $\mathcal{C}_\alpha$ and $\mathcal{C}_\beta$ are two groups identified by k-Means: $\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta \doteq \{(u,v) | \, u \in \mathcal{C}_\alpha, v \in \mathcal{C}_\beta\}$. Let $\mathcal{U}$ and $\mathcal{V}$ be the VDCs created by $\mathcal{C}_\alpha$ and $\mathcal{C}_\beta$, respectively. The logical link $(\mathcal{U}, \mathcal{V})$ has availability $A_{(\mathcal{U},\mathcal{V})}$ and network delay $d_{(\mathcal{U},\mathcal{V})}$ defined as the average availability and the average network delay of the links in $\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta$, respectively. We have:

$$A_{(\mathcal{U},\mathcal{V})} = \frac{\sum_{(u,v) \in \mathcal{C}_\alpha \bowtie \mathcal{C}_\beta} A_{(u,v)}}{|\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta|}$$
$$d_{(\mathcal{U},\mathcal{V})} = \frac{\sum_{(u,v) \in \mathcal{C}_\alpha \bowtie \mathcal{C}_\beta} d_{(u,v)}}{|\mathcal{C}_\alpha \bowtie \mathcal{C}_\beta|}$$

5.4.2 ODP-PS: ODP on a Pruned Space

ODP on a Pruned Space (ODP-PS) computes the operator placement in two steps. First, it identifies a suitable subset of computing resources that can possibly host the application operators. Then, it executes ODP only on the candidate resources so to determine the operators placement. If the set of candidates leads to an unfeasible placement, the heuristic tries to expand this set before solving again ODP.

Algorithm 2 describes how ODP-PS works. To identify a subset of resources that can be shrunk or expanded as needed (line 4), the heuristic proceeds as follows. First, it combines resources in pairs $(u,v) \in E_{res}$ that minimize the penalty function $\delta(u,v)$. Then, as represented in Figure 5.1,

Algorithm 2 ODP-PS

```
1:  function ODP-PS(G_dsp, G_res)
2:      Input: G_dsp, DSP application graph
3:      Input: G_res, computing resource graph
4:      H_res ← create tree-like structure of subsets of resources
5:      P ← resources hosting the pinned operators of G_dsp
6:      T_res ← smallest set in H_res that strictly contains P
7:      do
8:          S ← solve ODP (G_dsp, T_res)
9:          if placement S not found and T_res equals G_res then
10:             return NOT_FEASIBLE
11:         end if
12:         T_res ← smallest set in H_res that strictly contains T_res
13:     while (placement S not found)
14:     return S
15: end function
```

the heuristic further combines pairs in sets, so that the summation of the penalty function $\delta(u, v)$ over every pair of resources within the same set is minimized. By leveraging on this representation (referred as H_{res}), ODP-PS can quickly prune the solution space and limit the number of computing nodes to be considered as candidates for solving ODP (lines 6 and 12). To define the application placement, ODP-PS first solves ODP on the smallest set of resources that host the pinned operators (i.e., data sources, sinks), named T_{res} in Algorithm 2. If this set of resources cannot execute the whole DSP application, the heuristic expands T_{res}, by considering the smallest set in H_{res} that strictly contains T_{res} (line 12); then, it solves again ODP on the updated resource set T_{res} (line 8). If the infrastructure does not contain enough computing resources (line 9), the heuristic terminates by asserting that a placement cannot be found. Differently from Hierarchical-ODP, ODP-PS computes the application placement directly on a subset of computing resources in G_{res}, whose cardinality can increase, in the worst case, up to the whole infrastructure size.

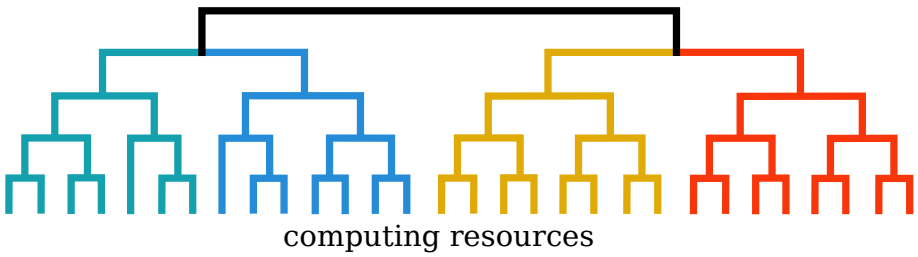


Figure 5.1: ODP-PS organizes the computing infrastructure in subsets of resources, so that the penalty function within each set is minimized.

Algorithm 3 RES-ODP

1: **function** RES-ODP(G_{dsp}, G_{res}, k)
2: **Input:** G_{dsp}, DSP application graph
3: **Input:** G_{res}, computing resource graph
4: **Input:** k, number of new neighbors to include
5: $S \leftarrow$ solve ODP (G_{dsp}, G_{res}), no integrality constraints
6: $\mathcal{G}_{res} \leftarrow$ candidate resources used in S
7: $\mathcal{G}_{res} \leftarrow$ add k neighbor for each resource in $\mathcal{G}_{res}$
8: $S \leftarrow$ solve ODP (G_{dsp}, $\mathcal{G}_{res}$), with integrality constraints
9: **return** S
10: **end function**

5.4.3 RES-ODP: Relax, Expand, and Solve ODP

Relax, Expand, and Solve ODP (for short, RES-ODP) uses the linear relaxation of ODP to identify an initial set of candidate computing nodes upon which ODP will be solved. As summarized in Algorithm 3, RES-ODP proceeds in three steps. First, it solves ODP by relaxing the integrality constraint for the placement variables $\boldsymbol{x}$ (line 5). Then, the computed placement solution is used to identify an initial set of candidates nodes (referred as $\mathcal{G}_{res}$ in line 6). While doing this, if the relaxed placement assigns an operator to multiple computing resources (i.e., for $i \in V_{dsp}, \exists u, v \in V_{res}, u \neq v$ such that $x_{i,u}, x_{i,v} > 0$), RES-ODP selects one of them as candidate node with a uniform probability. Afterwards, RES-ODP expands the set of candidates by adding k neighbors for each candidate (line 7). The best neighbors are identified exploiting the link penalty function and, to increase diversification, they are selected in a probabilistic manner: the lower the link penalty, the higher the probability to select the neighbor. We need to extend the set of candidate resources with respect to the one identified by the linear relaxation of ODP, because the latter can use a lower number of resources: indeed, by neglecting the integrality constraints, it can assign fractions of operators to resources, thus reducing resource wastage and fragmentation. Nevertheless, in our setting, a placement solution cannot fragment the operators. Finally, RES-ODP determines the application placement by solving ODP (with integrality constraints) on the extended set of candidate resources (line 8).

5.5 Model-free Heuristics

In this section, we present in detail the model-free heuristics, namely Greedy First-fit, Local Search, and Tabu Search.

Algorithm 4 Local Search

```
1: function LocalSearch(G_dsp, G_res)
2:     Input: G_dsp, DSP application graph
3:     Input: G_res, computing resource graph
4:     P ← resources hosting the pinned operators of G_dsp
5:     L ← resources of G_dsp, sorted by the cumulative
6:         link penalty with respect to nodes in P
7:     S ← solve GreedyFirst-fit(G_dsp, L)
8:     do                                                  ▷ local search
9:         F ← value of the objective function for S
10:        S ← improve S by co-locating operators
11:        S ← improve S by swapping resources
12:        S ← improve S by relocating a single operator
13:        F' ← value of the objective function for S
14:    while F' < F                          ▷ placement solution is improved
15:    return S
16: end function
```

5.5.1 Greedy First-fit

The *Greedy First-fit* heuristic determines the placement solution using a greedy approach [38]. For each DSP operator, this heuristic selects the computing resource from a sorted list L in a first-fit manner. Specifically, let P be the set of computing nodes that host the pinned operators (e.g., data sources, sinks). The heuristic adds to a list L the available resources $u \in V_{res}$, and sorts them in ascending order of overall link penalty; the latter is the summation of the link penalty between the node u in L and every resource in P, i.e., $\sum_{v \in P} \delta(u, v)$. Then, by navigating G_{dsp} in a breadth-first manner, the heuristic defines the placement of every DSP operator on the computing resources, which are selected from L in a first-fit manner.

5.5.2 Local Search

Local Search explores the solution space by moving from a placement configuration to the next one using a greedy approach. We summarize its behavior in Algorithm 4.

As first step, the heuristic creates L, a list of computing resources sorted in ascending order of cumulative penalty with respect to nodes hosting the pinned operators (see Section 5.5.1). Then, the heuristic computes the initial application placement using Greedy First-fit (line 7) and starts the local search. Specifically, it iterates to discover new placement configurations with lower value of the objective function F, until no further improvement can be achieved (lines 8–14). At each iteration, neighbor configurations of the current placement are explored, and the best one is chosen

as current configuration. Three exploration strategies are used, namely co-locate operators (line 10), swap resources (line 11), and move single operator (line 12).

Co-locate operators tries to assign two communicating operators on the same computing resource. Considering the initial configuration where $i \in V_{dsp}$ runs on $u \in V_{res}$ and $j \in V_{dsp}$ on $v \in V_{res}$, this strategy tries to co-locate i and j either on u or on v. *Swap resources* replaces an active computing resource u with a new one v from L; as a consequence, all the operators hosted on u are relocated on v. *Move single operator* relocates a single operator i from its location u to a new computing resource v, selected from L. Differently from the previous strategy, all other operators in u are not relocated. Observe that we run the local search until no further improvement can be found (line 14); however, a more stringent stopping condition can be used so as to limit the resolution time.

5.5.3 Tabu Search

The drawback of methods with local improvements (i.e., Greedy First-fit, Local Search) is that they might only find local optima, which nevertheless depend on the initial configuration, and miss the identification of global optima. Tabu Search increases the chances of finding a global optimum by moving, if needed, through non-improving placement configurations.

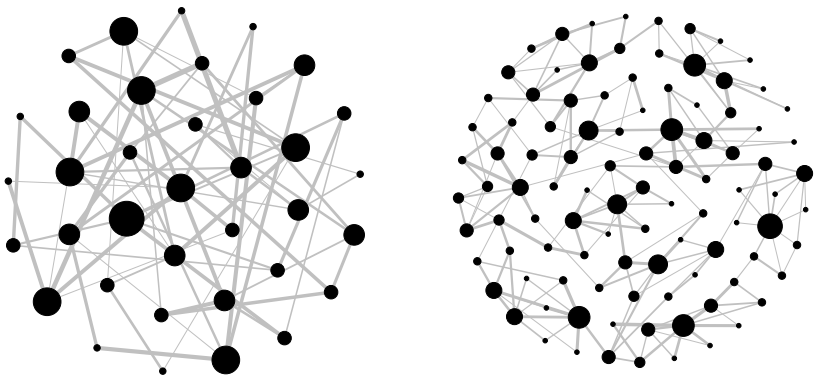
Algorithm 5 describes Tabu Search. It starts from an initial placement configuration, which is determined using Greedy First-fit. Then, it computes the neighbor configurations using the exploration strategies presented in Section 5.5.2 (i.e., *co-locate operators*, *swap resources*, and *move single operator*) and accepts the best improving placement (line 6). As soon as a local optimum is found, the heuristic continues to explore the search space by selecting the best non-improving configuration found in the neighborhood of the local optimum (line 12). This process increases the possibility of escaping from the local optimum and finding a new configuration that further decreases the objective function F (lines 17 and 24). To improve exploration and avoid cycles, the heuristic uses a tabu list (referred as tl), which contains the latest $tl_{\max}$ visited solutions which cannot be further explored. The heuristic terminates when no further improvement can be achieved (line 23). When the tabu search ends, the heuristic returns the overall best solution (line 24). Similarly to Local Search, although we run the heuristic until no further improvement can be found, a more stringent stopping condition can be used.

Algorithm 5 Tabu Search

```
1: function TABUSEARCH(G_dsp, G_res)
2:     Input: G_dsp, DSP application graph
3:     Input: G_res, computing resource graph
4:     S* ← NOT_DEFINED                                   ▷ best placement
5:     F* ← ∞
6:     S' ← LOCALSEARCH(G_dsp, G_res)                     ▷ local optimum
7:     F' ← objective function value for S'
8:     S ← S'
9:     tl ← create new tabu list and append S
10:    do
11:        improvement ← false
12:        S ← local search from S, excluding solutions in tl
13:        F ← objective function value for S
14:        if F = F* and S ∉ tl then
15:            tl.append(S)
16:        end if
17:        if F < F* and S ∉ tl then
18:            S* ← S; F* ← F
19:            tl.append(S)
20:            improvement ← true
21:        end if
22:        limit tl to the latest tl_max placement configurations
23:    while (improvement)
24:    if F' < F* then
25:        S* ← S'
26:    end if
27:    return S*
28: end function
```

5.6 Experimental Results

Relying on ODP as benchmark, we evaluate the efficacy and efficiency of the proposed heuristics under different utilization scenarios. We describe the experimental setup in Section 5.6.1. Then, in Section 5.6.2, we analyze the heuristics performance for different application topologies and computing infrastructures. Then, in Section 5.6.3, we discuss the impact of different objective functions on the heuristics performance, by evaluating single- and multi-objective optimization functions. Finally, in Section 5.6.4, we summarize the results and identify the heuristic that achieves, on average, a good trade-off between resolution time and quality of the placement solution.



(a) Smallest network, $|V_{res}| = 36$ (b) Largest network, $|V_{res}| = 100$

Figure 5.2: BRITE-generated reference networks. The size of nodes and links is proportional respectively to their connectivity degree and network delay, respectively

5.6.1 Experimental Setup

We run the experiments on a virtual machine with 4 vCPU and 8 GB RAM; CPLEX© (version 12.6.3), the state-of-the-art solver for ILP problems, is used to resolve ODP.

We consider several geographically distributed infrastructures, where computing nodes are interconnected with non-negligible network delays. We use BRITE [144] to generate infrastructures with $n^2 = \{36, 49, 64, 81, 100\}$ computing nodes, where the latter are interconnected in a two-layered top-down network: in the top-level, n autonomous systems (AS) communicate with high-speed links, whereas, within each AS, routers use slower links[3]. Each level is generated as a Waxman random graph [207]. The QoS attributes of computing and network resources, together with the random graph generation parameters, are summarized in Table 5.1. Figure 5.2 proposes a graphical representation of the smallest and the largest computing infrastructure. The size of nodes is proportional to their connectivity degree, whereas the size of (physical) links is proportional to their network delay. We assume that a *logical link* $(u, v) \in E_{res}$ between any pair of re-

[3]In BRITE, HS and LS specify the dimensions of the plane that will contain the generated topology. The plane is a square with side HS and it is internally subdivided into smaller squares with side LS.

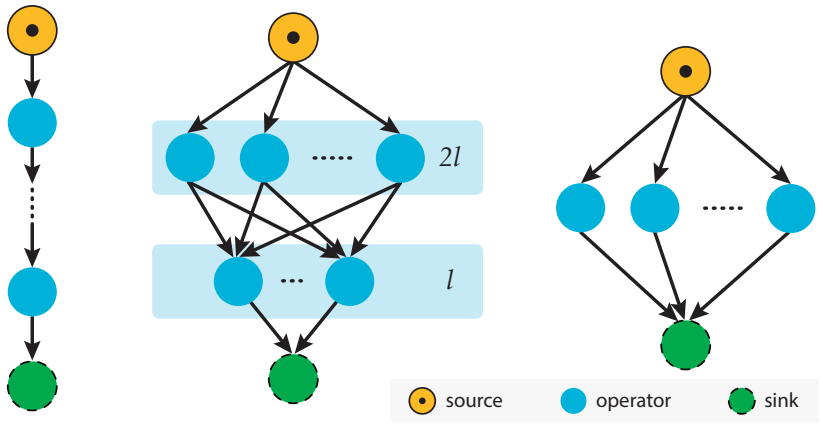


Figure 5.3: Sequential, replicated, and diamond applications

sources $u, v \in V_{res}$ always exists; it results by the underlying physical network paths and a shortest-path routing strategy.

We consider three alternatives of layered topology, representing *sequential*, *replicated*, and *diamond* DSP applications, where each layer has one or more operators. The first and last layer contain the sources and sinks of the application, respectively. The DAGs of sequential, replicated, and diamond applications are shown in Figure 5.3. The replicated application has one operator in the first and in the last layer, $2l$ operators in the second layer, and l in the third one. All the topologies contain the same number of operators. We assume that source and sink are pinned on the same node.

The baseline scenario defines applications, computing and network resources with homogeneous characteristics. This represents the worst-case scenario for the CPLEX solver that, using a branch-and-cut resolution strategy, has to explore the whole solution space in order to find and certificate the optimum. Applications and resources are parametrized as reported in Table 5.1. The latter also reports the normalization factors used by ODP, which have been computed using ODP with different optimization objectives[4].

Together with the model-based and model-free heuristics presented in Sections 5.4 and 5.5, we also consider two additional baseline approaches: ODP+T and Greedy First-fit (no δ). *ODP+T* limits the time interval granted to CPLEX for solving ODP through a timeout, that we set to 300 s: if the optimal solution has not been identified within this time interval, ODP+T returns the best solution it has computed. *Greedy First-fit (no δ)* determines

[4]Different normalization factors should be used for each combination of application and network topologies. However, in our experimental setting, the different network topologies have a limited impact on these factors, so we only consider different normalization factors for the different application topologies.

the placement by assigning DSP operators to the computing resources using a first-fit approach; differently from Greedy First-fit, this heuristic does not rely on the penalty function δ to sort resources in L (as other solutions in literature usually do, e.g., [9, 213]). We parametrize the different heuristics after preliminary experiments, aimed to minimize the quality degradation of the computed placement solutions: Hierarchical ODP uses a grouping factor $g = 2$; RES-ODP includes at most $k = 5$ neighbors for each candidate node; and Tabu Search limits the tabu list to $tl_{\max} = 1000$ configurations. Furthermore, every heuristic uses a timeout on the resolution time equal to 24 hours (value defined for practical reasons).

We compare the proposed heuristics against ODP in terms of resolution time and performance degradation. The *resolution time* (*rt*) represents the time needed to compute the placement solution. Based on this information, we define the *speed-up* (*sp*) as the ratio between the resolution time of ODP and the resolution time of the considered heuristic h, i.e., $sp_h = rt_{\text{ODP}}/rt_h$. Usually, the heuristics reduce the resolution time but also decrease the quality of the computed placement solution. To quantify how far is the placement solution by the heuristic h to the optimal placement, we define the *performance degradation* (*pd*) as $pd_h = (F_h - F_{\text{ODP}})/(1 - F_{\text{ODP}})$, where F_h is the objective function value by h and F_{ODP} is the optimal value by ODP. By definition, F_h ranges between F_{ODP} and 1; in turn, pd_h ranges between the best value 0 and the worst value 1.

Table 5.2 summarizes the experimental results that we will discuss in the following sections.

Table 5.1: Parameters of the experimental setup.

Infrastructure		**Application**	
$\lvert V_{res}\rvert$	$\{36, 49, 64, 81, 100\}$	$\lvert V_{dsp}\rvert$	20
A_u	$\mathcal{U}(97\%, 99.99999\%)$	C_i	1
C_u	2	R_i	3 ms
S_u	1.0	$\lambda_{(i,j)}$	100 tuples/s
$A_{(u,v)}$	100%		
avg $d_{(u,v)}$	17 ms		

BRITE's parameters used to generate the infrastructure network

Resource	**Random Graph**	**Latency Space**
AS	Waxman (α: 0.15; β 0.20)	HS: 1000; LS: 100
Routers in AS	Waxman (α: 0.15; β 0.20)	HS: 10000; LS: 1000

Normalization factors for the ODP objective function

Diamond Application					
$A_{\min}$	$A_{\max}$	$R_{\min}$	$R_{\max}$	$Z_{\min}$	$Z_{\max}$
58.8 %	97.2 %	74 ms	410 ms	132.2 KB	1409.2 KB
Sequential Application					
$A_{\min}$	$A_{\max}$	$R_{\min}$	$R_{\max}$	$Z_{\min}$	$Z_{\max}$
58.8 %	97.2 %	114 ms	3098 ms	8.4 KB	303.8 KB
Replicated Application					
$A_{\min}$	$A_{\max}$	$R_{\min}$	$R_{\max}$	$Z_{\min}$	$Z_{\max}$
58.8 %	97.2 %	49 ms	247 ms	52.0 KB	446.4 KB

Table 5.2: Heuristics comparison. For each heuristic, the first row reports the resolution time speed-up (sp), whereas the second one represents the performance degradation (pd). The last column reports the average value of speed-up and performance degradation obtained by considering the performance of all the experiments.

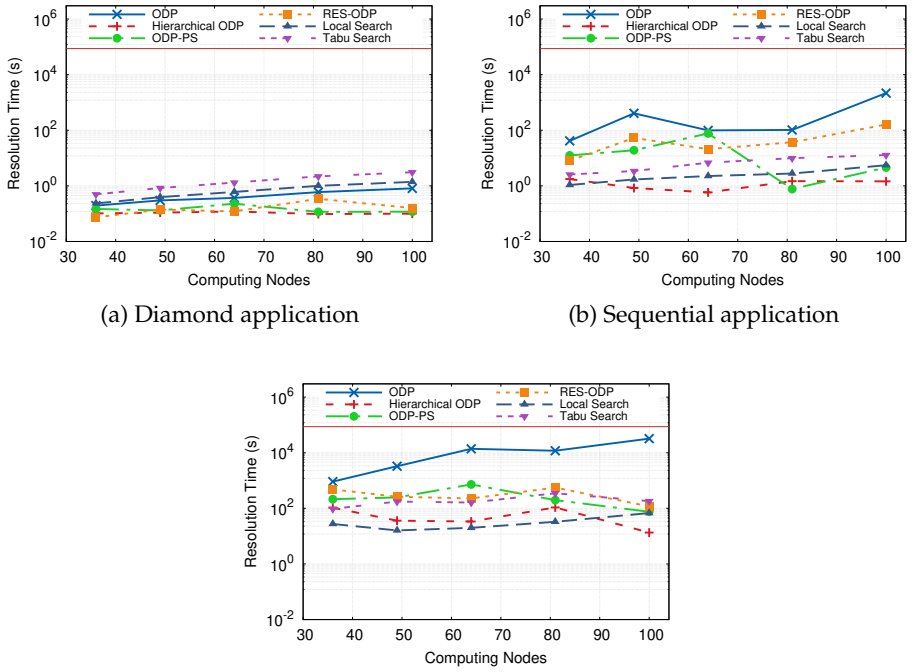
Algorithm		**Diamond Application**				**Sequential Application**				**Replicated Application**				**Average**
		w_a	w_r	w_z	$*$	w_a	w_r	w_z	$*$	w_a	w_r	w_z	$*$	
ODP	rt 36 (s)	0.1	0.1	0.1	0.7	0.7	41.4	25.2	31.1	8.2	915.2	19682.8	86404.5	
	rt 100 (s)	0.7	0.8	0.7	2.6	4.9	2174.8	388.1	5225.4	168.6	32193.9	86407.0	86411.6	
Hierarchical ODP	sp	17.19	4.40	14.46	6.26	74.25	448.39	170.30	1425.05	52.21	602.77	110.65	269.64	266.30
	pd	5%	17%	9%	14%	7%	3%	3%	8%	22%	11%	4%	11%	10%
ODP-PS	sp	0.88	3.45	10.79	3.29	1.28	127.17	71.07	896.99	1.02	104.71	52.44	180.92	121.17
	pd	0%	7%	2%	4%	0%	0%	0%	3%	0%	2%	2%	2%	2%
RES-ODP	sp	0.01	2.93	4.95	1.93	0.05	6.77	9.96	6.27	0.03	73.80	16.05	10.51	11.10
	pd	2%	0%	0%	1%	0%	0%	0%	0%	0%	0%	2%	0%	0%
Local Search	sp	0.09	0.68	0.45	1.64	0.47	150.54	72.47	333.99	0.38	353.07	587.85	1088.04	215.81
	pd	2%	0%	0%	1%	0%	1%	1%	2%	0%	4%	2%	3%	1%
Tabu Search	sp	0.07	0.31	0.21	0.84	0.26	65.91	30.64	151.26	0.16	64.93	207.63	480.08	83.53
	pd	2%	0%	0%	1%	0%	1%	1%	2%	0%	4%	1%	3%	1%
ODP+T	sp	1.74	1.88	1.79	2.22	1.98	2.32	0.96	5.90	1.42	40.75	134.63	261.35	38.08
	pd	0%	0%	0%	1%	0%	0%	0%	1%	0%	49%	0%	22%	6%
Greedy First-fit	sp	354.60	454.40	338.80	1949.32	2821.80	$56 \cdot 10^4$	$19 \cdot 10^4$	$168 \cdot 10^4$	$6 \cdot 10^4$	$12 \cdot 10^6$	$40 \cdot 10^6$	$78 \cdot 10^6$	$11 \cdot 10^6$
	pd	29%	0%	0%	2%	29%	7%	7%	12%	29%	5%	7%	8%	11%
Greedy First-fit (no δ)	sp	354.60	454.40	338.80	1949.32	2821.80	$56 \cdot 10^4$	$19 \cdot 10^4$	$168 \cdot 10^4$	$6 \cdot 10^4$	$12 \cdot 10^6$	$40 \cdot 10^6$	$78 \cdot 10^6$	$11 \cdot 10^6$
	pd	29%	34%	9%	15%	29%	7%	7%	10%	29%	24%	15%	16%	19%

5.6.2 Application Topologies and Network Size

In this experiment, we compare the performance of the heuristics against ODP, when the optimization objective is the minimization of the application response time R. This corresponds to set the weights $w_r = 1$, $w_a = w_z = 0$ to the objective function F (see Section 5.2). We evaluate the heuristics for different application topologies (i.e., diamond, sequential, replicated) and different size of the computing infrastructure (i.e., when the number of nodes grows from 36 to 100). Figure 5.4 reports the heuristic resolution time as the number of computing resources increases, and, as a horizontal red line, the timeout (set at 24 h). For sake of clarity, we do not represent ODP+T, whose resolution time is like the one by ODP limited to 300 s, and the Greedy First-fit approaches, whose resolution time is at most 1 ms. By aggregating performance over the different configurations of infrastructure size, Figure 5.5 shows the average speed-up on resolution time and the average performance degradation of the heuristics for the different application topologies.

Diamond Application. From Figure 5.4, we readily see that the application topology strongly influences the overall behavior of ODP and the heuristics. The diamond application has the lowest computational demand, which leads to a resolution time always below 10 s (see Figure 5.4a). Conversely, the replicated application is the most demanding one and, when ODP is used, the resolution time reaches 3×10^4 s, which is 3 orders of magnitude higher than the one experienced for the diamond application (see Figure 5.4c). When the diamond application has to be deployed, ODP and the model-based heuristics are very competitive and can quickly compute the placement solution in less than 1 s, even when the infrastructure includes 100 computing nodes. In this case, Local Search and Tabu Search perform worse than the others: they register a resolution time that grows up to 19 s on the largest infrastructure. Interestingly, these heuristics are slower then ODP (they obtain a speed-up factor lower than 1). Greedy First-fit, with and without the penalty function δ, computes the placement solution in 1 ms, independently from the infrastructure size.

Considering the performance degradation reported in Figure 5.5a, we can identify two main groups of heuristics. In the first one, we find the approaches that compute lower quality placement solutions (i.e., with not negligible performance degradation), namely ODP-PS, Hierarchial ODP, and Greedy First-fit (no δ). As expected, since Greedy First-fit (no δ) does not consider the QoS attributes of computing resources, it determines placement solutions having a very limited quality. Hierarchical ODP, the fastest model-based heuristic, shows a performance degradation of about 17%, whereas ODP-PS has a degradation of 7%. All the other heuristics identify the optimal placement. Interestingly, besides being very fast (with a speed-



(a) Diamond application

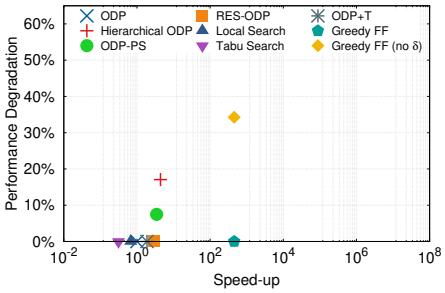
(b) Sequential application

(c) Replicated application

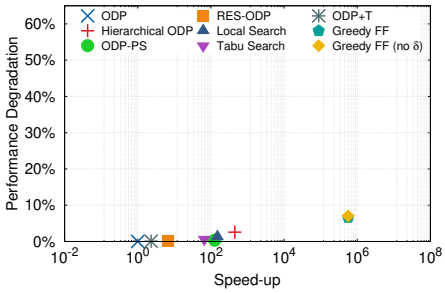
Figure 5.4: Resolution time of different policies to define the application placement that minimizes the response time R, when several application topologies and size of the computing infrastructure are considered.

up of 454 times), Greedy First-fit also identifies the optimal placement solution. As a consequence, also Local Search and Tabu Search identify the optimum; nevertheless, they are even slower than ODP.

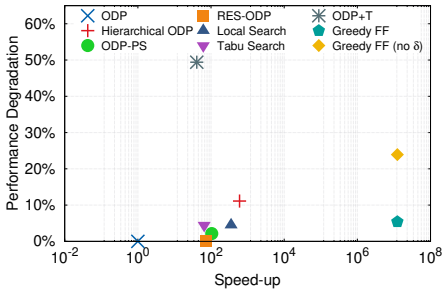
Sequential Application. When the sequential application has to be deployed on the computing infrastructure, the benefits of the heuristics are clearer: they can reduce the resolution time up to 3 orders of magnitude with respect to ODP. From Figure 5.4b, we can observe a non-monotonic trend on the resolution time, especially for ODP. It depends on the different network topologies that, being randomly generated, do not preserve exactly the same connectivity as the network size increases. Differently from the case of diamond applications, here all the heuristics have a resolution time smaller than the one by ODP, and they present only a limited performance degradation (see Figure 5.5b). More precisely, Tabu Search, ODP-PS, Local Search, and Hierarchical ODP have speed-up from 66 to 448 times, respectively, with a performance degradation always below 3% (see Table 5.2). Observe that, in this case, Local Search and Tabu Search are effective, because they improve the sub-optimal placement solutions identified by Greedy First-fit.



(a) Diamond application



(b) Sequential application



(c) Replicated application

Figure 5.5: Policies performance to compute the application placement that minimizes the response time R, when different application topologies are considered. Each point reports the average performance on the different infrastructure settings.

Replicated Application. The replicated application is characterized by a higher number of streams exchanged between operators; this makes the optimization function harder to minimize. In general, all the placement policies have a resolution time greater by one order of magnitude than the case of sequential application. From Figure 5.4c, we observe an exponential growth of the ODP resolution time when the number of computing resources increases. The model-based heuristics successfully restrict the solution space and present a resolution time that increases slowly as the infrastructure size grows. Considering the case of 100 resources, the slowest model-based heuristic, i.e., RES-ODP, has a resolution time in the order of 10^2 seconds. Local Search and Tabu Search obtain similar performance. As appears from Figure 5.5c, ODP-PS and RES-ODP are as fast as Tabu Search, but they compute a better placement solution (performance degradation of 2%, 0%, and 4%, respectively). In this case, Hierarchical ODP is faster than Tabu Search and Local Search with a speed-up of 603 times (instead of 353 and 65, respectively), however it has a higher performance degradation, i.e., 11%. The resolution time of ODP is prohibitively high; surprisingly,

ODP+T performs very badly in this case, reporting a 49% of performance degradation. Greedy First-fit is very beneficial for replicated applications, because of its limited resolution time (1 ms); moreover, the penalty function further improves this heuristic, by reducing performance degradation from 24% to 5%.

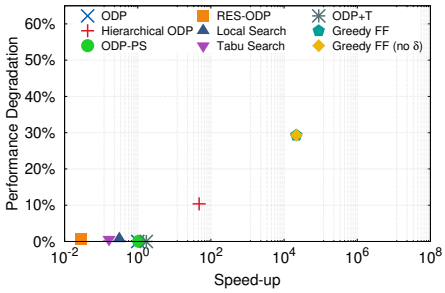
Discussion. In these experiments, the heuristics achieved different trade-offs between speed-up and performance degradation. ODP+T turned out not to be a good resolution approach: in case of replicated application, it obtained a performance degradation higher than Greedy First-fit (no δ). In most of the cases, Local Search and Tabu Search improved the placement solution with respect to Greedy First-fit. ODP-PS and RES-ODP behaved similarly to Local Search and Tabu Search, determining placement solutions with a limited performance degradation. The fastest model-based heuristic is Hierarchical ODP, which obtained, in the worst case, a rather high 17% of performance degradation (which is still lower than the one by Greedy First-fit (no δ)).

We summarize the outcomes of these experiments as follows. First, the application topology strongly influences the complexity of computing the optimal placement: a diamond application is less demanding than a sequential one, which, in turn, is less demanding than a replicated application. Solving the ODP model is feasible for diamond applications (it takes at most 0.8 s); nevertheless, this does not hold true for sequential and replicated applications, which lead to an increment of resolution time up to 9 hours. Second, the infrastructure size increases the resolution time of ODP; nevertheless, working on a subset of resources, the heuristics are less prone to increase their resolution time. In these experiments, when the infrastructure size increased from 36 to 100 resources, the resolution time of ODP has grown up to 35 times, whereas the resolution time of the model-based heuristics at most up to 20 times and of the other heuristics at most up to 6 times. Third, the penalty function δ, presented in Section 5.3, helps to improve the quality of placement solutions. Greedy First-fit is the fastest heuristic and, independently from the application topology, determines a placement solution in 1 ms. When the penalty function δ is used, Greedy First-fit has strongly reduced the performance degradation of the computed solutions.

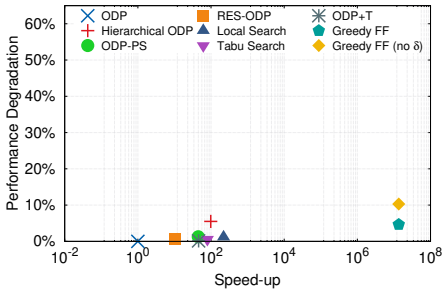
5.6.3 Optimization Objectives

In the previous set of experiments, we have investigated the heuristics behavior when they minimize the application response time. In this experiment, we first consider the other single-objective optimization functions (i.e., maximization of the application availability and minimization of network usage), and then we focus on a multi-objective function. In Ta-

(a) Minimization of the response time R



(b) Maximization of the availability A



(c) Minimization of the network usage N

Figure 5.6: Policies performance to compute the application placement, when different single-objective optimization functions are considered. Each point reports the average performance on different infrastructure settings and application topologies.

ble 5.2, we report the performance metrics for each combination of application topology and optimization objective. In the following, we describe the heuristics behavior aggregated by optimization objective: for each one, we consider the average value of speed-up and performance degradation which have been experienced for all the application topologies.

Response Time. In Section 5.6.2, we have considered the minimization of the response time as optimization objective (i.e., $w_r = 1$, $w_a = w_z = 0$). Figure 5.6a reports the heuristics performance when this objective function is considered (note that, in this figure, we aggregate results on the different application topologies). Figures 5.6b and 5.6c report the heuristic performance when the placement goal is the maximization of the application availability (i.e., $w_a = 1$, $w_r = w_z = 0$) and the minimization of network usage (i.e., $w_z = 1$, $w_a = w_r = 0$), respectively. From Figure 5.6, we can easily observe that the different optimization objectives lead to different performance of the heuristics. The minimization of response time and network usage result in similar trade-offs between speed-up and performance degradation. However, the maximization of availability deeply changes

the heuristics behavior.

Availability. In the evaluated settings, the maximization of the application availability imposes a limited computational demand. Even in the most complex configuration, i.e., ODP deploying a replicated application on an infrastructure with 100 nodes, the resolution time is at most 169 s (see Table 5.2). Interestingly, Local Search, Tabu Search, and RES-ODP perform worse than ODP (they have speed-up lower than 1). Being ODP very fast, ODP-PS does not significantly reduce its resolution time, obtaining a limited speed-up (see Figure 5.6b). We can also observe that most of the heuristics (i.e., all but Hierarchical ODP and Greedy First-fit) determine near-optimal placement solutions, introducing at most 2% of performance degradation. In particular, Table 5.2 reports that this is especially true for sequential and replicated applications, where the performance degradation is reduced to 0%. Similarly to the previous scenario, Hierarchical ODP has a rather high speed-up (up to 2 orders of magnitude), albeit it degrades the quality of the computed solution (10% on average). For this optimization function, Greedy First-fit is not very effective, even when it is equipped with the penalty function δ. With and without the penalty function, the performance degradation of the computed solution is close to 30%; this result clearly shows the benefit of performing a local or a tabu search so as to escape from the local optimum and improve the solution quality.

Network Usage. As we can see from Figure 5.6c, when network usage is optimized, the heuristics behave similarly to the case of response time minimization. Most of the heuristics have a speed-up of 2 order of magnitude and achieve a very limited performance degradation (always below 9%, except for Greedy First-fit (no δ)). As reported in Table 5.2, the resolution time changes widely with respect to the application topology. In general, ODP and the other heuristics can determine the placement of the sequential and diamond applications rather quickly (at worst, ODP takes 388 s). As regards the sequential application, optimizing the network usage is apparently less computational demanding than optimizing response time. Nevertheless, when the replicated application is considered, ODP has a very high resolution time and the need of heuristics is very well motivated. More precisely, when the infrastructure contains 100 computing resources, ODP always reaches the timeout and returns the best feasible solution (i.e., it requires more than 24 h to identify the optimal solution). Albeit not the optimum, the computed placement solutions after 24 h are better than the ones by the other heuristics. As shown in Figure 5.6c, the Greedy First-fit heuristics are still the fastest ones, computing the placement in 1 ms. Excluding the case of sequential applications, the penalty function δ improves the placement quality by reducing the solution performance degradation from 9% to 0%, for diamond applications, and from 15% to 7%, for replicated applications. Also in this setting, Local Search and

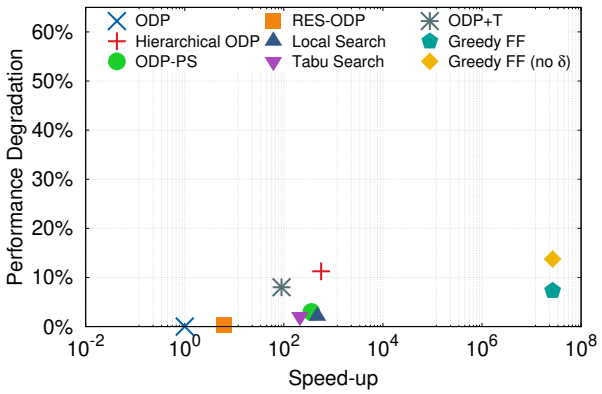


Figure 5.7: Policies performance to compute the application placement, when a multi-objective optimization function is considered. The optimized QoS metrics are equally weighted. Each point reports the average performance on different infrastructure settings and application topologies.

Tabu Search are beneficial for improving the solution quality; they obtain a performance degradation of 2% and 1%, respectively, for the replicated application (i.e., the most demanding one). Figure 5.6c also shows that most of the heuristics achieve a speed-up of 2 orders of magnitude with a very limited performance degradation. Hierarchical ODP has slightly higher performance degradation. A very good trade-off is obtained by Local Search with a speed-up of 588 times (i.e., in the worst case, it takes up to 3 minutes to compute the placement solution) and only 2% of performance degradation. ODP+T has overall good performance: with replicated applications, where the resolution time is very high, the early stop due to the timeout does not compromise the solution quality. This happens because, even though CPLEX has found the best solution within the first 300 ms, it needs to further explore the solution space so to certify the solution optimality. By taking up to 1.5 hours to compute the placement, RES-ODP is not well suited to be applied in online DSP systems.

Multi-objective Optimization. We now consider the case of multi-objective optimization function: the application requires a placement solution that minimizes response time and network usage and, at the same time, maximizes the availability. This corresponds to assign the weights $w_a = w_r = w_z = 0.33$ to the optimization function F. Figure 5.7 summarizes the experimental results. From Table 5.2, we can see that this is the most challenging scenario: the resolution time of ODP is higher than all the other configurations of objective functions. ODP takes at most 2.6 s to determine the placement of the diamond application, meaning that, in this case, ODP is very competitive (only Tabu Search has, on average, longer resolution time — its speed-up is 0.84). Conversely, for sequential applica-

tions, ODP shows its scalability issues, by requiring about 1.5 h to compute the placement. The case of replicated applications is even worse: in most of the cases (even with the 36 computing resources), ODP reaches the time-out at 24 h and does not certify the computed best solution as the optimal one. By observing Figure 5.7, we can classify the heuristics in four groups, according to their performance. The first group contains RES-ODP, which has a very limited speed-up: on average, it is one order of magnitude faster than ODP (i.e., in case of replicated application, RES-ODP is prohibitively slow — it requires 2.3 h to compute the placement solution). The second group contains ODP-PS, Tabu Search, and Local Search. They achieve a very good trade-off between resolution time and solution quality, having speed-up from 2 to 3 orders of magnitude with respect to ODP and performance degradation at most of 4%. In the most challenging setting (i.e., replicated application), ODP-PS and Local Search compute the placement in 8 and 1.3 minutes, respectively. The third group comprises ODP+T and Hierarchical ODP. Their speed-up is very similar to the one by the previous group of heuristics; nevertheless, their performance degradation is slightly higher: the average performance degradation by Hierarchical ODP and ODP+T is around 10%, which, in the worst case, grows up to 14% and 22%, respectively. The fourth group comprises the Greedy First-fit heuristic, which is characterized by a very high speed-up and a rather limited performance degradation (it is always below 15%). The penalty function δ improves the computed solution quality in, basically, all the experiments. Interestingly, for sequential applications, there is an inversion of tendency and δ reduces the application quality by 2%: this is an outlier behavior, which could be caused by the computing infrastructure topology[5] or by the complexity of the objective function.

Discussion. This set of experiment has extended the results of Section 5.6.2 by investigating the heuristics under different optimization objectives. We have seen that, together with the application topology and infrastructure size, also the optimization goal impacts on the heuristics performance. In general, we have empirically observed that some single-objective functions can be more easily optimized (e.g., availability) than others (e.g., network usage). In the worst case, i.e., minimize the network usage for a replicated application, the resolution time of ODP grows from 169 s to 24 h, whereas the resolution time of Hierarchical ODP, the fastest heuristics excluding Greedy First-fit, grows from 3.2 s to 13 min. Interestingly, when we are interested in maximizing the application availability, the well-know meta-heuristics, i.e., Local Search and Tabu Search, are rather slow. Conversely, Greedy First-fit is tremendously fast but computes low quality solutions (up to 29% of performance degradation).

[5]Our results present this anomaly only for the infrastructure with 49 and 64 computing nodes.

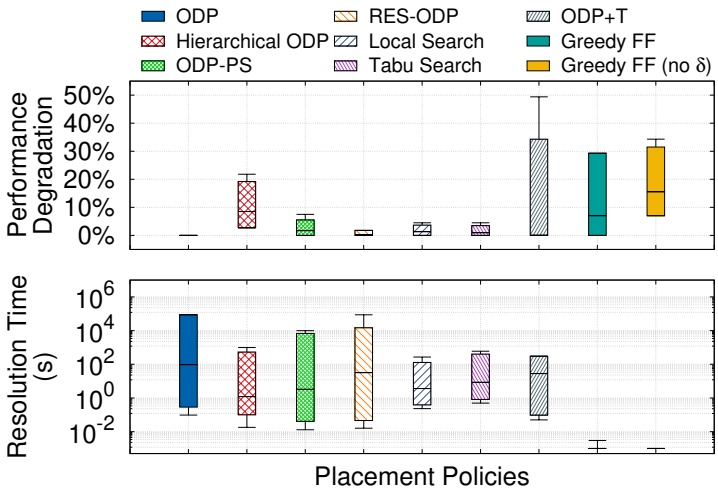


Figure 5.8: Performance distribution of ODP and the heuristics, considering their speed-up and performance degradation obtained throughout the whole experimental session.

Determining an application placement that optimizes the multi-objective function (i.e., when $w_a = w_r = w_z = 0.33$ in F) represents the hardest case. Here, although Hierarchical ODP performs well in terms of speed-up and performance degradation (at most, 14%), Local Search shows even better performances (resolution time in the order of minutes and at most 3% of performance degradation).

All these experiments clearly show that it is not easy to identify *the best* heuristic: in the different scenarios, the heuristics achieve a different trade-off between speed-up and performance degradation. Nevertheless, these experiments provide us enough data to discuss which are the heuristics that, on average, show a good behavior.

5.6.4 Heuristics Overall Performance

In this section, we analyze the heuristics performance obtained during the whole experimental evaluation, aiming to identify their average behavior. To this end, in Table 5.2, we report the average values of speed-up and performance degradation obtained for each combination of infrastructure size, application topology, and optimization objective. Figure 5.8 also reports the distribution of these two performance indexes (i.e., speed-up and performance degradation): each boxplot reports the minimum value, the 5th percentile, the median, the 95th percentile, and the maximum value. We summarize the experimental results as follows.

The diamond application presents a topology whose complexity can be

efficiently handled by ODP and the model-based heuristics. Interestingly, in this case, Tabu Search and Local Search perform poorly. With different application topologies, we need to use heuristics to efficiently deploy the application. This is especially true when complex objective functions should be optimized (e.g., multi-objective functions).

The Greedy First-fit heuristic is the fastest one, although it obtains solution with lowest quality (19% of average performance degradation). However, when equipped with our penalty function δ, the quality of this heuristic increases: it decreases the quality degradation of the computed placement solution from 19% to 11%, on average. These results empirically show the benefits of our penalty function δ, which is adopted also by the other heuristics.

In several configurations, ODP has a prohibitively high resolution time. We have shown that the idea of applying a stringent timeout to the CPLEX solver (as ODP+T does) does not always work fine. We have shown that, in some cases, ODP+T computes low quality solution, obtaining a performance degradation higher than Greedy First-fit (no δ). Figure 5.8 shows that the 95th percentile and the maximum value of performance degradation by ODP+T are the highest achieved in our experiments.

The heuristic ODP-PS, RES-ODP, Local Search, and Tabu Search have a very good trade-off between speed-up and performance degradation. The latter is always below 10%, whereas the resolution time of these heuristics is distributed on a wide range. We observe that RES-ODP shows an interesting behavior, indeed it always computes near-optimal solutions (with at most 2% of performance degradation), by reducing the resolution time of ODP by 11 times on average. Nevertheless, in the worst case, it takes several hours to compute the placement solution. As regards the other heuristics (i.e., ODP-PS, Local Search, and Tabu Search), there is not a net supremacy of one on the others. However, we consider Local Search to be the one having the best performance trade-off: it shows an average speed-up of 216 times and performance degradation of 1% with respect to ODP. In the worst case, Local Search took 269.8 s to compute the placement solution. Furthermore, this heuristic can easily extended to further limit the resolution time by using time-based stopping criteria, e.g., by setting a timeout on the exploration phase or by limiting the number of neighbor configurations to evaluate (see Section 5.5.2).

We also observe that Hierarchical ODP is very fast (average speed-up of 266 and 1.2 s as median value of resolution time) and shows an average performance degradation of 10%. As such, it determines, on average, higher quality solutions with respect to those by Greedy First-fit (independently whether the latter is equipped with or without the penalty function δ).

To conclude, we have shown that the combination of application topology, optimization function, and infrastructure size can deeply change the complexity of the placement problem to be solved. Moreover, we have shown that is not easy to identify a single heuristic that always achieves the best performance: e.g., see Local Search for diamond applications in Table 5.2. By analyzing the overall behavior in our experiments, we have concluded that Local Search achieves the best performance trade-off between efficiency (i.e., speed-up) and efficacy (i.e., performance degradation).

5.7 Summary

In this chapter, we have presented several heuristics for computing the placement of DSP applications over geo-distributed infrastructures. Relying on a penalty function, these heuristics explicitly take into account the heterogeneity of optimization goals and computing and network resources. Some of them are built around the ODP model (i.e., Hierarchical ODP, ODP-PS, and RES-ODP), whereas others implement well-known meta-heuristics for the problem at hand (i.e., Greedy First-fit, Local Search, and Tabu Search).

We have conducted a thoroughly experimental evaluation, where we investigated the heuristics' performance under different configurations of application topology, computing infrastructure size, and deployment optimization objective. To this end, we have used ODP as benchmark to determine the heuristics speed-up on resolution time and the quality of the computed placement solution (i.e., closeness to the optimal placement solution).

The experimental results have shown that the heuristics achieve different speed-up and performance degradation for the different combinations of application topology, infrastructure size, and optimization objective. There is not a *one-size-fits-all* heuristic, and we have discussed how the different approaches behave under different deployment configurations. By aggregating the results over every evaluated configuration, we have identified Local Search as the heuristic that achieves the best trade-off between high speed-up and reduced performance degradation.

Chapter 6

Optimal Operator Replication and Placement

To keep up with the high volume of daily produced data, the operators of a DSP application can be replicated and placed on multiple, possibly distributed, computing nodes, so to process data in parallel. We propose a general formulation of the optimal DSP replication and placement problem that, differently from most existing works, jointly optimizes these two deployment actions.

In the previous chapters, we have seen the importance of the placement problem for geo-distributed infrastructures and how our unified general formulation of the problem helps to evaluate existing heuristics as well as to develop new ones. In the Big Data era, DSP applications should be capable to seamlessly process huge amount of data, which require to scale their execution on multiple computing nodes, because a single machine cannot provide enough processing power. To this end, these applications usually exploit *data parallelism*, which consists in increasing or decreasing the number of parallel instances for the operators, so that each instance can process a subset of the incoming data flow in parallel (e.g., [65, 81]). Moreover, since data sources can be geographically distributed, the execution of DSP applications should also take advantage of the ever increasing presence of distributed Cloud and Fog computing resources. Indeed, the latter can improve the system scalability and reduce latency by moving the computation close to data sources and consumers.

In this chapter, we present and evaluate a general formulation of the optimal DSP replication and placement (ODRP) as an ILP problem. ODRP represents a unified general formulation of the operator replication and placement problem, which takes into account the heterogeneity of application requirements as well as computing and network resources. Differ-

ently from most works in literature [53, 132, 145, 196], ODRP can jointly determine the application placement and the replication of its operators, while optimizing the QoS attributes of the application. At the same time, ODRP provides a benchmark against which other centralized and decentralized placement and replication algorithms can be compared. The main contributions of this chapter are as follows.

- We model the ODRP problem as an ILP problem, which can be used to optimize different QoS metrics, such as response time, cost, and availability (Sections 6.2 and 6.3).
- We present a prototype scheduler for Distributed Storm, where the DSP application operators are replicated and placed according to the ODRP solution (Section 6.4).
- We thoroughly validate and evaluate the proposed ODRP model, relying on a set of numerical experiments, aimed to show the benefit of a joint optimization of replication and placement (Section 6.5).
- Relying on the Storm-based prototype and the benchmark application that addresses the DEBS 2015 Grand Challenge [99], we propose a second set of experiments. The latter aims to show strengths and drawbacks of the proposed solution in a real setting. Moreover, the prototype-based experiments aim to confirm the outcomes of the numerical experiments and show how ODRP can contextually optimize several QoS metrics (Section 6.6).

6.1 Related Work

To deploy a DSP application, a DSP system needs to determine the replication degree of the application operators and their placement on the computing infrastructure. The literature analysis presented in Chapter 2 has shown that, although widely investigated, these problems are often considered separately and only few works study their joint optimization.

Specifically, most works in literature first compute the operator placement without determining the replication degree for each operator. Then, in response to performance deterioration, they use a different policy to accordingly change the number of operator replicas (e.g., [78]). This two-stage approach requires to reschedule the DSP application multiple times, thus possibly incurring in higher overhead.

The placement problem has been investigated in literature under different modeling assumptions and optimization goals, e.g., [53, 117, 196]. Our problem formulation can be adjusted to take into account these different utility functions. As regards the replication problem, many research efforts focus on scaling the amount of operator replicas in response to changes ob-

served in some monitored performance metric. Some works, e.g., [30, 80], exploit threshold-based policies based on the utilization of either the system nodes or the operator instances. Other works, e.g., [65, 132, 145], use more complex policies to determine the scaling decisions. Lohrmann et al. [132] propose a strategy that enforces latency constraints by relying on a predictive latency model based on queuing theory. Mencagli [145] present a game-theoretic approach where the control logic is distributed on each operator.

Differently from the above mentioned works, in this chapter, we propose a *single-stage* approach to determine both the placement and the parallelism degree of the operators in a DSP application. Specifically, we build on the ODP model presented in Chapter 4, which provides a general formulation of the optimal DSP placement, taking into account the heterogeneity of computing and networking resources. ODRP extends the formulation of ODP so as to determine the optimal number of replicas for each operator contextually to their placement on the underlying infrastructure.

The most closely related work to ours has been proposed by Madsen et al. [138]. The authors propose an ILP model that computes replication of co-location groups, together with load balancing among them. These co-location groups are pre-determined relying on a heuristic that aims to minimize inter-node traffic. Nevertheless, differently from our solution, Madsen et al. do not consider network delays among computing nodes.

Interestingly, Heinze et al. [78, 81] propose a model to estimate the latency spike created by a set of operator movements. This model is then used to define an operator placement algorithm based on a bin packing heuristic, which minimizes the latency violations and focuses only on the placement of the newly added operators. We present an optimal problem formulation that targets the initial placement decision and can be used to benchmark existing heuristics.

An important issue related to operator replication regards the management of stateful operators (see Chapter 2). The approach we propose in this chapter jointly places and replicates operators, thus saving the overhead and latency penalty incurred by stateful operator migrations in case of disjointed replication and placement decisions. Anyway, the latter still occur when the deployment is adapted at runtime and we will address this issue in the next chapters.

6.2 System Model and Problem Statement

In this section, we recall the resource model from Chapter 4, extend the DSP application model for representing replication, and define the operator replication and placement problem. For the sake of clarity, in Table 6.1 we

Table 6.1: Main notation adopted in this chapter

Symbol	Description
G_{dsp}	Graph representing the DSP application
V_{dsp}	Set of vertices (operators) of G_{dsp}
E_{dsp}	Set of edges (streams) of G_{dsp}
C_i	Cost of deploying operator $i \in V_{dsp}$
R_i	Latency of $i \in V_{dsp}$ on a reference processor
Res_i	Resources required to execute $i \in V_{dsp}$
k_i	Maximum replication degree of $i \in V_{dsp}$
$\lambda_{(i,j)}$	Average tuple rate exchanged on $(i,j) \in E_{dsp}$
$b_{(i,j)}$	Avg. number of byte per tuple on $(i,j) \in E_{dsp}$
G_{res}	Graph representing computing and network resources
V_{res}	Set of vertices (computing nodes) of G_{res}
E_{res}	Set of edges (logical links) of G_{res}
A_u	Availability of node $u \in V_{res}$
Res_u	Amount of resources available at $u \in V_{res}$
S_u	Processing speed-up of $u \in V_{res}$
$A_{(u,v)}$	Availability of $(u,v) \in E_{res}$
$C_{(u,v)}$	Transmission cost per data on $(u,v) \in E_{res}$
$d_{(u,v)}$	Network delay on $(u,v) \in E_{res}$
$V^i_{res} \subseteq V_{res}$	Subset of nodes where $i \in V_{dsp}$ can be placed
$\mathcal{X} \sqsubset X$	Multiset of elements in X
$x_{i,\mathcal{U}}$	Placement of $i \in V_{dsp}$ on nodes in $\mathcal{U} \sqsubset V^i_{res}$
$y_{(i,j),(\mathcal{U},\mathcal{V})}$	Placement of $(i,j) \in E_{dsp}$ on the network paths from nodes in $\mathcal{U} \sqsubset V^i_{res}$ to nodes in $\mathcal{V} \sqsubset V^j_{res}$
z_u	Activation variable for $u \in V_{res}$
$z_{(u,v)}$	Activation variable for $(u,v) \in E_{res}$

summarize the notation used throughout the chapter.

6.2.1 Resource Model

Computing and network resources can be represented as a labeled fully connected directed graph $G_{res} = (V_{res}, E_{res})$, where the set of nodes V_{res} represents the distributed computing resources, and the set of links E_{res} represents the *logical connectivity* between nodes. At this level, links represent the logical connections across the network, which correspond to network paths between nodes (as determined by the network operator routing strategies). Each node $u \in V_{res}$ is characterized by: Res_u, the amount of available resources; S_u, the processing speed-up on a reference processor; and A_u, its availability, i.e., the probability that u is up and running. Each

link $(u, v) \in E_{res}$, with $u, v \in V_{res}$, is characterized by: $d_{(u,v)}$, the network delay between node u and v; $A_{(u,v)}$, the link availability, i.e., the probability that the link between u and v is active; and $C_{(u,v)}$, the cost per unit of data transmitted along the network path between u and v. This model considers also edges of type (u, u) (i.e., loops); they capture network connectivity between operators placed on the same node u, and are considered as perfect links, i.e., always active with no network delay. We assume that the considered QoS attributes can be obtained by means of either active/passive measurements or through some network support (e.g., SDN).

6.2.2 DSP Model

A DSP application can be represented at different levels of abstraction. We distinguish between a user-defined *abstract model* and an *execution model*, which is used to run the application.

A DSP abstract model can be represented as a labeled DAG $G_{dsp} = (V_{dsp}, E_{dsp})$, where the nodes in V_{dsp} represent the application operators, data sources and sinks, and the links in E_{dsp} represent the streams between nodes. Due to the difficulties in formalizing the non-functional attributes of an abstract operator, we characterize it with the non-functional attributes of a reference implementation on a reference architecture: Res_i, the amount of resources required for running the operator; R_i, the operator latency (which accounts for the waiting time on the input queues as well as the execution time of a unit of data); C_i, the cost of deploying an instance of the operator. We characterize the stream exchanged from operator i to j, $(i, j) \in E_{dsp}$, with its average tuple rate $\lambda_{(i,j)}$ and average number of bytes per tuple $b_{(i,j)}$. To model load-dependent latency, we assume that the latency is function of λ_i, the operator input tuple rate, $R_i = R_i(\lambda_i)$, where $\lambda_i = \sum_{j \in V_{dsp}} \lambda_{(j,i)}$; without loss of generality, we also assume that R_i is an increasing function in λ_i. We assume that Res_i is a scalar value, but our placement model can be easily extended to consider Res_i as a vector of required resources.

The DSP *execution model* is obtained from the abstract model by replacing each operator with its replicas. Differently from most of the existing solutions, ODRP computes the execution model by optimizing, in a single stage, the number of operator instances and their placement.

6.2.3 Operator Replication and Placement

The DSP replication and placement problem consists in determining, for each operator $i \in V_{dsp}$, the number of replicas and where to deploy them on the computing nodes in V_{res}. Figure 6.1 represents a simple instance of the problem. Since a DSP operator cannot be usually placed on every

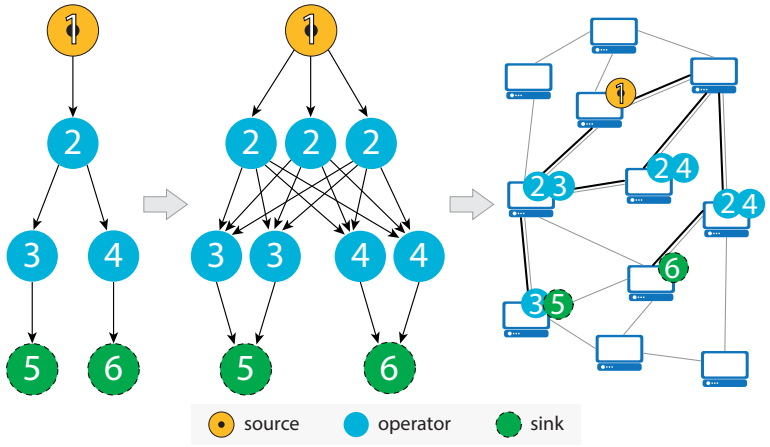


Figure 6.1: Replication of the application operators and their placement on the computing resources

node in V_{res}, we consider for each operator $i \in V_{dsp}$ a subset of candidate resources $V_{res}^i \subseteq V_{res}$ where it can be deployed. For example, if sources and sinks ($I \subset V_{dsp}$) are external applications, their placement is fixed, that is $\forall i \in I, |V_{res}^i| = 1$. The operator placement can be represented by a function map which maps an operator $i \in V_{dsp}$ to a multiset of computing nodes in V_{res}^i. We recur to multisets because a deployment can place multiple replicas of the same operator on the same computing node. For instance, $map(i) = \{u, u, v\}$, $i \in V_{dsp}$, $u, v \in V_{res}^i$, indicates that operator i deployment consists of 3 replicas, two of which on node u and one on node v. A multiset $\mathcal{X}$ over a set X, which we denote as $\mathcal{X} \sqsubset X$, is defined as a mapping $\mathcal{X} : X \to \mathbb{N}$ where, for $x \in X$, $\mathcal{X}(x)$ denotes the multiplicity of x in $\mathcal{X}$. $x \in \mathcal{X}$ if and only if $\mathcal{X}(x) \geq 1$. The cardinality of a multiset $\mathcal{X}$, denoted $|\mathcal{X}|$, is defined by the number of elements in $\mathcal{X}$, that is $|\mathcal{X}| = \sum_{x \in \mathcal{X}} \mathcal{X}(x)$. Hereafter, without lack of generality, we will assume that in a deployment each operator $i \in V_{dsp}$ can be replicated at most k_i times. Therefore, we also find convenient to define the power multiset $\mathcal{P}(X)$ of a set X as the set of all multisets with elements taken from X and the subset $\mathcal{P}(X; k) \subset \mathcal{P}(X)$ of the multiset over X with cardinality no greater of k, that is $\mathcal{P}(X; k) = \{\mathcal{X} \in \mathcal{P}(X) | \sum_{x \in \mathcal{X}} \mathcal{X}(x) \leq k\}$.

6.3 Optimal Replication and Placement Model

In this section we present our model for the ODRP problem. As the optimal solution depends on non-functional attributes, we first derive the expression for the different QoS metrics of interest and then present the ODRP formulation.

6.3.1 ODRP Variables

We model the ODRP problem with binary variables $x_{i,\mathcal{U}}$, $i \in V_{dsp}$ and $\mathcal{U} \sqsubset V_{res}^i$: $x_{i,\mathcal{U}} = 1$ if and only if the operator $map(i) = \mathcal{U}$, that is, i is replicated in $|\mathcal{U}|$ instances with exactly $\mathcal{U}(u)$ copies deployed in u, with $u \in \mathcal{U}$. We also find convenient to consider binary variables associated to links, namely $y_{(i,j),(\mathcal{U},\mathcal{V})}$, $(i,j) \in E_{dsp}$, $\mathcal{U} \sqsubset V_{res}^i$, $\mathcal{V} \sqsubset V_{res}^j$, which denotes whether the data stream flowing from operator i to operator j traverses the network paths from nodes in $\mathcal{U}$ to nodes in $\mathcal{V}$. By definition, we have $y_{(i,j),(\mathcal{U},\mathcal{V})} = x_{i,\mathcal{U}} \wedge x_{j,\mathcal{V}}$.

Finally, we also consider the variables z_u, $u \in V_{res}$ which denote whether at least one operator is deployed on node u and the variables $z_{(u,v)}$, $(u,v) \in E_{res}$, which denote whether a stream (or a portion of it) traverses the network path (u,v). By definition, we have $z_u = \vee_{i \in V_{dsp}, \mathcal{U} \in \mathcal{P}(V_{res}^i; k_i)} x_{i,\mathcal{U}}$ and $z_{(u,v)} = \vee_{(i,j) \in E_{dsp}, \mathcal{U} \in \mathcal{P}(V_{res}^i; k_i), \mathcal{V} \in \mathcal{P}(V_{res}^j; k_j)} y_{(i,j),(\mathcal{U},\mathcal{V})}$. For short, in the following we denote by $\boldsymbol{x}$ and $\boldsymbol{y}$ the placement vectors for nodes and edges, respectively, where $\boldsymbol{x} = \langle x_{i,\mathcal{U}} \rangle$, $\forall i \in V_{dsp}$, $\forall \mathcal{U} \in \mathcal{P}(V_{res}^i; k_i)$, and $\boldsymbol{y} = \langle y_{(i,j),(\mathcal{U},\mathcal{V})} \rangle$, $\forall x_{i,\mathcal{U}}, x_{j,\mathcal{V}} \in \boldsymbol{x}$. Similarly, we denote by $\boldsymbol{z_V}$ and $\boldsymbol{z_E}$ the vectors $\boldsymbol{z_V} = \langle z_u \rangle$, $\forall u \in V_{res}$, and $\boldsymbol{z_E} = \langle z_{(u,v)} \rangle$, $\forall (u,v) \in E_{res}$.

6.3.2 Qos Metrics

Operator QoS Metrics

Let us first consider an operator in isolation. For each $i \in V_{dsp}$, the QoS of the operator deployment depends on the deployment $\mathcal{U}$. Let $R_{i,\mathcal{U}}$, $C_{i,\mathcal{U}}$, and $A_{i,\mathcal{U}}$ denote the maximum latency[1], the cost, and the availability of the deployment $\mathcal{U}$, respectively. We readily have:

$$R_{i,\mathcal{U}} = \max_{u \in \mathcal{U}} \frac{R_i(\frac{\lambda_i}{|\mathcal{U}|})}{S_u} \tag{6.1}$$

$$C_{i,\mathcal{U}} = \sum_{u \in \mathcal{U}} \mathcal{U}(u) C_i Res_i \tag{6.2}$$

$$A_{i,\mathcal{U}} = \prod_{u \in \mathcal{U}} A_u \tag{6.3}$$

under the assumption that the traffic is equally split among the different operator replicas.

[1]This definition of the operator response time might appear counter intuitive. Nevertheless, it is consistent with our definition of the DSP application response time (see (6.8)–(6.11) below) which we define as the expected delay along the critical path of the DSP application. It is indeed easy to verify that, with this definition of operator response time $R_{i,\mathcal{U}}$, since different replicas can experience different average response time, (6.8) corresponds to the average latency along the DSP critical path.

Stream QoS Attributes

We now turn our attention to the QoS attributes related to a stream. For a stream (i,j), the QoS depends on the upstream and downstream operators' deployments $\mathcal{U}$ and $\mathcal{V}$. Let $d_{(i,j),(\mathcal{U},\mathcal{V})}$, $C_{(i,j),(\mathcal{U},\mathcal{V})}$, and $A_{(i,j),(\mathcal{U},\mathcal{V})}$ denote the maximum latency[2], the cost, and the availability of the deployments $\mathcal{U}$ and $\mathcal{V}$, respectively. We readily have:

$$d_{(i,j),(\mathcal{U},\mathcal{V})} = d_{(\mathcal{U},\mathcal{V})} = \max_{u\in\mathcal{U},v\in\mathcal{V}} d_{(u,v)} \tag{6.4}$$

$$C_{(i,j),(\mathcal{U},\mathcal{V})} = \sum_{u\in\mathcal{U},v\in\mathcal{V}} \lambda_{(i,j),(\mathcal{U},\mathcal{V})} C_{u,v} \tag{6.5}$$

$$A_{(i,j),(\mathcal{U},\mathcal{V})} = \prod_{u\in\mathcal{U},v\in\mathcal{V}} A_{(u,v)} \tag{6.6}$$

where

$$\lambda_{(i,j),(\mathcal{U},\mathcal{V})} = \frac{\lambda_{(i,j)}}{|\mathcal{U}||\mathcal{V}|} \quad u\in\mathcal{U}, v\in\mathcal{V} \tag{6.7}$$

is the amount of stream (i,j) traffic exchanged between two operator replicas under the deployments $\mathcal{U}$ and $\mathcal{V}$.

DSP Application QoS Metrics

We consider both user-oriented and system-oriented QoS metrics, such as application response time, cost, and availability for the former, and network related metrics for the latter.

Response Time: We consider as response time R the critical path average delay (we refer the reader to Section 4.3 for the definition of critical path delay). Formally, we have that:

$$R = \max_{\pi\in\Pi_{dsp}} R_\pi \tag{6.8}$$

being R_π the delay along path π and Π_{dsp} the set of all source-sink paths in G_{dsp}. Given a placement vector $\boldsymbol{x}$ (and resulting $\boldsymbol{y}$) and a path $\pi = (i_1, i_2, \ldots, i_{n_\pi})$, we have $R = R(\boldsymbol{x},\boldsymbol{y}) = \max_{\pi\in\Pi_{dsp}} R_\pi(\boldsymbol{x},\boldsymbol{y})$ with $R_\pi(\boldsymbol{x},\boldsymbol{y})$ defined as:

$$R_\pi(\boldsymbol{x},\boldsymbol{y}) = \sum_{p=1}^{n_\pi} R_{i_p}(\boldsymbol{x}) + \sum_{p=1}^{n_\pi-1} D_{(i_p,i_{p+1})}(\boldsymbol{y}) \tag{6.9}$$

[2]Similarly to the response time definition (6.1), this definition might appear counter intuitive. Nevertheless, it is easy to verify that it is consistent with the DSP application response time definitions, (6.8)-(6.11).

where

$$R_i(\boldsymbol{x}) = \sum_{\mathcal{U} \in \mathcal{P}(V_{res}^i; k_i)} R_{i,\mathcal{U}} x_{i,\mathcal{U}} \tag{6.10}$$

$$D_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{\mathcal{U} \in \mathcal{P}(V_{res}^i; k_i) \\ \mathcal{V} \in \mathcal{P}(V_{res}^j; k_j)}} d_{(\mathcal{U},\mathcal{V})} y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{6.11}$$

denote respectively the execution time of operator i when deployed over the multiset $\mathcal{U}$ and the worst case network delay for transferring data from i to j when the two operators are mapped over $\mathcal{U}$ and $\mathcal{V}$, respectively.

Cost: We define the cost C of the DSP application as the monetary cost of all the computing resources and paths involved in the processing and transmission of the application data streams. We have:

$$C(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i \in V_{dsp}} C_i(\boldsymbol{x}) + \sum_{(i,j) \in E_{dsp}} C_{(i,j)}(\boldsymbol{y}) \tag{6.12}$$

where

$$C_i(\boldsymbol{x}) = \sum_{\mathcal{U} \in \mathcal{P}(V_{res}^i; k_i)} C_{i,\mathcal{U}} x_{i,\mathcal{U}} \tag{6.13}$$

$$C_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{\mathcal{U} \in \mathcal{P}(V_{res}^i; k_i) \\ \mathcal{V} \in \mathcal{P}(V_{res}^j; k_j)}} C_{(i,j),(\mathcal{U},\mathcal{V})} y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{6.14}$$

Availability: We define the application availability A as the availability of all the nodes and paths involved in the processing and transmission of the application data streams. For the sake of simplicity, we assume the availability of the different components to be independent; we have:

$$A(\boldsymbol{z_V}, \boldsymbol{z_E}) = \prod_{u \in V_{res} : z_u = 1} A_u z_u \cdot \prod_{(u,v) \in E_{res} : z_{(u,v)} = 1} A_{(u,v)} z_{(u,v)} \tag{6.15}$$

To obtain a linear expression, we consider the logarithm of the availability, obtaining:

$$\log A(\boldsymbol{z_V}, \boldsymbol{z_E}) = \sum_{u \in V_{res}} a_u z_u + \sum_{(u,v) \in E_{res}} a_{(u,v)} z_{(u,v)} \tag{6.16}$$

where $a_u = \log A_u$ and $a_{(u,v)} = \log A_{(u,v)}$. It is worth observing that in (6.16) we can take the summation over all $u \in V_{res}$ and $(u, v) \in E_{res}$ since the terms not appearing in (6.15) are those corresponding to $z_u = 0$ or $z_{(u,v)} = 0$, which do not affect the summation in (6.16).

Network Related QoS Metrics: Similarly to Chapter 4, we model the following network related QoS metrics: *inter-node traffic* T, *network usage* N,

and equivalent *elastic energy* EE. Let $Z(\boldsymbol{y})$, $Z = T|N|EE$, denote QoS attribute of the DSP application under the placement policy $\boldsymbol{y}$; we have:

$$Z(\boldsymbol{y}) = \sum_{(i,j)\in E_{dsp}} Z_{(i,j)}(\boldsymbol{y}) \tag{6.17}$$

where $Z_{(i,j)}(\boldsymbol{y})$ is defined as follows. Given a stream $(i,j) \in E_{dsp}$ and the placement vector $\boldsymbol{y}$, we define the *inter-node traffic* T, the *network usage* N, and the equivalent *elastic energy* EE as follows:

$$T_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{u\in\mathcal{U},v\in\mathcal{V},u\neq v\\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)\\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)}} b_{(i,j)}\lambda_{(i,j),(\mathcal{U},\mathcal{V})}y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{6.18}$$

$$N_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{u\in\mathcal{U},v\in\mathcal{V},u\neq v\\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)\\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)}} b_{(i,j)}\lambda_{(i,j),(\mathcal{U},\mathcal{V})}d_{(u,v)}y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{6.19}$$

$$EE_{(i,j)}(\boldsymbol{y}) = \sum_{\substack{u\in\mathcal{U},v\in\mathcal{V},u\neq v\\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)\\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)}} b_{(i,j)}\lambda_{(i,j),(\mathcal{U},\mathcal{V})}d^2_{(u,v)}y_{(i,j),(\mathcal{U},\mathcal{V})} \tag{6.20}$$

where $d_{(u,v)}$ is the network delay among nodes $u, v \in V_{res}$, with $u \neq v$.

6.3.3 ODRP Formulation

Depending on the usage scenario, a DSP replication and placement strategy could be aimed at optimizing different, possibly conflicting, QoS attributes. To this end, we use again the SAW technique to define the utility function $F(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z_V}, \boldsymbol{z_E})$ as a weighted sum of the normalized QoS attributes of the application, as follows:

$$\begin{aligned} F(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z_V}, \boldsymbol{z_E}) = {} & w_r \frac{R_{\max} - R(\boldsymbol{x}, \boldsymbol{y})}{R_{\max} - R_{\min}} + w_a \frac{\log A(\boldsymbol{z_V}, \boldsymbol{z_E}) - \log A_{\min}}{\log A_{\max} - \log A_{\min}} \\ & + w_c \frac{C_{\max} - C(\boldsymbol{x}, \boldsymbol{y})}{C_{\max} - C_{\min}} + w_z \frac{Z_{\max} - Z(\boldsymbol{x}, \boldsymbol{y})}{Z_{\max} - Z_{\min}} \end{aligned} \tag{6.21}$$

where $w_r,\ w_a,\ w_c,\ w_z \geq 0$, $w_r + w_a + w_c + w_z = 1$, are weights associated to the different QoS attributes. $R_{\max}$ ($R_{\min}$), $A_{\max}$ ($A_{\min}$), $C_{\max}$ ($C_{\min}$), and $Z_{\max}$ ($Z_{\min}$) denote, respectively, the maximum (minimum) value for the overall expected response time, availability, cost and network related metric. Observe that after normalization, each metric ranges in the interval $[0, 1]$, where the value 1 corresponds to the best possible case and 0 to the worst case.

We formulate the ODRP problem as an ILP model as follows:

$$\max_{\boldsymbol{x},\boldsymbol{y},r} F'(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z_V},\boldsymbol{z_E},r)$$

subject to:

$$r \geq \sum_{\substack{p=1,\dots,n_\pi \\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)}} R_{i_p,\mathcal{U}} x_{i,\mathcal{U}} + \sum_{\substack{p=1,\dots,n_\pi-1 \\ q=p+1 \\ \mathcal{U}\in\mathcal{P}(V^{i_p}_{res};k_{i_p}) \\ \mathcal{V}\in\mathcal{P}(V^{i_q}_{res};k_{i_q})}} d_{(\mathcal{U},\mathcal{V})} y_{(i_p,i_q),(\mathcal{U},\mathcal{V})} \qquad \forall \pi \in \Pi_{dsp} \qquad (6.22)$$

$$Res_u \geq \sum_{\substack{i\in V_{dsp} \\ \mathcal{U}\in\mathcal{P}(V^i_{res})}} \mathcal{U}(u) Res_i x_{i,\mathcal{U}} \qquad \forall u \in V_{res} \qquad (6.23)$$

$$z_u \geq \frac{\sum_{\substack{i\in V_{dsp} \\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)}} x_{i,\mathcal{U}}}{M} \qquad u \in V_{res} \qquad (6.24)$$

$$z_{(u,v)} \geq \frac{\sum_{\substack{(i,j)\in E_{dsp} \\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i) \\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)}} y_{(i,j),(\mathcal{U},\mathcal{V})}}{N} \qquad (u,v) \in E_{res} \qquad (6.25)$$

$$1 = \sum_{\mathcal{U}\in\mathcal{P}(V^i_{res};k_i)} x_{i,\mathcal{U}} \qquad \forall i \in V_{dsp} \qquad (6.26)$$

$$x_{i,\mathcal{U}} = \sum_{\mathcal{V}\in\mathcal{P}(V^j_{res};k_j)} y_{(i,j),(\mathcal{U},\mathcal{V})} \qquad \substack{\forall (i,j)\in E_{dsp}, \\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)} \qquad (6.27)$$

$$x_{j,\mathcal{V}} = \sum_{\mathcal{U}\in\mathcal{P}(V^i_{res};k_u)} y_{(i,j),(\mathcal{U},\mathcal{V})} \qquad \substack{\forall (i,j)\in E_{dsp}, \\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)} \qquad (6.28)$$

$$x_{i,\mathcal{U}} \in \{0,1\} \qquad \substack{\forall i\in V_{dsp}, \\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i)} \qquad (6.29)$$

$$y_{(i,j),(\mathcal{U},\mathcal{V})} \in \{0,1\} \qquad \substack{\forall (i,j)\in E_{dsp}, \\ \mathcal{U}\in\mathcal{P}(V^i_{res};k_i) \\ \mathcal{V}\in\mathcal{P}(V^j_{res};k_j)} \qquad (6.30)$$

$$z_u \in \{0,1\} \qquad \forall u \in V_{res} \qquad (6.31)$$

$$z_{(u,v)} \in \{0,1\} \qquad \forall (u,v) \in E_{res} \qquad (6.32)$$

In the problem formulation we use the objective function $F'(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z_V},\boldsymbol{z_E},r)$ that is obtained from $F(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z_V},\boldsymbol{z_E})$ by replacing $R(\boldsymbol{x},\boldsymbol{y})$ with the auxiliary variable r, which represents the application response time in the optimization problem, in order to obtain a linear objective function. Observe that, indeed, while F is nonlinear in $\boldsymbol{x}$, $\boldsymbol{y}$ since $R(\boldsymbol{x},\boldsymbol{y}) = \max_{\pi\in\Pi_{dsp}} R_\pi(\boldsymbol{x},\boldsymbol{y})$ is a nonlinear term, F' is linear in r as well as in $\boldsymbol{x}$ and $\boldsymbol{y}$. Equation (6.22)

follows from (6.8)–(6.11). Since r must be larger or equal than the response time of any path and, at the optimum, r is minimized, we have that $r = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x}, \boldsymbol{y}) = R(\boldsymbol{x}, \boldsymbol{y})$. The constraint (6.23) limits the placement of operators on a node $u \in V_{res}$ according to its available resources. Constraints (6.24) and (6.25) are the activation constraints for the variable z_u and $z_{(u,v)}$, respectively, with M and N large constants. Equation (6.26) guarantees that each operator $i \in V_{dsp}$ is placed on one and only one node $u \in V^i_{res}$. Finally, constraints (6.27)–(6.28) model the logical `AND` between the placement variables, that is, $y_{(i,j),(u,v)} = x_{i,u} \wedge x_{j,v}$.

Theorem 6.1. *The ODRP problem is an NP-hard problem.*

Proof. It is sufficient to observe that the Optimal DSP Replication and Placement problem is a generalization of the Optimal DSP Placement problem we presented in Chapter 4, which has been shown to be NP-hard. □

6.4 Storm Integration: S-ODRP

To enable the usage of ODRP in a real DSP framework, we develop a prototype scheduler for Apache Storm, named **S-ODRP**. To this end, we have to address two issues: (1) to adapt the DSP and resource model to consider the specific execution entities of Storm, and (2) to instantiate the ODRP model with the proper QoS information about computing and networking resources.

As regards the first issue, we have to model the fact that Storm runs multiple executors to replicate an operator, and that a Storm scheduler deploys these executors on the available worker slots, considering that at most $EPS_{\max}$ executors can be co-located on the same slot. Hence, S-ODRP defines $G_{dsp} = (V_{dsp}, E_{dsp})$, with V_{dsp} as the set of operators and E_{dsp} as the set of streams exchanged between them. Since in Storm an operator is considered as a black box element, we conveniently assume that its attributes are unitary, i.e., $C_i = 1$ and $Res_i = 1$, $\forall i \in V_{dsp}$. By solving the replication and placement model, S-ODRP determines the number of executors for each operator $i \in V_{dsp}$, leveraging on the cardinality of $\mathcal{U}$ when $x_{i,\mathcal{U}} = 1$, with $\mathcal{U} \sqsubset V^i_{res}$. The resource model $G_{res} = (V_{res}, E_{res})$ must consider that a worker node $u \in V_{res}$ offers some worker slots $WS(u)$, and each worker slot can host at most $EPS_{\max}$ executors. For simplicity, S-ODRP considers the amount of available resources C_u on a worker node $u \in V_{res}$ to be equal to the maximum number of executors it can host, i.e., $C_u = WS(u) \times EPS_{\max}$. To enable the parallel execution of executors, C_u must be equal (or proportional) to the number of CPU cores available on u.

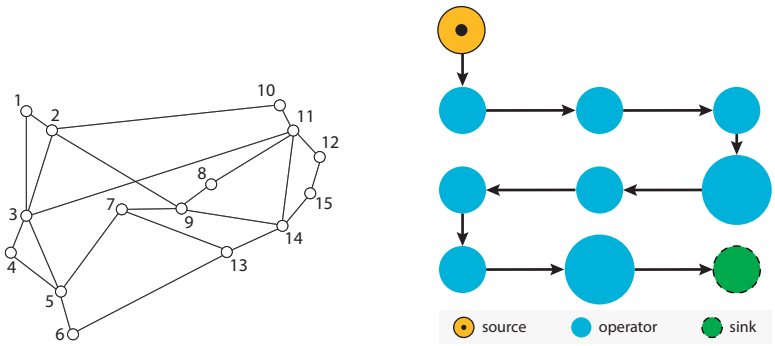
As regards the second issue, Storm allows us to easily develop new centralized schedulers with the pluggable scheduler APIs. However, Storm is

unaware of the QoS attributes of its networking and computing resources, except for the number of available worker slots. Since we need to know these QoS attributes in order to apply the ODRP model, we rely on Distributed Storm, that enables the QoS awareness of the scheduling system by providing intra-node (i.e., availability) and inter-node (i.e., network delay and exchanged data rate) information. S-ODRP retrieves, from the monitoring components of the extended Storm, the information needed to parametrize the nodes and edges in G_{dsp} and G_{res}. Specifically, it considers: the average data rate exchanged between communicating executors (i.e., $\lambda_{(i,j)}, \forall (i,j) \in E_{dsp}$), the node availability ($A_u, \forall u \in V_{res}$), and the network latencies ($d_{(u,v)}, \forall u, v \in V_{res}$). Once built the ODRP model, S-ODRP relies on CPLEX©, the state-of-the-art solver for ILP problems, for its resolution.

From an operational prospective, Nimbus uses S-ODRP to compute the optimal operator replication and placement when a new application is submitted to Storm and when a failure of the worker process compromises the application execution. In the latter case, S-ODRP invalidates the existing assignment and computes the new optimal placement. Algorithm 6 summarizes the runtime execution of S-ODRP, which has to face two main issues: to collect the exchanged data rate between the operators, and to replicate the operators as needed. When information on the exchanged data rate is unknown (line 5), e.g., the first time the application is scheduled, S-ODRP defines an early assignment and monitors the application execution to harvest the needed information (lines 6–8). As soon as this information is available, S-ODRP reassigns the application by solving the updated ODRP model with the network-related QoS attributes (line 10). To enact the replication decision computed by ODRP (lines 7 and 11), S-ODRP leverages on the Storm API `rebalance`, which restarts the application with the correct number of executors, before assigning them to the worker nodes as specified by the computed placement solution.

Algorithm 6 Application placement with S-ODRP

```
1:  function SCHEDULE(G_dsp, G_res)
2:      Input: G_dsp, graph representing a DSP application
3:      Input: G_res, graph representing computing and network resources
4:      RP = [ ]                                  ▷ replication and placement
5:      if not streamsDatarateAvailable(G_dsp) then
6:          RP ← ODRP(G_dsp, G_res)
7:          enact(RP)
8:          G_dsp ← collectStreamsDatarate(G_dsp)
9:      end if
10:     RP ← ODRP(G_dsp, G_res)
11:     enact(RP)
12: end function
```



(a) Subset of ANSNET nodes (b) Reference application

Figure 6.2: Reference infrastructure and DSP application

6.5 Numerical Experiments

In this section, we evaluate the ODRP model through a set of numerical experiments that aim at demonstrating the flexibility of the formulation and the benefits that derive from the joint optimization of operators placement and replication. In Section 6.5.2, we show the impact of replication when the DSP application is subject to an increasing load. Then, in Section 6.5.3, we analyze how the ODRP formulation allows us to consider the optimization of several QoS metrics, such as response time, cost, availability, and inter-node traffic.

6.5.1 Experimental Setup

The ODRP model allows us to define the operators placement and replication by optimizing different QoS attributes, whose importance depends on the utilization scenario.

We solve the ILP problem using CPLEX© (version 12.6.2) on an Amazon EC2 virtual machine (c4.xlarge with 4 vCPU and 7.5 GB RAM). In the experiments, G_{res} models a portion of ANSNET, a geographically distributed network where 15 computing nodes are interconnected with non-negligible network delays. Within this network, we assume that a logical link $(u, v) \in E_{res}$ between any two computing resources $u, v \in V_{res}$ always exists; each logic link results by the underlying physical network paths, which are represented in Figure 6.2a, and a shortest-path routing strategy.

As reference application, we consider the topology represented in Figure 6.2b, which is made of a sequence of 10 operators, where the first and last one are respectively the source and sink. If not otherwise specified, the application includes two CPU-intensive operators, represented with a big-

Table 6.2: Parameters of the experimental setup

Processing and network resources

Parameter	Value	Parameter	Value
$\|V_{res}\|$	15	$A_{(u,v)}$	100 %
A_u	Uniform in $[97, 99.99999]$ %	$C_{(u,v)}$	0.02
Res_u	5	avg $d_{(u,v)}$	32 ms
S_u	1.0		

Application

Parameter	Value	Parameter	Value
$\|V_{dsp}\|$	10	Res_i	1
C_i	1	μ_i	0.02 ms^{-1}
$R_i(\lambda_i/\|\mathcal{U}\|)$	$\frac{1}{\mu_i - \lambda_i/\|\mathcal{U}\|}$ ms	$b_{(i,j)}$	1500 B/tuple
λ_i	14 tuples/s		

Normalization factors for the ODRP utility function

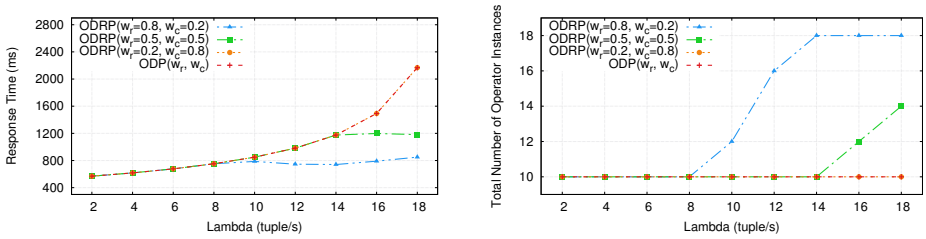
Parameter	Value	Parameter	Value
$R_{\min}$	1230 ms	$R_{\max}$	2810 ms
$A_{\min}$	85 %	$A_{\max}$	98.8 %
$C_{\min}$	11.0	$C_{\max}$	25.0
$Z_{\min}$	8.0 KB/s	$Z_{\max}$	40.0 KB/s

ger shape, which require twice of the computing resources Res_i and have a service time five times longer than the other operators. We summarize the application and infrastructure configuration parameters in Table 6.2.

We rely on ODRP to define the optimal placement and replication of each operator but the source and sink, which are placed a-priori. In the experiments, we assume that each operator can be replicated at most twice (i.e., $k_i = 2, \forall i$), while the source and sink are not replicated. In the ODRP model, we obtain the response time R_i of the operator i subject to the incoming load $\lambda_i/|\mathcal{U}|$ by modeling the underlying computing node as an M/M/1 queue.

6.5.2 Impact of Replication

In this experiment, we want to investigate the replication benefits when the application is subject to an increasing traffic load. Differently from the previous experiment, now each operator emits only 95 % of the incoming data; in this way, every operator imposes a different load which decreases from the source to the sink. We compare the performance achieved by ODRP with ODP, which optimizes only the operator placement. Note that ODP is a special case of ODRP where $k_i = 1, \forall i \in V_{dsp}$. Both the models are solved with the following configurations of weights: $(w_r = 0.8, w_c = 0.2)$,



(a) Response time

(b) Total number of operator instances

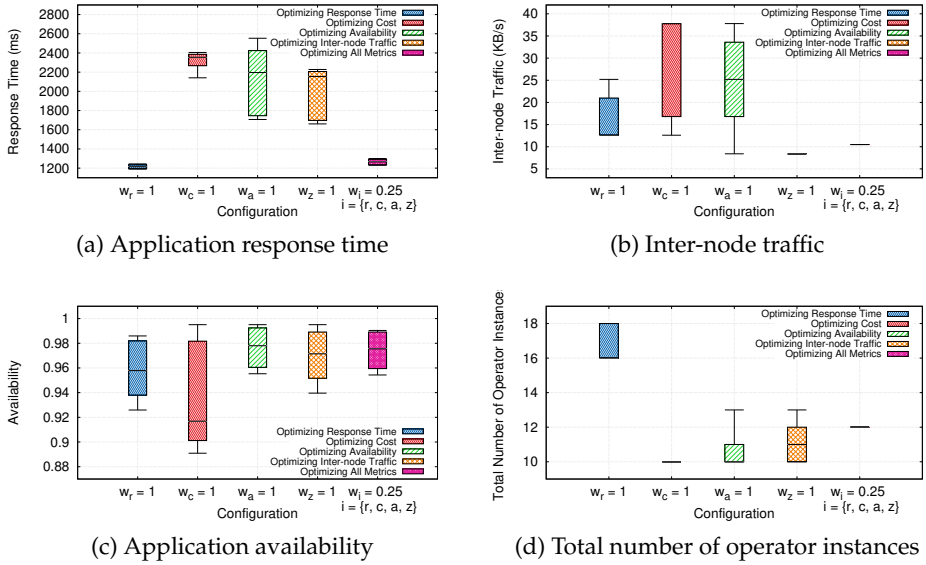
Figure 6.3: Impact of replication on the application performance

$(w_r = 0.5, w_c = 0.5)$, and $(w_r = 0.2, w_c = 0.8)$. The other weights are set to 0. In Figure 6.3 we indicate with "ODP(w_r, w_c)" the curves obtained by ODP under the different weight configurations, which overlap one another. Note also that "ODP(w_r, w_c)" overlaps with "ODRP($w_r = 0.2, w_c = 0.8$)".

During the experiment, the incoming load increases from 2 to 18 tuples/s, therefore the bottleneck operator imposes a load on the underlying computing node that ranges from 10% to 90% of its capacity. Although ODP cannot replicate the operators, it can find the optimal placement that minimizes the response time. As shown in Figure 6.3a, when the system becomes overloaded, i.e., after 14 tuples/s, the response time grows quickly, up to become 4 times higher than the case of medium load. A similar result is obtained when ODRP is configured with $(w_r = 0.2, w_c = 0.8)$: being the application cost much more important than the response time, ODRP tries not to replicate the operators. The opposite result is achieved when ODRP is configured with $(w_r = 0.8, w_c = 0.2)$, because replication is exploited to reduce the application response time. Figure 6.3b shows the number of operator instances. We can observe that, when the incoming load exceeds 8 tuples/s, ODRP with $(w_r = 0.8, w_c = 0.2)$ starts replicating the bottleneck operators one by one up to 18 instances, i.e., when every operator is replicated. As shown in Figure 6.3a, this strategy is beneficial for the response time, which does not increase more than 1.5 times with respect to the case of lightly loaded system.

6.5.3 Optimal Replication and Placement

This second experiment evaluates the effect on QoS metrics of different optimization objectives. We first compute the replication and placement solution by optimizing a single QoS metric. For example, to optimize the response time we set the weights as $w_r = 1$, $w_a = w_c = w_z = 0$. Then, we optimize the multi-objective function by uniformly weighting each metrics contribution (i.e., $w_r = w_a = w_c = w_z = 0.25$); we report in Table 6.2 the normalization factors used in Equation (6.21). Since the position of data



(a) Application response time

(b) Inter-node traffic

(c) Application availability

(d) Total number of operator instances

Figure 6.4: Impact of different optimization objectives on the QoS metrics

source and sink affects the optimal solution, we execute a run for each configuration that results by pinning these components on every pair of distinct nodes. Figure 6.4 presents the results in term of the different QoS metrics. For each optimization objective, a boxplot represents the performance distribution that results from the different locations of data source and sink. Each boxplot reports the minimum value, the 5th percentile, the median, the 95th percentile, and the maximum value.

When ODRP optimizes the response time R, the solution achieves the minimum achievable response time, as shown in Figure 6.4a, but also requires that all the operators are replicated, as shown in Figure 6.4d. Since the cost of running a configuration is directly proportional to the total number of operator instances, this is also the most expensive solution. We can also observe from Figure 6.4b that the inter-node traffic is indirectly minimized, because transmitting data over the network rather than locally introduces network delays that penalize the response time.

When ODRP optimizes the cost C, the placement solution tries to use less computing resources as possible, therefore no operator is replicated as shown in Figure 6.4d. This strategy penalizes the response time which almost doubles with respect to the minimum value, as shown in Figure 6.4a. Nothing can be concluded about the application availability (Figure 6.4c) and inter-node traffic (Figure 6.4b): since these metrics are not optimized, there is a set of equally optimal solutions that differ each other only in the operator placement on the same set of computing resources.

At a first sight, the results obtained when ODRP optimizes the application availability seem to be unexpected, because, as shown in Figure 6.4c, the application availability is not maximized. However, this behavior can be easily explained recalling that the data source and sink are placed a-priori on each pair of computing nodes, so their availability drives the overall application availability. In some cases ODRP increases the number of operator instances to improve the application availability: from Figure 6.4d, we can see that the maximum number of instances in this configuration is 13, i.e., 3 operators are replicated with degree 2.

When ODRP optimizes the inter-node traffic, the solution tries to co-locate the operator instances on the least number of nodes, so to reduce the amount of data transmitted over the network (see Figure 6.4b). An interesting behavior is highlighted by Figure 6.4d: sometimes ODRP takes advantage of replication in order to reduce the amount of data transmitted on the network; this behavior is analytically explained by Equation (6.7).

When ODRP optimizes all the considered QoS metrics, the resulting placement and replication solution has a response time (Figure 6.4a) and an inter-node traffic (Figure 6.4b) which are very close to the values achievable when optimizing a single-objective function. The total number of operator instances is always equal to 12, as shown in Figure 6.4d, therefore only two operators are replicated. Recalling the topology of our application, we can readily identify that these two replicated operators are the application bottlenecks.

6.6 Prototype-based Experiments

In this section, we present a series of experiments that revolve around S-ODRP, the prototype scheduler for Storm whose core is ODRP. They aim to show the generality and flexibility of the proposed formulation as well as its impact in terms of achievable application performance. After introducing the experimental setup and the reference application (Section 6.6.1), we provide a general overview on the runtime execution of S-ODRP (Section 6.6.2). Then, in Section 6.6.3, we show the benefits of the joint optimization of placement and replication when the DSP application is subject to an increasing load. Finally, in Section 6.6.4, we evaluate how S-ODRP can optimize several QoS metrics, such as response time, cost, availability, and inter-node traffic.

6.6.1 Experimental Setup

We perform the experiments using Apache Storm 0.9.3 on a cluster of 6 worker nodes, each with 2 worker slots, and a further node to host Nim-

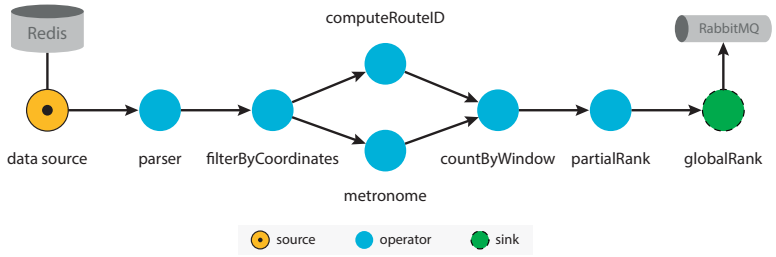


Figure 6.5: Reference DSP application

bus and ZooKeeper. Each node is a machine with a dual CPU Intel Xeon E5504 (8 cores at 2 GHz) and 16 GB of RAM. To better exploit the presence of independent CPU cores, we define that a worker slot can host at most 4 executors, i.e., $EPS_{\max} = 4$; therefore, a worker node can host at most 8 operator replicas, one for each available CPU core. We emulate wide-area network latencies among the Storm nodes using `netem`, which applies to outgoing packets a Gaussian delay with mean and standard deviation in the ranges $[12, 32]$ ms and $[1, 3]$ ms, respectively. As regards the pricing policy, we charge only the usage of computing resources, i.e., we set a unitary cost for each operator replica. We solve the ILP problem using CPLEX© (version 12.6.3) on the node hosting Nimbus.

As test-case application we developed a benchmarking application that solves the first query of the DEBS 2015 Grand Challenge [99]: by processing data streams originated from the New York City taxis, the goal of the query is to find the top-10 most frequent routes during the last 30 minutes. Its topology is represented in Figure 6.5. The *data source* reads the dataset from Redis, an in-memory data store, and pushes data towards a *parser* operator, which parses them and filters out irrelevant and invalid data. Afterwards, *filterByCoordinates* forwards only the events related to a specific observation area, whose extension is about 22 500 Km^2. The operator *computeRouteID* is in charge of identifying the route covered by taxis, and *countByWindow* counts the route frequency in the last 30 minutes; the notion of time is managed by a coordinator component, called *metronome*, which pulses when the time related to the dataset events advances. The following operators, *partialRank* and *globalRank*, compute the top-10 most frequent routes by leveraging on a two-step approach that enables to compute the ranking in a distributed and parallel manner. Finally, *globalRank* publishes the top-10 updates on a message queue, implemented with RabbitMQ. We assume that *data source* and *globalRank* are pinned operators. Moreover, since we investigate the initial application placement, we have set the data source so to feed the topology with a constant data rate, defined a-priori.

In the experiments, we define that each operator can be replicated at

Table 6.3: Parameters of the experimental setup

Application: service rate per operator, expressed in tuples/s (tps)

Operator	μ_i	Operator	μ_i
data source	284 tps	metronome	190 tps
parser	233 tps	countByWindow	335 tps
filterByCoordinates	253 tps	partialRank	2371 tps
computeRouteID	253 tps	globalRank	185 tps

Normalization factors for the ODRP utility function

Parameter	Value	Parameter	Value
$R_{\min}$	5 ms	$R_{\max}$	450 ms
$A_{\min}$	95%	$A_{\max}$	100%
$C_{\min}$	8	$C_{\max}$	18
$Z_{\min}$	0 tps	$Z_{\max}$	3400 tps

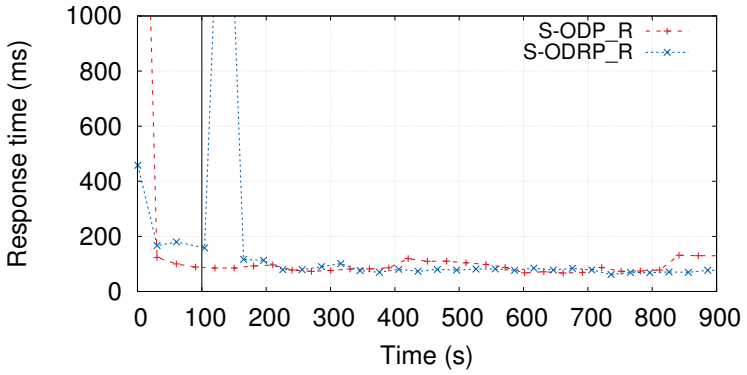


Figure 6.6: Runtime execution of S-ODRP and S-ODP

most three times (i.e., $k_i = 3, \forall i$), except for the pinned ones (i.e., *data source* and *globalRank*) and the *metronome*, which cannot be easily parallelized. Without loss of generality, in the ODRP model we estimate the response time R_i of operator i subject to the incoming load $\lambda_i/|\mathcal{U}|$ by modeling the underlying computing node as an M/M/1 queue, i.e., $R_i(\lambda_i/|\mathcal{U}|) = (\mu_i - \lambda_i/|\mathcal{U}|)^{-1}$, where μ_i is the service rate of i measured on a reference processor. The operators service rate and the other configuration parameters have been obtained through preliminary experiments and are shown in Table 6.3.

6.6.2 Evaluation of S-ODRP

This first experiment aims at showing the runtime execution of the S-ODRP scheduler, described by Algorithm 6, and its impacts on the application performance. As baseline we use S-ODP, a prototype scheduler for Storm

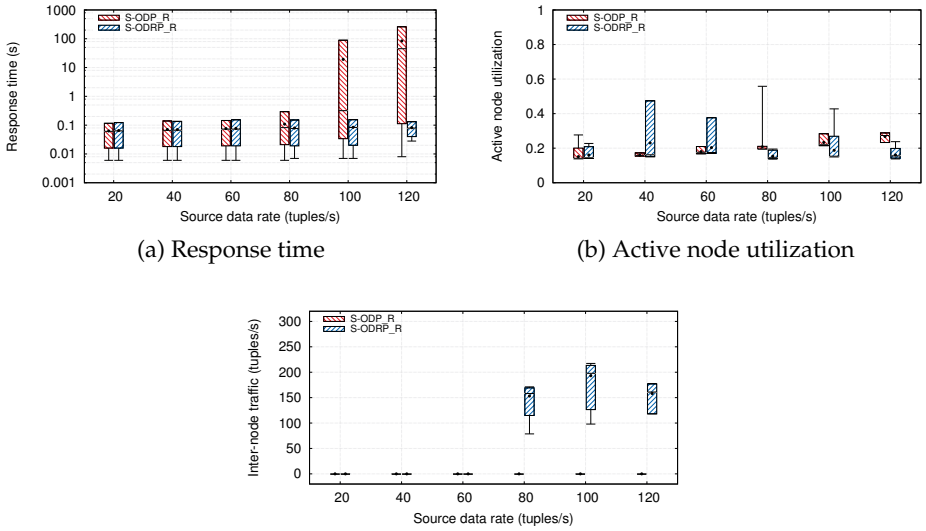
whose core is the ODP model that optimizes only the operator placement (see Chapter 4). Note that ODP is a special case of ODRP where $k_i = 1$, $\forall i \in V_{dsp}$. To simplify the presentation, we consider only the response time R and the monetary cost C as QoS metrics; all worker nodes have an availability of 100%. Both the optimization models focus on the minimization of the application response time R (i.e., $w_r = 1$, $w_c = 0$), so we name them as **S-ODRP_R** and **S-ODP_R**, respectively. Differently from S-ODP_R, S-ODRP_R computes R relying on the exchanged data-rate between the operators; as presented in Section 6.4, it deploys the application with a preliminary placement so to harvest the relevant data; this preliminary placement is computed by minimizing the deployment cost (i.e., $w_c = 1$, $w_r = 0$). The rescheduling event takes place after 100 s of execution and is represented in Figure 6.6 with a vertical line.

We set the data rate of the source operator to 80 tuples/s and launch the application. Figure 6.6 reports the resulting application response time. We observe that as soon as the placement is defined, i.e., after 0 s and after 100 s (the latter for S-ODRP_R only), a transient period is experimented where R is quite high and exceeds 1 s. This behavior depends on a well-known issue of the Storm framework [213], which starts the operators as soon as they are ready without coordination at level of the whole application; as a consequence, data emitted by an operator wait in inter-operator buffers until the following operator is up and running for processing. When the transient period ends, after 200 s, the applications deployed with the two schedulers experience quite similar performance in terms of response time.

Table 6.4 reports the total number of operator replicas deployed by the two schedulers. S-ODP_R cannot replicate the operators, therefore it instantiates a replica for each of them. With a source data rate of 80 tuples/s, S-ODRP_R replicates twice two operators, namely *computeRouteID* and *partialRank*, and runs the application with a total of 10 executors. S-ODRP_R replicates the operators as much as possible while considering that, when a new replica has to be located on a new worker node, the latter introduces network latencies that can overcome the benefits of replication in reducing the operator execution time. In this experiment, it is worth to observe that, although the scheduler is forced to use a second worker node, it places the replicas so to minimize the response time; in particular, only the replicas of *computeRouteID*, which have the lowest data rate exchanged with the other operators, are located on a separate node.

6.6.3 Impact of Replication

In the second set of experiments, we want to investigate the replication benefits when the application is subject to different incoming loads. We use the same settings of the previous experiment except for the source data rate



(a) Response time

(b) Active node utilization

(c) Inter-node traffic

Figure 6.7: Impact of replication on the application performance

and we compare how S-ODRP_R and S-ODP_R determine the placement of the reference application. In each single experiment, which lasts 900 s, the source data rate is constant and is set in the range $[20, 120]$ tuples/s with step 20. We collect the resulting QoS metrics as soon as the transient period ends (i.e., after 200 s) and we summarize the results in Figure 6.7 leveraging on a boxplot, which represents the QoS metric distribution through the minimum value, the 5th percentile, 50th percentile, 95th percentile, and the maximum value; the average value is also represented using a full dot. We define as *active node utilization*, the average utilization of all the worker nodes involved in the application execution; each node contributes with its

Table 6.4: Operator replication

Operator	**S-ODP**	**S-ODRP**		
		$20, 40, 60$ tuples/s	$80, 100$ tuples/s	120 tuples/s
data source	1	1	1	1
parser	1	1	1	3
filterByCoordinates	1	1	1	2
computeRouteID	1	1	2	2
metronome	1	1	1	1
countByWindow	1	1	1	3
partialRank	1	1	2	3
globalRank	1	1	1	1

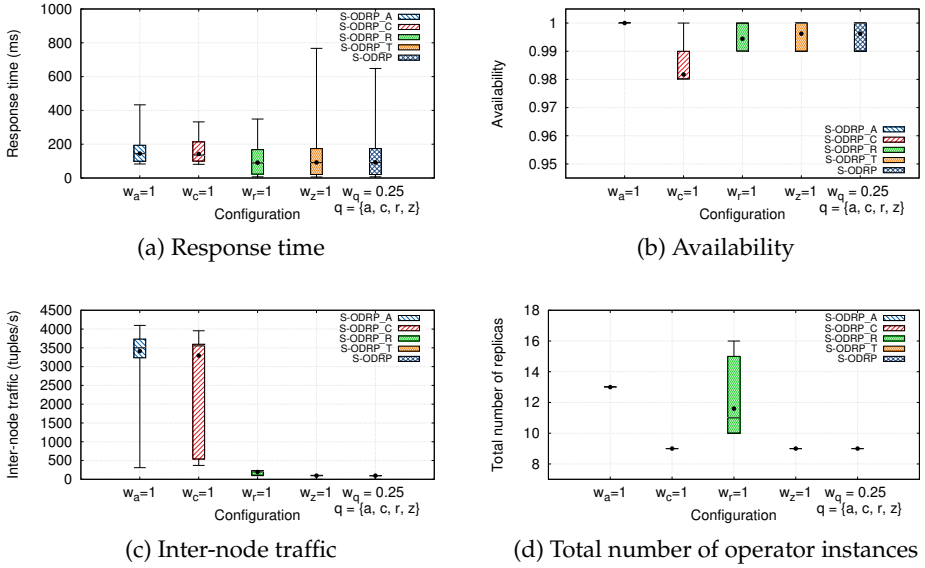
average utilization calculated on a time window of 60 s.

Figure 6.7a reports the application response time and Table 6.4 the number of replicas per operator. Although S-ODP_R cannot replicate the operators, it finds the optimal placement that minimizes R, as S-ODRP_R does. Indeed, when the replication is not needed, i.e., up to 60 tuples/s, the two schedulers achieve the same application performance. Note that, also in this case, network delays prevent S-ODRP_R from further replicating the operators: due to the absence of inter-node traffic (see Figure 6.7c), we can easily detect that all the 8 replicas run on the same node.

When the data source emits 80 tuples/s, the replication is needed: the *partialRank* operator represents a bottleneck, because it receives on average 2500 tuples/s, i.e., 5% more than its service rate. The utilization of the worker node that hosts the whole application for S-ODP_R, reported in Figure 6.7b, is around 20%, therefore the overload situation cannot be easily detected if not relying on fine-grain monitoring tools, which work at the level of single operator or CPU core. Conversely, S-ODRP_R detects the bottleneck and replicates the operator, which is then executed on a second worker node (see Table 6.4 and Figure 6.7c).

We observe a similar behavior when the data source emits 100 tuples/s. This time the bottleneck operator, *partialRank*, receives on average 30% more tuples than a single replica can process per unit of time. With S-ODP_R, the application response time is unstable and continuously grows during the experiment, up to 106 s per single tuple. Conversely, thanks to replication, with S-ODRP_R the application maintains the same response time of the configuration with 80 tuples/s.

The need of replication is further exacerbated when the data source emits 120 tuples/s. With S-ODP_R, the application response time explodes up to about 300 s per tuple. On the contrary, S-ODRP_R obtains an application response time that is not influenced by the increased load. To keep up with the incoming data rate, S-ODRP_R needs to replicate every operator. It instantiates 2 replicas for *filterByCoordinates* and *computeRouteID*, and 3 replicas for *parser*, *countByWindow*, and *partialRank*; comprising also the other operators, the application runs with a total of 16 executors (see Table 6.4) on 2 worker nodes. In spite of the increased incoming data rate, Figure 6.7c shows that the application deployment produces a fairly limited inter-node traffic. Finally, we observe from Figure 6.7b that, although the need of replication, the average value of the active node utilization is quite low. This behavior highlights that, for this specific use case, a static mapping between replicas and CPU cores does not lead to an efficient usage of resources; a possible solution to better exploit the available resources might be based on a dynamic multiplexing of replicas on the same CPU core. This optimization calls for the runtime adaptation of the application deployment, which will be the topic of the next chapters of this thesis.



(a) Response time

(b) Availability

(c) Inter-node traffic

(d) Total number of operator instances

Figure 6.8: Impact of different optimization objectives on the QoS metrics

6.6.4 Optimal Replication and Placement

This experiment evaluates the effect of different optimization objectives on QoS metrics, i.e., availability A, cost C, response time R, and inter-node traffic T. To this end, we set our experimental environment so that half of the worker nodes has an availability of 99% and the other of 100%, whereas the links are always available. We place the pinned operators (i.e., *data source* and *globalRank*) on a single worker node chosen randomly among those 100% available. S-ODRP determines the placement of the reference application when the data source emits 100 tuples/s. Figure 6.8 summarizes 25 runs, each of 900 s, where we collect QoS metrics after the initial transient period of 200 s. We first compute the replication and placement solution by optimizing a single QoS metric. For example, to optimize the response time we set the weights as $w_r = 1$, $w_a = w_c = w_z = 0$. Then, we optimize the multi-objective function by uniformly weighting each metrics contribution (i.e., $w_a = w_c = w_r = w_z = 0.25$); we report in Table 6.3 the normalization factors used in Equation (6.21). Figure 6.8 presents the results in term of the different QoS metrics.

When S-ODRP optimizes the application availability A (for short, S-ODRP_A), the solution finds the configuration where all the replicas are on the most available worker nodes. Observe that the placement of pinned operators, which are not relocated by S-ODRP, impacts on the overall application availability. In our experiments we placed these operators on nodes with 100% of availability. From Figure 6.8d and Equation (6.15), we observe

that, although this configuration of S-ODRP does not optimize the number of operator instances, multiple replicas can be executed until a new worker node with availability lower than 100% has to be activated. This setting of weights does optimize neither the response time nor the inter-node traffic (see Figure 6.8a and 6.8c): on average, the response time is almost 1.6 times higher than the minimum achievable, whereas the inter-node traffic is almost 35 times higher than the optimal one.

When S-ODRP optimizes the cost C (S-ODRP_C), the placement solution tries to use fewer replicas as possible, as shown in Figure 6.8d. However, the explicit modeling of the operator service rate enables to instantiate a configuration that can properly handle the incoming traffic: S-ODRP instantiates 9 replicas, i.e., replicates twice the bottleneck operator (*partialRank*). Nothing can be concluded about the other QoS metrics, i.e., response time, application availability, and inter-node traffic (see Figures 6.8a, 6.8b, and 6.8c): since these metrics are not optimized, there is a set of equally optimal solutions that differ each other only for the operator placement on the same set of computing resources.

When S-ODRP optimizes response time (S-ODRP_R), the placement solution experiences the minimum achievable value for this metric, as shown in Figure 6.8a, but it uses up to 16 replicas (Figure 6.8d). Observe that 16 replicas completely occupy two worker nodes, and the presence of network delays prevents the scheduler from instantiating other replicas. Since the cost of running a configuration is directly proportional to the total number of operator instances, this is also the most expensive solution. From Figure 6.8c, we can also observe that the inter-node traffic is quite low (almost double with respect to the optimal value), because transmitting data over the network rather than locally introduces network delays that penalize the response time.

When S-ODRP optimizes the network-related QoS metric, that is the inter-node traffic T (S-ODRP_T), the solution tries to co-locate the operator instances on the least number of nodes, so to reduce the amount of data transmitted over the network (see Figure 6.8c). In this configuration the number of replicas is neither minimized nor maximized; however, if the load is equally split among replicas, S-ODRP might take advantage of replication in order to reduce the amount of data transmitted on the network. As side effect of the co-location, the application response time is, on average, very close to the optimal one. Nothing can be concluded about the application availability (Figure 6.8b), which is not considered as an optimization objective of S-ODRP_T.

When S-ODRP optimizes all the considered QoS metrics, the resulting placement and replication solution has response time and inter-node traffic which are very close to the values achievable when optimizing a single-objective function. The total number of operator instances is equal to 9, as

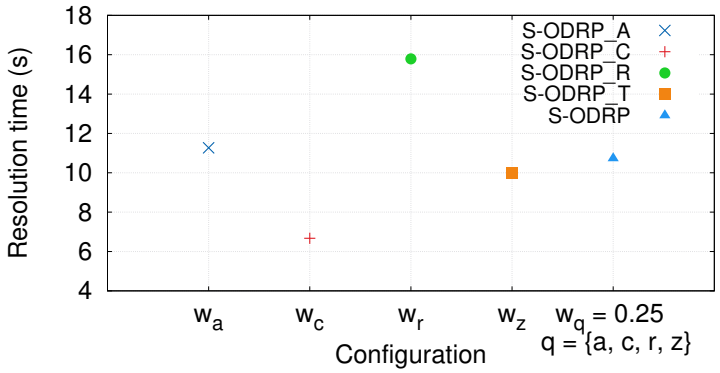


Figure 6.9: Average resolution time of ODRP with respect to the optimization objective

shown in Figure 6.8d, therefore only the bottleneck operator is replicated. The application availability ranges from 100% to 99%, and assumes 99.6% as average value. Although it might appear counter-intuitive, this result follows from the trade-off between the minimization of response and the maximization of the availability pursued in the utility function F. In particular, when the application availability is 99%, on the basis of the pinned operators placement, whose location is randomly defined a-priori, choosing a worker node that minimizes the response time, rather than one that maximizes the availability, provides a bigger contribution to the optimization of the utility function F.

On ODRP Resolution Time. We now discuss about the *resolution time* of ODRP and its relationship with the optimization goals. We consider as resolution time the time needed to compute the exact solution of the ILP problem. Although the investigated placement problem is fairly limited in size ($|V_{res}| = 6$, $|V_{dsp}| = 8$), the ODRP model includes about 55.8 K variables[3] and, considering all the 25 runs of the last experiment, its average resolution time is 10.84 s. Figure 6.9 provides more details on the average resolution time of ODRP with respect to the different weights (w_a, w_c, w_r, and w_z) used for the utility function F. Determining a placement that minimizes the application response time is the most computationally demanding configuration; conversely, the minimization of the application deployment costs registers the fastest resolution time. Note that the former is twice slower than the latter. Being ODRP an NP-hard problem, as demonstrated in Theorem 6.1, it does not scale well as the problem instance

[3]Observe that the number of multisets $\mathcal{X}$ of cardinality k, with elements taken from a finite set of cardinality n, grows very quickly, as $\frac{(n+k-1)!}{k!(n-1)!}$. This clearly shows the combinatorial nature of the problem at hand. Nevertheless, as we showed in Chapter 5 and we will show in Chapter 8, the problem structure can often be exploited so to reduce the computational cost of solving the problem formulation with a reduced quality degradation.

increases in size. Nevertheless, by determining the optimal replication and placement of DSP operators, ODRP provides a benchmark for evaluating heuristics, for developing new ones, and for identifying the most suitable ones with respect to the specific optimization objectives.

6.7 Summary

In this chapter, we have presented and evaluated ODRP, an ILP formulation that jointly optimizes the replication and placement of DSP applications. ODRP is a general and flexible model that can take into account the heterogeneity of computing and networking resources and can be conveniently configured to optimize different QoS metrics, whose importance depends on the application scenario. We have thoroughly evaluated the proposed contribution using both numerical and prototype-based experiments. The set of numerical experiments has validated the proposed model and has shown the benefits of a joint optimization of the operators placement and their replication on the application performance. The set of prototype-based experiments has investigated how S-ODRP, an ODRP-based scheduler for Apache Storm, manages the deployment of a real and well-known application. The latter is the DEBS 2015 Grand Challenge application, which processes real time data generated by taxis moving in a urban environment. Besides confirming the ODRP flexibility and the benefits of a joint optimization of replication and placement, this second set of experiment has shown how ODRP can contextually optimize several QoS metrics.

Chapter 7

Elastic Storm

> Due to the unpredictable rate and varying nature of incoming workloads, DSP applications demand for adaptation capabilities. We extend Storm with two mechanisms that support the runtime reconfiguration of DSP applications: elasticity and stateful migration of DSP operators.

In the previous chapters, we investigated the initial deployment of DSP applications. From this chapter onwards, we turn our attention on application runtime and the need for changing the application deployment so to keep satisfying performance during the application execution. Indeed, DSP applications are usually long-running and subject to periodic or unpredictable workload variations. Elasticity consists in automatically scaling the number of parallel instances for the DSP operators (i.e., data parallelism), so that each instance can process a subset of the incoming data flow in parallel (e.g., [65, 81, 86]).

For the execution of DSP applications, users commonly rely on DSP frameworks (e.g., Apache Storm [199], Spark [221], Flink [27]), that offer simple programming interfaces, abstracting away the underlying infrastructure and complexity of distributing the operators. Although including numerous features, in most cases these open-source frameworks only support a static or manual definition of the operator parallelism. Moreover, most DSP frameworks supporting elasticity are equipped with mechanisms in an embryonic stage: they dynamically scale the application in a disruptive manner, because they enact reconfigurations by killing and restarting the whole application. As a consequence, to support workload fluctuations, the user determines the number of parallel instances for the operators on the expected maximal workload, achieving either an average under-utilization of the system, because load peaks can rarely occur, or being unable to manage a bursty workload with unexpected fluctuations. Therefore, they deploy the application on a fixed number of computing

nodes and do not fully exploit the Cloud computing principles, which promote the elastic usage of on-demand resources. Among the open source DSP systems, Apache Storm has received increasing interest in the last few years and, as shown in Chapter 2, different works have proposed extensions, defined new scheduling policies, and build applications on top of it (e.g., [9, 28, 58, 213]).

In this chapter, we extend Storm by introducing two mechanisms that support the runtime adaptation of DSP applications: automatic elasticity and stateful migration. The *elasticity* mechanism implements scaling decisions at the framework level, i.e., it allows to automatically adapt the number of parallel instances for each application operator, according to a scaling policy. Different scaling policies can be defined, as surveyed in Chapter 2; as proof of concept, we propose a simple threshold-based policy that elastically changes the number of parallel instances of each operator according to the incoming workload. Once equipped with elasticity at the framework level, Storm can be then properly coupled with a lower-level scaling system that realizes elasticity at the infrastructure level by acquiring and releasing the computational nodes as needed, therefore encompassing the on-demand resource principle of Cloud computing. The *stateful migration* mechanism supports the relocation of the operator internal state on a different node and enables Storm to change the application deployment at runtime, without compromising the application integrity in terms of extracted information. This mechanism operates at fine granularity with respect to the execution model adopted by Storm and allows multiple and concurrent migrations.

The main contributions of this chapter are as follows.

- We extend Storm with an automatic *elasticity* mechanism that changes the number of parallel instances for the operators at runtime; we realize, as proof of concept, a threshold-based scaling policy that aims at maximizing the system utilization (Section 7.3).
- We enhance Storm with stateful operator migrations, which enable the self-adaptation capabilities of DSP applications in a non-destructive way, preserving the operators state (Section 7.4).

A set of experiments run with the enhanced Storm shows the benefits and overhead of the newly introduced mechanisms (Section 7.5). Our extension is fully modular and loosely coupled from the existing architecture of Storm, therefore existing solutions based on or proposed for Storm in the literature can transparently reuse the new functionalities. To this end, we publicly release our source code to the community[1].

[1]Elastic Storm is available on GitHub: `http://bit.ly/1oUjZAi`

7.1 Related Work

As analyzed in Chapter 2, the most popular open-source DSP frameworks (e.g., Storm, Spark Streaming, Flink, and Heron) do not fully support elasticity and stateful migration, which are two features that have recently received an increasing attention.

As regards the operator elasticity, in most cases these frameworks require their users to manually tune the number of replica per operator, thus potentially leading to a sub-optimal provisioning of resources. In recent years[2], some of these frameworks have been equipped with elasticity mechanisms, which nevertheless are in a embryonic stage: indeed, they dynamically scale the application in a disruptive manner, because they enact reconfigurations by killing and restarting the whole application (which introduces a significant downtime). Storm does not natively support elasticity. Yang and Ma [216] have investigated the internal architecture of Storm and have proposed different strategies for relocating stateless executors, achieving a reduction of the application latency degradation. Since we introduce the new mechanisms on top of the existing architecture of Storm, enhancements on its internal components are beneficial also for our extension. Differently from our work that exploits replication to implement data parallelism, Liu et al. [129] exploit replication for achieving fault-tolerance in Storm; in this case, multiple operator replicas process the same input stream and generate the same output stream (i.e., active replication — see Chapter 2). When equipped with Dhalion [60], Heron exposes elastic capabilities and allows to update the application topology without restarting its execution. Nevertheless, at the time of writing, Dhalion has not yet been open-sourced. From version 2.0, the micro-batched streaming module of Apache Spark (i.e., Spark Streaming) introduces the dynamic allocation feature. The latter uses a simple heuristic where the number of executors is scaled up when there are pending tasks and is scaled down when executors have been idle for a specified time. Finally, Flink, although supporting stateful operators (and their stateful reconfiguration upon request), does not yet autonomously adapt the application deployment [26, 27].

In Chapter 2, we have shown that most elasticity policies are threshold-based and rely on the measured CPU utilization of the system nodes or of the operator replicas (e.g., [31, 71, 80]). Other solutions use more complex policies to determine the scaling decisions (e.g., [65, 78, 81, 132]). For example, Lohrmann et al. [132] propose a strategy that enforces latency constraints by relying on a predictive latency model based on queueing theory. Heinze et al. [78] propose a model to estimate the latency spike created by a set of operator movements and use it to define an elastic oper-

[2] At the moment of writing this contribution, published in [30], the most popular open-source DSP frameworks did not support elasticity.

ator placement algorithm that minimizes the latency violations. De Matteis and Mencagli [46] present an interesting elasticity policy, which relies on a proactive and control-theoretic method that takes into account a limited future time horizon to choose the reconfigurations to execute. Similarly to the approaches in [81, 80, 132], our proof-of-concept scaling policy is reactive and threshold-based; however, we decouple the elasticity mechanism from the corresponding policy, which can be easily changed. Bilal and Canini [20] propose a framework for offline parameter tuning. This framework determines the best configuration (e.g., in terms of operator replication) according to the performance goals through a set of preliminary DSP application runs. Differently from our approach, they assume the scheduling is static, therefore the application cannot be conveniently reconfigured when the incoming workload changes.

While scaling stateless operators can be achieved by just starting or stopping new operator instances, elasticity of stateful operators requires state migration to preserve the consistency of the operations [65]. The most common solutions, as seen in [78], are the pause-and-resume approach (that we adopt in this thesis) and the parallel track approach (see Chapter 2). To identify the portion of state to migrate, Castro Fernandez et al. [31] expose an API to let the user manually manage the state, while Gedik et al. [65] automatically determine, on the basis of a partitioning key, the optimal number of state partitions to be used and to migrate. In our approach, the minimum unit of migratable state is defined by the user through the Storm execution model. ChronoStream [210] natively supports stateful migrations and uses a lightweight protocol that leverages on distributed checkpoints to minimize the amount of state relocated during a migration. Conversely, our work is driven by the existing Storm architecture. Storm also includes Trident, which provides high-level processing abstractions such as joins, aggregations, and filters. Differently from our extension, Trident can persist a state which is obtained by applying a sequence of Trident transformations on the input data. However, this approach requires to play the stream as a sequence of micro-batches, processed in a commit-like fashion, thus causing a constant latency overhead.

7.2 Elastic Storm Architecture

Neither stateful migrations nor elastic scaling decisions are supported by the current architecture of Storm. However, Storm provides the `rebalance` function which allows to manually change the number of executors for the topology operators. We extend Storm to support elasticity and stateful migration, aiming at providing mechanisms that can be reused by the Storm-based extensions in the literature, including those focusing on the operator

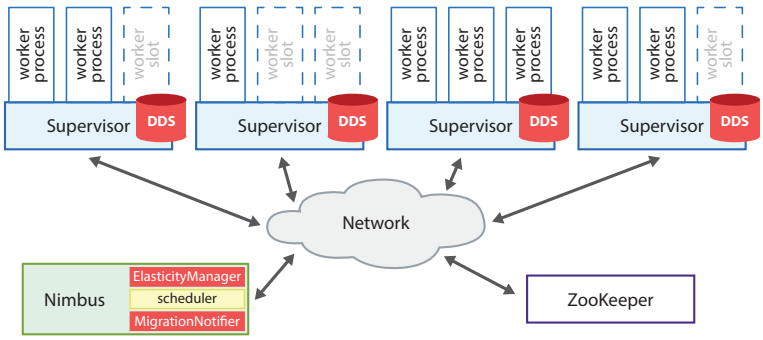


Figure 7.1: Extended Storm architecture.

placement policies (e.g., [9, 28, 34, 58, 213]). Figure 7.1 illustrates in red the newly introduced components, namely the ElasticityManager, the MigrationNotifier, and the Distributed Data Store (DDS).

The *ElasticityManager* is located on the centralized component Nimbus and is in charge of evaluating scaling decisions for the topologies managed by Storm. A scaling decision changes the operator parallelism, i.e., it increases or decreases the number of executors for an operator, improving resource utilization in relationship to the incoming data rate. Since the newly added executors have to be placed on the worker nodes, the ElasticityManager is executed before the Storm scheduler, so that the latter can analyze and define the placement of the newly introduced executors.

The *MigrationNotifier* is executed after the scheduler; it initiates the migration by notifying the tasks of the executors that have changed their placement to save their internal state. The MigrationNotifier will resume the execution as soon as all the migrating tasks can be terminated without loss. Afterwards, the new assignment plan, defined by the scheduler, will become effective and the migrating tasks can be restored on the new worker nodes.

The *Distributed Data Store* (DDS) is a data store introduced on each worker node and allows the migrating tasks to save their internal state before terminating their execution. It acts as a local repository for migrating tasks, which can retrieve and restore their state as soon as they are instantiated on the new worker nodes. The presence of a locally available data store allows us to minimize the amount of state moved across the network during a migration.

We observe that the presence of centralized components that oversee the adaptation of DSP applications (such as the ElasticityManager and the MigrationNotifier) represents a scalability bottleneck for the execution of Elastic Storm in geo-distributed environments. In the next chapters we will further extend the architecture of Storm to overcome this scalability issue.

7.3 Operator Elasticity

The *ElasticityManager* dynamically reconfigures a topology by scaling horizontally (i.e., scaling out and in) the number of executors for an operator. A *scale-out* decision increases the number of executors when the operator needs more computing resources. A *scale-in* decreases the number of executors when the operator under-uses its resources. Recalling the execution model of Storm (Section 3.2), a scaling operation changes how tasks are grouped into executors, thus leading to a possible relocation of the operator state on a different worker node.

7.3.1 Design Overview

The *ElasticityManager* is designed as a loosely coupled component of Storm, which works independently from the placement policy. Periodically, every T_{nbs}, Nimbus activates the ElasticityManager; the latter analyzes each of the topologies running in Storm by possibly taking scaling decisions for at most a single topology at a time. This limitation allows to evaluate the effects of scaling decisions on the other running topologies. We describe the simple yet effective threshold-based policy adopted by the ElasticityManager to compute the scaling decisions in Section 7.3.2. The resulting decisions are then implemented exploiting the `rebalance` function provided by Storm, which allows to change the set of executors for the operators of a topology and to adapt the Storm execution environment. As side effect, `rebalance` suspends the execution of the topology spouts until the scheduler defines a new placement for the topology. To avoid stressing a topology with frequent scaling decisions that may lead to instability, after a `rebalance` we let the topology enter in a cooldown state for the next EM_{cld} invocations of the ElasticityManager.

7.3.2 Scaling Policy

We design a reactive and threshold-based policy: the ElasticityManager takes scaling decisions comparing the fraction of CPU time utilized by each executor against some thresholds. Specifically, U_e measures the fraction of CPU time used by the executor $e \in E(op)$ of the operator $op \in OP(q)$ in the topology q[3]. That is, U_e is the CPU utilization per executor e. The ElasticityManager considers a single topology q at a time and decides first the scale-out actions, then the scale-in ones.

[3]Since an executor is a Java thread, the CPU time for the thread can be obtained relying on the `ThreadMXBean` class.

Scale-out. It adds a new executor for each one in overload. Formally, for each $e \in E(op)$, with $op \in OP(q)$, such that

$$U_e > ScaleOutThr$$

where $ScaleOutThr$ is the upper usage threshold, one new executor is added to $E(op)$. According to this policy, the number of executors for op can be at most doubled in a run of the ElasticityManager. All the operators subject to a scale-out operation compose the set $OP^{sc}(q)$.

Scale-in. It halves the number of executors of an operator, if all those existing are underloaded. Scale-in decisions are evaluated for all the operators not already under a scaling-out operation. Formally, for each operator $op \in OP(q) \backslash OP^{sc}(q)$ such that

$$U_e < ScaleInThr \quad \forall e \in E(op)$$

where $ScaleInThr$ is the lower usage threshold, the number of executors is halved or set to $minExec(op)$, which is the configurable minimum for operator op.

This scaling strategy doubles or halves the number of executors for an operator. Therefore, we conveniently define the scaling thresholds in a such a way that $ScaleOutThr, ScaleInThr \in [0, 1]$ and we can guarantee a stability gap $S \in [0, 1]$ between them such that $ScaleOutThr > 2 \times ScaleInThr + S$. The higher the values for S, the more conservative are the scaling decisions.

7.3.3 Elasticity and Placement

When a topology is subject to a scaling decision, the ElasticityManager conveniently marks it with a special label. The Storm scheduler can thus adopt a specific placement policy, which assigns only the added or changed executors by minimizing, for example, the amount of relocated operator state. Since in Storm the number of tasks for each operator is defined a-priori and cannot change at runtime (see Section 3.2), the elasticity changes how the tasks are grouped into executors. If the operator is stateful, a scaling decision leads to the relocation of a partition of the operator state, which could be costly or negatively impact the application performance. Therefore, we implement a simple placement policy for those topologies subject to scaling decisions, which places the new or updated executors by minimizing the number of tasks that should be relocated with respect to the previous configuration. This strategy assumes that the operator state is uniformly distributed across its partitions (i.e., tasks); if this condition does not hold true, more sophisticated strategies can perform better. Furthermore, this strategy consolidates the application executors on fewer worker nodes.

7.4 Stateful Operator Migration

After either a scaling operation or the definition of a new application placement, some executors may be relocated on the worker nodes. If the executor is stateless (i.e., it contains tasks of a stateless operator), the relocation can be easily performed by terminating the executor on the old location, moving its code to the new location, and restarting it. On the other hand, if the executor is stateful (i.e., it contains tasks of a stateful operator), we also need to efficiently *migrate* its internal state, so to preserve the integrity and consistency of the outputted streams. As a consequence, this kind of migration can involve a sophisticated cooperation among several components.

7.4.1 Overview

We propose a stateful migration solution that uses a pause-and-resume approach [78], which extracts the current state from the old instance and replays it within the new instance. To this end, the executor needs to be paused to ensure a semantically correct migration. Since in Storm a task represents the smallest entity that handles a partition of the operator state, we extend Storm to support *task-level*, or fine-grained, migrations. Note that this kind of migration covers the needs not only of scaling decisions, which define a new tasks-to-executors mapping, but also of replacement decisions, which move already existing executors. Furthermore, thanks to the fine granularity, we can parallelize the migrations towards different computing locations, for example when an executor is split in tasks that will be relocated in different locations.

7.4.2 Extended Architecture

The design of the migration protocol aims at satisfying the following requirements: (1) to be transparent to and reusable by the other existing Storm components; (2) to preserve the operator semantics by avoiding tuple loss and tuple reordering; and (3) to minimize the amount of data transferred using the network. Our intent is also to minimize the impact on the existing Storm architecture, reducing the amount of new code and reusing properly the Storm functionalities. The key idea is to enhance the tasks with the ability of exporting the operator state from the old worker node and of importing it to the new one. We realize this idea thanks to the cooperation of the following new components of Storm (see Figure 7.1):

- a Distributed Data Store (DDS) which enables to decouple the operator state from the related task during the migration;

- an extension of the Storm API that allows to define the code of spouts (i.e., data sources) and bolts (i.e., operators and final consumers). Specifically, we introduce the *StatefulSpout* and *StatefulBolt* classes, which enrich the tasks with the ability of storing and retrieving the partition of the operator state from the DDS in a user-transparent way;
- a centralized *MigrationNotifier* that orchestrates the migration and prevents Storm from propagating new placement decisions until all the tasks involved in a migration have saved their state to DDS. Indeed, in the official release of Storm, after a new placement decision the executors that have changed location are restarted, making the tasks loose their state.

The proposed solution also exploits ZooKeeper, which provides a coordination and synchronization service that simplifies the cooperation among the distributed components.

DDS. Each worker node is equipped with a data store, which is accessible to all the other worker nodes. This allows a migrating task to save its state on a local storage, so to minimize the amount of data transferred on the network. The data store is implemented as an in-memory caching system with Hazelcast[4]. We do not use ZooKeeper for this purpose, because it is not designed to hold large data values.

StatefulSpout and StatefulBolt. These classes support and execute the stateful migration protocol presented in the next section and should be used when the topology has at least one stateful operator. That is, also stateless operators should be defined leveraging on these new classes. Differently from the default implementation of spouts and bolts, these new classes are enhanced to: (1) provide a common interface that defines the task state and allows to export and import it with `getState()` and `setState()`, respectively; and (2) define two execution modes for the task, namely the *operational mode* and the *migration mode*. The former represents the traditional execution mode of the task, which runs the operator logic that is defined by the user. The latter is used when the task or a neighbor (i.e., upstream, downstream) task is migrating; it allows to safely stop the task execution, save and restore the internal state, and avoid tuple loss and tuple reordering — thus preserving the operator integrity. Moreover, since the coordination among tasks relies on ZooKeeper, the StatefulSpout and StatefulBolt classes include a Watcher component that asynchronously observes and retrieves information published by the MigrationNotifier or by other tasks.

MigrationNotifier. This component executes on the centralized entity Nimbus. Basically, the MigrationNotifier intercepts the scheduler assignment plan, notifies the tasks that should export their internal state, and

[4]Hazelcast: `https://hazelcast.com/`

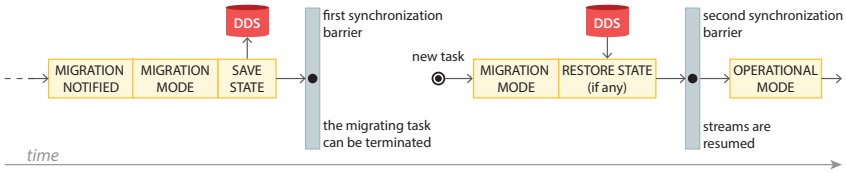


Figure 7.2: Sequence of operations performed by a migrating task.

waits until each of them has correctly completed this operation (i.e., it reaches the first synchronization barrier, as presented next). When the MigrationNotifier resumes the execution, Nimbus can disseminate the new placement decisions to the worker nodes; the latter will terminate and launch the worker processes (and related executors) according to the new placement.

7.4.3 Migration Mode and Migration Protocol

When the MigrationNotifier communicates, through ZooKeeper, the set of tasks involved in a migration, these tasks, as well as the tasks that precede and follow them in the topology, enter in the migration mode. Each task relies on the Watcher component to asynchronously check for this kind of notifications. The migration mode runs the migration protocol, which specializes the task behaviour with respect to the role played during the migration. We identify three roles: the task precedes a migrating task in the topology, the task itself is migrating, and the task follows a migrating task in the topology. For short, we refer to them as *upstream* task, *migrating* task, and *downstream* task, respectively.

To avoid tuple loss or their reordering, we need to pause the streams directed to a migrating task and save the tuples in transit, so to replay them as soon as the migration is completed. To this end, we introduce an *OutputBuffer* that resides on the upstream task and allows to temporary store the tuples directed towards a migrating task that could change its location. The upstream task explicitly notifies the last tuple sent on a stream, leveraging a special end-of-stream (EOS) message. For sake of efficiency, we introduce also an *InputBuffer*, which resides on the migrating task and on the downstream ones. Using this buffer, the migration protocol can stop the execution of the operator logic and rapidly retrieve the incoming tuples from the communication link.

We now present the migration protocol according to the role played by a task during a migration: upstream task (of a migrating one), downstream task (of a migrating one), and migrating task. If a task plays different roles during a migration (e.g., upstream and downstream task), the following procedures are combined. As depicted in Figure 7.2, the migration proto-

col is characterized by two synchronization barriers, which indicate respectively that a task has correctly saved its state and has completely recovered it.

Upstream Task. Entering the migration mode, the upstream task stops (only) the streams directed to the migrating task: it emits the EOS message and stores the subsequently produced tuples for that stream on the OutputBuffer. Besides the migrating task, this EOS message can be transmitted to other tasks (e.g., the other operator tasks); the latter directly join the second synchronization barrier. As soon as also the migrating task reaches the second synchronization barrier, the migration is completed: the upstream task emits the tuples stored within the OutputBuffer and switches back to the operational mode.

Downstream Task. Entering the migration mode, the downstream task stops the computation and rapidly downloads the streams coming from the migrating task; all the received tuples that precede the EOS message are stored into the InputBuffer. Afterwards, it switches back to the operational mode and resumes the computation.

Migrating Task. Although logically sequential, the execution of the migration mode of the migrating task is divided in two parts, namely *save state* and *restore state*, which can be executed on two different computing locations.

a) Save State. Entering the migration mode, the migrating task stops the computation, emits the EOS message to its downstream tasks, and buffers the incoming tuples on the InputBuffer until the EOS message is received from its upstream tasks. As soon as these operations are completed, the task pushes on the local DDS the operator state and the InputBuffer. The operator state is extracted relying on the `getState()` function that the user implements. Since computation is stopped, the OutputBuffer is always empty. At this point, the task is ready to be safely terminated, reaches the first synchronization buffer, and disseminates this information to the other tasks using ZooKeeper.

b) Restore State. When a new task is launched, it automatically enters the migration mode and checks on ZooKeeper if it is involved in a migration. If so, it contacts the DDS on the old worker node and recovers the operator state together with the InputBuffer. The operator state is imported using the `setState()` function that the user implements. Afterwards, the migration is considered as completed, the task reaches the second synchronization barrier and spreads this information to the other tasks using ZooKeeper. When all the tasks involved in a migration reach the second synchronization barrier, the paused streams will be resumed and the application will continue the execution. In the meanwhile, the task retrieves and processes the tuples from the InputBuffer and, then, switches to the

operational mode.

The proposed protocol supports concurrent migrations, because each task can autonomously relocate its state.

7.5 Experimental Results

To analyze the benefits and overhead introduced by the new elastic scaling and stateful migration mechanisms, we run a set of experiments executing a stateful DSP application on Storm, by first enabling and then disabling the new mechanisms.

7.5.1 Experimental Setup

We run the experiments using Storm 0.9.3 on a cluster of 5 worker nodes and one further node for Nimbus and ZooKeeper. Each node is a *m4.xlarge* AWS EC2 virtual machine with 4 vCPUs on Intel Xeon E5-2676 and 16 GB of RAM.

In our experiments, Storm determines the operators placement using the scheduler designed by Xu et al. [213], which assigns the operators on the worker nodes in descending order of incoming and outgoing traffic exchanged using the network so to minimize the inter-node traffic. Differently from the default round-robin scheduling policy, this one triggers executor reassignments if it finds a new configuration that reduces the inter-node traffic. Moreover, the policy ensures an even distribution of the executors on all the worker nodes, because a node can run at most E/N executors, where E is the total number of executors and N the number of worker nodes. To avoid inter-process communication overhead, the scheduler by Xu et al. uses only a worker slot per node. Since its source code is not publicly available, we implement the scheduler according to the description in [213] and integrate it with the special placement policy for the new or updated executors that we presented in Section 7.3.3.

The ElasticityManager is executed together with the scheduler every $T_{nbs} = 10$ s, and *ScaleOutThr* and *ScaleInThr* are set to 0.7 and 0.2, respectively. After a scaling decision, the topology enters in a cooldown state for the next $EM_{cld} = 12$ invocations of the ElasticityManager (i.e., for 120 s), where no new scaling decision can be applied. In preliminary experiments, not reported for space limits, we found that this setting ensures a stable behavior of the system.

As testing application we implement the Frequent Pattern Detection (FPD) [51], which analyzes tweets from Twitter and retrieves the most frequent patterns (i.e., those that occur more than 20 times) on a sliding window of 60 s. Figure 7.3 shows the FPD topology and Table 7.1 reports the

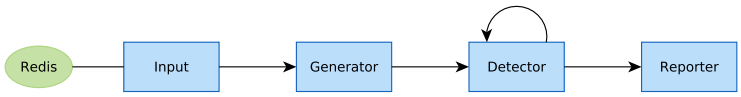


Figure 7.3: Frequent Pattern Detection topology.

Table 7.1: Number of executors and tasks, and minimum number of executors for each operator of the FPD application.

Operator op	$\|E(op)\|$	$\|T(op)\|$	$minExec(op)$
Input	1	1	1
Generator	2	2	1
Detector	5	20	1
Reporter	1	1	1

configuration of its operators. The topology spout reads input data from Redis, a shared memory, and emits them according to an exponential distribution with parameter λ.

Each experiment comprises five sequential phases, each lasting 900 s, and with input data rate according to the sequence $\lambda = \{120, 350, 900, 250, 120\}$ tweets/s. This workload stresses the infrastructure of Storm, which has to repeatedly evaluate the application placement at runtime. In the following figures, the beginning of each phase is represented with a vertical dotted line with a rhombus on the extremities. A scaling decision, which changes the number of executors for the topology, is represented with a vertical dash line, whereas a scheduling decision, which changes the placement of the executors, is represented with a vertical dot-dash line and a symbol "+" on top. The performance metrics are collected through the Storm metric system, which every 5 s provides an average value computed on a sliding window of 600 s, made of samples harvested every 5 s. Since this metric system is stateless, after a migration the old samples are lost and thus the following figures will show some zero values.

7.5.2 On the Elastic Scaling Mechanism

Figure 7.4 shows the application latency, i.e., the average latency experienced to traverse the entire FPD topology. When the elastic scaling mechanism is deactivated (referred to as "w/o E+SM" in Figure 7.4), the application ability of handling the incoming data rate depends on the parallelism that is statically defined by the user at design time. In this case, the application can manage the data source with $\lambda = 120$ tweets/s, but cannot keep the pace when the data rate reaches $\lambda = 350$ tweets/s; the system becomes unstable, as confirmed by the continuous increase of the application latency after 1500 s. When the input data rate reaches 900 tweets/s, the applica-

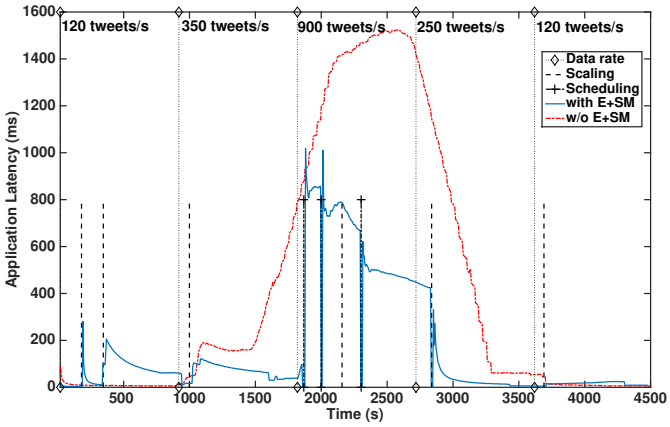


Figure 7.4: Application latency with and without elasticity and stateful migration.

tion latency grows up to 1500 ms. When the source reduces its data rate to $\lambda = 250$ tweet/s, the system can properly handle the buffered elements, until 3300 s, when all the buffers are empty, and the experienced application latency is below 90 ms. The elastic scaling and stateful migration mechanisms (referred to as "with E+SM" in Figure 7.4) improves significantly the application performance. When the data source emits 350 tweets/s, a scaling decision allows to efficiently handle the incoming load by doubling the number of executors for the Detector operator. Similarly, when the input data rate reaches 900 tweets/s, three reassignments and a scaling decision lead to an application latency lower than 900 ms. Soon after the occurrence of either a scaling or scheduling decision, we observe a transient period, where the application latency increases due to the processing of the collected buffers and the overhead imposed by the extended Storm to restart the executors on the new worker nodes.

Figure 7.5 shows how the ElasticityManager scales in or out the number of running executors during the experiment. In the first phase, the unneeded executors are terminated, ensuring that the utilization of the running executors is higher than the $ScaleInThr$ threshold. When the load imposed to the system increases, new executors are launched up to the third phase, when 23 executors run concurrently. As the incoming load starts decreasing, the number of executors decreases as well and, at the end of the experiment, only 8 executors are running. Except for the Generator operator whose parallelism is halved in the first phase, only the Detector operator is scaled out and in, since it constitutes the bottleneck of the FPD application. The ElasticityManager changes the Detector parallelism degree in the following sequence: $\{20, 10, 5, 10, 20, 10, 5\}$.

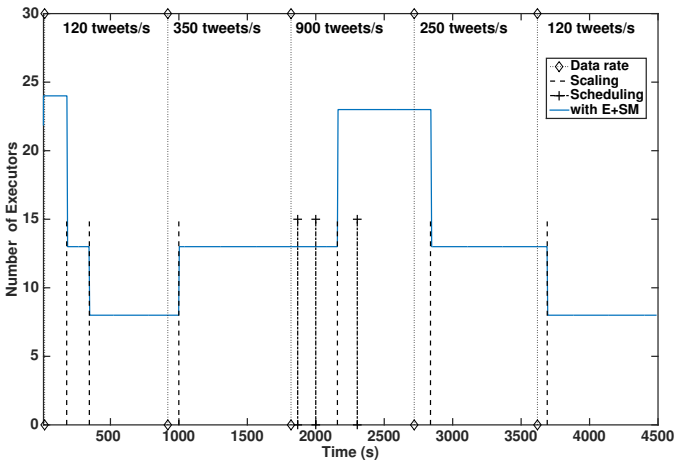


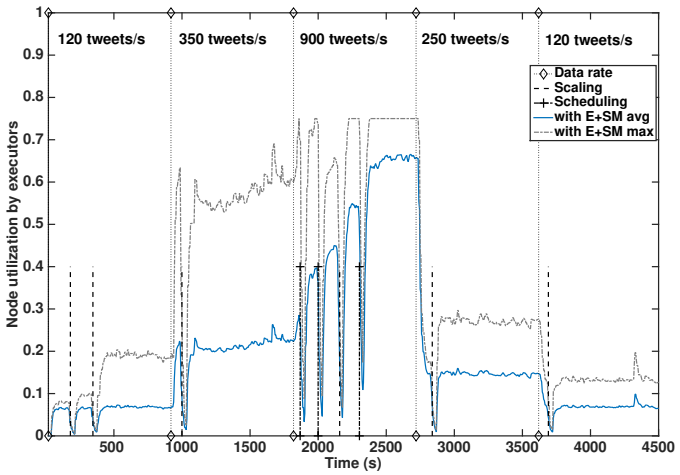
Figure 7.5: Effects of the elasticity on the number of executors.

When the elastic scaling and stateful migration mechanisms are both active, the application can better exploit the available resources. Figure 7.6a represents the average and maximum node utilization by the application executors. We can observe that the benefits of the scaling and scheduling decisions are complementary. The former allow to change the set of executors in order to better exploit the available computing resources, whereas the latter allow to balance the load among the worker nodes, enabling a more efficient usage of the available resources. When the elasticity and stateful migration mechanisms are both disabled, as results from Figure 7.6b, the system utilization is low and the application cannot process the increasing load in a timely way. This happens because the fixed number of executors overloads a subset of the computing resources, whereas the remaining subset of computing resources is free and not utilized. For example, with $\lambda = 900$ tweets/s, on the worker node with the maximum utilization (of about 50%), the half of the CPU cores with running executors is overloaded, whereas the other half is almost idle.

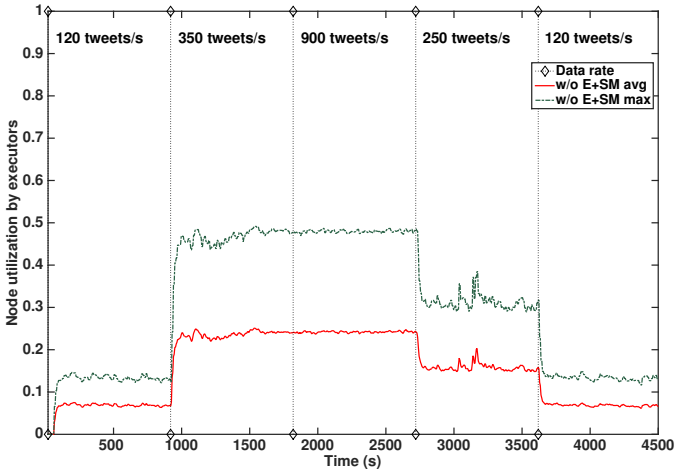
7.5.3 On the Stateful Migration Mechanism

We now analyze in detail the stateful migrations occurred during the experiments, with a special focus on two characteristics: the amount of state transferred using the network and the overhead introduced by the stateful migration.

The system has performed 9 migrations, 6 of which are related to a scaling decisions and 3 to placement decisions. In the following, we refer to each of them using the *migration index*, a progressive number that reflects the chronological order when the migration has been performed. We first



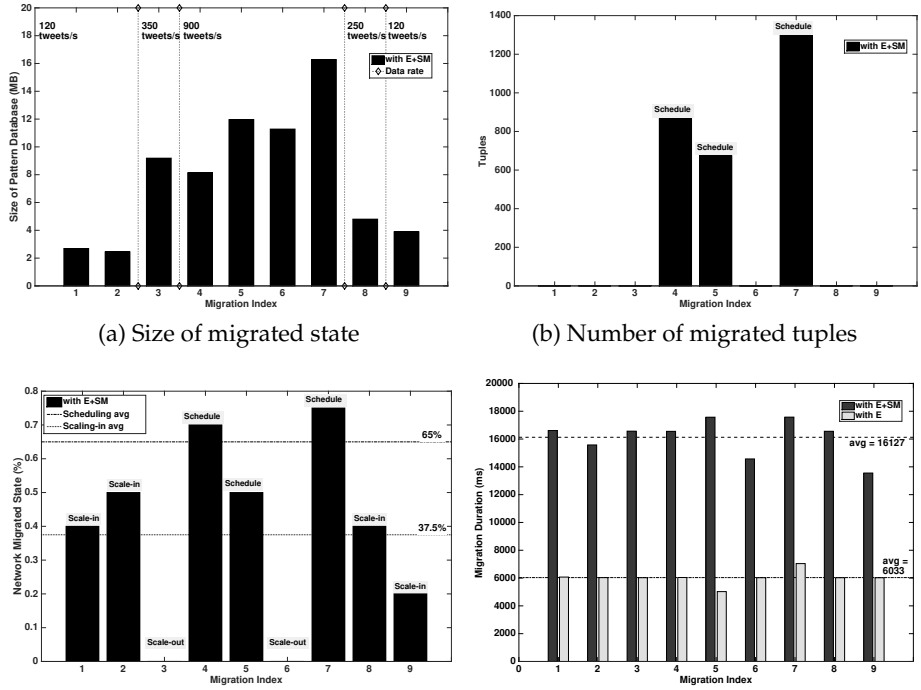
(a) With elasticity and stateful migration.



(b) Without elasticity and stateful migration.

Figure 7.6: Node utilization by executors.

analyze the amount of state transferred using the network. Figures 7.7a and 7.7b illustrate the operator state and the InputBuffer that *a migrating task* saves on the DDS. As expected, Figure 7.7a shows that there is a general tendency which links the size of the migrating operator with the incoming data rate: the higher the load, the larger the saved state. From Figure 7.7b, we can clearly see that the InputBuffers are always empty after the scaling decisions; this happens because the time elapsed between the `rebalance` command, that suspends the spouts activity, and the beginning of the migration is enough to consume the tuples already emitted. Figure 7.7c shows the percentage of saved state that is relocated during each migration. Specifically, after a scheduling decision on average 65% of



(a) Size of migrated state

(b) Number of migrated tuples

(c) Percentage of migrated state on network

(d) Overhead of stateful migration

Figure 7.7: Analysis of stateful migration.

the state is migrated. For the scaling decisions, the placement policy proposed in Section 7.3.3 reduces this percentage and obtains a relocation of only 37.5% and 0% for the scale-in and scale-out decisions, respectively. To analyze the overhead introduced by the stateful migration, we run again the same experiment by enabling the elastic scaling mechanism but disabling the stateful migration. The result is shown in Figure 7.7d, where we compare the time elapsed to perform the stateful migration with the time needed to reassign the executors in Storm (i.e., stateless reassignment). On average the stateful migration introduces an overhead of about 10 s comprising: (1) the time to save the state on the DDS; (2) the time to retrieve and replay the state from the DDS; and (3) the time waited on the synchronization barriers. In this experiment, the third component dominates on the others, while the size of the operators state (from 2 to 16 MB, see Figure 7.7a) does not affect the migration overhead.

7.6 Summary

In this chapter, we have designed and implemented two mechanisms, i.e., elasticity and stateful task migration, that allow Storm to address at runtime the highly dynamic nature of DSP applications. The proposed solution is modular and loosely coupled with the existing Storm architecture as well as transparent and fully reusable by other Storm-based solutions in literature. The experimental results have shown that Storm can elastically increase or decrease the number of operator executors as needed, improving resource utilization of the underlying infrastructure and properly processing the incoming workload.

In recent years, Storm has been widely adopted by research and industrial communities. Therefore, many elasticity policies as well as stateful migration protocols have been proposed in literature. In our work, even tough we have implemented simple proof-of-concept policies, we have designed mechanisms to host also other, more efficient, algorithms. Elastic Storm can represent a useful tool for the DSP-related community; therefore, we have publicly released its source code, which is available as an open source project on GitHub (`http://bit.ly/1oUjZAi`). In the next chapters, we will use Elastic Storm to prototype and evaluate our runtime adaptation solutions.

Chapter 8

Elastic Operator Placement and Replication

In the Big Data era, DSP applications should be capable of seamlessly processing huge amount of data with varying workload patterns, thus requiring to dynamically scale their execution on multiple computing nodes. We investigate and model the problem of adapting at runtime the replication and placement of DSP operators. By explicitly considering reconfiguration costs, the proposed formulation can determine whether the application should be more conveniently redeployed.

In the previous chapters, we have shown the importance of a reference model to evaluate existing deployment heuristics as well as to develop new ones. Moreover, we have investigated the benefits of a joint optimization of replication and placement of DSP operators. From Chapter 7, we started to explore the challenges of adapting at runtime the application deployment and, in this chapter, we show the importance of adaptation costs for running high-performance DSP applications.

DSP applications should be capable of seamlessly processing massive amount of data with varying workload patterns; therefore, they require to dynamically scale their execution on multiple computing nodes. To deal with the fact that some operators in the application can be overloaded and thus become a bottleneck, a commonly adopted optimization consists of scaling-out or scaling-in the number of parallel instances (i.e., replicas) for the operators, so that each replica can process a subset of the incoming data flow in parallel (e.g., [65, 81, 86]). Due to the unpredictable rate at which the sources may produce the data streams, a static or manual configuration of the operator replication degree (which in literature is also referred to as parallelization degree) does not allow to effectively manage changes

that can either occur in the execution environment (e.g., workload fluctuations) or in the QoS requirements of the DSP application (which is typically long-running). Therefore, a key design choice in a DSP framework is to enable it with elastic data parallelism, where the replication degree is self-configured at runtime. As expected, it is more complex to achieve elasticity for stateful operators. Furthermore, since data sources can be geographically distributed, the deployment and execution of DSP applications can also take advantage of the ever increasing presence of distributed Cloud and Fog computing resources, which can reduce latency by moving the computation towards the network edges.

This chapter focuses on the initial deployment and runtime reconfiguration of DSP applications over geo-distributed computing nodes. We propose Elastic DSP Replication and Placement (for short, EDRP), a unified general formulation of the elastic operator replication and placement problem. EDRP takes into account the heterogeneity of infrastructural resources and the QoS application requirements and determines the number of replicas for each operator and where to deploy them on the geo-distributed computing infrastructure. Moreover, EDRP takes into account the reconfiguration costs, which arise when an operator is migrated from one computing resource to another, as well as when a scaling-in/out decision changes the replication degree of an operator. These costs are significant in case of stateful operators, which require the relocation of the internal state on a different node in such a way that no state information is lost. As in most recent research, e.g., [65], we consider partitioned stateful operators, for which the internal state can be decomposed into partitions based on a partitioning key and each partition can be assigned to a distinct replica.

Differently from most works in literature [53, 132, 145, 196], EDRP can jointly determine the application placement and replication of its operators, while optimizing the QoS attributes of the DSP application. At runtime, by modeling the reconfiguration costs of migration and scaling operations, EDRP can be used to determine whether the application should be more conveniently redeployed.

The main research contributions of this chapter are as follows.

- We model the EDRP problem as an Integer Linear Programming (ILP) problem (Sections 8.2, 8.3, and 8.4) which can be used to optimize different QoS metrics. Specifically, in this chapter, we minimize the response time, defined as the critical path delay to traverse the application DAG, and the monetary cost of all the computing and networking resources involved in the processing and transmission of the application data streams. We propose a general formulation of the reconfiguration costs caused by operator migration and scaling in terms of application

downtime. The latter depends on the fact that most DSP frameworks do not support seamlessly reconfigurations, since the ongoing computation is interrupted while the migration and/or scaling operations are being carried out.

- Leveraging on Distributed Storm (presented in Chapter 3), we develop a prototype scheduler named S-EDRP. S-EDRP can replicate and place the DSP application operators according to the EDRP solution (Section 8.5). To this end, we adapt the reconfiguration costs in order to consider the migration protocol used for relocating stateful operators.
- We extensively evaluate our proposal through numerical experiments as well as experiments based on our S-EDRP prototype (Section 8.6). The former aims to investigate the effectiveness of EDRP in taking the reconfiguration decisions under different scenarios of reconfiguration costs. The latter aims to evaluate the prototyped solution in a real setting; to this purpose, we use the DSP application that solves the DEBS 2015 Grand Challenge and that we already used in Chapter 6.

8.1 Related Work

Starting from the literature analysis presented in Chapter 2, in this section, we review the most relevant approaches aiming to pinpoint the research contributions of this chapter.

From Chapter 2, we have seen that the DSP deployment problem has been widely investigated in literature under different modeling assumptions and optimization goals. Therefore, in Chapter 4 and 6, we have proposed two general formulations of the optimal DSP deployment problem, which take into account the heterogeneity of computing and networking resources. The first one determines only the operator placement, whereas the second one optimizes also the replication degree of DSP operators. Differently from existing solutions, our *single-stage* approach jointly optimizes replication and placement of the DSP application operators. The contributions of Chapters 4 and 6 focus only on the initial deployment problem, thus neglecting the DSP systems runtime dynamics and the reconfiguration costs. In this chapter, we further extend our results to explicitly address the challenges of performing an elastic runtime adaptation.

Elasticity is a key feature for today's DSP systems, and many research efforts have proposed efficient policies. Some works, e.g., [31, 71, 80], exploit best-effort threshold-based policies based on some utilization metric. Other works, e.g., [65, 132, 145, 214], use more complex policies to determine the scaling decisions, including latency models based on queuing theory, game-theoretic and control theoretic approaches. These approaches usually do not consider stateful operators (e.g., [132, 167, 205, 214, 219]) or

cannot be easily utilized in geo-distributed environment (e.g., [31, 47, 60, 78, 162]), because they neglect the impact of network latencies while planning reconfigurations. Furthermore, the lack of a reference benchmark does not allow to assess the quality of the proposed heuristics.

Since most DSP frameworks require to restart the application or rely on a pause-and-resume approach, performing a reconfiguration may incur in a significant downtime (e.g., [199]). Therefore, several solutions explicitly consider the adaptation cost while determining the adaptation actions. Specifically, some works only limit the number of deployment changes [48, 47], whereas others predict the resulting performance penalties (e.g., application downtime, latency spikes), with the aim of enacting only the less expensive reconfigurations [78, 95, 219] or to enforce constraints on the adaptation cost [82, 138]. Our solution uses the adaptation cost while determining the optimal reconfiguration strategy. Indeed, by modeling the DSP system, EDRP can determine whether it is more convenient to redeploy the application by evaluating, at the same time, the performance improvement and the cost of adaptation (which results in a short term performance degradation).

The works most closely related to ours have been presented in [78, 81, 138]. Heinze et al. [78, 81] propose a model to estimate latency spikes caused by operator reallocations, and use it to define a placement heuristic. It results a solution that places only the newly added operators and minimizes the latency violations. Our approach differs in that we propose an optimal problem formulation which deals with the reconfiguration costs of the entire DSP application. Moreover, our solution represents a benchmark against which heuristics can be compared. Madsen et al. [138] formulate a MILP optimization problem aimed to control load balancing and horizontal scaling, which works in combination with a heuristic in charge of collocating operators on computing nodes. Similarly to our work, their solution considers operator collocation (in our case, it follows from minimizing the response time) and state-migration overheads. However, we further consider the network among computing nodes, which impacts on placement, replication, and also on migration costs.

Differently from the above cited works that present reactive scaling strategies, De Matteis and Mencagli [48] propose a proactive strategy for a DSP distributed environment that takes into account a limited future time horizon to choose the reconfigurations. Differently from our solution, their approach is not integrated in an existing DSP framework.

DSP Frameworks. Aside the specific functionalities, the most popular open-source DSP frameworks (i.e., Storm, Spark Streaming, Flink, and Heron) use directed graphs to model DSP applications. This is perfectly in line with the model we adopt in EDRP. Some of these frameworks are equipped with elasticity mechanisms in an embryonic stage, because they

dynamically scale the application in a disruptive manner. To support elasticity and stateful migrations, we originally developed in Elastic Storm several runtime mechanisms, which have been presented in Chapter 7. In this chapter, we combine these functionalities with our Distributed Storm to exploit the distributed monitoring capabilities of the latter. We present in Section 8.5 the resulting enhanced framework.

An approach to support elastic scaling of DSP applications in Storm has been also presented at the same time in [123]; interestingly, their proposal reduces the interruption due to scaling operations by keeping the application running while scaling, instead of shutting down the application operators and restarting them. However, they considered a clustered architecture and their improved version of Storm has not been released publicly. Therefore, in our experiments we use the standard rebalancing command of Storm.

As regards the deployment of DSP applications in geo-distributed environments, we rely on Distributed Storm, that has been presented in Chapter 3. However, we observe that our EDRP formulation is general enough and it could be integrated in other DSP frameworks designed to operate in a geo-distributed infrastructure (e.g., Foglets).

DSP systems are also offered as Cloud services (e.g., Google Cloud Dataflow, Amazon Kinesis Streams, Azure Stream Analytics). These services support dynamic scaling of the computing resources; however, it appears that they execute in a single data center, conversely to the geo-distributed environment we investigate in this thesis.

8.2 System Model

In this section, we briefly recall the resource and application model presented in Chapter 6, and introduce the reconfiguration model. We summarize the notation used throughout the chapter in Table 8.1.

8.2.1 Resource Model

Computing and network resources can be represented as a labeled fully connected directed graph $G_{res} = (V_{res}, E_{res})$: the set of nodes V_{res} represents the distributed computing resources, and the set of links E_{res} represents the logical connectivity between nodes. Each node $u \in V_{res}$ is characterized by Res_u, the amount of available resources, and S_u, the processing speed-up on a reference processor. Each link $(u, v) \in E_{res}$, with $u, v \in V_{res}$ is characterized by: $d_{(u,v)}$, the network delay between node u and v; $r_{(u,v)}$, the transfer rate between node u and v; and $C_{(u,v)}$, the cost per unit of data

Table 8.1: Description of main symbols used in the model.

Symbol	Description
G_{dsp}	Graph representing the DSP application
V_{dsp}	Set of vertices (operators) of G_{dsp}
E_{dsp}	Set of edges (streams) of G_{dsp}
C_i	Cost of deploying $i \in V_{dsp}$
R_i	Latency of $i \in V_{dsp}$
Res_i	Required resources to execute $i \in V_{dsp}$
$I_{S,i}$	Internal state size of $i \in V_{dsp}$
$I_{C,i}$	Operator code size of $i \in V_{dsp}$
$\lambda_{(i,j)}$	Avg. tuple rate on $(i,j) \in E_{dsp}$
λ_i	Avg. incoming tuple rate on $i \in V_{dsp}$
G_{res}	Graph representing computing resources
V_{res}	Set of vertices (comp. nodes) of G_{res}
E_{res}	Set of edges (network links) of G_{res}
Res_u	Resources available on $u \in V_{res}$
S_u	Processing speed-up of $u \in V_{res}$
$d_{(u,v)}$	Network delay on $(u,v) \in E_{res}$
$r_{(u,v)}$	Transfer rate on $(u,v) \in E_{res}$
$C_{(u,v)}$	Cost for data unit on $(u,v) \in E_{res}$
$t_{s,u}$	Time for instance spawning on $u \in V_{res}$
t_{syn}	Synchronization time for reconfiguration
$V^i_{res} \subseteq V_{res}$	Subset of nodes where to place $i \in V_{dsp}$
$\mathcal{S} \sqsubset S$	Multiset of elements in set S
$\mathcal{P}(S;k)$	Set of all multisets with elements taken in S and cardinality no greater than k
$\mathcal{U}_{0,i}$	Current deployment for $i \in V_{dsp}$
$x_{i,\mathcal{U}}$	Placement of $i \in V_{dsp}$ on nodes in $\mathcal{U}$
$y_{(i,j),(\mathcal{U},\mathcal{V})}$	Placement of $(i,j) \in E_{dsp}$ on network paths between $\mathcal{U}$ and $\mathcal{V}$

transmitted along the network path between u and v. This model also considers edges of type (u,u) (i.e., loops), as in Section 6.2.1.

8.2.2 DSP Model

To describe a DSP application, also in this chapter we distinguish between a user-defined abstract model and an execution model.

A DSP abstract model can be represented as a labeled directed acyclic graph (DAG) $G_{dsp} = (V_{dsp}, E_{dsp})$, where the nodes in V_{dsp} represent the application operators as well as the data stream sources and sinks, and the links in E_{dsp} represent the streams between nodes. We characterize an operator with the non-functional attributes of a reference implementation on

a reference architecture: Res_i, the amount of resources required for running the operator; R_i, the average operator instance latency (which accounts for the waiting time on the input queues as well as the execution time of a data unit); and C_i, the cost of deploying an operator instance. We characterize the stream exchanged from operator i to j, $(i, j) \in E_{dsp}$, with its average tuple rate $\lambda_{(i,j)}$. To model load-dependent latency, we assume that R_i is a function of the operator input tuple rate λ_i, that is, $R_i = R_i(\lambda_i)$, where $\lambda_i = \sum_{j \in V_{dsp}} \lambda_{(j,i)}$; without loss of generality, we also assume that R_i is an increasing function in λ_i. We assume that Res_i is a scalar value, but our placement model can be easily extended to consider Res_i as a vector of required resources (e.g., CPU cores, memory).

The DSP *execution model* is obtained from the abstract model by replacing each operator with the current number of operator replicas. Since the load can vary over time, the number of replicas in the execution model can change accordingly as to optimize QoS requirements (e.g., response time).

8.2.3 Reconfiguration Model

When a DSP application is launched, an initial number of replicas and their placement is determined based on the current (expected) load and QoS requirements. Then, at runtime, changes of QoS attributes of the application and the execution environment (e.g., load, latency) can call for the application reconfiguration. The latter aims to preserve high application performance, facing the dynamism of the runtime environment. A DSP application can be reconfigured by combining migrations and scaling operations. A *migration* moves an operator replica to a different computing resource, so to optimize resource utilization and, in turn, application performance. A *scaling* operation changes the replication degree of an operator. Specifically, a scale-out decision increases the number of replicas when the operator needs more computing resources, whereas a scale-in decreases the number of replicas when the operator under-uses its resources.

To perform a reconfiguration while preserving the application integrity in terms of extracted information, we assume a simple pause-and-resume approach, as introduced in Chapter 2. Its drawback is the application downtime for the entire duration of the reconfiguration process, which negatively impacts on the application perceived QoS.

Replica Migration. Migrating an operator replica using the pause-and-resume approach involves the following operations. First, the DSP system terminates the replica running on the old location and stops the related upstream operators from emitting data towards the operator under recon-

figuration. Then, the operator *code*[1] is copied from an external repository to the new location, if the latter does not hold it, i.e., if no other replica is currently deployed there. Moreover, if the replica is an instance of a stateful operator, the DSP system has to migrate the replica internal state from the old location to the new one. Finally, the DSP system starts the new replica and resumes the application execution. We consider that the DSP system performs the code and state migrations leveraging on a storage system, named *DataStore*. The DataStore acts as repository for the operators code and allows replicas to save and restore their state during reconfigurations.

Scaling Operation. The complexity of changing at runtime the replication degree of an operator depends on the presence of its internal state. If the operator is stateless, a scaling operation involves only adding or removing replicas. When a new replica is added, the DSP system also determines *where* the replica should be executed; as a consequence of this decision, the operator code may be transferred from the DataStore to the new location. If the operator is stateful, a scaling operation has also to redistribute the operator internal state among its replicas, so to preserve the application integrity. Similarly to most of the existing solutions (e.g., [65, 71, 137, 199, 210]), we model the operator internal state as a set of key-value pairs, where the key identifies the smallest and indivisible state entity. These keys are partitioned and assigned to the operator replicas. After a scaling operation, the keys are redistributed with the help of the DataStore.

8.3 Elastic Operator Replication and Placement Model

In this section, we present the EDRP elastic replication and placement model and derive the expressions of the QoS metrics of interest.

8.3.1 Operator Replication and Placement

The EDRP problem consists in determining, for each operator $i \in V_{dsp}$, the number of replicas and where to deploy them on the computing nodes in V_{res}. EDRP can evaluate the application deployment either periodically or in response to changes on the execution environment (e.g., workload variations), so to execute application with high QoS even in dynamic running conditions. EDRP can identify the optimal reconfiguration strategy and evaluate its convenience in terms of application downtime. Figure 8.1 represents a simple instance of the problem. Similarly to Chapter 6, we represent the operator replication and placement through multisets, be-

[1]We use the term *code* to refer to the entity that encapsulates the execution logic of an operator. Depending on the specific system implementation, it can correspond, e.g., to Java classes, executables, or container/virtual machine images.

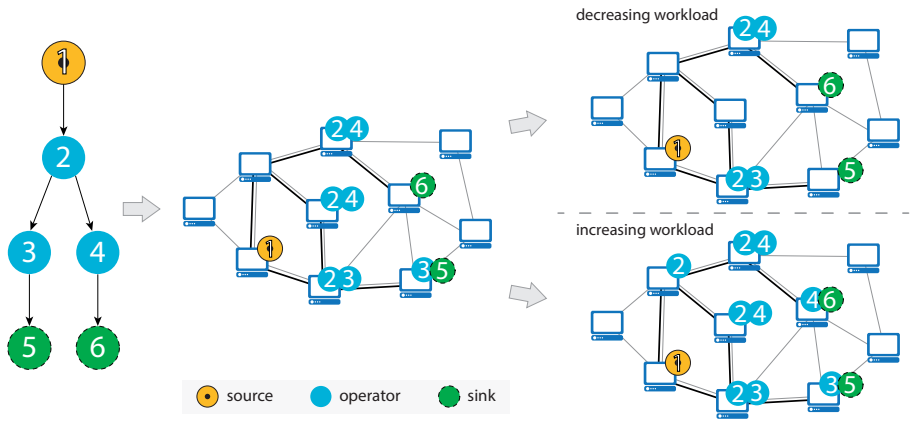


Figure 8.1: Elastic replication of the application operators and their placement. The right part of the figure shows: (top) scale-in of operators 2, 3 and 4 due to load decrease, (bottom) scale-out of operators 2 and 4 due to load increase. The figure does not differentiate between stateful or stateless operators.

cause a deployment can place multiple replicas of the same operator on the same computing node. A multiset $\mathcal{S}$ over a set S, which we denote as $\mathcal{S} \sqsubset S$, is defined as a mapping $\mathcal{S} : S \to \mathbb{N}$ where, for $s \in S$, $\mathcal{S}(s)$ denotes the multiplicity of s in $\mathcal{S}$. Hence, $s \in \mathcal{S}$ if and only if $\mathcal{S}(s) \geq 1$. The cardinality of a multiset $\mathcal{S}$, denoted $|\mathcal{S}|$, is defined by the number of elements in $\mathcal{S}$, that is $|\mathcal{S}| = \sum_{s \in \mathcal{S}} \mathcal{S}(s)$. Hereafter, without lack of generality, we assume that in a deployment each operator $i \in V_{dsp}$ can be replicated at most k_i times. We also define the power multiset $\mathcal{P}(S)$ of a set S as the set of all multisets with elements taken from S and the subset $\mathcal{P}(S;k) \subset \mathcal{P}(S)$ of the multiset over S with cardinality no greater than k, that is $\mathcal{P}(S;k) = \{\mathcal{S} \in \mathcal{P}(S) | \sum_{s \in \mathcal{S}} \mathcal{S}(s) \leq k\}$.

We model the EDRP problem with binary variables $x_{i,\mathcal{U}}$, $i \in V_{dsp}$ and $\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)$: $x_{i,\mathcal{U}} = 1$ if and only if i is replicated in $|\mathcal{U}|$ instances with exactly $\mathcal{U}(u)$ copies deployed in u, with $u \in \mathcal{U}$. We also find convenient to consider binary variables associated to links, namely $y_{(i,j),(\mathcal{U},\mathcal{V})}$, with $(i,j) \in E_{dsp}$, $\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)$, and $\mathcal{V} \in \mathcal{P}(V^j_{res}; k_j)$, which denotes whether the data stream flowing from operator i to operator j traverses the network paths from nodes in $\mathcal{U}$ to nodes in $\mathcal{V}$. By definition, we have $y_{(i,j),(\mathcal{U},\mathcal{V})} = x_{i,\mathcal{U}} \wedge x_{j,\mathcal{V}}$. For short, in the following we denote by $\boldsymbol{x}$ and $\boldsymbol{y}$ the placement vectors for nodes and edges, respectively, where $\boldsymbol{x} = \langle x_{i,\mathcal{U}} \rangle$, $\forall i \in V_{dsp}, \forall \mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)$, and $\boldsymbol{y} = \langle y_{(i,j),(\mathcal{U},\mathcal{V})} \rangle$, $\forall x_{i,\mathcal{U}}, x_{j,\mathcal{V}} \in \boldsymbol{x}$.

8.3.2 QoS Metrics

We consider the application response time and deployment cost as QoS metrics of interest. Nevertheless, the analysis can be easily extended to include other QoS metrics, as application availability and network related metrics (as shown in Chapters 4 and 6).

Since DSP applications are usually employed in latency-sensitive domains (e.g., [99]), a desired placement should minimize the response time resulting from the operators deployment. However, if on the one hand a higher replication degree leads to lower response time, on the other hand it incurs in higher resource wastage, which leads to a higher deployment cost. We first formulate these metrics for operators and streams in isolation, then derive the expressions for a DSP application.

Operator QoS Metrics.

For each $i \in V_{dsp}$, the QoS of the operator depends on the hosting resources, that is its deployment $\mathcal{U}$. Let $R_{i,\mathcal{U}}$ and $C_{i,\mathcal{U}}$ denote the maximum latency and the cost experienced when i runs on $\mathcal{U}$, respectively. We readily have:

$$R_{i,\mathcal{U}} = \max_{u \in \mathcal{U}} \frac{R_i\left(\frac{\lambda_i}{|\mathcal{U}|}\right)}{S_u} \tag{8.1}$$

$$C_{i,\mathcal{U}} = \sum_{u \in \mathcal{U}} \mathcal{U}(u) C_i Res_i \tag{8.2}$$

under the assumption that the traffic is equally split among the different operator replicas. Good load balancing can be achieved even for stateful operators relying on the stream partitioning schemes mentioned in Chapter 2 (e.g., [65, 150]). It is worth pointing out that here we do not make any particular assumption on the actual expression of R_i. We can adopt either a queuing model closed-form expression, e.g., the response time of a M/M/1 queue (as in Section 8.6.2), or an experimental characterization of the operator response time as function of the load (as in Section 8.6.3).

Stream QoS Attributes.

For a stream $(i,j) \in E_{dsp}$, the QoS depends on the upstream and downstream operators deployments $\mathcal{U}$ and $\mathcal{V}$. Let $d_{(i,j),(\mathcal{U},\mathcal{V})}$ and $C_{(i,j),(\mathcal{U},\mathcal{V})}$ denote the maximum latency and the cost experienced when i runs on $\mathcal{U}$ and j on $\mathcal{V}$, respectively. We readily have:

$$d_{(i,j),(\mathcal{U},\mathcal{V})} = d_{(\mathcal{U},\mathcal{V})} = \max_{u \in \mathcal{U}, v \in \mathcal{V}} d_{(u,v)} \tag{8.3}$$

$$C_{(i,j),(\mathcal{U},\mathcal{V})} = \sum_{u \in \mathcal{U}, v \in \mathcal{V}} \lambda_{(i,j),(\mathcal{U},\mathcal{V})} C_{u,v} \tag{8.4}$$

where $\lambda_{(i,j),(\mathcal{U},\mathcal{V})} = \frac{\lambda_{(i,j)}}{|\mathcal{U}||\mathcal{V}|} \quad u \in \mathcal{U}, v \in \mathcal{V}$ is the amount of stream (i,j) traffic exchanged between two replicas for the deployments $\mathcal{U}$ and $\mathcal{V}$.

DSP Application QoS Metrics.

We derive the following expressions for a DSP application.

Response Time: For any placement vector $\boldsymbol{x}$ (and resulting $\boldsymbol{y}$), we consider as response time $R(\boldsymbol{x},\boldsymbol{y})$ the critical path average delay (we refer the reader to Section 4.3 for the definition of critical path delay). Formally, we have:

$$R(\boldsymbol{x},\boldsymbol{y}) = \max_{\pi \in \Pi_{dsp}} R_{\pi}(\boldsymbol{x},\boldsymbol{y}) \tag{8.5}$$

where $R_{\pi}(\boldsymbol{x},\boldsymbol{y})$ is the end-to-end delay along path π and Π_{dsp} the set of all source-sink paths in G_{dsp}. For any path $\pi = (i_1, i_2, \ldots, i_{n_\pi}) \in \Pi_{dsp}$, where i_p and n_π denote the p^{th} operator and the number of operators in the path π, respectively, we obtain:

$$R_{\pi}(\boldsymbol{x},\boldsymbol{y}) = \sum_{p=1}^{n_\pi} R_{i_p}(\boldsymbol{x}) + \sum_{p=1}^{n_\pi - 1} D_{(i_p,i_{p+1})}(\boldsymbol{y}) \tag{8.6}$$

where for any $i \in V_{dsp}$ and $(i,j) \in E_{dsp}$

$$\begin{aligned} R_i(\boldsymbol{x}) &= \sum_{\mathcal{U} \in \mathcal{P}(V_{res}^i;k_i)} R_{i,\mathcal{U}} x_{i,\mathcal{U}} \\ D_{(i,j)}(\boldsymbol{y}) &= \sum_{\substack{\mathcal{U} \in \mathcal{P}(V_{res}^i;k_i) \\ \mathcal{V} \in \mathcal{P}(V_{res}^j;k_j)}} d_{(\mathcal{U},\mathcal{V})} y_{(i,j),(\mathcal{U},\mathcal{V})} \end{aligned} \tag{8.7}$$

denote respectively the execution time of operator i when deployed on $\mathcal{U}$ and the network delay for transferring data from i to j when the two operators are mapped on $\mathcal{U}$ and $\mathcal{V}$, respectively.

Cost: For any placement vector $\boldsymbol{x}$ (and resulting $\boldsymbol{y}$), we consider the cost $C(\boldsymbol{x},\boldsymbol{y})$ of all the resources and links involved in the processing and transmission of the application data streams. We have:

$$C(\boldsymbol{x},\boldsymbol{y}) = \sum_{i \in V_{dsp}} C_i(\boldsymbol{x}) + \sum_{(i,j) \in E_{dsp}} C_{(i,j)}(\boldsymbol{y}) \tag{8.8}$$

where

$$\begin{aligned} C_i(\boldsymbol{x}) &= \sum_{\mathcal{U} \in \mathcal{P}(V_{res}^i;k_i)} C_{i,\mathcal{U}} x_{i,\mathcal{U}} \\ C_{(i,j)}(\boldsymbol{y}) &= \sum_{\substack{\mathcal{U} \in \mathcal{P}(V_{res}^i;k_i) \\ \mathcal{V} \in \mathcal{P}(V_{res}^j;k_j)}} C_{(i,j),(\mathcal{U},\mathcal{V})} y_{(i,j),(\mathcal{U},\mathcal{V})} \end{aligned} \tag{8.9}$$

denote the cost of deploying i on $\mathcal{U}$ and the cost of transferring data from i to j when the two operators are mapped on $\mathcal{U}$ and $\mathcal{V}$, respectively.

Reconfiguration-related Metric.

The management operations that change the application deployment temporarily degrade the application performance introducing a downtime.

Application downtime: We express the cost of reconfigurations in terms of application downtime, so to capture the trade-offs between benefits and costs of changing the deployment. Under the assumption that different operators can be reconfigured in parallel, the overall application downtime $T_D(\boldsymbol{x})$ corresponds to the longest operator downtime:

$$T_D(\boldsymbol{x}) = \max_{i \in V_{dsp}} T_{D,i}(\boldsymbol{x}) \tag{8.10}$$

where $T_{D,i}(\boldsymbol{x})$ is the downtime duration for operator $i \in V_{dsp}$, which depends on its new deployment configuration and can be expressed as:

$$T_{D,i}(\boldsymbol{x}) = \sum_{\mathcal{V} \in \mathcal{P}(V^i_{res}; k_i)} t_D(i, \mathcal{U}_{0,i}, \mathcal{V})\, x_{i,\mathcal{V}} \tag{8.11}$$

where $t_D(i, \mathcal{U}, \mathcal{V})$ represents the time needed to reconfigure the operator i from the deployment $\mathcal{U}$ to $\mathcal{V}$, and $\mathcal{U}_{0,i}$ represents the current deployment of i. The expression for $t_D(i, \mathcal{U}, \mathcal{V})$ depends on the actual type of operator reconfiguration (i.e., no reconfiguration, operator migration, and operator scaling). For the sake of readability, the derivation of their expressions is postponed to Appendix 8.A.1.

8.4 EDRP Optimization Problem

In this section we present the EDRP optimization problem. We assume that, at each reconfiguration, the system goal is to optimize a suitable objective function which, depending on the scenario, could be aimed at optimizing different, possibly conflicting, QoS attributes. We further assume that each QoS metric must not exceed an application dependent worst case behavior. We use the SAW technique to define the objective cost function $F(\boldsymbol{x}, \boldsymbol{y})$ as a weighted sum of the normalized QoS attributes of the application:

$$F(\boldsymbol{x}, \boldsymbol{y}) = w_r \frac{R(\boldsymbol{x}, \boldsymbol{y})}{R_{max}} + w_c \frac{C(\boldsymbol{x}, \boldsymbol{y})}{C_{max}} + w_d \frac{T_D(\boldsymbol{x})}{T_{D,max}} \tag{8.12}$$

where $w_r,\ w_c,\ w_d \geq 0$, $w_r + w_c + w_d = 1$, are weights associated to the different QoS attributes. $R_{\max}$, $C_{\max}$, and $T_{D,\max}$ are user-defined parameters

which denote, respectively, the worst case bound on the expected response time, cost, and reconfiguration downtime. After normalization, each metric ranges in the interval $[0, 1]$, where the value 1 corresponds to the worst possible value.

We formulate the EDRP problem as an ILP minimization problem as follows:

$$\min_{\boldsymbol{x},\boldsymbol{y},r,t_D} F'(\boldsymbol{x},\boldsymbol{y},r,t_D)$$

subject to:

$$r \geq R_\pi(\boldsymbol{x},\boldsymbol{y}) \qquad \forall \pi \in \Pi_{dsp} \tag{8.13}$$

$$t_D \geq T_{D,i}(\boldsymbol{x}) \qquad \forall i \in V_{dsp} \tag{8.14}$$

$$R(\boldsymbol{x},\boldsymbol{y}) \leq R_{max} \tag{8.15}$$

$$C(\boldsymbol{x},\boldsymbol{y}) \leq C_{max} \tag{8.16}$$

$$T_D(\boldsymbol{x}) \leq T_{D,max} \tag{8.17}$$

$$Res_u \geq \sum_{\substack{i \in V_{dsp} \\ \mathcal{U} \in \mathcal{P}(V_{res}^i)}} \mathcal{U}(u) Res_i x_{i,\mathcal{U}} \qquad \forall u \in V_{res} \tag{8.18}$$

$$1 = \sum_{\mathcal{U} \in \mathcal{P}(V_{res}^i; k_i)} x_{i,\mathcal{U}} \qquad \forall i \in V_{dsp} \tag{8.19}$$

$$x_{i,\mathcal{U}} = \sum_{\mathcal{V} \in \mathcal{P}(V_{res}^j; k_j)} y_{(i,j),(\mathcal{U},\mathcal{V})} \qquad \begin{matrix} \forall (i,j) \in E_{dsp}, \\ \mathcal{U} \in \mathcal{P}(V_{res}^i; k_i) \end{matrix} \tag{8.20}$$

$$x_{j,\mathcal{V}} = \sum_{\mathcal{U} \in \mathcal{P}(V_{res}^i; k_u)} y_{(i,j),(\mathcal{U},\mathcal{V})} \qquad \begin{matrix} \forall (i,j) \in E_{dsp}, \\ \mathcal{V} \in \mathcal{P}(V_{res}^j; k_j) \end{matrix} \tag{8.21}$$

$$x_{i,\mathcal{U}} \in \{0,1\} \qquad \begin{matrix} \forall i \in V_{dsp}, \\ \mathcal{U} \in \mathcal{P}(V_{res}^i; k_i) \end{matrix} \tag{8.22}$$

$$y_{(i,j),(\mathcal{U},\mathcal{V})} \in \{0,1\} \qquad \begin{matrix} \forall (i,j) \in E_{dsp}, \\ \mathcal{U} \in \mathcal{P}(V_{res}^i; k_i) \\ \mathcal{V} \in \mathcal{P}(V_{res}^j; k_j) \end{matrix} \tag{8.23}$$

In this formulation we use the linear objective function $F'(\boldsymbol{x},\boldsymbol{y},r,t_D)$, obtained from $F(\boldsymbol{x},\boldsymbol{y})$ by replacing $R(\boldsymbol{x},\boldsymbol{y})$ and $T_D(\boldsymbol{x})$ with the auxiliary variables r and t_D, respectively. Observe that, while F is nonlinear in $\boldsymbol{x}$ and $\boldsymbol{y}$, since $R(\boldsymbol{x},\boldsymbol{y}) = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x},\boldsymbol{y})$ and $T_D(\boldsymbol{x}) = \max_{i \in V_{dsp}} T_{D,i}(\boldsymbol{x})$ are nonlinear terms, F' is linear in r and t_D as well as in $\boldsymbol{x}$ and $\boldsymbol{y}$. Equation (8.13) follows from (8.5)–(8.6). Since r must be larger or equal than the response time of any path and, at the optimum, r is minimized, we have that $r = \max_{\pi \in \Pi_{dsp}} R_\pi(\boldsymbol{x},\boldsymbol{y}) = R(\boldsymbol{x},\boldsymbol{y})$. Similarly, Equation (8.14) defines t_D that, at the optimum, is $t_D = \max_{i \in V_{dsp}} T_{D,i}(\boldsymbol{x})$. Constraints (8.15)–(8.17) are the worst case bounds on the QoS metrics. The constraint (8.18) limits the placement of operators on a node $u \in V_{res}$ according to its available resources. Equation (8.19) guarantees that each operator $i \in V_{dsp}$ is placed on

one and only one node $u \in V_{res}^i$. Finally, constraints (8.20)–(8.21) model the logical AND between the placement variables, that is, $y_{(i,j),(u,v)} = x_{i,u} \wedge x_{j,v}$.

Theorem 8.1. *The EDRP problem is NP-hard.*

Proof. It suffices to observe that the EDRP problem is a generalization of the DSP Placement problem, presented in Chapter 4, which has been shown to be NP-hard. □

8.5 Storm Integration: S-EDRP

To enable the usage of EDRP in a real DSP framework, we have developed a prototype scheduler for Apache Storm, named S-EDRP.

8.5.1 Elasticity in Storm

To elastically adapt the applications deployment in Storm, we resort on Distributed Storm, which is our extension of Storm with QoS-aware scheduling capabilities (see Chapter 3), and on S-EDRP, which is a new custom scheduler for Storm we propose in this work. Distributed Storm enhances the official Storm architecture by introducing key components that support the execution of the MAPE control loop. In our case, the MAPE loop controls the runtime reconfiguration of the application deployment in response to changes in the workload conditions. As shown in Figure 8.2, we have designed a partially decentralized control loop, that follows the master-worker pattern presented in [208]. Specifically, we have centralized the Analyze and Plan phases on Nimbus, and decentralized the Monitor and Execute phases on Nimbus and on the worker nodes, which contribute with local components. The Analyze component collects the monitored data from the decentralized Monitors on the worker nodes and periodically triggers the Plan component. The latter solves the EDRP model, thus determining if needed the reconfiguration actions (i.e., scale-in, scale-out, migrate) that improve application performance. In such a case, the decentralized Execute components perform the corresponding adaptation actions.

Monitor. The first phase of the control loop enriches S-EDRP with infrastructure and application QoS-awareness. Specifically, the monitoring components of Distributed Storm running on each worker node provide to Nimbus inter-node information, such as network delay and exchanged data rate. Further details on our extension can be found in Chapter 3.

Analyze and Plan. In S-EDRP, the Analyze phase collects the monitored data and periodically instantiates the EDRP model. The latter has been properly adjusted, as described in Section 8.5.3, to represent the DSP and resource models used by Storm. The Analyze phase instantiates the

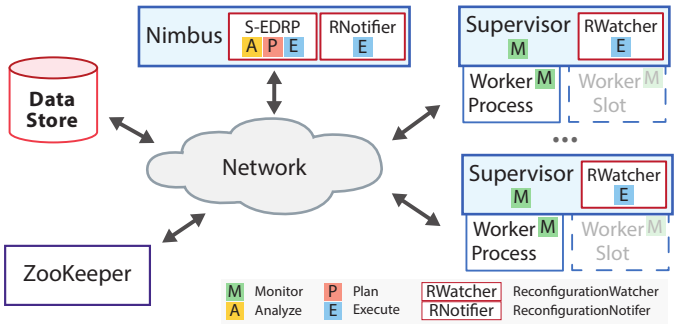


Figure 8.2: Extended Storm architecture.

EDRP model using the collected monitored data to parametrize nodes and edges in G_{dsp} and G_{res} and give value to the average data rate exchanged between communicating operator replicas, $\lambda_{(i,j)}$, and the network latencies, $d_{(u,v)}$. When the Plan phase is activated, it resolves the EDRP model relying on CPLEX©. Observe that, to smoothly integrate our solution with the existing Storm architecture, the Analyze phase uses a time-based triggering strategy to activate the Plan phase. However, more efficient strategies can be designed to recompute the deployment only when specific events occur, e.g., when QoS bounds are violated.

Execute. When EDRP devises a new application deployment, S-EDRP needs to accordingly reconfigure the replication and placement of the application operators. Storm provides an API to change the operators replication at runtime (i.e., `rebalance`); nevertheless, this operation restarts the application topology, compromising the integrity of stateful operators (which lose their internal state). To overcome this issue, we further extend the Storm architecture to include mechanisms that support the migration and replication of stateful operators. In Section 8.5.2, we describe the newly introduced components and the adopted stateful migration protocol.

Besides the runtime adaptation, Nimbus uses S-EDRP to determine the deployment of new applications as well. In such a case, since information on the exchanged data rate is not yet available, S-EDRP defines an early assignment and monitors the application execution to harvest the needed information. Then, at the first MAPE execution, S-EDRP can reassign the application by solving the updated EDRP model and neglecting reconfiguration costs (i.e., $w_d = 0$).

8.5.2 Stateful Migrations in Storm

To perform stateful migrations, we integrate in Distributed Storm the features of Elastic Storm, presented in Chapter 7. The resulting Distributed Storm mainly differs from Elastic Storm for the presence of a centralized

DataStore (instead of a decentralized one). For sake of clarity, in this section we describe the stateful migration features of our extension.

In Storm, a task represents the smallest entity that handles a partition of the operator state. Therefore, the key idea is to introduce, within the Storm architecture, new components that perform reconfigurations by following the pause-and-resume approach. To this end, a task is enhanced with the ability to be paused and resumed, and to export and import the managed operator state. The newly introduced components, represented in red in Figure 8.2, are DataStore, ReconfigurationNotifier, and ReconfigurationWatcher. The *DataStore* provides a staging area for the operator state during reconfigurations. It is a key-value storage component implemented using Redis; we could not rely on ZooKeeper, because it is not designed to hold large data values. The *ReconfigurationNotifier* is a centralized component that we added on Nimbus in order to coordinate the reconfiguration actions, in such a way to apply them while preserving the operators integrity (i.e., saving and restoring their internal state) during adaptations. This component notifies tasks when a reconfiguration is going to be performed, so that they can export their state, wait the reconfiguration, and finally import again their state. On each worker node, the *ReconfigurationWatcher* handles these notifications; it is a watchdog component that pauses and resumes the execution of the application tasks and executes the migration protocol.

To interact with the newly introduced components, we rely on the *StatefulSpout* and *StatefulBolt* classes, described in Section 7.4.2.

Stateful Migration Protocol. When the number of executors or the placement for an operator is changed, the ReconfigurationNotifier publishes a reconfiguration message on ZooKeeper to instruct the involved worked nodes. As a consequence, on each worker node the ReconfigurationWatcher activates the migration mode for the application tasks, which, in turn, run the stateful migration protocol. The latter uses the pause-and-resume approach for reconfiguring the stateful operator. We refer the reader to Section 7.4.3 for further details on the stateful migration protocol.

8.5.3 S-EDRP: EDRP in Storm

To conclude this section, we present how the EDRP model has been adjusted to represent the abstractions of Storm. We first describe the representation of applications and resources and then the modeling of the stateful migration protocol developed in Storm.

DSP and Resource Model of Storm. We have to model the fact that Storm runs multiple executors to replicate an operator, and that a Storm scheduler deploys these executors on the available worker slots, considering that at most $EPS_{\max}$ executors can be co-located on the same slot.

Hence, S-EDRP defines $G_{dsp} = (V_{dsp}, E_{dsp})$, with V_{dsp} as the set of operators and E_{dsp} as the set of streams exchanged among them. Since in Storm an operator is considered as a black box element, we conveniently assume that its attributes are unitary, i.e., $C_i = 1$ and $Res_i = 1$, $\forall i \in V_{dsp}$. By solving the replication and placement model, S-EDRP determines the number of executors for each operator $i \in V_{dsp}$, leveraging the cardinality of $\mathcal{U}$ when $x_{i,\mathcal{U}} = 1$, with $\mathcal{U} \in \mathcal{P}(V^i_{res}; k_i)$ and k_i equals to the number of tasks of i. The resource model $G_{res} = (V_{res}, E_{res})$ must take into account that a worker node $u \in V_{res}$ offers some worker slots $WS(u)$, and each worker slot can host at most $EPS_{\max}$ executors. For simplicity, S-EDRP considers the amount of available resources C_u on a worker node u to be equal to the maximum number of executors it can host, i.e., $C_u = WS(u) \times EPS_{\max}$. To enable the parallel execution of executors, C_u should be equal (or proportional) to the number of CPU cores available on u.

Modeling the Stateful Migration Protocol. Differently from the generic formulation described in Section 8.3.2, to model the downtime induced by our the stateful migration protocol we need to account that, in Storm, the entire application is restarted when a configuration is carried out. Therefore, to preserve the application integrity, even the replicas of the operators not directly involved in a reconfiguration need to save and restore the internal state on their local swapping area. This significantly impacts the application downtime during reconfiguration. For the sake of readability, we postpone the new downtime expression derivation to Appendix 8.A.2.

8.6 Experimental Results

In this section, we evaluate EDRP through two different sets of experiments. First, we analyze the elasticity capabilities of the EDRP model through numerical investigations. Then, we evaluate the elasticity mechanisms introduced in Storm by our extension and the QoS achieved by the S-EDRP scheduler. In both cases, we compare the results achieved using EDRP with a baseline solution that does not consider reconfiguration costs (it suffices to set $w_d = 0$ to ignore the downtime effects). Observe that this baseline EDRP configuration corresponds to ODRP.

8.6.1 Reference DSP Application

As a test-case application, we consider the prototype application that solves the first query of *DEBS 2015 Grand Challenge*, presented in Section 6.6.1. By processing data streams originated from the New York City taxis, the goal of the query is to find the top-10 most frequent routes during the last 30 minutes. Differently from the previous setting, in these experiments we

Table 8.2: Parameters of the experimental setup.

Service rate expressed in tuples per second (tps) and internal state size in bytes per operator

Operator	μ_i (tps)	**State** (B)	**Operator**	μ_i (tps)	**State** (B)
data source	284	82	metronome	300	328
parser	233	-	countByWindow	335	1376
filterByCoordinates	253	-	partialRank	2371	1536
computeRouteID	253	-	globalRank	3000	480

Default values used for metrics computation

Parameter	**Value**	**Parameter**	**Value**
C_i	1 \$/h	$r_{(u,DS)}, r_{(DS,u)}$	100 Mbps
$C_{(u,v)}$	0.02 \$/tps	$I_{C,i}$	300 KB
$t_{s,u}$	500 ms	t_{syn}	250 ms

define that each operator can be replicated at most three times (i.e., $k_i = 3, \forall i$), except for the pinned ones (i.e., *data source* and *globalRank*) and the *metronome*, which cannot be easily parallelized. If not otherwise specified, we use the configuration parameters reported in Table 8.2.

8.6.2 Evaluation of EDRP Model

This set of experiments evaluates the behavior of the EDRP optimization model for the reference DSP application through numerical investigation. We have implemented EDRP in CPLEX© (version 12.6.3) and we have executed the experiments on an Amazon EC2 virtual machine (c4.xlarge with 4 vCPU and 7.5 GB RAM). In these experiments, G_{res} models a portion of ANSNET, a geographically distributed network where 15 computing nodes are interconnected with non-negligible network delays (whose average is 32 ms). Within this network, we assume that a logical link $(u, v) \in E_{res}$ between any two computing resources $u, v \in V_{res}$ always exists; each logic link results by the underlying physical network paths and a shortest-path routing strategy. We assume that each computing node u has $Res_u = 2$ available resources and provides a unitary $S_u = 1$ processing speed-up. Additionally, in this scenario, we assume that each operator can be replicated at most twice (i.e., $k_i = 2, \forall i \in V_{dsp}$), except for the pinned ones (i.e., *data source* and *globalRank*) and the *metronome*, which cannot be easily parallelized. EDRP estimates the response time R_i for operator i subject to the incoming load $\lambda_i/|\mathcal{U}|$ by modeling the underlying computing node as an M/M/1 queue, i.e., $R_i(\lambda_i/|\mathcal{U}|) = (\mu_i - \lambda_i/|\mathcal{U}|)^{-1}$, where μ_i is the service rate of i measured on a reference processor. The operators service rate has been obtained through preliminary experiments and it is shown in Table 8.2. We assume that the network latency to and from DataStore is constant and set to 5 ms, and the data transfer rate between all the pairs of nodes is 100 Mbps. We consider a 1 hour experiment during which the *data source* emission rate linearly increases from 70 to 220 tuples per sec-

ond, during the first half of the experiment, and then decreases down to 70 tuples per second, during the second half. We solve EDRP periodically, every minute, possibly reconfiguring the application deployment. The initial placement is computed by EDRP as well, ignoring the downtime-related metrics, with a balanced objective function having $w_r = w_c = 0.5$. Considering the execution environment, we impose the following bounds on the QoS metrics: $R_{max} = 175$ ms, $C_{max} = 30$ \$/h, and $T_{D,max} = 7.5$ s.

We consider different objective function configurations by changing the weights w_X, $X \in \{r, c, d\}$, which translate to different QoS optimization goals into the model. We first consider the case whereby we do not account for reconfiguration costs, i.e., $w_d = 0$, and assign equal weights to response time and cost, i.e., $w_r = w_c = 0.5$. Then, we consider a balanced scenario with $w_d = 0.4$ and $w_r = w_c = 0.3$. Finally, we consider configurations which focus on either response time or deployment cost: $w_r = 0.6$, $w_d = 0.4$, and $w_c = 0.6$, $w_d = 0.4$.

Results. We first study the DSP application behavior when we do not consider reconfiguration cost, with $w_r = w_c = 0.5$ and $w_d = 0$. In the initial deployment at $t = 0$ minutes of simulated time (min), each component gets exactly one instance. As shown in Figure 8.3a, as load starts growing, EDRP repeatedly migrates components to exploit better operator placement configurations. At $t = 7$ min, a second instance of *partialRank* is eventually launched. Then, at $t = 24$ min, the model scales-out at once other components: *parser*, *filterByCoordinates*, and *computeRouteID*. A third scale-out is performed when the input rate reaches the peak, and *countBy-Window* is replicated as well. In the second half of the experiment, EDRP progressively comes back to the initial configuration. Table 8.3 summarizes the experiment results, reporting the mean response time and cost, and total downtime. Observe that, although the application achieves good performance, it suffers appreciably longer downtime than other scenarios (from 240% to 500% higher values).

We now turn our attention to the scenarios where EDRP accounts for the reconfiguration costs. In the following experiments we set $w_d = 0.4$ and vary the other weights. Figure 8.3b illustrates the results for the balanced scenario with $w_r = w_c = 0.3$ and $w_d = 0.4$. Compared to the previous case, here we have only three reconfigurations overall. First, as the input rate grows, after 8 min EDRP launches a second replica for *partialRank*. This new instance is placed on the same node where *partialRank* is already running along with *countByWindow*, which minimizes the amount of state moved across the network. The latter operator is concurrently migrated to a different node. Then, at $t = 27$ min, since response time is approaching the user-defined threshold, EDRP scales-out the *parser*, *filterByCoordinates*, and *computeRouteID* operators, while also migrating others. some components. When the load decreases, EDRP is quite conservative and does not

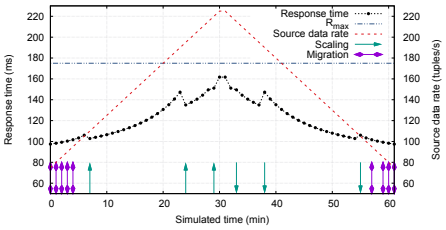
Table 8.3: Mean response time and cost, and total downtime with different EDRP objective configurations.

Objective	Resp. time	Cost	Downtime
Baseline ($w_d = 0$, $w_c = w_r = 0.5$)	120.8 ms	9.6 \$/h	12.0 s
Balanced ($w_d = 0.4$, $w_c = w_r = 0.3$)	121.7 ms	9.9 \$/h	2.4 s
Response time ($w_d = 0.4$, $w_r = 0.6$)	121.5 ms	10.3 \$/h	2.4 s
Cost ($w_d = 0.4$, $w_c = 0.6$)	136.0 ms	9.2 \$/h	4.9 s

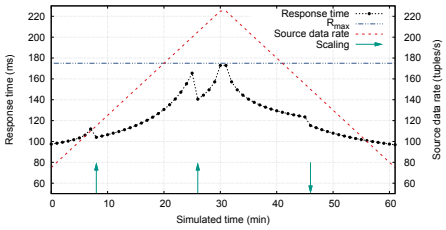
immediately scale-in operators. After 46 min, the model terminates all the replicas launched at $t = 27$ min. So all the operators are left with just one replica, except for *partialRank* because, according to the model, terminating this extra replica is not worth more downtime. We observe that the scale-in operation also results in a reduction of the application latency. This can be explained by observing that, along with the scale-in, EDRP takes advantage from co-locating communicating operators on the same node, which reduces network latency and, in turn, overall application latency.

If we focus on response time optimization, setting $w_r = 0.6$ and $w_d = 0.4$, we get results which are similar to the previous balanced objective scenario with better latency but higher cost. As shown in Figure 8.3c, EDRP scales-out operators in two phases in the first part, and performs a single scale-in after 50 min. Finally, we consider the deployment cost minimization, setting $w_c = 0.6$ and $w_d = 0.4$. As expected, we experience lower cost and higher response time with respect to the previous cases. As shown in Figure 8.3d, EDRP scales-out *partialRank* at $t = 8$ min, similarly to other settings. However, it delays other scaling operations until it is forced to use more replicas to satisfy the latency bound; EDRP progressively launches new instances from $t = 26$ min to $t = 30$ min. Nevertheless, consistently with the cost minimization objective, EDRP immediately terminates those replicas as input rate decreases. Finally, at $t = 53$ min, EDRP scales-in *partialRank*, leaving each operator with a single running instance.

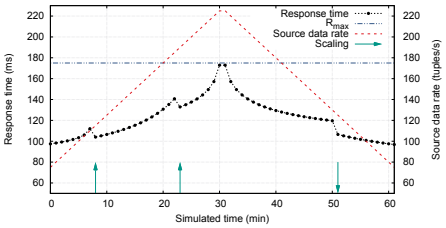
Scalability. For any given topology and number of computing nodes, the EDRP optimization problem complexity depends on the maximum degree of operator replication k_i, which impacts the number of operator configuration alternatives, that is the cardinality of the sets $\mathcal{P}(V^i_{res}; k_i)$ and thus the number of operator and link variables, $x_{i,\mathcal{U}}$ and $y_{(i,j),(\mathcal{U},\mathcal{V})}$, respectively. We computed the number of variables and measured the EDRP resolution time for our reference DSP application for an increasing number of computing nodes and maximum replication degree k. The results are shown in Table 8.4 (at the left of the slash sign /). We can observe that the number of variables and resolution time rapidly grow even for $k = 2$ and small number of nodes and it is not possible even to generate the entire instance for $k = 3$ but for the smallest network size. This is due to the huge number of



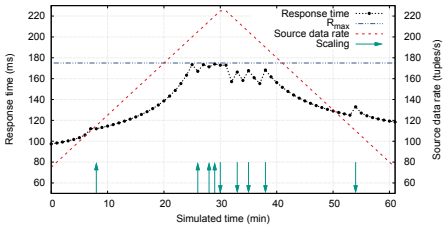
(a) Baseline ($w_r = 0.5$, $w_c = 0.5$, $w_d = 0$): minimization of response time and cost, with no reconfiguration costs.



(b) Balanced ($w_r = 0.3$, $w_c = 0.3$, $w_d = 0.4$): minimization of response time and cost, with reconfiguration costs.



(c) Response time ($w_r = 0.6$, $w_d = 0.4$, $w_c = 0$): minimization of response time, with reconfiguration costs.



(d) Cost ($w_c = 0.6$, $w_d = 0.4$, $w_r = 0$): minimization of deployment cost, with reconfiguration costs.

Figure 8.3: EDRP model: application response time under different optimization configurations.

link variables which is $O(|E_{dsp}||V_{res}|^{2k})$ and thus exponentially grows with k.

These results clearly show that EDRP can be directly applied only for small instances and limited replication degree. Therefore, in order to support runtime operation computationally efficient heuristics are required. In this respect, EDRP nevertheless represents a benchmark against which the performance of heuristics solutions can be assessed. To address this limitation, we can take advantage of the fact that, as confirmed by our experiments, most of the times an operator reconfiguration entails a single scale-out, scale-in operation or a migration. As a consequence, in practice, for each operator i we do not need to consider all possible configurations $\mathcal{V} \in \mathcal{P}(V^i_{res}; k_i)$, but only a (significantly smaller) subset. For example, given the current operator i deployment $\mathcal{U}$, we can consider, as a heuristic, the set of configurations $\mathcal{P}(V^i_{res}; k_i; \mathcal{U}_{i,0}) \subset \mathcal{P}(V^i_{res}; k_i)$ which only comprises the deployments $\mathcal{V}$ which differ from the current deployment $\mathcal{U}_{i,0}$ by only one element (one more node $u \in V^i_{res}$ for a scale-out operation, one less node $u \in \mathcal{U}_{i,0}$, $\mathcal{U}(u) > 0$ for a scale-in operation, and a node replacement $u \in \mathcal{U}_{i,0}$, $\mathcal{U}_{i,0}(u) > 0$ replaced by a node $v \in V^i_{res}$ for a migration). More formally, $\mathcal{P}(V^i_{res}; k_i; \mathcal{U}_{i,0}) = \{\mathcal{V} \in \mathcal{P}(V^i_{res}; k_i) \mid \wedge_{v \in V^i_{res}} (|\mathcal{U}_{i,0}(v) - \mathcal{V}(v)| \leq$

Table 8.4: Number of variables and execution time under EDRP and *r*EDRP (expressed as EDRP / *r*EDRP).

$\|V_{res}\|$	**Max Replication Degree =** 2		**Max Replication Degree =** 3		**Max Replication Degree =** 4	
	Variables ($\times 10^3$)	Resolution Time (s)	Variables ($\times 10^3$)	Resolution Time (s)	Variables ($\times 10^3$)	Resolution Time (s)
8	9.4 / 1.6	3.7 / 0.8	113.7 / 1.8	298.0 / 0.7	994.5 / 1.8	- / 1.0
12	37.3 / 2.8	30.5 / 0.6	848.7 / 3.3	- / 0.9	- / 3.4	- / 0.8
16	103.2 / 5.5	122.6 / 1.1	- / 6.1	- / 1.2	- / 6.3	- / 1.2
20	231.6 / 8.8	415.2 / 1.6	- / 10.0	- / 1.9	- / 9.7	- / 2.5
24	453.3 / 12.7	593.7 / 3.1	- / 13.8	- / 3.9	- / 13.9	- / 4.2

$1) \wedge \sum_{v \in V_{res}^i} |\mathcal{U}_{i,0}(v) - \mathcal{V}(v)| \leq 2 \wedge \sum_{v \in V_{res}^i} (\mathcal{U}_{i,o}(v) - \mathcal{V}(v)) \cdot \sum_{v \in V_{res}^i} (\mathcal{V}(v) - \mathcal{U}_{i,0}(v)) \leq 0\}$.

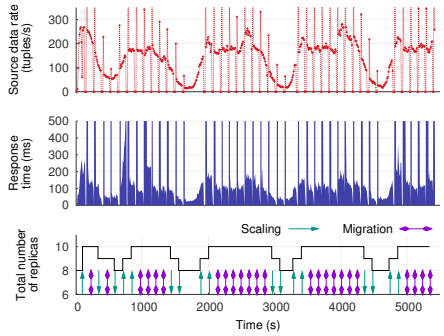
We call this approach *restricted* EDRP problem, *r*EDRP for short. As shown in Table 8.4, *r*EDRP (at the right of / sign) is characterized by a much smaller set of variables (it is easy to verify that the number of variables is $O(|V_{res}|^2|V_{dsp}||E_{dsp}|)$, which is only quadratic on the number of nodes) and order of magnitudes faster execution times. The results with *r*EDRP (not shown for space limitation) are very close to the optimal behavior but require a fraction of the computational costs.

8.6.3 Evaluation of S-EDRP Scheduler

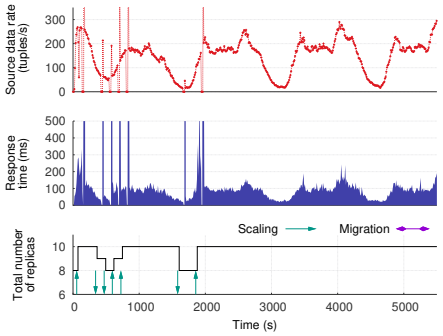
We perform the experiments using Apache Storm 0.9.3 on a cluster of 5 worker nodes, each with 2 worker slots, and a further node to host Nimbus and ZooKeeper. Each node is a machine with a dual CPU Intel Xeon E5504 (8 cores at 2 GHz) and 16 GB of RAM. To better exploit the presence of independent CPU cores, we configure the system so that a worker slot can host at most 4 executors, i.e., $EPS_{\max} = 4$; therefore, a worker node can host at most 8 operator replicas, one for each available CPU core. In S-EDRP, the Analyze phase triggers every 60 s the Plan phase, which solves the ILP problem using CPLEX© (version 12.6.3).

As regards the model parameters, we rely on the same configuration used in the numerical investigation (see Table 8.2) upon which we introduce the following adjustments. First, in these experiments we do not rely on a closed formula expression, e.g., the response time of an M/M/1 queue, to compute the response time as function of the input load. Instead, we conducted a preliminary set of experiments to measure the response time of each operator for different values of the input load[2]. Then, we increase

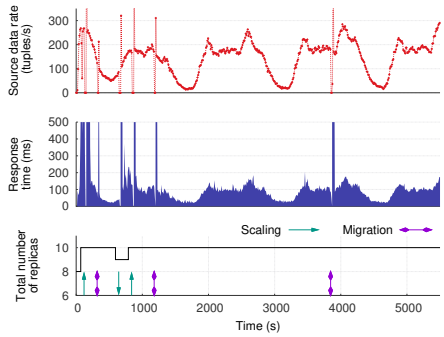
[2]Our earlier experiments revealed that a M/M/1 queue approximation provide a poor approximation of the operator response time. For a more realistic response time modeling, for each operator i we measured the average response time for a set of possible inputs and then derived closed expressions for $R_i(\lambda)$ through Least Squares polynomial interpolation.



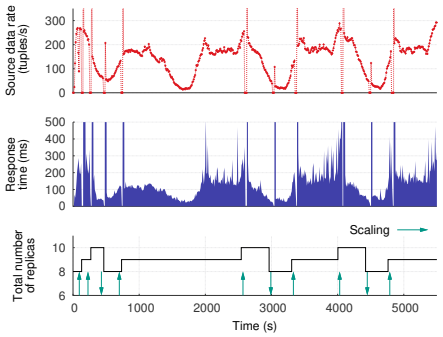
(a) Baseline ($w_r = 0.5$, $w_c = 0.5$, $w_d = 0$): minimization of response time and cost, with no reconfiguration costs.



(b) Balanced ($w_r = 0.3$, $w_c = 0.3$, $w_d = 0.4$): minimization of response time and cost, with reconfiguration costs.



(c) Response Time ($w_r = 0.6$, $w_d = 0.4$, $w_c = 0$): minimization of response time, with reconfiguration costs.



(d) Cost ($w_c = 0.6$, $w_d = 0.4$, $w_r = 0$): minimization of deployment cost, with reconfiguration costs.

Figure 8.4: S-EDRP and different optimization configurations: evolution of the application response time and operators replication under a varying source data rate.

the synchronization overhead t_{sync} from 250 ms to 6 s, accounting for the execution time of Storm `rebalance` command.

In these experiments we use a portion of the real dataset provided by DEBS for the 2015 Grand Challenge. We replay the taxi dataset 60 times faster than the original dataset, thus obtaining that 1 minute of the original dataset time is equal to 1 second of time in our experiments. In particular, we feed the application with data collected during 4 days, characterized by different load levels during the various parts of the day: the tuple emission rate ranges from about 20 to 300 tuples per second. We impose the following bounds on the QoS metrics: $R_{max} = 150$ ms, $C_{max} = 15$ \$/h, $T_{D,max} = 60$ s. We compare different configurations for the objective function. Similarly to the numerical investigation, we first consider the baseline

Table 8.5: Mean response time and cost, and total downtime with different S-EDRP objective configuration.

Objective	Resp. time	Cost	Downtime
Baseline ($w_d = 0$, $w_c = w_r = 0.5$)	103.6 ms	9.3 \$/h	10.18%
Balanced ($w_d = 0.4$, $w_c = w_r = 0.3$)	90.0 ms	9.8 \$/h	2.36%
Response time ($w_d = 0.4$, $w_r = 0.6$)	88.8 ms	9.9 \$/h	1.27%
Cost ($w_d = 0.4$, $w_c = 0.6$)	116.6 ms	8.9 \$/h	4.00%

case that neglects reconfiguration costs, then we consider a balanced scenario with $w_d = 0.4$, and finally we consider the configurations that focus on either response time or deployment cost. Table 8.5 provides a summary of the obtained results in the different settings, showing mean response time (computed excluding tuples buffered during reconfigurations) and monetary cost, and total application downtime during each experiment.

Figure 8.4a shows the results for the baseline scenario, where S-EDRP reconfigures the application at almost every scheduling round. It adjusts the parallelism of *partialRank*, which appears to be a bottleneck in our application, following the variations of the input rate. The total number of replicas ranges from 8 (i.e., one per operator) to 10. Moreover, S-EDRP frequently migrates operators when the load is high, in order to exploit more promising deployments. As consequences of this behavior, we note that (i) the application downtime lasts for 10.18% of the experiment duration, and (ii) the latency and the source data rate present frequent spikes, which are caused by the buffering of tuples during the reconfigurations at both the source and operators.

Figure 8.4b shows the results obtained with a balanced objective function, which equally weights response time and cost, and limits downtime (i.e., $w_r = w_c = 0.3$, $w_d = 0.4$). In this setting, the number of performed reconfigurations is significantly smaller. The balanced objective configuration allows to achieve a 75% reduction of the total application downtime. The mean response time, 90 ms, is lower with respect to the baseline scenario, whereas the deployment cost is slightly higher, amounting to 9.8 \$/h. We observe that, after 2000 s, S-EDRP finds more convenient to run with a higher number of replica, so to avoid the downtime introduced by a possible scale-in operation.

When S-EDRP focuses on response time minimization (i.e., $w_r = 0.6$, $w_d = 0.4$), the scheduler does not need to keep the number of replicas as low as possible. Nevertheless, it does not blindly scale-out operators, because a larger number of replicas would require using more nodes, thus involving a higher network delay in exchanging data streams. Indeed, S-EDRP scales-out only the bottleneck operator, *partialRank*. As shown in Figure 8.4c, S-EDRP achieves a lower mean latency, 88.8 ms, launching two

extra replicas as input rate begins to grow. Then, other reconfigurations are driven by slight changes in the measured network latency, which make S-EDRP enact a few migrations to exploit better placement configurations. Running with a total of 10 replicas for most the time, the monetary cost of executing the topology is the highest with respect to the other scenarios (its mean value is equal to 9.9 \$/h). The cumulative application downtime is the 1.27% of the experiment duration.

Figure 8.4d reports the source rate and the application latency when S-EDRP minimizes the deployment cost and downtime (i.e., $w_c = 0.6$, $w_d = 0.4$). In this setting, S-EDRP keeps the operators replication as low as possible. The number of replicas goes up to 10 when the load is high (with *partialRank* running up to 3 replicas), but, as the load significantly decreases, the scheduler scales-in the operator, restoring the initial count of 8 replicas. The mean deployment cost is indeed lower, amounting to 8.9 \$/h. The application latency is not minimized throughout the experiment, and ranges from 40 ms to 300 ms (except for the spikes soon after the reconfigurations). The cumulative application downtime is 4% of the experiment duration. Observe that, in this experiment, S-EDRP performs scale-out operations only when the bounds are violated: the violation of $R_{\max}$ and the minimization of cost lead to scale-out and subsequent scale-in operations, respectively.

8.7 Summary

In this chapter, we have presented and evaluated EDRP, an ILP formulation that jointly optimizes the replication and placement of DSP applications running in geo-distributed environments. At runtime, by monitoring the application deployment, EDRP can identify the optimal reconfiguration strategy and evaluate its convenience in terms of application downtime. Our formulation of EDRP is general and flexible, thus it can be easily extended or customized for considering different needs. We have shown how it can be adapted and integrated in Apache Storm, one of the most used open-source DSP frameworks. Then, relying on an application that processes real-time data generated by taxis moving in a urban environment, we have conducted a thorough experimental evaluation. The latter has shown the benefits on the application performance that steam from a joint optimization of operators replication and placement and the detailed modeling of the reconfiguration costs. In particular, our results show the importance of taking into account reconfiguration cost to the overall application performance. In the considered scenario, which can be regarded as a best case scenario as far as reconfiguration is concerned, being executed on a local cluster, the reconfiguration downtime were as large as 10% of

the overall experiment duration and with EDRP we were able to achieve up to a tenfold reduction of the system downtime. Moreover, by adjusting the metric weights, we were able to trade-off the different performance metrics stressing latency and/or resource utilization. Since reconfiguration costs are expected to be significantly higher in a distributed scenario due to non negligible network delays and limited bandwidth, these results clearly show that importance of taking into account reconfiguration costs in the operation of self-adaptive DSP platforms.

Appendix

8.A Reconfiguration Metrics

In this appendix we present the detailed expressions of the reconfiguration cost metrics. In Section 8.A.1, we derive the expressions for the reconfiguration downtime metric under the *pause-and-resume* approach. Then, in Section 8.A.2, we provide the detailed downtime expressions for our implementation of the stateful migration protocol in Storm.

8.A.1 Reconfiguration Downtime

As shown in Section 8.3.2, the reconfiguration downtime of the operator $i \in V_{dsp}$ is defined as:

$$T_{D,i}(\boldsymbol{x}) = \sum_{\mathcal{V} \in \mathcal{P}(V_{res}^{i}; k_i)} t_D(i, \mathcal{U}_{0,i}, \mathcal{V})\, x_{i,\mathcal{V}} \tag{8.24}$$

where $t_D(i, \mathcal{U}, \mathcal{V})$ represents the time needed to reconfigure the operator i from the deployment $\mathcal{U}$ to $\mathcal{V}$, and $\mathcal{U}_{0,i}$ represents the current deployment of operator i. The expression for $t_D(i, \mathcal{U}, \mathcal{V})$ depends on the actual type of reconfiguration to be enacted when the deployment of i changes from $\mathcal{U}$ to $\mathcal{V}$:

$$t_D(i, \mathcal{U}, \mathcal{V}) = \begin{cases} 0 & \text{if } \mathcal{U} = \mathcal{V} \\ t_{syn} + t_{D,MI}(i, \mathcal{U}, \mathcal{V}) & \text{if } |\mathcal{U}| = |\mathcal{V}|, \mathcal{U} \neq \mathcal{V} \\ t_{syn} + t_{D,SO}(i, \mathcal{U}, \mathcal{V}) & \text{if } |\mathcal{U}| > |\mathcal{V}| \\ t_{syn} + t_{D,SI}(i, \mathcal{U}, \mathcal{V}) & \text{if } |\mathcal{U}| < |\mathcal{V}| \end{cases} \tag{8.25}$$

where t_{syn} is a constant synchronization overhead, $t_{D,X}(i, \mathcal{U}, \mathcal{V})$, with $X \in \{MI, SO, SI\}$, represents the time needed to perform a migration, a scale-out, and a scale-in operation on i, respectively. Since each type of reconfiguration requires data transfers between multiple pairs of nodes, which can be concurrently executed, the time to complete the reconfiguration corre-

sponds to the longest of any such operation, that is:

$$t_{D,X}\left(i,\mathcal{U},\mathcal{V}\right) = \max_{\substack{u\in\mathcal{U},v\in\mathcal{V}\\ u\neq v}} \left\{\tau_{D,X}\left(i,u,v,\mathcal{U},\mathcal{V}\right)\right\} \tag{8.26}$$

where $\tau_{D,X}\left(i,u,v,\mathcal{U},\mathcal{V}\right), X \in \{MI, SO, SI\}$ represents the time required to exchange data between node u to node v when the operator i is migrated, scaled out, or scaled in, respectively, during the reconfiguration from deployment $\mathcal{U}$ to deployment $\mathcal{V}$.

The expression for $\tau_{D,X}\left(i,u,v,\mathcal{U},\mathcal{V}\right)$ has the following general form:

$$\begin{aligned}\tau_{D,X}\left(i,u,v,\mathcal{U},\mathcal{V}\right) = \max\left\{\tau_{C}^{dwl}(i,v)\cdot\mathbb{1}_{\{\mathcal{U}(v)=0\}}, \tau_{S,X}^{upl}(i,u,v,\mathcal{U},\mathcal{V})\right\} + \\ + \tau_{S,X}^{dwn}(i,u,v,\mathcal{U},\mathcal{V}) + t_{s,v}\cdot\mathbb{1}_{\{\mathcal{V}(v)>\mathcal{U}(v)\}}\end{aligned} \tag{8.27}$$

where $\mathbb{1}_{\{.\}}$ is the indicator function. The expression of $\tau_{D,X}\left(i,u,v,\mathcal{U},\mathcal{V}\right)$ accounts for: 1) $\tau_{C}^{dwl}(i,v)$, the time to download time the operator code to node v, if there was not a replica of operator i in v in the (current) deployment $\mathcal{U}$, i.e., if $\mathcal{U}(v)=0$; 2) $\tau_{S,X}^{upl}(i,u,v,\mathcal{U},\mathcal{V})$, the time to upload the internal state of the replicas present in u to the DataStore; 3) $\tau_{S,X}^{dwn}(i,u,v,\mathcal{U},\mathcal{V})$, the time to download the internal state of replicas from the DataStore to node v; and 4) $t_{s,v}$, the time to (concurrently) initialize new replicas in v, if any, i.e., if $\mathcal{V}(v) > \mathcal{U}(v)$.

The time needed to transfer the operator code, i.e., $t_{C}^{dwl}(i,v)$, and the time needed to redistribute the replicas state, i.e., $t_{S,X}^{upl}(i,u,v,\mathcal{U},\mathcal{V})$ as well as $t_{S,X}^{dwn}(i,u,v,\mathcal{U},\mathcal{V})$, are function of: 1) the round-trip network delay between the DataStore and the related computing resource, $d_{x,DS} + d_{DS,x}$, with $x \in \{u,v\}$; 2) the amount of data to transfer data from or to the DataStore, that is, the code $I_{C,i}$ to be downloaded to node v, the state $I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{upl}$ to be uploaded from u to the DataStore and the state $I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{dwn}$ to be then downloaded to v, respectively; and 3) the data rate $r_{(x,DS)}$ and $r_{(DS,x)}$, with $x \in \{u,v\}$, to and from the DataStore. We readily get the following expressions:

$$\tau_{C}^{dwn}(i,v) = d_{(v,DS)} + \frac{I_{C,i}}{r_{(DS,v)}} + d_{(DS,v)} \tag{8.28}$$

$$\tau_{S,X}^{upl}(i,u,v,\mathcal{U},\mathcal{V}) = d_{(u,DS)} + \frac{I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{upl}}{r_{(u,DS)}} + d_{(DS,u)} \tag{8.29}$$

$$\tau_{S,X}^{dwn}(i,u,v,\mathcal{U},\mathcal{V}) = d_{(v,DS)} + \frac{I_{S,X,(i,u,v,\mathcal{U},\mathcal{V})}^{dwn}}{r_{(DS,v)}} + d_{(DS,v)} \tag{8.30}$$

We conclude by deriving the expressions for the amount of state that has to be redistributed, which depends on the specific reconfiguration action. When a migration is performed, i.e., $X = MI$, a replica first stores

and then retrieves its portion of the state on the DataStore during the reconfiguration. A scaling operation redistributes the operator state among its replicas. Specifically, when a scale-out is performed, i.e., $X = SO$, each replica provides a uniform fraction of its internal state to the new ones; similarly, when a scale-in is performed, i.e., $X = SI$, the internal state of the terminated replicas is redistributed among the other operator replicas. Let $I_{S,i}$ denote operator i state size. Under the assumption that the state is uniformly redistributed among replicas, we have the following expressions:

$$I^{upl}_{S,X,(i,u,v,\mathcal{U},\mathcal{V})} = \begin{cases} I_{S,i}\left[\frac{\delta^-_u(\mathcal{U},\mathcal{V})}{|\mathcal{U}|} + \mathcal{U}(u)\frac{|\mathcal{V}|-|\mathcal{U}|}{|\mathcal{U}||\mathcal{V}|}\right] & \text{if X = SO} \\ I_{S,i}\frac{\delta^-_u(\mathcal{U},\mathcal{V})}{|\mathcal{U}|} & \text{otherwise} \end{cases} \tag{8.31}$$

$$I^{dwn}_{S,X,(i,u,v,\mathcal{U},\mathcal{V})} = \begin{cases} I_{S,i}\left[\frac{\delta^+_v(\mathcal{U},\mathcal{V})}{|\mathcal{V}|} + \min\left\{\mathcal{U}(v),\mathcal{V}(v)\right\}\frac{|\mathcal{U}|-|\mathcal{V}|}{|\mathcal{U}||\mathcal{V}|}\right] & \text{if X = SI} \\ I_{S,i}\frac{\delta^+_v(\mathcal{U},\mathcal{V})}{|\mathcal{V}|} & \text{otherwise} \end{cases} \tag{8.32}$$

where $\delta^+_u(\mathcal{U},\mathcal{V})$ (and $\delta^-_u(\mathcal{U},\mathcal{V})$) denotes the number of replicas to be added (and removed) in node $u \in V_{res}$ when the configuration changes from $\mathcal{U}$ to the deployment $\mathcal{V}$, that is: $\delta^+_u(\mathcal{U},\mathcal{V}) = \max\{\mathcal{V}(u) - \mathcal{U}(u), 0\}$ and $\delta^-_u(\mathcal{U},\mathcal{V}) = \max\{\mathcal{U}(u) - \mathcal{V}(u), 0\}$, respectively. In (8.31)-(8.32), $\frac{\delta^-_u(\mathcal{U},\mathcal{V})}{|\mathcal{U}|}$ is the percentage of state uploaded from the node u to the DataStore during a migration or scale in operation, which is function of the number of terminated replicas $\delta^-_u(\mathcal{U},\mathcal{V})$. Similarly, $\frac{\delta^+_v(\mathcal{U},\mathcal{V})}{|\mathcal{V}|}$ is the percentage of new state transfered to v from the DataStore, going from deployment $\mathcal{U}$ to $\mathcal{V}$. The terms proportional to $\frac{|\mathcal{U}|-|\mathcal{V}|}{|\mathcal{U}||\mathcal{V}|}$ represent the amount of state that is redistributed among the other operator replicas after a scaling operation.

8.A.2 Stateful Migration Downtime in Storm

As mentioned in Section 8.5.3, in Storm we need to take into account that, for any reconfiguration, the entire application must be restarted. In this case we can rewrite (8.24) as:

$$T_{D,i}(\boldsymbol{x}) = \max\left\{\sum_{\mathcal{V}\in\mathcal{P}(V^i_{res};k_i)} t_D(i,\mathcal{U}_{0,i},\mathcal{V})\, x_{i,\mathcal{V}}\,, t_{D,R}(i,\mathcal{U}_{0,i})\cdot \mathbb{1}_{\{\sum_i \sum_{\mathcal{V}\neq\mathcal{U}_{0,i}} x_{i,\mathcal{V}}>0\}}\right\} \tag{8.33}$$

where the first term is the time needed to reconfigure the operator i from the deployment $\mathcal{U}$ to $\mathcal{V}$, $\mathcal{U}_{0,i}$ is the current deployment of i, whereas $t_{D,R}(i,\mathcal{U})$ is the time needed to restart i, should any configuration occurs in the application (condition expressed by $\sum_i \sum_{\mathcal{V}\neq\mathcal{U}_{0,i}} x_{i,\mathcal{V}} > 0$).

Since our Storm migration protocol does not distinguish the different reconfiguration actions (i.e., migration or scaling), we have a single expression for $t_D(i,\mathcal{U},\mathcal{V})$, i.e., the time to reconfigure the operator i from $\mathcal{U}$ to $\mathcal{V}$. It needs to account for a constant synchronization overhead, t_{syn}, and the longest downtime over any possible replica reconfiguration. Thus, we have:

$$t_D(i,\mathcal{U},\mathcal{V}) = t_{syn} + \max_{\substack{u\in\mathcal{U},v\in\mathcal{V}\\ u\neq v}} \{\tau_D(i,u,v,\mathcal{U},\mathcal{V})\} \quad \mathcal{U}\neq\mathcal{V} \tag{8.34}$$

where

$$\begin{aligned}\tau_D(i,u,v,\mathcal{U},\mathcal{V}) = \tau_C^{dwl}(i,v)\mathbb{1}_{\{v\notin\bigcup_i \mathcal{U}_{0,i}\}} \\ + \tau_S^{upl}(i,u,v,\mathcal{U},\mathcal{V}) + \tau_S^{dwn}(i,u,v,\mathcal{U},\mathcal{V}) + \mathcal{V}(v)t_{s,v},\end{aligned} \tag{8.35}$$

and $t_D(i,\mathcal{U},\mathcal{V}) = 0$ if $\mathcal{U}=\mathcal{V}$. Comparing the expression above to (8.27), we note that: (i) we need to consider the sum of the different terms in place of the maximum, because the pause-and-resume approach serializes the different phases; (ii) the time to download the operator code on v, $\tau_C^{dwl}(i,v)$, needs to be considered only if no replica of *any* operator is currently deployed on that node, because Storm packages the code of the whole application in a single archive, which is transferred to each worker node the first time it has to execute a component of the topology; (iii) the whole application is restarted after a reconfiguration, thus all the $\mathcal{V}(v)$ replicas deployed on v have to be launched at the end of the migration process, as expressed by the last term. We also obtain slightly different expressions for the time spent moving state and code as well:

$$\tau_C^{dwn}(i,v) = d_{(v,DS)} + \frac{\sum_{i\in V_{dsp}} I_{C,i}}{r_{(DS,v)}} + d_{(DS,v)} \tag{8.36}$$

$$\tau_S^{upl}(i,u,v,\mathcal{U},\mathcal{V}) = d_{(u,DS)} + \frac{I_{S,i}\frac{\mathcal{U}(u)}{|\mathcal{U}|}}{r_{(u,DS)}} + d_{(DS,u)} \tag{8.37}$$

$$\tau_S^{dwn}(i,u,v,\mathcal{U},\mathcal{V}) = d_{(v,DS)} + \frac{I_{S,i}\frac{\mathcal{V}(v)}{|\mathcal{V}|}}{r_{(DS,v)}} + d_{(DS,v)} \tag{8.38}$$

where we model that, differently from the general formulation (8.28)–(8.30), in Storm: (i) a node has to download the whole topology code; and (ii) each node has to save (and restore) the internal state for all the hosted replicas, because the entire application is restarted during a reconfiguration.

Finally, we consider $t_{D,R}(i,\mathcal{U})$, the time needed to restart i on the same location. This term accounts for a constant synchronization overhead, t_{syn}, and the restart time of any replica. We have:

$$t_{D,R}(i,\mathcal{U}) = t_{syn} + \max_{u\in\mathcal{U}} \{\tau_{D,R}(i,u,\mathcal{U})\} \tag{8.39}$$

where

$$\tau_{D,R}\left(i,u,\mathcal{U}\right)=\frac{I_{S,i}\frac{\mathcal{U}(u)}{|\mathcal{U}|}}{r_{(u,LS)}}+\frac{I_{S,i}\frac{\mathcal{U}(u)}{|\mathcal{U}|}}{r_{(LS,u)}}+\mathcal{U}\left(u\right)t_{s,v} \tag{8.40}$$

models the time to store and retrieve the state from the swapping area and restart the replicas of i.

Chapter 9

Hierarchical Autonomous Control for Elastic DSP

We present a hierarchical and distributed architecture for the autonomous control of elastic DSP applications, which relies on a two layered approach. At the lower level, distributed components issue requests for adapting the deployment of DSP operators as to adjust to changing workload conditions. At the higher level, a per-application centralized component works on a broader time scale; it oversees the application behavior and grants reconfigurations to control the application performance while limiting the negative effect of their enactment (i.e., application downtime).

In the previous chapter, we have shown the importance of controlling the adaptation costs while reconfiguring the application deployment. Being the EDRP problem NP-hard, it suffers from scalability issues and does not represent a viable approach when the system needs to quickly solve large problem instances. Therefore, we need efficient heuristics that can quickly determine the DSP application deployment adaptation at runtime. As previously discussed, recently we have witnessed a paradigm shift with the deployment and execution of DSP applications over distributed Cloud and Fog computing resources. The latter *de facto* bring applications closer to the data, rather than the other way around. Nevertheless, this very idea makes it difficult to control DSP application performance. Most of the approaches proposed in the literature have been designed for cluster environments with a centralized control component overlooking the DSP operations. These solutions typically do not scale well in a distributed environment given the spatial distribution, heterogeneity, and sheer size of the infrastructure itself. While scalable decentralized solutions have been proposed, e.g., [158], their inherent lack of coordination might result in frequent reconfigurations which negatively affect the application perfor-

mance due to continuous system downtime.

In this chapter, to take the best of the two worlds, we propose a hierarchical distributed approach to the autonomous control of elastic DSP applications in Fog-based environment. Indeed, by working at different levels of abstractions, we believe that a hierarchical approach can improve performance and scalability without compromising stability. Specifically, by decentralizing the resource and operator control, we relieve the control load on the centralized component of the system. This allows a scalable management of a multitude of computing resources and DSP applications. However, by conveniently centralizing the high-level management capabilities, we can overcome the lack of coordination, which represents one of the main drawbacks of the existing fully decentralized approaches.

The contributions of this chapter are as follows.

- We present *Elastic and Distributed DSP Framework* (EDF), a hierarchical distributed architecture for the autonomous control of elasticity (Section 9.2). The control is organized according to the MAPE model for self-adaptive systems. Specifically, the proposed architecture relies on a high-level centralized MAPE-based *Application Manager* that coordinates the runtime adaptation of subordinated MAPE-based *Operators Managers*, which, in turn, locally control the adaptation of single DSP operators.
- We present a simple reference control strategy for each component (in Section 9.3). Specifically, we present a *local policy* for the Operator Managers and a *global policy* for the Application Manager. The first monitors and analyzes the operator performance to determine whether it needs to be reconfigured by scaling the number of replicas or by migrating a replica. The global policy identifies the most effective reconfigurations proposed by the Operator Managers, accepting or declining the proposed reconfigurations in order to control their number, and hence the application downtime.
- We have implemented EDF on Distributed Storm and evaluated the proposed solution on our prototype. We implemented two simple policies: the local policy employs a threshold approach to request operator reconfigurations to the Application Manager; the global policy adopts a token bucket scheme to control the number of allowed reconfigurations in any control interval. As shown in Section 9.4, our results are promising and show the effectiveness of the proposed solution in achieving a good trade-off between application performance and reconfiguration cost.

Although this chapter presents only preliminary results, it proposes a general DSP system architecture for geo-distributed environments, and sheds light on the benefits of using hierarchical decentralized placement

policies. The results are encouraging and foster the design of new hierarchical heuristics, which can be efficiently employed in distributed Cloud and Fog computing environments.

9.1 Related Work

Runtime adaptation of DSP applications achieved through elastic data parallelism is attracting many research and industrial efforts. As analyzed in Chapter 2, albeit most of the approaches for elasticity use a centralized approach (e.g., [65, 71, 78, 83, 132]), several decentralized solutions have also been proposed (e.g., [88, 145]). Either way, all these solutions are implicitly or explicitly organized as self-adaptive systems based on the MAPE model.

To determine when the DSP application should be reconfigured, some works, e.g., [31, 71, 80], exploit best-effort threshold-based policies based on the utilization of either the system nodes or the operator instances. Other works, e.g., [48, 65, 132, 145, 214], use more complex centralized policies to plan the scaling decisions. However, all these works rely on a centralized planner for the runtime adaptation of DSP applications, that may suffer from network latencies in a geo-distributed operating environment.

Several works also propose decentralized approaches for adapting the DSP application deployment. Mencagli [145] presents a game-theoretic approach where the control logic is distributed on each operator; specifically, each decentralized agent seek to change its operator parallelism degree as needed, when multiple applications compete for resources in a non-cooperative manner. Nevertheless, this solution is not integrated in a DSP system. Hochreiner et al. [88] introduces a managing component for each DSP operator, which uses a threshold-based policy to elastically adapt the number of operator replicas at runtime. Other decentralized approaches (e.g., [50, 158, 171, 222]) focus only on adapting the operator placement at runtime and do not consider the operator replication problem. Examples are the distributed placement heuristics by Pietzuch et al. [158] and by Rizou et al. [171], that we have implemented in Distributed Storm (Chapter 3). Even though these decentralized heuristics overcome scalability issues, they can suffer from lack of global coordination. Therefore, they can introduce frequent and uncoordinated reconfigurations, which degrade the application performance.

Differently from all the above mentioned solutions, we introduce a hierarchical control architecture that can be equipped with hierarchical elasticity policies. Our approach reacts to changes of working conditions (e.g., workload variations) by performing migration and scaling operations. Furthermore, by working at different levels of abstractions, it can improve performance and scalability without compromising stability. Few other

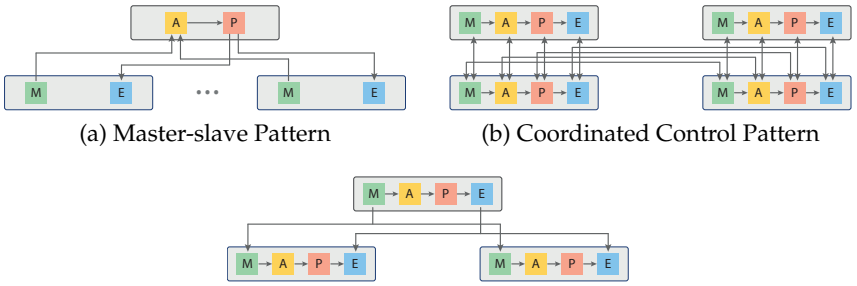
research efforts also rely on a multi-level strategy with separation of concerns and time-scale to adapt DSP applications [83, 146]. Hidalgo et al. [83] propose a centralized elasticity control algorithm, where a reactive and a proactive policy work interleaved on two different time scales. Mencagli et al. [146] study the problem of parallelizing sliding-window preference queries on a single multi-core computing node. These queries are implemented using a special kind of windowed operators. The authors design a two-level adaptation solution that controls load balancing (at the lower-level) and resource allocation through vertical elasticity (at the higher-level). Differently from these solutions, we design a hierarchical approach that considers the adaptation of the whole DSP application and makes no assumptions on the type of DSP operators. Furthermore, our solution works in distributed environments, where computing nodes can be interconnected with not-negligible network latency.

Most of the existing DSP systems have been designed to work in centralized cluster or Cloud computing environments (e.g., [65, 71, 77, 113, 199, 221]). As more extensively discussed in Chapter 2, so far, only few solutions have been explicitly proposed for Fog computing environments (e.g., [11, 175, 178]). The recently presented SpanEdge [175] considers the execution of Storm in decentralized data centers and, similarly to our works, places the application operators so to minimize network latencies. Nevertheless, it does not support operator migrations. Foglets [178] proposes a decentralized architecture and a Fog-specific programming model that supports the migration of application components. Differently from all the existing solutions, we propose a hierarchical architecture that supports the execution of multi-level autonomous control policies, that drive the deployment adaptation of DSP applications at runtime.

9.2 System Architecture

9.2.1 Architectural Options for Decentralized Control

The MAPE loop is a widely adopted model to organize the autonomous control of a software system (see Chapter 2). When the controlled system is geo-distributed as in Fog computing, a single MAPE loop, where analysis and planning decisions are centralized on a single component, introduces a single point of failure and scalability limitation. The latter depends on the capacity of the centralized control entity, which can efficiently manage the adaptation of only a limited number of entities, as well as on the presence of network latencies among the system components. As described by Weyns et al. in [208], different patterns to design multiple MAPE loops have been used in practice by decentralizing the functions of self-adaptation. Here, we describe some key configurations, aiming to identify the most suitable



(a) Master-slave Pattern (b) Coordinated Control Pattern

(c) Hierarchical Control Pattern

Figure 9.1: Different options for decentralizing the MAPE loop

approach to control DSP applications in the geo-distributed execution environment under investigation.

Master-slave Pattern. In a *master-slave pattern*, the system includes a single master component, which runs the Analyze and Plan phases, and multiple independent worker components, which run the Monitor and Execute phases in a decentralized manner. We represent this pattern in Figure 9.1a. Differently from a fully centralized approach, this design pattern decentralizes the execution of the Monitor and Execute components, relieving the burden from the centralized control node. The latter is in charge of determining when and how a reconfiguration should be performed. Having single and centralized Analyze and Plan components, this pattern can be equipped with self-adaptation policies that can be more easily designed and, moreover, can more easily determine globally optimum reconfiguration strategies. Nevertheless, the centralized components of the MAPE loop can still represent the system bottleneck, especially when they have to control a multitude of entities scattered in a large-scale geo-distributed system. Moreover, due to the system distribution, collecting monitoring data on the master component and then dispatching the subsequent scaling actions to the decentralized executors may introduce a not negligible communication overhead.

Coordinated Control Pattern. Sometimes controlling the elasticity of a system using a single centralized component is unfeasible, e.g., because of scale, administrative, or privacy issues. Anyway, we still need to efficiently control the application elasticity so to meet certain QoS metrics. As represented in Figure 9.1b, the *coordinated control pattern* employs multiple decentralized MAPE loops, where each control loop oversees one specific part of the system. The control loops must also coordinate with one another, as peers, so to reach joint adaptation decisions, when needed. With respect to the degree of cooperation, a great variety of inter-node behaviors can be devised, ranging from a fully uncoordinated to a tightly coordinated one. Each degree of coordination exhibits pros and cons. As observed in

Chapter 3, the lack of coordination between the distributed agents may introduce too frequent and uncoordinated decisions that can be detrimental for the application performance. Conversely, a tightly coupled coordination reduces the system ability to quickly react to changes. Although this pattern allows to obtain highly scalable solutions, designing efficient decentralized control policies is, in general, not easy, because of the difficulty of guaranteeing convergence properties in a decentralized manner.

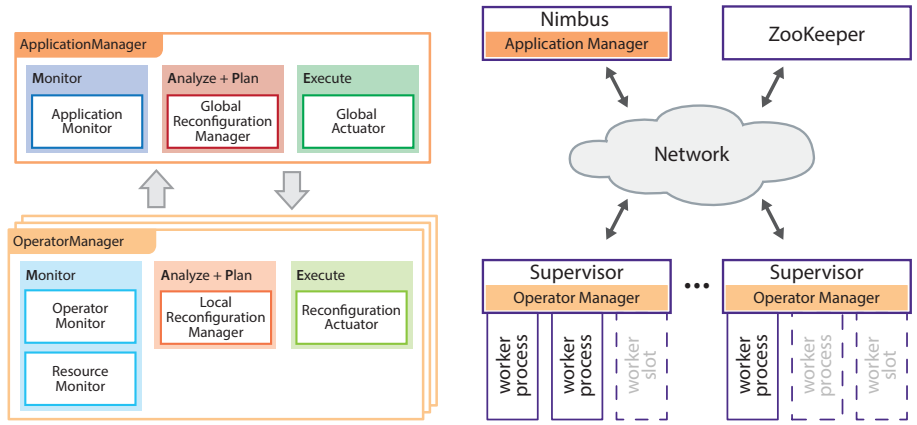
Hierarchical Control Pattern. The *hierarchical control pattern* revolves around the idea of a layered architecture, where each layer works at a different level of abstraction. In this pattern, multiple MAPE control loops work with time scales and concerns separation. Lower levels operate on a shorter time scale and are in charge of performing local adaptation. Exploiting a broader view on the system, higher levels steer the overall adaptation by providing guidelines to the lower levels. As represented in Figure 9.1c, each layer usually includes a full MAPE loop with all the four components.

We believe that this approach is well suited for controlling DSP applications in a Fog environment: it promises to exploit the benefits of both centralized and decentralized architectures, thus improving performance and scalability without compromising stability. By working at different levels of abstraction, the system can more efficiently deal with a great number of near-edge and Cloud computing resources, which can also expose very different features. Near-edge resources are usually characterized by lower computing capacity, are interconnected by not negligible network latency, and can possibly have limited energy capacity. Conversely, Cloud resources expose (practically) infinite computing capacity and are interconnected with almost negligible network latency. A hierarchical control allows to rule the complexity by decentralizing as much as possible the low-level adaptation, while, at the same time, exploiting the benefit of lightweight higher-level coordination elements, which take advantage of a broader view of the system.

9.2.2 Hierarchical Architecture

We present a hierarchical distributed architecture, named *Elastic and Distributed DSP Framework* (EDF), for the autonomous control of elastic DSP applications in a Fog environment. The proposed solution is organized according to the hierarchical control pattern, where higher-level MAPE components control subordinate MAPE components. Specifically, our proposal revolves around a two layered approach with separation of concerns and time scale between layers. Figure 9.2a illustrates the conceptual architecture of EDF, highlighting the hierarchy of the multiple MAPE loops and the system components in charge of the MAPE loop phases.

At the lower level (i.e., at the per-operator grain) and a faster time scale,



(a) The EDF conceptual architecture: hierarchical MAPE loops

(b) Implementation of the system architecture in Storm

Figure 9.2: System architecture

the *Operator Manager* is the distributed entity in charge of controlling the adaptation of a single DSP application operator/subset of the DSP application operators through a local MAPE loop. It monitors the system logical and physical components used by the operator(s) through the *Operator Monitor* and the *Resource Monitor*, and then, through the *Local Reconfiguration Manager*, it analyzes the monitored data and determines if and which local reconfiguration action (among operator scale-in, scale-out, or migration) is needed. When the Operator Manager determines that some adaptation should occur, it issues an operator adaptation request to the higher layer.

At the higher level (i.e., at the per-application grain) and a slower time scale, the *Application Manager* is the centralized entity that coordinates the adaptation of the overall DSP application through a global MAPE loop. By means of the *Application Monitor* it oversees the global application behavior. Then, through the *Global Reconfiguration Manager* it analyzes the monitored data and the reconfiguration requests received by the multiple Operator Managers, and decides which reconfigurations should be granted. These decisions are then communicated by the *Global Actuator* to each Operator Manager, which can, finally, execute the operator adaptation actions by means of the its local *Reconfiguration Actuator*.

The EDF architecture is general enough to not limit the specific internal policies and goals that can be designed for each component in the two layers. For example, the planning components can be either activated periodically or on event-basis, can rely on optimization problem formulation or heuristics with the goal to minimize the application response time, maximize its availability or a combination of the two. As a proof-of-concept of

the proposed architecture, we present, in Section 9.3, simple heuristic adaptation policies whose overall adaptation goal is to preserve the application performance, avoiding unnecessary or too frequent reconfigurations which might result in excessive application downtime.

We have implemented the proposed EDF architecture in Apache Storm. Figure 9.2b shows the high-level instantiation of the EDF components on the Storm architecture. EDF leverages on the monitoring system and the elasticity and stateful migration capabilities exposed by Distributed Storm (see Chapters 3, 7, and 8). Furthermore, we consider that, to agree on satisfying execution conditions, the user and the DSP system provider stipulate a Service Level Agreement (SLA). The SLA specifies as Service Level Objective (SLO) the maximum acceptable response time $R_{\max}$, that is the worst end-to-end delay from a data source to a data sink, and the maximum tolerable downtime during normal execution conditions. The latter indicates how often the application can be reconfigured when its response time is far from the critical value $R_{\max}$.

9.3 Multi-level Elasticity Policy

The proposed two-layered architecture for self-adaptive DSP elasticity control identifies the different macro-components (i.e., Application Manager and Operator Managers) that, by means of abstraction layers and separation of concerns, cooperate to adapt the deployment of DSP applications at runtime. By properly selecting each component internal policy, the proposed solution can address the needs of different execution contexts, which can comprise applications with different requirements, infrastructures with different computing resources, and different user preferences. For example, specific policies can execute the application by minimizing its response time, maximizing its availability, or limiting the adaptation efforts (i.e., executing the application in a best-effort manner). The Operator Manager works at the granularity of a single DSP operator and implements what we call a *local policy*. By monitoring and analyzing the performance of each operator replica, the local policy can plan a reconfiguration of number and location of the operator replicas. Specifically, by scaling the number of replicas, the operator exploits parallelism to quickly process its incoming data, whereas by migrating some of the operator replicas, the operator better distributes the incoming load among computing resources. The Operator Manager sends the planned reconfiguration to the Application Manager, which runs periodically and decides, according to its so called *global policy*, which reconfigurations should be enacted. The global policy works at the granularity of the whole application, thus it coordinates the reconfigurations so to limit them and avoid deployment oscillations, if needed. On the

basis of the monitored application performance and the stipulated SLA, the global policy identifies the most effective reconfigurations proposed by the Operator Managers: it accepts or declines each reconfiguration with the aim to adapt the DSP application to changing working conditions while meeting the SLA.

9.3.1 Local Policy

The Operator Manager local policy implements the Analyze and Plan phases of the decentralized MAPE loop, which controls the execution of a single DSP operator. Running on a decentralized component, this policy has only a local view of the system, which results from the monitoring components (i.e., Operator Monitor and Resource Monitor). The local view consists of the status (i.e., resource utilization) of each operator replica and of a restricted suitable set of computing nodes (i.e., located in the neighborhood). By analyzing this information, the policy can plan a reconfiguration of the operator deployment, either by changing the number of replicas, or by migrating some of them. The proposed reconfiguration plan is then communicated to the centralized Application Manager which, based on all the Operator Manager's reconfiguration plans and the global policy, determines which plan can be executed and which not.

Reconfiguration Plan. A reconfiguration plan is expressed through the following information: *adaptation actions*, *reconfiguration gain*, and *reconfiguration cost*[1]. We consider two types of adaptation actions: replica migration and operator scaling. Actions can be of the form: "move replica α of op from r_i to r_j", "add a new replica to op on r_i", or "remove replica α of op from r_i", where op and r_i denote an operator and a computing resource, respectively. The *reconfiguration gain* is a function, adopted by every Operator Manager, which captures the benefits of the planned adaptation action. It can express, for instance, the reduction of the operator's processing latency, the reduction of monetary cost for running the operator, or the improvement of some utility function. We assume a simple gain function that induces an order relation among the reconfiguration actions, namely `scale-out` $>$ `migration` $>$ `scale-in`. The *reconfiguration cost* expresses the cost of reconfiguring the system. In this chapter, we express it in terms of application downtime. It results from the time required to add/remove an operator replica, to relocate the operator code, and to migrate its internal state (if any). We now discuss the two types of adaptation action.

Replica Migration. A computing resource can host replicas of one or more operators, which, in turn, are controlled by dedicated Operator Managers. When the computing resource becomes overloaded, the hosted repli-

[1]For the sake of simplicity, we assume that the local policy proposes, for an operator, a single reconfiguration decision (i.e., migration, scaling) at a time.

cas can experience a performance degradation. To overcome this issue, an Operator Manager proposes to move some of the operator replicas away from the resource.

We adopt a reactive and threshold-based policy in order to decide when and how to perform the migration. The local policy analyzes the monitoring data coming from the computing resources that host at least one operator replica. We denote with U_r the overall CPU utilization of the resource r. When U_r is above a critical value $U_{\max}$, the policy plans to migrate at most one operator replica to a new location. The latter is identified in two steps. First, the policy sorts the known neighbor resources according to their distance, measured in terms of network delay. Then, it selects the new location using a randomized approach: the closer the resource, the higher the probability of being selected. The policy checks if the new selected location has room to run the migrating replica; in negative case, a new resource is selected from the sorted list.

Reconfiguration Cost. If the operator is stateless, the migration of a replica can be easily performed by terminating the replica on the old location, moving its code to the new location, and restarting it. On the other hand, if the operator is stateful, we also need to efficiently migrate its internal state, so to preserve the integrity and consistency of the outputted streams. Our migration protocol follows a pause-and-resume approach with the help of a data store as staging area for the replica internal state (details on our migration protocol in Chapter 7).

Operator Scaling. When an operator replica receives an increasing workload, it can saturate the capacity of the hosting computing resource. To prevent the performance penalty associated to overloading, the Operator Manager proposes to add an additional replica and redistribute the incoming workload accordingly. Conversely, when the incoming workload decreases, the Operator Manager can reduce the number of replicas in order to decrease the number of allocated resources, and redistribute the workload among the remaining ones. Let us denote by S_α the resource utilization of the hosting resource by replica α, which measures the fraction of CPU time used by α. We adopt a simple threshold-based scale-out policy to each replica. When the utilization of α exceeds a usage threshold $S_{\text{s-out}} \in [0, 1]$ (i.e., $S_\alpha > S_{\text{s-out}}$), the Operator Manager proposes to add a new replica. Its placement is computed relying on the same strategy used for the replica migration. Conversely, the Operator Manager proposes a scale-in operation, which removes one of the running n replicas, when the sum of their utilization divided by $n - 1$ is significantly below the usage threshold, i.e., when $\sum_{\alpha=1}^{n} S_\alpha/(n-1) < cS_{\text{s-out}}$, being $c < 1$. The replica to be removed is randomly chosen between the two replicas with the highest utilization.

Reconfiguration Cost: If the operator is stateless, a scaling operation im-

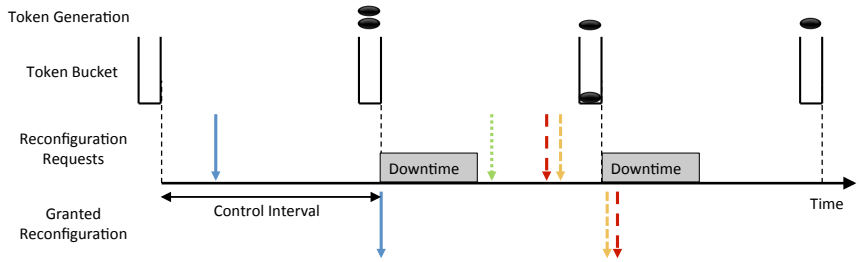


Figure 9.3: Global policy behavior

plies only to start or stop a replica. If the operator is stateful, we also need to reallocate its internal state among the new set of replicas. We assume that each replica can work on a well-defined state partition [65]. A scale-out operation redistributes equally the partitions among replicas, whereas a scale-in operation aggregates the partitions from the merged replicas.

9.3.2 Global Policy

The Application Manager global policy implements the Analyze and Plan steps of the centralized MAPE loop. Its main goal is to satisfy the DSP application SLA, while minimizing the allocated resources (or their cost). To this end, it monitors the application response time and analyzes its behavior with respect to the SLO specified in the SLA. In the planning phase, the policy determines which reconfiguration plans, proposed by the decentralized Operator Managers, should be enacted as to improve performance while controlling the number of application reconfigurations (which cause application downtime). In this thesis, we consider a simple global policy scheme which is exemplified in Figure 9.3. Time is divided in control intervals of fixed length T. During each interval, the global policy collects reconfiguration requests from the Operator Managers: these requests can take different forms, e.g., replica migrations (the continuous arrows in the figure), operator scale-out (the dotted arrow), and operator scale-in (the dashed arrow). At the end of each interval, the policy determines how many and which reconfigurations should be enacted by the Operators Managers. In order to control the number of reconfigurations, and hence the downtime, we adopt a simple token bucket scheme whereby each reconfiguration consumes a token. Tokens are generated at the end of each control interval T and are accumulated in a token bucket, which has a finite capacity (i.e., when the bucket is full, it cannot store any other token). The number of reconfigurations allowed at the end of each control interval is thus limited by the number of available tokens. If the number of requests is higher than the number of available tokens, the global policy has to iden-

tify the most valuable reconfigurations to accept. As simple scheme, the policy uses a greedy approach by prioritizing the requests according to the gain to cost ratio; the higher this index, the better the reconfiguration.

In the proposed scheme, a key role is played by the token generation rate. Ideally, when the application response time is well within the SLO (defined by $R_{\max}$), reconfigurations should be limited since performance is guaranteed and the possibly sub-optimal behavior is preferable to the downtime caused by reconfigurations. On the other hand, should the performance degrades, the system should be more prone to reconfigure itself. As such, the token generation frequency depends on how far is the response time from $R_{\max}$, with increasing token generation rates as performance gets close to $R_{\max}$.

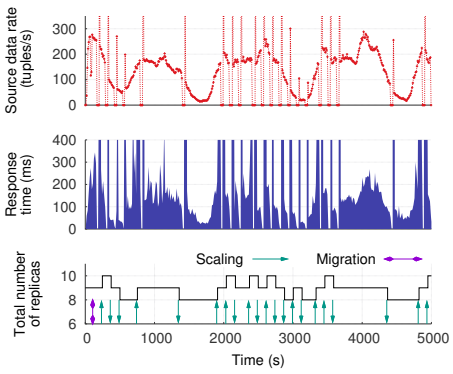
9.4 Evaluation

We evaluate EDF equipped with the proposed proof-of-concept policies, using Apache Storm 0.9.3 on a cluster with 5 worker nodes and one further node to host Nimbus and ZooKeeper. Each node has a dual CPU Intel Xeon E5504 (8 cores at 2 GHz) with 16 GB of RAM.

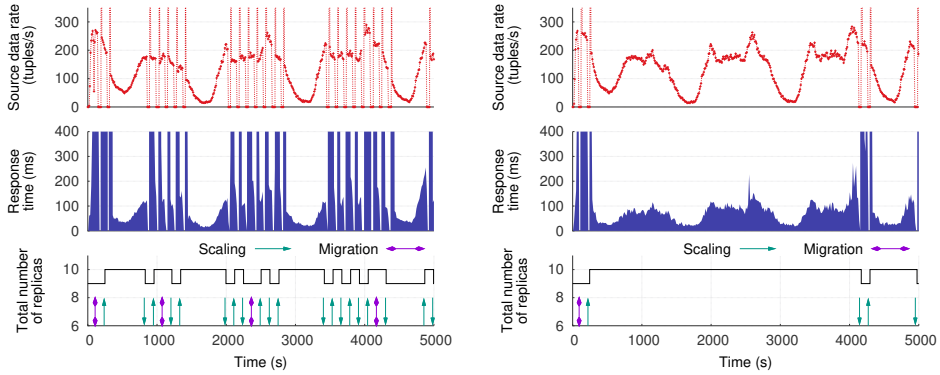
The reference application solves a query of DEBS 2015 Grand Challenge [99], where data streams originated from the New York City taxis are processed to find the top-10 most frequent routes during the last 30 min. A detailed description of the DEBS 2015 Grand Challenge application can be found in Chapter 6.

We feed the application with a sample dataset provided by DEBS, and process real data collected during 2 days. The taxi service utilization significantly changes during the day, thus the application input rate is variable as well. As regards the Operator Manager local policy, we set the scale-out and migration thresholds, $U_{\max}$ and $S_{\text{s-out}}$, to 0.7 and the scale-in parameter c to 0.75. Both OperatorManager and ApplicationManager run once every 30 s, respectively proposing and accepting/rejecting reconfigurations. We compare the baseline approach in which all reconfiguration requests are always accepted by the ApplicationManager to one in which the global policy in Section 9.3.2 is employed in order to determine which reconfigurations will be enacted. In particular, the token bucket stores at most 1 token at any time and the token generation rate is 1 per min only if the achieved application response time is above $\beta R_{\max}$, where $\beta \in [0, 1]$, otherwise no token is generated. In these experiments we set $R_{\max} = 200$ ms and vary β.

Figure 9.4a shows the application response time and number of replicas during the experiment when using the baseline approach. Since every reconfiguration proposed by any OperatorManager is accepted (like in a



(a) All reconfigurations



(b) $\beta = 0.50$ (c) $\beta = 0.75$

Figure 9.4: Response time and number of replicas using different policies for ApplicationManager: in (a) accepting all the reconfiguration requests, in (b) and (c) generating a token only when response time is greater than $\beta R_{\max}$

fully decentralized policy), the application is frequently reconfigured. As a consequence, the application is available only for 93.7% of the time. The measured response time shows many spikes, which are caused by tuples buffering during reconfiguration.

Figure 9.4b shows the application response time and number of operator replicas during the experiment using the full reconfiguration policy, with $\beta = 0.5$. As the response time frequently rises above $\beta R_{\max} = 100$ ms, the number of granted reconfigurations is not significantly reduced with respect to the baseline approach in Figure 9.4a (and so the application downtime). Nevertheless, we can observe that, by performing less reconfigurations, the total number of replicas is never reduced to 8, due to the lack of tokens and the low priority of the scale-in action.

Figure 9.4c shows the results when $\beta = 0.75$. As tokens are now generated in a more conservative manner (being $\beta R_{\max} = 150$ ms), the number

of reconfigurations is significantly reduced. In the initial part of the experiment, the input rate grows up to 300 tuples/s, resulting in high response time; therefore, EDF generates tokens for performing a migration and for increasing the total number of replicas to 10. Then, the application is stable until a new input peak (at around 4000 s), when a scale-in followed by a scale-out of the bottleneck operator are accepted. The application downtime is limited (only 1.7%), which is beneficial for response time, but it might lead to higher cost, having more active replicas.

9.5 Summary

In this chapter, we have presented Elastic and Distributed DSP Framework (EDF), a hierarchical autonomous control for elastic DSP applications. Designed according to the decentralized MAPE control pattern, our proposal revolves around a two layered approach with separation of concerns and time scale between layers. At the lower level, distributed components control the adaptation of DSP operators, so to improve their performance by means of scaling and migration actions. At the higher level, a per-application centralized component oversees the overall DSP application performance and coordinates its deployment by accepting or declining the proposed reconfiguration actions. Then, relying on an application that processes real-time data generated by taxis, we have conducted an experimental evaluation. The results have shown the effectiveness of our solution in achieving good trade-off in terms of application performance and number of application reconfigurations even adopting simple control policies. These results are encouraging and foster the design of new hierarchical heuristics, which can efficiently deal with the DSP application deployment problem.

Chapter 10

Conclusion

In this thesis, we have addressed the problem of running DSP applications over geo-distributed environments. First, we have investigated the initial deployment problem of DSP applications with different QoS requirements. Our contributions regard the formalization of two deployment problems (i.e., placement and replication problem) and the design of several QoS-aware heuristics for quickly computing high-quality application placement solutions. Then, we have explored the challenges of adapting the application deployment at runtime. Indeed, to preserve the application performance within acceptable bounds and avoid costly over-provisioning of the system resources, the deployment of DSP applications should be conveniently reconfigured at runtime, through migration and scaling operations. Specifically, we have shown the importance of taking into account the adaptation costs while determining reconfiguration actions, thus controlling the short term application performance degradation. Our contributions are twofold. First, we have formalized the elastic replication and placement problem, which determines whether the application should be more conveniently redeployed by explicitly considering the adaptation costs. Then, we have developed a hierarchical approach for the autonomous control of elastic DSP applications, which can efficiently deal with runtime adaptation over large scale and geo-distributed infrastructures.

10.1 Major Contributions

In this thesis, we have investigated the placement and replication problem of DSP applications using an engineering-oriented approach, which comprises the following steps: problem identification and modeling, design of resolution approaches, prototype development, and experimental evaluation. The main contributions of this thesis are as follows.

- We have proposed a taxonomy of the existing deployment and runtime

adaptation approaches for DSP systems (Chapter 2). First of its kind for DSP systems, the taxonomy summarizes the main design choices of the existing solutions and helps to identify new research directions.

- We have provided three main contributions to the initial deployment problem of DSP applications over heterogeneous resource. First, we have formalized the operator placement problem (Chapter 4), obtaining ODP. It introduces to the state of the art a tool that can be easily adopted as a benchmark against which centralized and decentralized placement heuristics can be evaluated.

  Second, we have proposed several heuristics for efficiently solving the operator placement problem (Chapter 5). We have proposed two main classes of heuristics, i.e., model-based and model-free, which take into account the QoS attributes of applications and computing and network resources. We have evaluated the heuristics in terms of resolution time and, differently from the existing solutions, in terms of quality of the computed placement solution. To this end, we have used ODP as benchmark. As a result, we have identified in Local Search the heuristic that achieves, on average, the best trade-off between resolution time and performance degradation.

  Third, we have formulated the optimal operator replication and placement problem, obtaining ODRP (Chapter 6). Differently from existing solutions, the proposed solution jointly optimizes the replication and placement of the DSP operators, while maximizing the QoS attributes of the application. ODRP also provides a general framework that can be used to evaluate existing heuristics.

- We have provided two main contributions regarding the runtime adaptation of DSP application deployment. First, we have proposed a formulation of the elastic operator replication and placement problem, obtaining EDRP (Chapter 8). This formulation computes the optimal strategy to adapt at runtime the replication and placement of DSP operators while explicitly accounting for reconfiguration costs.

  Second, we have proposed a hierarchical control approach for elastic DSP applications (Chapter 9). Most of the existing solutions are either centralized or decentralized. Differently, our proposal revolves around a two layered hierarchical approach with separation of concerns and time scale between layers. As such, this approach can take the best of centralized and fully decentralized architectures, thus improving performance and scalability without compromising stability.

- We have design and implemented two extensions of Apache Storm. The first one is Distributed Storm (Chapter 3), which supports the execution of decentralized, QoS-aware, and self-adaptive scheduling policies on heterogeneous infrastructures. The second one is Elastic Storm (Chap-

ter 7), which introduces mechanisms for elasticity and stateful migration in Storm. These extensions have been released as open source projects.

10.2 Future Research Directions

In this thesis, we have addressed several issues regarding the initial deployment and runtime adaptation of DSP application over geo-distributed infrastructures. However, a number of challenges remain to be faced in future research to further improve the runtime support of DSP applications, while better exploiting the presence of near-edge/Fog computing resources.

Multi-application Deployment Decisions. As shown in the literature analysis (Chapter 2), optimizing the deployment of multiple concurrent applications represents a largely unexplored research direction. In a more general setting, the infrastructure hosts multiple DSP applications, each arriving and departing over time with unforeseen requirements and characteristics. In this setting, it is worth investigating the trade-offs between the possibly conflicting QoS requirements of multiple DSP applications sharing the same infrastructure and the service provider objectives, e.g., profit maximization and/or optimal resource utilization.

Enforcing stronger SLA. When multiple DSP applications run on a shared computing infrastructure, the application provider and the infrastructure provider stipulate a SLA. The latter expresses service level objectives (e.g., in terms of application performance) that should be met at runtime (see Chapter 2). In this thesis, we started to explore how to meet a SLA in a distributed manner. However, the enforcement of stronger SLAs, that may include percentiles and higher moments of QoS attributes, deserves further investigations, especially to address the challenges of the new environment, resulting from the convergence of IoT, Fog, and Cloud computing resources. New and more sophisticated adaptation policies should be designed so to efficiently rule the complexity of the next-generation DSP systems. Indeed, these systems have to execute multiple DSP applications, which are subject to varying workloads and are characterized by stringent and possibly conflicting QoS requirements. Moreover, being the infrastructure characterized by a large number of heterogeneous (and possibly dynamic) computing and network resources, DSP systems must be able to handle failures and changes of the execution environment without introducing prolonged latency spikes or downtimes.

Resource Heterogeneity and Mobility. Today's computing environments include computing resources characterized by a high degree of heterogeneity. To better exploit the ever increasing presence of these resources, deployment policies should be accordingly designed. Recentely, some re-

search efforts investigate the possibility of jointly running the computation on CPU and GPGPU (e.g., [111]) and on fixed and mobile resources (e.g., [153]). The convergence of these resources in a single, highly heterogeneous, infrastructure comes with further challenges, such as location awareness, power consumption, architecture dependent code. Nevertheless, their seamlessly integration enables the development of more efficient DSP systems, which better exploit resources and better support the execution of different types of workloads. For example, the combined exploitation of mobile resources enables the development of new services, that pro-actively send valuable information to users while they are moving around the city. Observe that, by providing a generic representation of computing and network resources, our deployment models (i.e., ODP, ODRP, EDRP) can be easily extended to consider different kind of heterogeneous resources.

Self-Adaptation and Hierarchical Control. Since DSP applications are long-running and characterized by strict QoS requirements, the ability to adapt the application deployment represents a key operation that should be computed as quickly as possible. This thesis has investigated benefits and limitations of using heuristics to compute the initial application placement as well as to control its runtime adaptation, showing the importance of a suitable representation of applications and computing infrastructures. Therefore, this thesis encourages the design and evaluation of new QoS-aware heuristics for geo-distributed environments, like the distributed Cloud and Fog computing environments. The EDRP model can be used to evaluate the self-adaptation policies already proposed in literature as well as to design new approaches. A promising research direction regards the design of hierarchical self-adaptive control policies. Although at its infancy, our hierarchical control architecture has shown interesting preliminary results. New and more sophisticated policies deserve to be investigated. Importantly, these policies should be evaluated in a real Fog environment, where the computing resources are hierarchically organized, geographically distributed, and can experience availability issues. As discussed in Chapter 2, several self-adaptation policies deal with failures of the computing infrastructure (e.g., [204]). Fault-tolerance is of key importance in the emerging Fog computing environment, where computing nodes are less resilient to failures and cannot be easily replaced by another machine with similar capacity and connectivity (as in the Cloud). Moreover, as regards self-adaptation, we observe that the approaches proposed in this thesis determine reconfigurations relying on a single snapshot of the system, i.e., the system status when the optimization model is executed. These approaches inherently compute greedy reconfiguration strategies which might be suboptimal if we are interested in the overall performance over a period of time. To overcome this issue, self-adaptation approaches should explicitly

take into account intervals of time. Possible solutions can include model based optimization, e.g., Markov Decision Processes and Model Predictive Control, if the system dynamic is known (or estimated), and model free optimization when the model is not available, e.g., Reinforcement Learning.

Cross-level Optimization. In this thesis, we have only considered the optimization of the application deployment, without performing optimization at the infrastructure level. Although simple and independent heuristics can be combined to acquire and release computing resources while changing the application deployment, we believe that a cross-level optimization can produce better adaptation strategies (e.g., [134]). Recently, the Cloud computing technologies made easy to acquire and release computing resources, whereas SDN simplified the management of large scale networks. Both these technologies enable the infrastructure to be programmable, thus allowing its runtime reconfiguration so as to better satisfy the application needs. Interestingly, SDN provides mechanisms for allocating network capacity for data flows and also enables a fine-grained control of the application deployment. Relying on SDN, a DSP application can determine the network paths that better satisfy the application QoS requirements (e.g., bandwidth, response time). Moreover, it allows to react to network congestion by re-routing data streams, i.e., without changing the operators placement or replication. We believe that the joint optimization of the application layer and the infrastructure layer represents another interesting research direction.

References

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A data stream management system. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 666–666. ACM, 2003.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research*, volume 5 of *CIDR 2005*, pages 277–289, 2005.

[3] B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran. Streamline: scheduling streaming applications in a wide area environment. *Multimedia Systems*, 13(1):69–85, Sep 2007.

[4] C. C. Aggarwal. *Data streams: models and algorithms*, volume 31. Springer Science & Business Media, 2007.

[5] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the 30th International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 456–467. VLDB Endowment, 2004.

[6] 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. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, Aug. 2013.

[7] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, Aug. 2015.

[8] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, Dec 2014.

[9] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in Storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218. ACM, 2013.

[10] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665. ACM, 2003.

[11] H. R. Arkian, A. Diyanat, and A. Pourkhalili. Mist: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowdsensing applications. *Journal of Network and Computer Applications*, 82:152–165, 2017.

[12] A. Artikis, M. Weidlich, F. Schnitzler, I. Boutsis, T. Liebig, N. Piatkowski, C. Bockermann, K. Morik, V. Kalogeraki, J. Marecek, et al. Heterogeneous stream processing and crowdsourcing for urban traffic management. In *Proceedings of 17th International Conference on Extending Database Technology*, EDBT'14, 2014.

[13] A. Avižienis, J.-C. Laprie, and B. Randell. Dependability and its threats: A taxonomy. In *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*, pages 91–120. Springer US, 2004.

[14] N. Backman, R. Fonseca, and U. Çetintemel. Managing parallelism for stream processing in the cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 1:1–1:5. ACM, 2012.

[15] C. Ballard, K. Foster, A. Frenkiel, B. Gedik, M. P. Koranda, S. Nathan, D. Rajan, R. Rea, M. Spicer, B. Williams, et al. *IBM Infosphere streams: Assembling continuous insight in the information revolution*. IBM Redbooks, 2012.

[16] R. S. Barga and H. Caituiro-Monge. Event correlation and pattern detection in CEDR. In *Current Trends in Database Technology - EDBT 2006*, pages 919–930. Springer Berlin Heidelberg, 2006.

[17] P. Basanta-Val, N. Fernández-García, A. Wellings, and N. Audsley. Improving the predictability of distributed stream processors. *Future Generation Computer Systems*, 52(Supplement C):22 – 36, 2015.

[18] P. Bellavista, A. Corradi, S. Kotoulas, and A. Reale. Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. In *Proceedings of the 17th International Conference on Extending Database Technology*, EDBT '14, pages 85–96, 2014.

[19] A. Benoit, H. Casanova, V. Rehn-Sonigo, and Y. Robert. Resource allocation for multiple concurrent in-network stream-processing applications. *Parallel Computing*, 37(8):331–348, 2011.

[20] M. Bilal and M. Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC'17. ACM, 2017.

[21] B. J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. *Telecommunication Systems*, 26(2):389–409, June 2004.

[22] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer International Publishing, 2014.

[23] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the 1st Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16. ACM, 2012.

[24] A. Brook. Low-latency distributed applications in finance. *Commun. ACM*, 58(7):42–50, June 2015.

[25] M. Caneill, A. El Rheddane, V. Leroy, and N. De Palma. Locality-aware routing in stateful streaming applications. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 4:1–4:13. ACM, 2016.

[26] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink. In *Proceedings of the 43rd International Conference on Very Large Data Bases - Volume 10*, volume 10 of *VLDB '17*, pages 1718–1729. VLDB Endowment, 2017.

[27] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, et al. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[28] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Distributed QoS-aware scheduling in Storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 344–347. ACM, 2015.

[29] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 69–80. ACM, 2016.

[30] V. Cardellini, M. Nardelli, and D. Luzi. Elastic stateful stream processing in Storm. In *Proceedings of the 2016 International Conference on High Performance Computing Simulation*, HPCS 2016, pages 583–590, July 2016.

[31] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736. ACM, 2013.

[32] J. Cerviño, E. Kalyvianaki, J. Salvachua, and P. Pietzuch. Adaptive provisioning of stream processing systems in the cloud. In *Proceedings of the 28th IEEE International Conference on Data Engineering Workshops*, ICDEW 2012, pages 295–301, Apr. 2012.

[33] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668. ACM, 2003.

[34] A. Chatzistergiou and S. D. Viglas. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 1579–1588. ACM, 2014.

[35] D. Cheng, Y. Chen, X. Zhou, D. Gmach, and D. Milojicic. Adaptive scheduling of parallel jobs in spark streaming. In *Proceedings of the 36th IEEE Conference on Computer Communications*, INFOCOM 2017, pages 1–9, May 2017.

[36] S. Chintapalli, D. Dagit, R. Evans, R. Farivar, Z. Liu, K. Nusbaum, K. Patil, and B. Peng. PaceMaker: When ZooKeeper arteries get clogged in Storm clusters. In *Proceedings of the 9th IEEE International Conference on Cloud Computing*, CLOUD 2016, pages 448–455, June 2016.

[37] Cisco Visual Networking Index. The zettabyte era: Trends and analysis. *Cisco White Paper*, 2017.

[38] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for np-hard problems. chapter Approximation Algorithms for Bin Packing: A Survey, pages 46–93. PWS Publishing Co., 1997.

[39] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 647–651. ACM, 2003.

[40] G. Cugola and A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012.

[41] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3):15:1–15:62, June 2012.

[42] L. Cui, F. R. Yu, and Q. Yan. When big data meets software-defined networking: SDN for big data and big data for SDN. *IEEE Network*, 30(1):58–65, Jan. 2016.

[43] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, Aug. 2004.

[44] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13. ACM, 2014.

[45] T. De Matteis. *Parallel Patterns for Adaptive Data Stream Processing*. PhD thesis, University of Pisa, Italy, 2016.

[46] T. De Matteis and G. Mencagli. Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 13:1–13:12. ACM, 2016.

[47] T. De Matteis and G. Mencagli. Elastic scaling for distributed latency-sensitive data stream operators. In *Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP 2017, pages 61–68, Mar. 2017.

[48] T. De Matteis and G. Mencagli. Proactive elasticity and energy awareness in data stream processing. *Journal of Systems and Software*, 127:302–319, 2017.

[49] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[50] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing*, SoCC '15, pages 97–110. ACM, 2015.

[51] J. Ding, T. Z. J. Fu, R. T. B. Ma, M. Winslett, Y. Yang, et al. Optimal operator state migration for elastic data stream processing. *CoRR*, abs/1501.03619, 2015.

[52] G. Du and I. Gupta. New techniques to curtail the tail latency in stream processing systems. In *Proceedings of the 4th Workshop on Distributed Cloud Computing*, DCC '16, pages 7:1–7:6. ACM, 2016.

[53] R. Eidenbenz and T. Locher. Task allocation for distributed stream processing. In *Proceedings of the 35th IEEE International Conference on Computer Communications*, INFOCOM 2016, pages 1–9, Apr. 2016.

[54] L. Eskandari, Z. Huang, and D. Eyers. P-scheduler: Adaptive hierarchical scheduling in Apache Storm. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '16, pages 26:1–26:10. ACM, 2016.

[55] G. Falcone, G. Nicosia, and A. Pacifici. Minimizing part transfer costs in flexible manufacturing systems: A computational study on different lower bounds. In *Proceedings of the 15th UKSim International Conference on Computer Modelling and Simulation*, pages 359–364, Apr. 2013.

[56] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu. Parallel stream processing against workload skewness and variance. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 15–26. ACM, 2017.

[57] L. Fischer and A. Bernstein. Workload scheduling in distributed stream processors using graph partitioning. In *Proceedings of the 2015 IEEE International Conference on Big Data*, Big Data 2015, pages 124–133, Oct. 2015.

[58] L. Fischer, T. Scharrenbach, and A. Bernstein. Network-aware workload scheduling for scalable linked data stream processing. In *Proceedings of the 12th International Semantic Web Conference (Posters & Demonstrations Track)*, ISWC-PD '13, pages 281–284, 2013.

[59] A. Floratou and A. Agrawal. Self-regulating streaming systems: Challenges and opportunities. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, BIRTE '17, pages 1:1–1:5. ACM, 2017.

[60] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. In *Proceedings of the VLDB Endowment, Vol. 10, No. 12*, pages 1825–1836, Aug. 2017.

[61] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. DRS: Auto-scaling for real-time stream analytics. *IEEE/ACM Transactions on Networking*, PP(99):1–15, 2017.

[62] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, 1979.

[63] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539, Aug. 2014.

[64] B. Gedik, H. Özsema, and O. Öztürk. Pipelined fission for stream programs with dynamic selectivity and partitioned state. *Journal of Parallel and Distributed Computing*, 96:106–120, 2016.

[65] B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, June 2014.

[66] J. Ghaderi, S. Shakkottai, and R. Srikant. Scheduling Storms and streams in the Cloud. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 1(4):14:1–14:28, Aug. 2016.

[67] L. Gu, D. Zeng, S. Guo, Y. Xiang, and J. Hu. A general communication cost optimization framework for big data stream processing in geo-distributed data centers. *IEEE Transactions on Computers*, 65(1):19–29, Jan. 2016.

[68] X. Gu, P. S. Yu, and K. Nahrstedt. Optimal component composition for scalable stream processing. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS 2005, pages 773–782, June 2005.

[69] Y. Gu and C. Q. Wu. Performance analysis and optimization of distributed workflows in heterogeneous network environments. *IEEE Transactions on Computers*, 65(4):1266–1282, Apr. 2016.

[70] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. Streamcloud: A large scale data streaming system. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS 2010, pages 126–137, June 2010.

[71] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, Dec. 2012.

[72] Q. Guo and Y. Zhou. CBP: A new parallelization paradigm for massively distributed stream processing. In *Database Systems for Advanced Applications*, DASFAA 2017, pages 304–320. Springer International Publishing, Cham, 2017.

[73] Z. Han, R. Chu, H. Mi, and H. Wang. Elastic allocator: An adaptive task scheduler for streaming query in the cloud. In *Proceedings of the 8th IEEE International Symposium on Service Oriented System Engineering*, SOSE 2014, pages 284–289, Apr. 2014.

[74] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni. Securestreams: A reactive middleware framework for secure data stream processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 124–133. ACM, 2017.

[75] B. Heintz, A. Chandra, and R. K. Sitaraman. Optimizing grouped aggregation in geo-distributed streaming analytics. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 133–144. ACM, 2015.

[76] B. Heintz, A. Chandra, and R. K. Sitaraman. Optimizing timeliness and cost in geo-distributed streaming analytics. *IEEE Transactions on Cloud Computing*, PP(99):1–1, 2017.

[77] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak. Cloud-based data stream processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 238–245. ACM, 2014.

[78] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22. ACM, 2014.

[79] T. Heinze, Y. Ji, Y. Pan, F. J. Grueneberger, Z. Jerzak, and C. Fetzer. Elastic complex event processing under varying query load. In *Proceedings of the 1st International Workshop on Big Dynamic Distributed Data*, BD3 2013, pages 25–31, 2013.

[80] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *Proceedings of the 30th*

IEEE International Conference on Data Engineering Workshops, ICDEW 2014, pages 296–302, Mar. 2014.

[81] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online parameter optimization for elastic data stream processing. In *Proceedings of the 6th ACM Symposium on Cloud Computing*, SoCC '15, pages 276–287. ACM, 2015.

[82] T. Heinze, M. Zia, R. Krahn, Z. Jerzak, and C. Fetzer. An adaptive replication scheme for elastic data stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 150–161. ACM, 2015.

[83] N. Hidalgo, D. Wladdimiro, and E. Rosas. Self-adaptive processing graph with operator fission for elastic stream processing. *Journal of Systems and Software*, 127:205–216, 2017.

[84] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. Technical Report UCB/EECS-2010-87, EECS Department, University of California, Berkeley, May 2010.

[85] B. Hindman, A. Konwinski, M. Zaharia, and I. Stoica. A common substrate for cluster computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09. USENIX Association, 2009.

[86] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):46:1–46:34, Mar. 2014.

[87] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar. Cost-efficient enactment of stream processing topologies. *PeerJ Computer Science*, 3:e141, Dec. 2017.

[88] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar. Elastic stream processing for the internet of things. In *Proceedings of the 9th IEEE International Conference on Cloud Computing*, CLOUD 2016, pages 100–107, June 2016.

[89] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, pages 15–20. ACM, 2013.

[90] M. Hoseiny Farahabady, A. Y. Zomaya, and Z. Tari. QoS- and contention- aware resource provisioning in a stream processing engine. In *Proceedings of the 2017 IEEE International Conference on Cluster Computing*, CLUSTER 2017, pages 137–146, Sept. 2017.

[91] M. R. Hoseiny Farahabady, H. R. D. Samani, Y. Wang, A. Y. Zomaya, and Z. Tari. A QoS-aware controller for Apache Storm. In *Proceeding of the 15th IEEE International Symposium on Network Computing and Applications*, NCA '16, pages 334–342, Oct. 2016.

[92] L. Hu, K. Schwan, H. Amur, and X. Chen. Elf: Efficient lightweight fast stream processing at scale. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 25–36. USENIX, 2014.

[93] J. Huang, G. Liu, Q. Duan, and Y. Yan. QoS-aware service composition for converged network-cloud service provisioning. In *Proceedings of the 2014 IEEE International Conference on Services Computing*, SCC 2014, pages 67–74, June 2014.

[94] Y. Huang, Z. Luan, R. He, and D. Qian. Operator placement with QoS constraints for distributed stream processing. In *Proceedings of the 7th International Conference on Network and Service Management*, CNSM '11, pages 1–7, Oct. 2011.

[95] W. Hummer, P. Leitner, B. Satzger, and S. Dustdar. Dynamic migration of processing elements for optimized query execution in event-based systems. In *On the Move to Meaningful Internet Systems*, OTM 2011, pages 451–468. Springer Berlin Heidelberg, 2011.

[96] J. H. Hwang, U. Cetintemel, and S. Zdonik. Fast and reliable stream processing over wide area networks. In *Proceedings of the 23rd IEEE International Conference on Data Engineering Workshop*, ICDEW 2007, pages 604–613, Apr. 2007.

[97] A. Ishii and T. Suzumura. Elastic stream computing with clouds. In *Proceedings of the 4th IEEE International Conference on Cloud Computing*, CLOUD 2011, pages 195–202, July 2011.

[98] Z. Jerzak and H. Ziekow. The DEBS 2014 grand challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 266–269. ACM, 2014.

[99] Z. Jerzak and H. Ziekow. The DEBS 2015 grand challenge. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 266–268. ACM, 2015.

[100] J. Jiang, Z. Zhang, B. Cui, Y. Tong, and N. Xu. StroMAX: Partitioning-based scheduler for real-time stream processing system. In *Database Systems for Advanced Applications*, volume 10178 of *DASFAA 2017*, pages 269–288. Springer, 2017.

[101] E. Kalyvianaki, T. Charalambous, M. Fiscato, and P. Pietzuch. Overload management in data stream processing systems with latency guarantees. In *Proceedings of the 7th IEEE International Workshop on Feedback Computing*, Feedback Computing '12, 2012.

[102] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch. Themis: Fairness in federated stream processing under overload. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 541–553. ACM, 2016.

[103] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. SQPR: Stream query planning with reuse. In *Proceedings of the 27th IEEE International Conference on Data Engineering*, ICDE 2011, pages 840–851, Apr. 2011.

[104] Y. H. Kao, B. Krishnamachari, M. R. Ra, and F. Bai. Hermes: Latency optimal task assignment for resource-constrained mobile computing. *IEEE Transactions on Mobile Computing*, PP(99):1–1, 2017.

[105] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. *Proceedings of the VLDB Endowment*, 10(11):1286–1297, Aug. 2017.

[106] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.

[107] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. Cola: Optimizing stream processing applications via graph partitioning. In *Middleware 2009: ACM/IFIP/USENIX, 10th International Middleware Conference*, pages 308–327. Springer Berlin Heidelberg, 2009.

[108] A. Khorlin and K. M. Chandy. Control-based scheduling in a distributed stream processing system. In *Proceedings of the 2006 IEEE Services Computing Workshops*, SCW 2006, pages 55–64, Sept. 2006.

[109] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch. Balancing load in stream processing with the cloud. In *Proceedings of the 27th IEEE International Conference on Data Engineering Workshops*, ICDEW 2011, pages 16–21, Apr. 2011.

[110] M. Kleppmann and J. Kreps. Kafka, samza and the unix philosophy of distributed data. *IEEE Data Engineering Bulletin*, 38(4):4–14, 2015.

[111] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. SABER: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 555–569. ACM, 2016.

[112] R. K. Kombi, N. Lumineau, and P. Lamarre. A preventive auto-parallelization approach for elastic stream processing. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*, pages 1532–1542, June 2017.

[113] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250. ACM, 2015.

[114] V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Proceedings of the 2nd International Conference on Autonomic Computing*, ICAC'05, pages 3–14, June 2005.

[115] A. Kumbhare, M. Frincu, Y. Simmhan, and V. K. Prasanna. Fault-tolerant and elastic streaming mapreduce with decentralized coordination. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, ICDCS 2015, pages 328–338, June 2015.

[116] A. G. Kumbhare, Y. Simmhan, M. Frincu, and V. K. Prasanna. Reactive resource provisioning heuristics for dynamic dataflows on cloud infrastructure. *IEEE Transactions on Cloud Computing*, 3(2):105–118, Apr. 2015.

[117] G. T. Lakshmanan, Y. Li, and R. Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, Nov. 2008.

[118] G. T. Lakshmanan, Y. Li, and R. Strom. Placement of replicated tasks for distributed stream processing systems. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 128–139. ACM, 2010.

[119] J. C. Laprie. *Dependability: Basic Concepts and Terminology*, pages 3–245. Springer Vienna, 1992.

[120] C. C. Lee. Fuzzy logic in control systems: fuzzy logic controller. i. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(2):404–418, Mar. 1990.

[121] H. Li, J. Wu, Z. Jiang, X. Li, and X. Wei. Task allocation for stream processing with recovery latency guarantee. In *Proceedings of the 2017 IEEE International Conference on Cluster Computing*, CLUSTER 2017, pages 379–383, Sept. 2017.

[122] J. Li, A. Deshpande, and S. Khuller. Minimizing communication cost in distributed multi-query processing. In *Proceedings of the 25th IEEE International Conference on Data Engineering*, ICDE 2009, pages 772–783, Mar. 2009.

[123] J. Li, C. Pu, Y. Chen, D. Gmach, and D. Milojicic. Enabling elastic stream processing in shared clusters. In *Proceedings of the 9th IEEE International Conference on Cloud Computing*, CLOUD 2016, pages 108–115, June 2016.

[124] T. Li, J. Tang, and J. Xu. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data*, 2(4):353–364, Dec. 2016.

[125] W. Li, D. Niu, Y. Liu, S. Liu, and B. Li. Wide-area Spark Streaming: Automated routing and batch sizing. In *Proceedings of the 2017 IEEE International Conference on Autonomic Computing*, ICAC 2017, pages 33–38, July 2017.

[126] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. StreamScope: Continuous reliable distributed processing of big data streams. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 439–453, 2016.

[127] X. Liu and R. Buyya. Performance-oriented deployment of streaming applications on cloud. *IEEE Transactions on Big Data*, PP(99):1–1, 2017.

[128] X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, and R. Buya. A step-wise auto-profiling method for performance optimization of streaming applications. *ACM Transactions on Autonomous and Adaptive Systems*, 12(4):24:1–24:33, Nov. 2017.

[129] X. Liu, A. Harwood, S. Karunasekera, B. Rubinstein, and R. Buyya. E-Storm: Replication-based state management in distributed stream processing systems. In *Proceedings of the 46th International Conference on Parallel Processing*, ICPP 2017, pages 571–580, Aug. 2017.

[130] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, Mar. 1982.

[131] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann. Stormy: An elastic and highly available streaming service in the cloud. In *Proceed-*

ings of the 2012 Joint EDBT/ICDT Workshops, EDBT-ICDT '12, pages 55–60. ACM, 2012.

[132] B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, ICDCS 2015, pages 399–410, June 2015.

[133] B. Lohrmann, D. Warneke, and O. Kao. Massively-parallel stream processing under QoS constraints with nephele. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 271–282. ACM, 2012.

[134] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2017.

[135] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, Dec 2014.

[136] K. G. S. Madsen, P. Thyssen, and Y. Zhou. Integrating fault-tolerance and elasticity in a distributed data stream processing system. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, SSDBM '14, pages 48:1–48:4. ACM, 2014.

[137] K. G. S. Madsen and Y. Zhou. Dynamic resource management in a massively parallel stream processing engine. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 13–22. ACM, 2015.

[138] K. G. S. Madsen, Y. Zhou, and J. Cao. Integrative dynamic reconfiguration in a parallel stream processing engine. In *Proceedings of the 33rd IEEE International Conference on Data Engineering*, ICDE 2017, pages 227–230, Apr. 2017.

[139] K. G. S. Madsen, Y. Zhou, and L. Su. Enorm: Efficient window-based computation in large-scale distributed stream processing systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 37–48. ACM, 2016.

[140] A. Martin, A. Brito, and C. Fetzer. Scalable and elastic realtime click stream analysis using streammine3g. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 198–205. ACM, 2014.

[141] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., 1st edition, 2015.

[142] R. Mayer, M. A. Tariq, and K. Rothermel. Minimizing communication overhead in window-based parallel complex event processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 54–65. ACM, 2017.

[143] M. D. Mcllroy. Coroutines: Semantics in search of a syntax. *Unpublished manuscript*, 1968.

[144] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: an approach to universal topology generation. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS 2001, pages 346–353, 2001.

[145] G. Mencagli. A game-theoretic approach for elastic distributed data stream processing. *ACM Transactions on Autonomous and Adaptive Systems*, 11(2):13:1–13:34, June 2016.

[146] G. Mencagli, M. Torquati, and M. Danelutto. Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Generation Computer Systems*, 79(Part 3):862–877, 2018.

[147] M. Migliavacca, D. Eyers, J. Bacon, Y. Papagiannis, B. Shand, and P. Pietzuch. SEEP: Scalable and elastic event processing. In *Middleware '10 Posters and Demos Track*, pages 4:1–4:2. ACM, 2010.

[148] S. M. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

[149] C. Mutschler, H. Ziekow, and Z. Jerzak. The DEBS 2013 grand challenge. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 289–294. ACM, 2013.

[150] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *Proceedings of the 31st IEEE International Conference on Data Engineering*, ICDE 2015, pages 137–148, Apr. 2015.

[151] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW 2010, pages 170–177, Dec. 2010.

[152] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream pro-

cessing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, Aug. 2017.

[153] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun, and U. Ramachandran. MCEP: A mobility-aware complex event processing system. *ACM Transactions on Internet Technology*, 14(1):6:1–6:24, Aug. 2014.

[154] O. Papaemmanouil, U. Çetintemel, and J. Jannotti. Supporting generic cost models for wide-area stream processing. In *Proceedings of the 25th IEEE International Conference on Data Engineering*, ICDE 2009, pages 1084–1095, Mar. 2009.

[155] Y. Park, R. King, S. Nathan, W. Most, and H. Andrade. Evaluation of a high-volume, low-latency market data processing system implemented with IBM middleware. *Software: Practice and Experience*, 42(1):37–56, 2012.

[156] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell. R-Storm: Resource-aware scheduling in Storm. In *Proceedings of the 16th Middleware Conference*, Middleware '15, pages 149–161. ACM, 2015.

[157] T. N. Pham, N. R. Katsipoulakis, P. K. Chrysanthis, and A. Labrinidis. Uninterruptible migration of continuous queries without operator state migration. *SIGMOD Rec.*, 46(3):17–22, Oct. 2017.

[158] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE'06, pages 49–49, Apr. 2006.

[159] N. Pollner, C. Steudtner, and K. Meyer-Wegener. Operator fission for load balancing in distributed heterogeneous data stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 332–335. ACM, 2015.

[160] N. Pollner, C. Steudtner, and K. Meyer-Wegener. Placement-safe operator-graph changes in distributed heterogeneous data stream systems. *Datenbank-Spektrum*, 15(3):203–211, Nov 2015.

[161] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy. Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE Network*, 17(6):27–35, Nov. 2003.

[162] M. Pundir, M. Kumar, L. M. Leslie, I. Gupta, and R. H. Campbell. Supporting on-demand elasticity in distributed graph processing. In *Proceedings of the 2016 IEEE International Conference on Cloud Engineering*, IC2E 2016, pages 12–21, Apr. 2016.

[163] W. Qian, Q. Shen, J. Qin, D. Yang, Y. Yang, and Z. Wu. S-Storm: A slot-aware scheduling strategy for even scheduler in Storm. In *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems*, HPCC/SmartCity/DSS, pages 623–630, Dec. 2016.

[164] 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 *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14. ACM, 2013.

[165] 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 *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14. ACM, 2013.

[166] C. Qin and H. Eichelberger. Impact-minimizing runtime switching of distributed stream processing algorithms. In *EDBT/ICDT Workshops*, 2016.

[167] S. Ravindra, M. Dayarathna, and S. Jayasena. Latency aware elastic switching-based stream processing over compressed data streams. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 91–102. ACM, 2017.

[168] T. Repantis, X. Gu, and V. Kalogeraki. QoS-aware shared component composition for distributed stream processing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(7):968–982, July 2009.

[169] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola. Online scheduling for shuffle grouping in distributed stream processing systems. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 11:1–11:12. ACM, 2016.

[170] S. Rizou, F. Durr, and K. Rothermel. Providing QoS guarantees in large-scale operator networks. In *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, HPCC 2010, pages 337–345, Sept. 2010.

[171] S. Rizou, F. Durr, and K. Rothermel. Solving the multi-operator placement problem in large-scale operator networks. In *Proceedings of 19th International Conference on Computer Communications and Networks*, ICCCN 2010, pages 1–6, Aug. 2010.

[172] S. Rizou, F. Durr, and K. Rothermel. Fulfilling end-to-end latency constraints in large-scale streaming environments. In *Proceedings of*

the 30th IEEE International Performance Computing and Communications Conference, PCCC 2011, pages 1–8, Nov. 2011.

[173] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. CAPE: Continuous query engine with heterogeneous-grained adaptivity. In *Proceedings of the 30th International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1353–1356. VLDB Endowment, 2004.

[174] M. Rychly, P. Koda, and P. Mr. Scheduling decisions in stream processing on heterogeneous clusters. In *Proceedings of the 8th International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS 2014, pages 614–619, July 2014.

[175] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov. SpanEdge: Towards unifying stream processing over central and near-the-edge data centers. In *Proceedings of the 2016 IEEE/ACM Symposium on Edge Computing*, SEC 2016, pages 168–178, Oct. 2016.

[176] M. Satyanarayanan, R. Schuster, M. Ebling, G. Fettweis, H. Flinck, K. Joshi, and K. Sabnani. An open ecosystem for mobile-cloud convergence. *IEEE Communications Magazine*, 53(3):63–70, Mar. 2015.

[177] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Proceedings of the 4th IEEE International Conference on Cloud Computing*, CLOUD 2011, pages 348–355, July 2011.

[178] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 258–269. ACM, 2016.

[179] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. L. Wu. Elastic scaling of data parallel operators in stream processing. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS 2009, pages 1–12, May 2009.

[180] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 53–64. ACM, 2012.

[181] S. Schneider, M. Hirzel, B. Gedik, and K. L. Wu. Safe data parallelism for general streaming. *IEEE Transactions on Computers*, 64(2):504–517, Feb. 2015.

[182] S. Schneider, J. Wolf, K. Hildrum, R. Khandekar, and K.-L. Wu. Dynamic load balancing for ordered data-parallel regions in distributed streaming systems. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 21:1–21:14. ACM, 2016.

[183] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12. ACM, 2009.

[184] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio. Control-theoretical software adaptation: A systematic literature review. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[185] A. Shukla and Y. Simmhan. Model-driven scheduling for distributed stream processing systems. *arXiv preprint arXiv:1702.01785*, 2017.

[186] P. Smirnov, M. Melnik, and D. Nasonov. Performance-aware scheduling of streaming applications using genetic algorithm. *Procedia Computer Science*, 108:2240–2249, 2017.

[187] D. Sow, A. Biem, M. Blount, M. Ebling, and O. Verscheure. Body sensor data processing using stream computing. In *Proceedings of the International Conference on Multimedia Information Retrieval*, MIR '10, pages 449–458. ACM, 2010.

[188] I. Stanoi, G. Mihaila, T. Palpanas, and C. Lang. Whitewater: Distributed processing of fast streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(9):1214–1226, Sept. 2007.

[189] F. Starks, V. Goebel, S. Kristiansen, and T. Plagemann. Mobile distributed complex event processing—ubi sumus? quo vadimus? In *Mobile Big Data: A Roadmap from Models to Technologies*, pages 147–180. Springer, 2018.

[190] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, Dec. 2005.

[191] D. Sun and R. Huang. A stable online scheduling strategy for real-time stream computing over fluctuating big data streams. *IEEE Access*, 4:8593–8607, 2016.

[192] D. Sun, H. Yan, S. Gao, X. Liu, and R. Buyya. Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams. *The Journal of Supercomputing*, Sep 2017.

[193] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[194] T. Tan, R. T. Ma, M. Winslett, Y. Yang, Y. Yu, and Z. Zhang. Resa: Realtime elastic streaming analytics in the cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1287–1288. ACM, 2013.

[195] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. On the use of hybrid reinforcement learning for autonomic resource allocation. *Cluster Computing*, 10(3):287–299, 2007.

[196] C. Thoma, A. Labrinidis, and A. J. Lee. Automated operator placement in distributed data stream management systems subject to user constraints. In *Proceedings of the 30th IEEE International Conference on Data Engineering Workshops*, ICDEW 2014, pages 310–316, Mar. 2014.

[197] L. Tian and K. M. Chandy. Resource allocation in streaming environments. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, ICGRID 2006, pages 270–277, Sept. 2006.

[198] Q. To, J. Soto, and V. Markl. A survey of state management in big data processing systems. *CoRR*, abs/1702.01596, 2017.

[199] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156. ACM, 2014.

[200] R. Tudoran, O. Nano, I. Santos, A. Costan, H. Soncu, L. Bougé, and G. Antoniu. Jetstream: Enabling high performance event streaming across cloud data-centers. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 23–34. ACM, 2014.

[201] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal. Ajira: A lightweight distributed middleware for mapreduce and stream processing. In *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems*, ICDCS 2014, pages 545–554, June 2014.

[202] J. S. van der Veen, B. van der Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer. Dynamically scaling Apache Storm for the analysis of streaming data. In *Proceedings of the 1st IEEE International Conference on Big Data Computing Service and Applications*, pages 154–161, Mar. 2015.

[203] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache

hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16. ACM, 2013.

[204] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389. ACM, 2017.

[205] C. Wang, X. Meng, Q. Guo, Z. Weng, and C. Yang. Orientstream: A framework for dynamic resource allocation in distributed data stream management systems. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pages 2281–2286. ACM, 2016.

[206] C. Wang, X. Meng, Q. Guo, Z. Weng, and C. Yang. Automating characterization deployment in distributed data stream management systems. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2669–2681, Dec. 2017.

[207] B. M. Waxman. Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622, Dec. 1988.

[208] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka. On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, pages 76–107. Springer Berlin Heidelberg, 2013.

[209] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware 2008: ACM/IFIP/USENIX 9th International Middleware Conference*, pages 306–325. Springer Berlin Heidelberg, 2008.

[210] Y. Wu and K. L. Tan. ChronoStream: Elastic stateful stream computation in the cloud. In *Proceedings of the 31st IEEE International Conference on Data Engineering*, ICDE 2015, pages 723–734, Apr. 2015.

[211] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 775–786. VLDB Endowment, 2006.

[212] Y. Xing, S. Zdonik, and J. H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE'05, pages 791–802, Apr. 2005.

[213] J. Xu, Z. Chen, J. Tang, and S. Su. T-Storm: Traffic-aware online scheduling in Storm. In *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems*, ICDCS 2014, pages 535–544, June 2014.

[214] L. Xu, B. Peng, and I. Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *Proceedings of the 2016 IEEE International Conference on Cloud Engineering*, IC2E 2016, pages 22–31, Apr. 2016.

[215] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *SIGMETRICS Perform. Eval. Rev.*, 40(4):23–32, Apr. 2013.

[216] M. Yang and R. T. Ma. Smooth task migration in Apache Storm. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 2067–2068. ACM, 2015.

[217] S. Yang. IoT stream processing and analytics in the fog. *IEEE Communications Magazine*, 55(8):21–27, 2017.

[218] K. P. Yoon and C.-L. Hwang. *Multiple Attribute Decision Making: an Introduction*, volume 104. Sage Pubns, 1995.

[219] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos. Elastic complex event processing exploiting prediction. In *Proceedings of the 2015 IEEE International Conference on Big Data*, Big Data 2015, pages 213–222, Oct. 2015.

[220] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10. USENIX Association, 2010.

[221] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438. ACM, 2013.

[222] J. Zhang, C. Li, L. Zhu, and Y. Liu. The real-time scheduling strategy based on traffic and load balancing in Storm. In *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems*, HPCC/SmartCity/DSS, pages 372–379, Dec. 2016.

[223] Q. Zhang, Y. Song, R. R. Routray, and W. Shi. Adaptive block and batch sizing for batched stream processing system. In *Proceedings of the 2016 IEEE International Conference on Autonomic Computing*, ICAC 2016, pages 35–44, July 2016.

[224] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *On the Move to Meaningful Internet Systems*, OTM 2006, pages 54–71. Springer Berlin Heidelberg, 2006.

[225] Y. Zhou, J. Wu, and A. K. Leghari. Multi-query scheduling for time-critical data stream applications. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 15:1–15:12. ACM, 2013.

[226] Q. Zhu and G. Agrawal. Resource allocation for distributed streaming applications. In *Proceedings of the 37th International Conference on Parallel Processing*, ICPP 2008, pages 414–421, Sept. 2008.