On the design of a framework integrating an optimization engine with streaming technologies

Cristóbal Barba-González *, Antonio J. Nebro, Antonio Benítez-Hidalgo, José García-Nieto, José F. Aldana-Montes

Departamento de Lenguajes y Ciencias de la Computación, Ada Byron Research Building, University of Málaga, 29071 Málaga, Spain

Keywords:

Dynamic multi-objective optimization
Streaming data processing
Big data
Software framework

A B S T R A C T

A number of streaming technologies have appeared in the last years as a result of the rising of Big Data applications. Nowadays, deciding which technology to adopt is not an easy task due not only to the number of available data streaming processing projects, but also because they are continuously evolving. In this paper, we focus on how these issues have affected jMetalSP, a framework for dynamic multi-objective optimization that incorporates streaming features. jMetalSP allows the development of three tier optimization workflows where the central component is an optimizer that is continuously solving a dynamic multi-objective optimization problem. This problem can change as a consequence of the analysis of data streams carried out by components that use the Apache Spark streaming engine. A third kind of components receive and process the Pareto front approximations being yielded by the optimization algorithm. However, all jMetalSP elements are tightly coupled and linked to Spark, making it difficult to use a different streaming system. To overcome this issue, we have redesigned the jMetalSP architecture to make it flexible enough to avoid the dependence of any particular streaming system. This way, popular Apache projects such as Spark Structured Streaming, Kafka Streams, or Flink can be used without requiring to change the rest of components of the application. Furthermore, Kafka can be used for inter-process communication, what enables the execution of components in different nodes of a cluster, independently of their implementation languages thanks to the serialization of data streams with Apache Avro. We show how the embraced solution provides a high degree of flexibility that enhances the usability of jMetalSP. To this end, a representative case study based on a transport problem is conducted that focuses on data representation and performance evaluation of the Spark, Flink, and Kafka systems.

1. Introduction

At present, Big Data has become a consolidated field in which a number of technologies are appearing and evolving quickly. This is the case, in particular, of streaming data processing tools, which are used in applications featuring some of the V's characterizing Big Data applications, such as volume, velocity, or variety. Examples of streaming engines include the Apache projects Spark [1], Flink [2], Kafka [3], Samza [4], Beam [5], Storm [6], and Apex [7].

Our focus in this paper is related to the application of streaming data processing in the context of an optimization framework. Let us consider route planning, which is a typical problem in dynamic optimization. To illustrate this problem, we can imagine an scenario in logistics where multiple product delivery routes have to be periodically recalculated during a working day using an application for this purpose. Initially, the routes to be followed by the vehicles are pre-calculated, but traffic conditions may change unexpectedly and these routes have to be readjusted to optimize cost, distances, or times. In addition, this application is continuously obtaining information from different data sources (sensors, cameras, social networks, etc.) about possible incidents (accidents, traffic jams, maintenance works), so it should be able to react and provide recalculated routes taking into account the new conditions that affect the initial planning. In this context, the generation of efficient frameworks for dynamic (online) optimization, incorporating advanced streaming processing capabilities, remains an open challenge which is what drives us in this study.

If we consider Apache Spark, which is probably the most popular project in this sense, we find that new versions appear every few months, not just concerning small upgrades in the basic library based on RDDs (Resilient Distributed Datasets), but

* Corresponding author.
E-mail addresses: cbarba@lcc.uma.es (C. Barba-González), antonio@lcc.uma.es (A.J. Nebro), antonio.benitez@lcc.uma.es (A. Benítez-Hidalgo), jnieto@lcc.uma.es (J. García-Nieto), jfam@lcc.uma.es (J.F. Aldana-Montes).

also adopting new engines for both streaming (denoted by *structured streaming*) and machine learning (based on the concept of dataframes that supplement the existing RDD-based ones). This could lead to situations in which some applications using Apache Spark are not prepared for its continuous evolution. Furthermore, the choice of a particular streaming technology can overwhelm the user, given the amount of available similar projects like the aforementioned ones. If an application implementing a given streaming project is too tight to it and another alternative technology could be more appropriated in a given context, substituting the streaming component by a different one can be a complex task.

In this paper, we describe and analyze how these issues have affected the jMetalSP [8] framework and how we have evolved its design to address them. The motivation is also to provide design and implementation experiences on the combination of dynamic optimization procedures with streaming processing, thus allowing to address new scenarios effectively.

The jMetalSP project was started as an attempt to combine a multi-objective optimization framework, jMetal [9], with the streaming engine of Apache Spark. The motivation was related to the fact that many real-world problems from different domains (e.g., engineering, economy, logistics, transportation, etc.) can be formulated as a multi-objective optimization problem (i.e., requiring to optimize two or more conflicting objectives at the same time), which can be solved with non-exact techniques such as metaheuristics [10]. Furthermore, as streaming technologies have become very popular, it was foreseeable that multi-objective optimization problems would arise in connection with the processing of streaming data in the context of Big Data. jMetalSP was born with the goal of offering a tool for dealing with dynamic multi-objective problems whose objectives, constraints and parameters can change over time.

The first version of jMetalSP revealed that, although it had an object-oriented architecture, Apache Spark was tightly incorporated into it, which caused a number of drawbacks. First, Spark was necessary to run any jMetalSP application, but we found some use cases not requiring it (e.g., solving dynamic multi-objective problems that did not depend on external streaming data sources). Second, we found that replacing Spark with another streaming technology would require many changes in the framework implementation. Third, all the components of a jMetalSP application run as a single process and communicate through shared memory, which did not allow to deploy some of them in different nodes of a cluster. Finally, jMetalSP is a 100% Java project, so it was very difficult to develop and integrate a component written in other programming languages (e.g., Python).

To address these issues, we have redesigned the architecture of jMetalSP, which includes the following features:

- jMetalSP incorporates jMetal as an optimization engine, which provides a large number of features to define dynamic multi-objective optimization problems and to develop dynamic optimization metaheuristics.
- The new architecture of jMetalSP is flexible enough to isolate the streaming engine. Now, it allows the use of Java Threads [11], Spark Streaming [12], Spark Structured Streaming [13], Kafka Streams [3], and Flink [14].
- The components of a jMetalSP application can communicate through shared memory (message passing) or by means of Kafka, which allows to deploy those components in different nodes of a cluster. In this last case, Apache Avro [15,16] is used for data serialization, bringing the possibility of mixing components developed in different programming languages.
- A number of dynamic multi-objective algorithms are provided, such as: InDM2[17] and dynamic versions of NSGA-II [18], NSGA-III [19], R-NSGA-II [20], SMPSO [21], SMPSO/RP [22] and WASF-GA [23]. In addition, a series of academic benchmark dynamic optimization problems have been also included in jMetalSP. In this work, we have studied a transportation problem (a bi-objective formulation of the traveling salesman problem) based on real-world data.

Fig. 1 shows the scheme of a workflow defined within jMetalSP. It is composed by three tiers: *Streaming processing*, *Optimization processing* and *Result processing*. In this paper, we focus on describing the streaming components that are members of the streaming processing tier and the inter-process communications between components.

The rest of this paper is structured as follows. In Section 2, background concepts and literature overview are presented. Section 3 describes the previous version of jMetalSP, focusing on its main features and limitations. Section 4 details the new architecture and how it addresses with those drawbacks detected in previous versions. Section 5 presents a use case for testing and validation, while Section 6 reports performance results obtained when using different streaming engines in a cluster. In Section 7, we include an analysis and discussion about the proposed framework. Conclusions and future work are drawn in Section 8.

2. Background concepts and technologies

In this section, we provide basic background concepts about dynamic multi-objective optimization with metaheuristics and streaming processing technologies.

2.1. Dynamic multi-objective optimization

Multi-objective optimization is a discipline aimed at solving optimization problems having two or more conflicting functions or objectives. As a consequence, the optimum of this kind of problems is not usually a single solution, but a set of trade-off solutions known as Pareto Optimal Set (and Pareto front when it is represented in the objective space). The meaning of optimality states that there is no other solution in the search space improving any solution in this set for all the objective values.

Finding the Pareto front of multi-objective optimization problems (MOPs) in practice can be a very difficult task due to a number of reasons, such as: NP-hard complexity, epistasis, nonlinearity, large number of decision variables, large number of objectives, etc. [24]. For these reasons, the use of exact techniques is frequently unfeasible and the alternative is to use approximate algorithms like metaheuristics [25]. These are a family of techniques including evolutionary algorithms [26], particle swarm optimization [27], ant colony optimization [28], and many others. Metaheuristics have proven to be very effective to find near-optimal solutions to single and multi-objective optimization problems [29].

Some real-world MOPs are dynamic, which means that they are characterized by having objectives, constraints, and parameters that can change over time [30]. Therefore, whenever there is a change in the problem being optimized, the metaheuristic to solve it must also be dynamic in the sense that it needs to detect those changes and react accordingly. Our idea in jMetalSP is to consider that changes are the result of analyzing data coming in streaming from one or more sources, such as: sensors, Web Services, Kafka topics, shared file systems, etc. The issue here is how to provide a "clean way" to incorporate the streaming processing part without affecting the rest of components of the application.

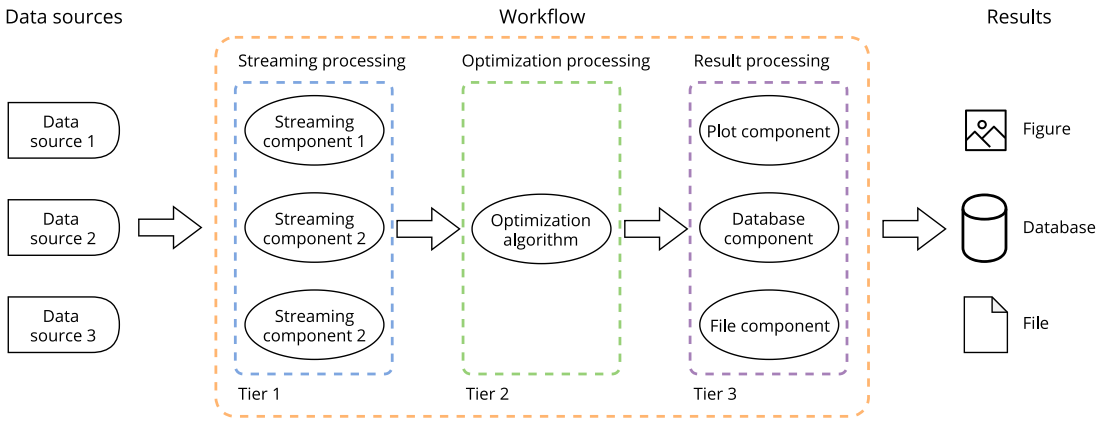

Fig. 1. General workflow scheme of a jMetalSP application.

2.2. Streaming data processing technologies

Big Data tools can be classified into three main classes, namely: batch processing tools, streaming processing tools, and interactive analysis tools [31]. Batch processing tools are only applied to the stored data, which does not change while being processed, while streaming processing allows us to process data in real-time as they arrive and quickly detect conditions within small time periods. Most batch processing tools are based on the Apache Hadoop ecosystem, such as Apache Mahout [32] or Hive [33]. Nevertheless, in case of interactive analysis, whose goal is to empower data analysts to formulate and assess hypotheses in a rapid and iterative manner [34], we find tools such as Apache Hadoop [35] with its distributed programming model MapReduce [36] and Apache Spark, and visualization tools like Data Wrangle [37] or Profiler [38].

Streaming data processing has two main issues to deal with: the first one is *high volume*, which implies that data arrives continuously and cannot be stored to be analyzed with standard database technologies; the second one, *low latency*, is related with real-time constraints, which requires data to be processed immediately, so computing capabilities beyond the CPU power of individual machines may be needed. As a consequence, streaming platforms must provide solutions able of handling high volumes of data in real-time with a scalable, highly available architecture [39].

These streaming technologies have a wide range of use cases, such as social networks, real-time trading analytics, malfunction detection, log processing, campaign, smart advertisement placement and metrics analytics [40]. The jMetalSP framework presented in this paper includes four representative open source distributed stream processing systems: Spark Streaming, Spark Structured Streaming, Kafka Streams and Flink. In order to provide a comparison among them, we focus on seven features:

- ***Cluster Management***. Platform where streaming processing systems can be deployed.
- ***Delivery Guarantee***. This feature is key when evaluating streaming systems, as it specifies whether a streaming processing system ensures that each record is processed once and only once, even if some failures are encountered in the middle of processing.
- ***Programming Language***. A streaming system can support more than one programming language.
- ***Programming Model***. It refers to the programming style. They are usually event- or batch-based.
- ***Streaming API***. Streaming systems usually provide two APIs for batch and streaming processing, which can be independent or very close to each other.
- ***Latency***. It refers to the delay between data production and data reading by the streaming engine.
- ***Throughput***. It determines the number of ingested and successfully processed records per time unit.

2.2.1. Spark streaming

Spark Streaming is a mechanism used to process real-time data with the built-in Spark modules for streaming, SQL, machine learning and graph processing. It is an extension of the core Spark RDD API that enables scalability, high-throughput, and guarantees an exactly-one read data semantics in quasi real-time streaming processing [41]. The Spark Streaming module provides a stream abstraction called discretized stream or DStream, whose main idea is to treat a streaming computation as a series of deterministic batch computations in small time intervals [42]. As a matter of fact, Spark Streaming follows a micro-batching programming model to simulate streaming processes. Instead of processing the streaming data one record at a time, Spark Streaming combines incoming data into small blocks (i.e., mini-batches), which are then processed together so as to keep high-throughput. Due to this fact, its main disadvantage is the growth of data latency. Stateless transformations (for instance, RDD transformations) of each batch does not depend on the data of its previous batches, while stateful transformations (i.e., based on sliding windows and on tracking state across time) use data or intermediate results from previous batches to compute the results of the current batch. In other words, RDD transformations are applied to data within each time slice, but they cannot be applied to all time slices if they are concatenated.

Spark Streaming is very versatile from two points of view, cluster manager and programming language. In the first case, Spark Streaming can connect to several types of cluster managers, such as either Spark's own standalone cluster manager, (Mesos [43] or YARN [44]). In the second case, Spark Streaming supports the programming languages Java, Scala, and Python.

2.2.2. Spark structured streaming

Spark Structured Streaming is a new high-level streaming API in Apache Spark to tackle the problem of processing fast-flowing data in real-time. It allows users to specify stream processing logic in the same way as batch analytics. Structured Streaming combines elements of Google Dataflow [5], such as separating processing time from event time and triggers, incremental queries [45,46] and, as its predecessor Spark Streaming, to enable stream processing under the Spark SQL API. Consequently, this streaming engine provides SQL operators such as selection, aggregation, and join, which are based on automatically extending a static relational query (expressed using SQL or DataFrames). Built

on the Spark SQL engine, Spark Structured Streaming, just like Spark Streaming, is a fault-tolerant and scalable system. The main concept in Structured Streaming is to manage live data streams as an unlimited table in a way that when new data arrives they are appended as new rows of a table of unlimited length, allowing low latency. This idea makes the stream processing model very similar to a batch processing model, i.e., batch processing may be seen as a special case of streaming processing [47]. As Spark Streaming, Spark Structured Streaming is based on micro-batching, it supports a number of programming languages (Java, Scala, Python, and R) and it can be executed with different cluster managers (Spark's own standalone cluster manager, Mesos and YARN).

2.2.3. Kafka streams

Apache Kafka [48,49] is a distributed, partitioned, replicated commit log service, which maintains feeds of messages in categories called topics. Kafka is designed using the *Producer Consumer pattern*, according to which producers are processes that publish messages to a Kafka topic and consumers subscribe to topics and read the published messages.

Kafka Streams is a client library which allows building applications and microservices, where the input and output data are stored in a Kafka cluster. Furthermore, Kafka Streams supports building streaming applications that allow calling external services like databases or even send them via Kafka topics. The most relevant features of this technology are:

- It is a simple and lightweight client library, whose API has two versions in Java and Scala.
- Kafka Streams has no external dependencies on systems other than Apache Kafka itself as the internal messaging layer.
- It guarantees that each record (message) will be processed once and only once.
- Kafka Streams supports one-record-at-a-time processing with a millisecond processing latency, as well as event-time based windowing operations as programming model (which allows to keep high-throughput).
- It offers two stream processing primitives, a high-level *Streams DSL* (Domain Specific Language) and a low-level *Processor API*. Kafka Streams does not prescribe a deployment mechanism [3].
- Kafka Streams can connect to Mesos and many other cluster systems.

In addition, Kafka Streams DSL provides common data transformation operations: map, filter, join and aggregations. The Processor API offers the possibility of developing custom operators close to the Kafka Core API.

2.2.4. Flink

Flink, like the aforementioned systems, is a top-level project of the Apache Software Foundation [14]. It is a stream processing engine making batch processing a special class of application. The main difference is that in batch processing with Flink data are bounded. The tool was developed as part of a research project at the Technical University of Berlin in 2009.

The approach followed by Flink is to built many classes of data processing applications, including real-time analytics, continuous data pipelines, historic data processing (batch), and iterative algorithms (machine learning, graph analysis) which can be expressed and executed as pipelined fault-tolerant dataflows [2].

Flink uses a stream abstraction called DataStream, which is a sequence of partially ordered records [50]. This framework guarantees exactly-once state consistency in case of failures by periodically and asynchronously checkpointing the local state to durable storage. Flink also allows the integration with all common cluster resource managers such as Yarn, Mesos, and Kubernetes, although it can be set up to run as a stand-alone cluster as well. Flink provides APIs in Java, Scala and Python.

A summary of streaming technologies with regards to their main features (cluster management, delivery guarantee, programming language, programming model, latency, and throughput) is included in Table 1. It is worth noting that in Flink and Kafka Streams every record is processed as soon as it arrives, allowing the frameworks to achieve the minimum possible latency. This implies that it is hard to achieve a high fault-tolerance degree. However, in Spark Streaming the latency is higher, because it uses micro-batching programming model instead of events. Nevertheless, Spark Structured Streaming enhances the latency compared with Spark Streaming thanks to the incremental batch queries it uses.

3. The jMetalSP framework

jMetalSP [8] is the result of combining the jMetal multi-objective optimization framework [9,51] and the Apache Spark cluster computing system [1], with the goal of solving dynamic optimization problems from multiple and diverse external streaming data sources in Big Data contexts.

jMetal is a widely used software in the field of multi-objective optimization and Spark is one of the dominant technologies in Big Data, so the idea of integrating them in jMetalSP was to generate a framework combining the features of the former (flexible and extensible architecture, lots of representative multi-objective metaheuristics and problems) and the latter (streaming processing, high level parallel model).

jMetalSP is an open source MIT licensed project hosted and maintained in GitHub.[1] It is offered as a white-box framework, as we provide the source code and to take advantages of some of its features it is necessary to know implementation details. The documentation of the project[2] includes guides about installing the project and several examples explaining use cases.

Since its launch in 2016 [8], jMetalSP has been continuously evolving. We detail next the first two developed versions.

3.1. First version of jMetalSP

When thinking in the design and development of jMetalSP, we were driven by some ideas. In particular, issues such as adding streaming data sources and connecting them to algorithms that deal with dynamic problems should be done in an easy a clean way. This was addressed by adopting an object-oriented architecture.

A jMetalSP application was originally conceived to be composed of these four types of components:

- ***Streaming data processing***. The role this kind of components is to receive and analyze incoming streaming data by using Spark Streaming.
- ***Problem***. These components represent dynamic multi-objective problems to be optimized. In jMetalSP, we focus on dynamic multi-objective optimization problems that are characterized by the fact that their objectives or their search space can undergo significant variations over time, which may affect their Pareto set, their Pareto front or both of them. In the context of jMetalSP, changes in the problems will be originated by the results of the processing and analysis of one or more streaming data sources.

[1] jMetalSP project site: https://github.com/jMetal/jMetalSP.
[2] https://jmetalsp.readthedocs.io/en/latest/.

Table 1
Feature comparison between streaming technologies.

Feature	Flink	Kafka streams	Spark streaming	Spark structured streaming
Cluster Management	Standalone / Yarn / Mesos	Mesos / Many[a]	Mesos / Yarn / Standalone	Mesos / Yarn / Standalone
Delivery Guarantee	Exactly Once	At Least Once	Exactly Once	Exactly Once
Programming Language	Java / Scala	Java / Scala	Java / Scala / Python	Java / Scala / Python / R
Programming Model	Event / Micro-batching	Event	Micro-batching	Micro-batching
Streaming API	Native	Separated from batch	Integrated with batch	Incremental batch queries
Latency	Low	Low	Medium	Low
Throughput	High	Medium	High	High

[a]Kurbenetes, Yarn, Swarm, ECS from Amazon, Cloud Foundry, etc.

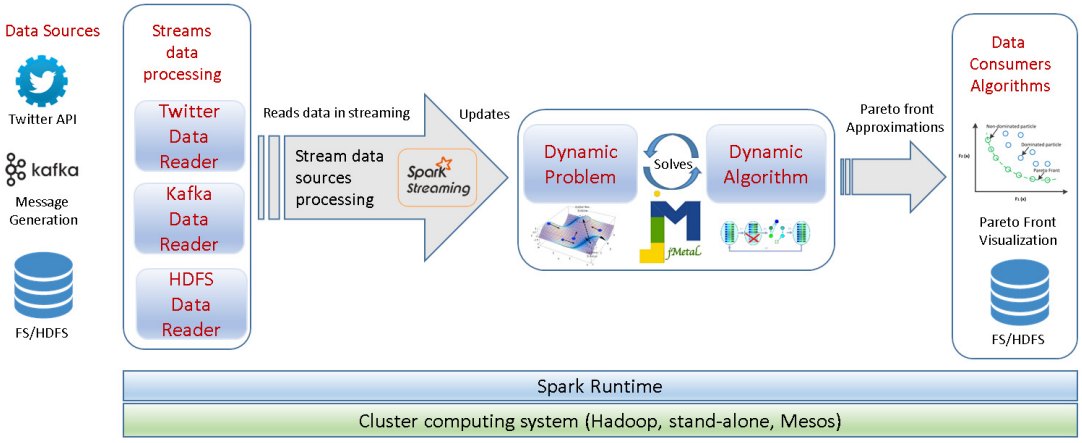

Fig. 2. Architecture of the original jMetalSP framework.

- ***Algorithm***. A dynamic algorithm in jMetalSP is a metaheuristic that considers two main aspects: first, the problem can change during the algorithm execution, so the state of the problem should be regularly checked and, when a change is detected, a re-starting procedure is applied; second, when the stopping condition is reached, the algorithm (instead of just terminating) starts again.
- ***Consumer components***. Dynamic algorithms are supposed to run forever, so Pareto front approximations are produced periodically. This means that any component "interested" in getting those fronts cannot wait until the completion of the algorithm (that never ends), as in the case of techniques dealing with static problems. Consumer components capture data from the algorithm in a way that they do not need to wait until their completion, e.g., to plot Pareto fronts or save them in a database or in files.

The architecture of the first version of jMetalSP is shown in Fig. 2, where we can observe the three tiers presented in Fig. 1. The first tier contains streaming data processing components, the second tier is always composed of the optimization component (metaheuristic and problem) that carries out the optimization process, and the third tier can have one or more consumer components.

There is a *Spark runtime* layer on top of which the rest of components are deployed. It is worth emphasizing that the data sources supported by jMetalSP were directly related to Spark API (version Spark 1.4.0), which allowed the deployment on the cluster management systems Mesos, Yarn, as well as the Standalone mode.

After using jMetalSP for a while, we observed some limitations of the adopted architecture, namely:

1. Spark was tightly integrated into jMetalSP, what led to the requirement of using it all the time and for every application. However, some jMetalSP users were only interested in solving benchmarks of dynamic multi-objective problems that did not required any streaming data source.
2. The second consequence of the strong Spark integration is that replacing Spark Streaming by another streaming engine (in particular, Spark Structured Streaming) was not easy and required many changes in other components.
3. All the components of a jMetalSP workflow were compiled into a single Java program, so all of them were executed concurrently using Java threads in a single node, so it was not possible to distribute them in a cluster.

3.2. Second version of jMetalSP

The second version of jMetalSP architecture was born with the goal of overcoming the main issues found in the first one. In this version, we focused particularly on dealing with the decoupling of Spark inside the jMetal architecture. For this reason, we added an abstract level called *Streaming Runtime*, which can be stacked on top of the former *Spark Runtime*, as well as on a new *Default runtime*, which is based on Java threads and it does not require Spark. The resulting architecture is depicted in Fig. 3, and it was presented in [52].

In this version, jMetalSP included (for the first time) a benchmark of dynamic multi-objective problems. Concretely, the FDA problem family [30], consisting in five dynamic problems with different features depending on whether their Pareto-optimal front (POF) and/or Pareto-optimal solutions (POS) change over time.

The adoption of the *Streaming Runtime* introduced a higher abstraction level, as it brought the opportunity to include different streaming systems, although this issue was not addressed in this second version yet. It is worth mentioning that only the streaming components were bound to a particular runtime (i.e., using Spark for reading from Kafka, Twitter, etc. requires the *Spark Runtime*); the rest of components (algorithms, problems, data consumers) are independent of the runtime system.

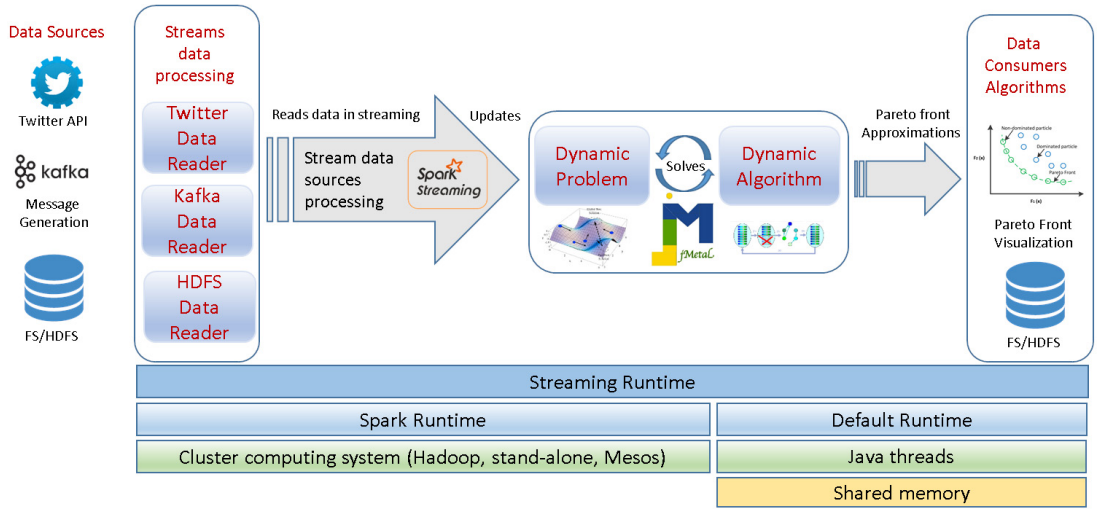

Fig. 3. Architecture of the second version of jMetalSP framework.

Anyway, as in the first version, the resulting application is a multi-threaded Java program, where its components cannot be executed in different nodes of a cluster and all of them must be written in that programming language; thus, it is not possible, for example, to implement a visualization component in Python, which provides a richer set of libraries for that purpose than Java.

4. New architectural design

In this section, we describe the new and most recent version of our framework. The goal of this architecture is to mitigate the problems that have been identified in former versions of jMetalSP. In this third approach, three additional streaming systems (Spark Structured Streaming, Kafka Streams and Flink) are included and, what is more, a new way of inter-process communication via Kafka and Avro for message serialization is added. The implications of these features are, first, the components of a jMetalSP application can be deployed in different nodes of a cluster; second, it allows to define and implement components in other programming languages besides Java.

A design requirement we impose is that adopting these new features must be done in a transparent way. Therefore, replacing a streaming component should not affect the rest of components.

The new jMetalSP architecture is depicted in Fig. 4. Compared to the previous version, we can observe that there are a number of runtime systems, one per streaming engine, which allow the deployment of components on clusters using Kafka and Avro for inter-component communication. The default runtime based on Java threads is still present as an option when none of these technologies are needed.

Avro is a data serialization system that provides data structures (in binary data format), object container files to store persistent data and remote procedure call (RPC) capabilities. Data are then serialized and deserialized using Avro schemes, so it brings a high degree of interoperability to jMetalSP, since components only need to know the Avro schema of the data in order to communicate between them. These data are then serialized and transmitted via Kafka. In this sense, as Kafka topics are fault-tolerant and highly scalable, these features are available to jMetalSP workflows.

For a better understanding of the new architecture of jMetalSP, Fig. 5 depicts the UML diagram of the final architecture. It is worth mentioning that the *observer pattern* has been adopted to be used for the communication between components. The classes of the UML diagram and their relationships are described below:

- ***jMetalSPApplication***. This class is a container of all the elements composing an application under jMetalSP. It acts as a template where all the components can be defined, configured, and started when the *run()* method is called.
- ***StreamingRuntime***. This class represents the underlying streaming engine. It has five sub-classes: (1) default plain-Java based runtime (Spark, Flink or Kafka are not required), which starts each streaming data source in a dedicated thread, (2) Spark RDD (Spark Streaming), (3) Spark SQL (Spark Structured Streaming), (4) Flink runtime, and (5) Kafka runtime.
- ***DynamicAlgorithm***. In jMetalSP, a DynamicAlgorithm is a conventional metaheuristic with a *restart()* method that is required when a change in the problem has been detected.
- ***AlgorithmDataConsumer***. These components receive the output of the dynamic algorithms.
- ***DynamicProblem***. This class represents the dynamic multi-objective problems to be optimized. It includes two methods, *isTheProblemModified()* and *reset()*, to respectively indicate when the problem has been modified and to reset that state when, after a change detection, the corresponding processing have been done.
- ***StreamingDataSource***. A jMetalSP application can have one o more streaming data source components, which are subclasses of *StreamingDataSource*.
- ***ObserverData***. All the communication between components are based on the observer pattern, which interchange instances of *ObservedData*.

A dynamic algorithm is considered as an observable entity, so when a new Pareto front approximation is produced it is notified to the registered *AlgorithmDataConsumer* observer objects. Similarly, the streaming data sources are observable entities that notify data to a particular observer, which is the dynamic problem.

5. Use case: transportation problem with real-world data

For validation purposes, a case of study has been developed, which is focused on streaming processing and optimization of mobility routes with real-world data in the domain of Smart Cities.

We have chosen a bi-objective formulation of the Traveling Salesman Problem (TSP) [53], where two conflicting objectives, minimizing travel time and minimizing the total travel distance,

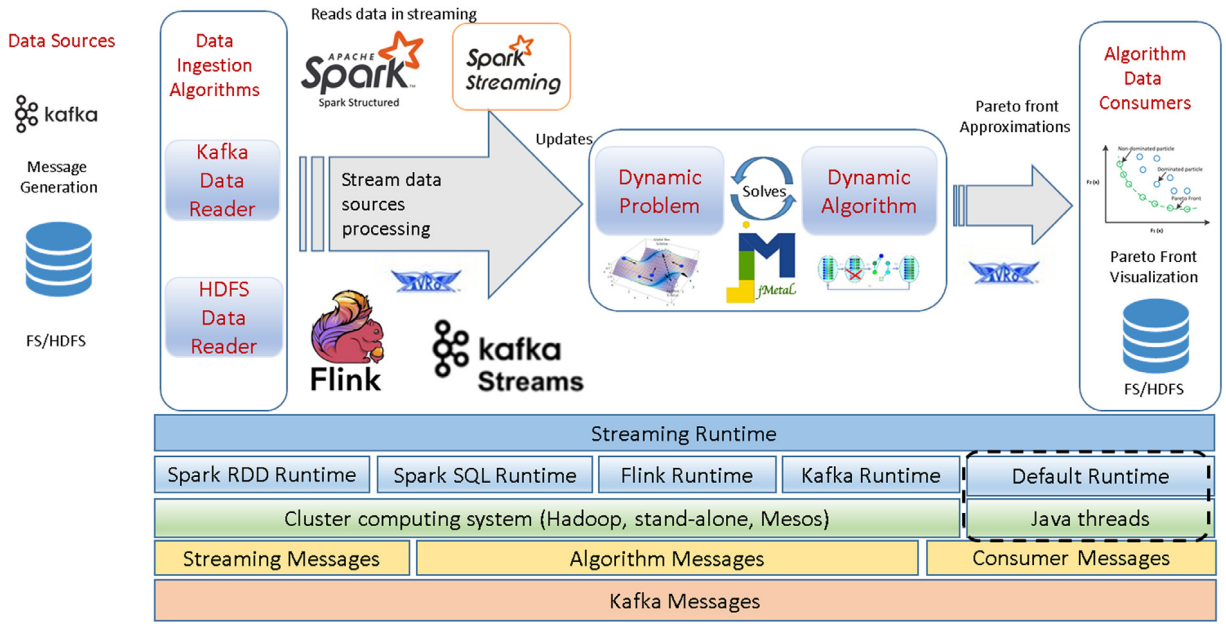

Fig. 4. Current architecture of jMetalSP framework. Dotted points means that messages are sent via shared memory.

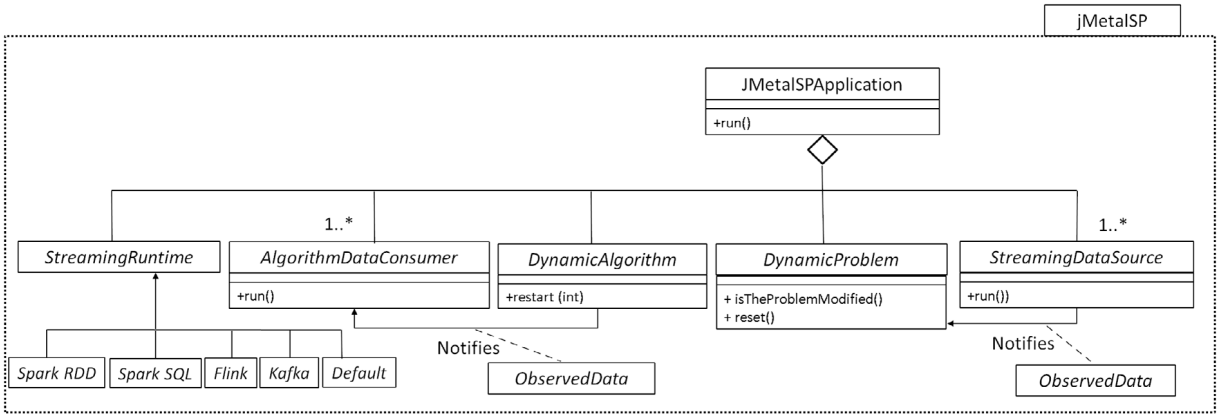

Fig. 5. UML diagram of the current architecture of jMetalSP framework.

have to be optimized at the same time. The data are obtained from the New York City Department of Traffic that provides open real-time traffic speed data.[3] Metadata in this source comprise the length of the links, the mean speed and the mean traveling time of the cars traversing the two end points that define the links. As data change with time, we are dealing with a dynamic multi-objective problem.

Fig. 6 illustrates a workflow with the three kinds of components involved in the workflow designed for this problem. Each component runs independently in one or more nodes of the cluster and the communications are carried out by means of Kafka with the data serialized according to the Avro schemes. In this example, two different data types are defined, *TSPMatrixData* and *AlgorithmData*, which are described in the next subsections.

Code Snippet 1: Definition of TSPMatrixData in Java.

```
public class TSPMatrixData
extends SpecificRecordBase
implements SpecificRecord {
  private String matrixIdentifier ;
  private int x ;
  private int y ;
  private double value ;
}
```

5.1. TSP matrix data

The data required to generate an instance of the TSP in the problem component are stored by means of two matrices: cost (travel time) and distance. The communication between streaming data and problem components is carried out throughout the class *TSPMatrixData*, which is summarized in Code Snippet 1 (for the sake of clarity, getters, setters and constructors have been bypassed). This class contains the information necessary to update the coordinates `x` and `y` of one of the two matrices. The matrix to be updated is identified by the variable *matrixIdentifier*.

The corresponding Avro schema of *TSPMatrixData* is included in Code Snippet 2. The fields of the TSP matrix data are represented in JSON notation. It is worth noting that Avro provides its own data type primitives that can be also found in commonly

[3] At the time of writing this paper, the traffic data can be obtained from https://data.cityofnewyork.us/view/qkm5-nuaq.

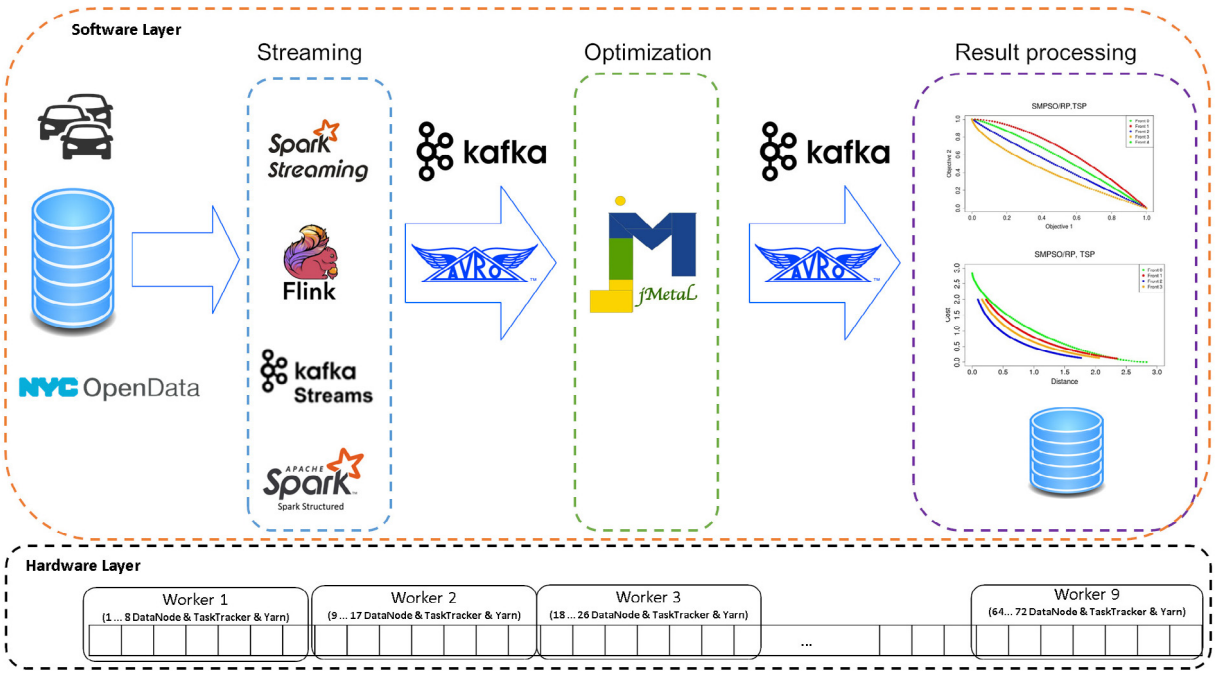

Fig. 6. Workflow for coping with optimization of the transportation problem.

used programming languages (Java or Python), such as: String, Double, Integer and List.

Code Snippet 2: Definition of TSPMatrixData Avro schema.

```
{"namespace": "tsp",
"type": "record",
"name": "TSPMatrixData",
"fields": [
    {"name": "matrixIdentifier",
    "type": "string"},
    {"name": "x", "type": "int"},
    {"name": "y", "type": "int"},
    {"name": "value", "type": "double"}
 ]
}
```

5.2. Algorithm data

The *AlgorithmData* class, described in Code Snippet 3, contains the information sent by the *DynamicAlgorithm* to the consumer components. This class defines fields to store the objectives and variables of the Pareto front approximation that the algorithm has found. In addition, it is used to manage metadata concerning the algorithm and the problem names, as well as additional information (e.g., reference points that can be used optionally by some consumers).

Code Snippet 3: Definition of AlgorithmData in Java.

```
public class AlgorithmData
extends SpecificRecordBase
implements SpecificRecord, ObservedData {
  private List<List<Double>> objectives;
  private List<List<Double>> variables;
  private int numberOfIterations;
  private String algorithmName;
  private String problemName;
  private int numberOfObjectives;
  private List <Double>referencePoints;
}
```

The corresponding Avro schema used to serialize the algorithm data is included in Code Snippet 4. We can observe how the objective and variables fields, implemented in Java as lists of double values, are translated into two dimensional arrays in JSON.

6. Experiments

In this section, we describe the experimental studies we have conducted to evaluate the jMetalSP we have proposed. We have designed two types of experiments aimed at, first, testing the working of NSGA-II when solving the dynamic bi-objective TSP problem and, second, to analyze the computation performance when using the Spark (RDD-based), Kafka, and Flink runtimes in a cluster.

6.1. Dynamic bi-objective TSP optimization

This experiment uses the Spark Streaming (RDD-based) runtime to process the streaming data obtained from the New York Department of Traffic Web site and a dynamic version of the NSGA-II metaheuristic to optimize the TSP problem, which reacts when changes in the problem data are detected. Spark Streaming can take as data source a directory which is monitored so that whenever a new file is stored into it, this fact is detected and the file is automatically read. We use this feature in such a way that we run an external process which periodically gets the latest traffic data information and write it in a new file that is stored in a directory monitored by the *StreamingDataSource* component of the jMetalSP application. As the traffic data are updated roughly every minute, the frequency of the external process is set to one minute.

The parameter settings of the dynamic NSGA-II algorithm are: the population size is 100, the crossover operator is PMX (applied with a 0.9 probability), the mutation operator is swap (applied with a probability of 0.2), and the algorithm computes 100,000

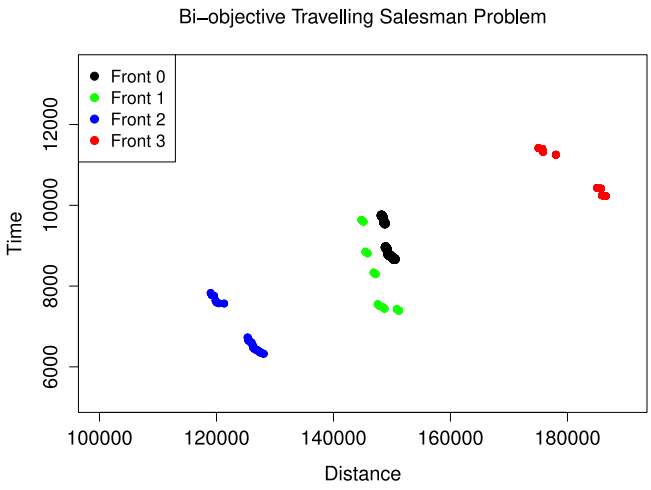

Fig. 7. Pareto front approximations obtained when solving the TSP of New York with Dynamic NSGA-II.

function evaluations before sending the front found to registered data consumers and restarting. In particular, the jMetalSP application for the TSP is configured with a data consumer that plots the fronts that are being produced by the algorithm. When running the application we obtain results as the one shown in Fig. 7, which depicts the fronts computed after four problem changes.

6.2. Computational performance

The former experiments allows us to test that our proposal works. However, the frequency update of the real traffic data is very low, so there is not a high demand of computational resources and hence we cannot assess the performance of our framework when large amounts data arrive and they have to be processed with real-time requirements.

For this reason, we have developed a second type of experiments in which we execute the same application as before but we produce artificially simulated traffic data at different speeds. The goal is to evaluate the performance of three streaming engines that can be used in jMetalSP, namely Spark Streaming, Flink, and Kafka Streams. Before this, we present the computational environment used to deploy a complete workflow as shown in Fig. 6.

Code Snippet 4: Definition of AlgorithmData schema.

```
{"namespace":"algorithmdata",
    "type": "record",
    "name": "AlgorithmData",
    "fields":[
      {
        "name":"objectives",
        "type":{
           "type":"array",
           "items":
           {
               "type":"array",
                "items":"double"
           }
        }
      },
      {
       "name":"variables",
       "type":{
          "type":"array",
           "items":
              {
               "type":"array",
                "items":"double"
               }
           }
      },
       {
         "name":"referencePoints",
         "type":{
           "type":"array",
            "items":"double"
           }
      },
      {
        "name": "numberOfIterations",
         "type":"int"
      },
      {
         "name": "algorithmName",
         "type":"string"
      },
      {
         "name": "problemName",
         "type":"string"
      },
      {
        "name": "numberOfObjectives",
        "type":"int"
      }
    ]
}
```

We have conducted these experiments in a computational environment deployed on a private high-performance cluster computing platform located in the *Ada Byron Research Center* at the University of Málaga (Spain). This infrastructure comprises a set of IBM hosting racks for storage, units of virtualization, server compounds and backup services. The virtualization platform used for experiments is made up of 9 virtual machines (VM1 to VM9), each one with 8 cores, 10 GB RAM and 33 GB virtual storage (summing up 72 cores, 90GBs of memory and 227GBs HD storage). The whole cluster is configured to execute the framework libraries Apache Spark v2.4.3, Apache Flink v2.8.1 and Apache Kafka v2.3.0.

For these experiments, have registered the execution traces of the entire workflow during one hour for two different test cases under different conditions of message passing load. A first test is performed by injecting 10,000 messages every 1 s from the data sources to the workflow, while a second test perform the same load of 10,000 messages but every 0.1 s. This way, we consider different stress conditions to assess the performance of Spark Streaming, Flink and Kafka Streams. The messages are written in a Kafka topic so as to the streaming engines are able to read them.

In addition, we have incorporated a Kafka message producer that generates a series of random messages (4 KBs) with information in form of *TSPMatrixData* to update the problem. In this regard, the complete workflow is continuously producing dynamic TSP routes according to these incoming data, so that the new Pareto front approximations could be plotted as shown in Fig. 7. This aspect was studied in our previous work [8], hence we focus now on assessing the computational effort from the perspective of managing streaming data. To this end, we have traced and plotted the *load_one* measure of the entire cluster, in order to check the overall CPU load. In particular, the *load_one* measure computes the number of threads at kernel level that

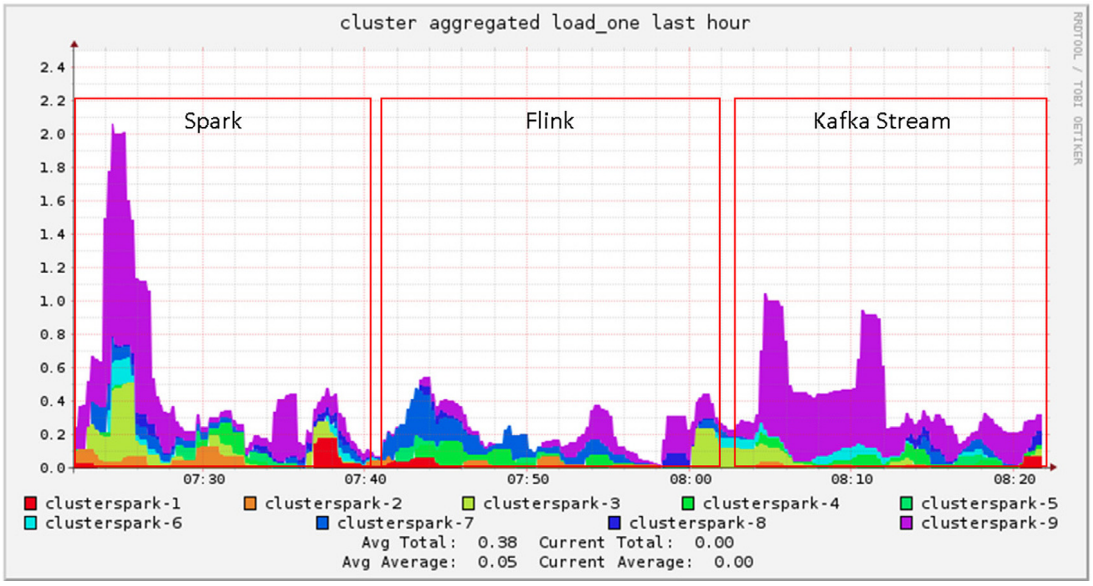

Fig. 8. Test 1. Load one. Number of threads per node.

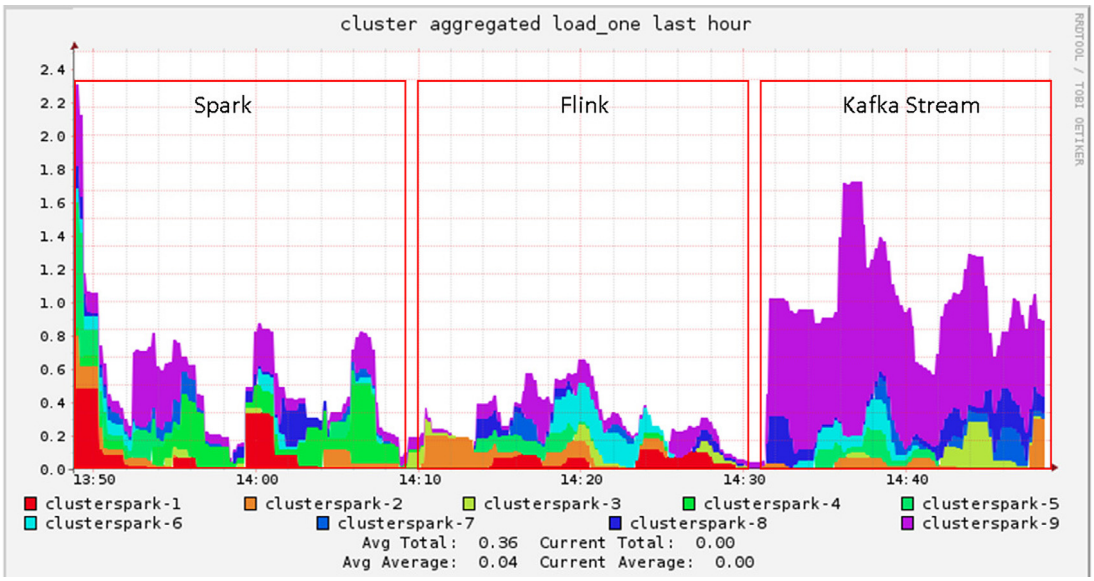

Fig. 9. Test 2. Load one. Number of threads per node.

are running and being queued while waiting for CPU resources, averaged over the last hour. We could interpret this number in relation with the number of hardware threads available on the machine and the time it takes to drain the run queue.

6.2.1. Test 1: 10,000 messages each 1 second

Fig. 8 depicts the CPU load_one trace of each node in the spark cluster through one hour of execution of jMetalSP for the transportation use case described in Section 5. This time period is divided in three intervals of twenty minutes for every streaming engine, i.e., each 20 min messages are managed by a different streaming processing engine.

In the first twenty minutes, Spark is in charge of reading the data from the Kafka topic that is sending messages. We can observe how the processing time in the *clusterspark-9* node (master node) is high in the initial minutes of execution. This behavior is caused by the initial scheduling process required to deploy all running tasks in the computational framework, afterwards it stabilizes. Nevertheless, when Flink is running, the *load_one* measure is steadier during the 20 min of message processing. Finally, the performance of the application with Kafka Streams has two threads consumption peaks, with the cluster being stabilized after five minutes. A main observation in this test case is that the three processing engines are able to manage the load of data derived from messages successfully, even though Flink demands lower computational resources than Kafka Streams and Spark.

6.2.2. Test 2: 10,000 messages each 0.1 second

In this second test, the goal is to assess the streaming engines in a highly demanding environment characterized by the ingestion of 10,000 messages every 0.1 s.

An execution trace of the load_one computed by jMetalSP in this test is captured in Fig. 9. As happened in the previous test, the cluster is initially stressed when Spark is reading streaming data, although in this case peaks appear in all the nodes of the cluster. Furthermore, the load is oscillating during the 20 min

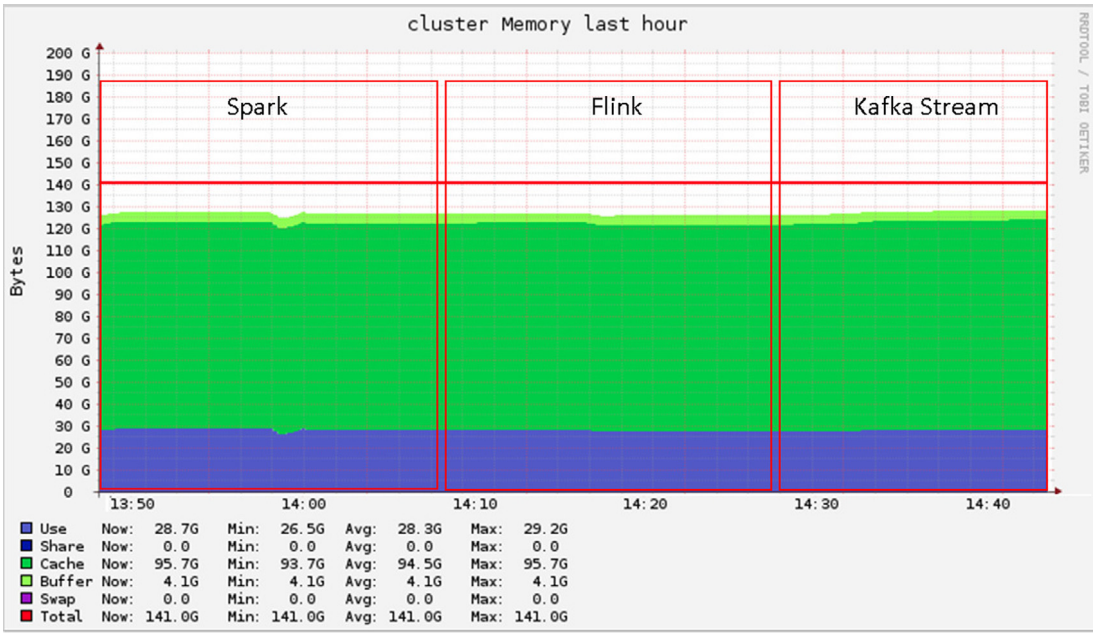

Fig. 10. Test 2. Memory usage level.

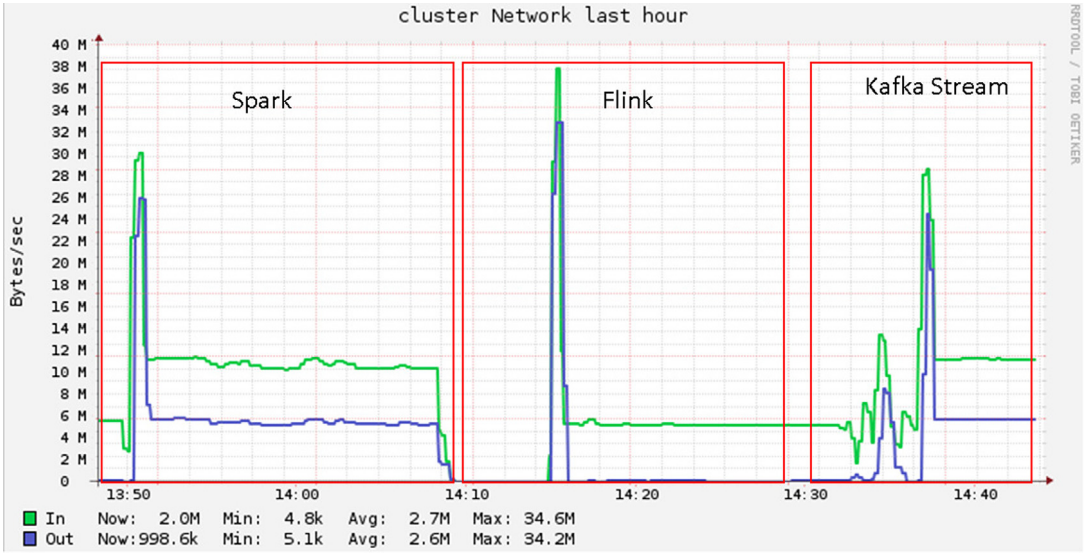

Fig. 11. Test 2. Network usage level.

of Spark processing, so it needs more time to be stabilized. In the case of Flink, incoming messages are properly managed, so the *load_one* is stable throughout the observation time. However, when Kafka stream is reading data, the system is not totally stabilized although the cluster is still able to read the data. In this regard, it is worth mentioning that although Kafka does not follow a master–worker architecture, the master node (node 9) registers a high *load_one* value, probably due to the way Kafka splits its tasks.

In terms of memory and network usage of the cluster, an interesting observation can be made when comparing Figs. 10 and 11, which show the trace of these two measures (respectively) in the context of test 2. According to them, the use of memory is practically stable during the complete execution time of jMetaSP when processing data with Spark Streaming, Flink and Kafka Streams. This is a highly desirable behavior, since it avoids the system collapse even with harder scenarios of message processing (as happens in Test 2). The network usage registers a series of peaks, mostly in the case of Flink that seems to derive certain network overhead, but with the aim of reaching stable balance in CPU load among the nodes in the cluster.

7. Discussion

Our new proposed architecture for jMetalSP enhances the capacity to deal with Big Data optimization problems, not only due to the flexibility of incorporating different streaming engines, but also by allowing the execution of its components in different nodes of an High Performance Computing cluster, as well as the interconnection with other components of a workflow, such as external data sink processes, data generators, analysers, visualizations, etc.

In terms of practical implications, the proposed architecture deals with four important issues of Big Data, namely variability,

variety, velocity and volume, since this new version of jMetalSP is able to use different streaming processes depending upon the properties (e.g., amount of data, velocity or data source) of the data to deal with. Furthermore, thanks to the new inter-process communication based on Kafka and Avro, jMetalSP allows to joint efforts with other tools in running time. In this sense, Avro schemes are essential for documenting and modeling the manipulated data, so its definitions capture a point in time of what data looked like when it was first recorded, since the schema is saved with the data inside it. Streams are often recorded in data repositories like Hadoop HDFS, and those records can represent historical data. It makes sense that data streams and data repositories have a less rigid, yet more evolving schema than the schema of the operational data base.

As a result, the proposed framework not only allows a better definition of an analytic workflow oriented to optimization using its components, but also can be used as part of a bigger workflow comprised of other external components or software that could complement its functionality when analyzing Big Data optimization problems.

All these features facilitate the adoption of optimization tasks in prescriptive modeling applications commonly found in industry, agriculture, domotics, wearables, and many other areas where the ingestion of multiple data from heterogeneous sensor sources are involved.

Finally, it is worth mentioning that all the streaming systems integrated with jMetalSP have fault-tolerance features. Although these characteristics are not currently used by jMetalSP, they are available to be used in cases of dealing with applications that require them.

8. Conclusions

In this work, a new jMetalSP architecture is proposed to enhance this framework with new streaming engines, as well as with the capacity to communicate components via Kafka and using Avro for data serialization.

To test this architecture, a case study has been presented based on a realistic online optimization of the well-known optimization problem TSP. The experience obtained from this case on different execution scenarios revealed that jMetalSP is able to not only incorporate different streaming engines, but also to switch between them without changing any component from its core.

The experiments show that jMetalSP improves its capacity when coping with Big Data optimization problems thanks to its new way to interconnect components, which can also be independently deployed on different nodes of a cluster, thus enhancing its capacity of defining workflows.

It is worthy to declare that although the proposed architecture has been developed for optimization purposes, those ideas can be transferred to other fields such as machine learning, deep learning, etc. What is more, the current architecture facilitates the interconnection between components from different fields, like data sources or optimization, which share data or belong to the same workflow and share analysis.

This motivates our future research agenda, which entails three phases: first, designing and developing components from different fields (optimization, machine learning, etc.); second, defining, through Avro schemes, their data structures; third, defining their communications and connections between the components in order to develop a workflow to analyze any kind of Big Data problems (and not only optimization ones). Finally, exploring its usability on different real-world use cases where Big Data optimization tasks are involved.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially funded by Grant TIN2017-86049-R (Spanish Ministry of Education and Science). Cristóbal Barba-González is supported by Grant BES-2015-072209 (Spanish Ministry of Economy and Competitiveness). Antonio Benítez-Hidalgo is supported by Grant PRE2018-084280 (Spanish Ministry of Science, Innovation and Universities) and José García-Nieto is the recipient of a Post-Doctoral fellowship of "Captación de Talento para la Investigación" Plan Propio at Universidad de Málaga.

References

[1] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, in: HotCloud'10, USENIX Association, 2010, p. 10.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, Bull. IEEE Comput. Soc. Tech. Committee Data Eng. 36 (4) (2015) 28–38.

[3] M. Kleppmann, J. Kreps, Kafka, Samza and the unix philosophy of distributed data, IEEE Data Eng. Bull. 38 (4) (2015) 4–14.

[4] S.A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, R.H. Campbell, Samza: stateful scalable stream processing at LinkedIn, Proc. VLDB Endow. 10 (12) (2017) 1634–1645.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R.J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al., The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, Proc. VLDB Endow. 8 (12) (2015) 1792–1803.

[6] M.H. Iqbal, T.R. Soomro, Big data analysis: Apache storm perspective, Int. J. Comput. Trends Technol. 19 (1) (2015) 9–14.

[7] H. Pathak, M. Rathi, A. Parekh, Introduction to real-time processing in Apache Apex, Int. J. Res. Advent Technol. (2016) 19.

[8] C. Barba-González, J. García-Nieto, A.J. Nebro, J.A. Cordero, J.J. Durillo, I. Navas-Delgado, J.F. Aldana-Montes, Jmetalsp: a framework for dynamic multi-objective big data optimization, Appl. Soft Comput. 69 (2017) 737–748, http://dx.doi.org/10.1016/j.asoc.2017.05.004.

[9] J.J. Durillo, A.J. Nebro, Jmetal: A java framework for multi-objective optimization, Adv. Eng. Softw. 42 (10) (2011) 760–771.

[10] J.D. Ser, E. Osaba, D. Molina, X.-S. Yang, S. Salcedo-Sanz, D. Camacho, S. Das, P.N. Suganthan, C.A.C. Coello, F. Herrera, Bio-inspired computation: Where we stand and what's next, Swarm Evol. Comput. 48 (2019) 220–250.

[11] P. Hyde, Java Thread Programming, vol. 1, Sams Indianapolis, 1999.

[12] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, et al., Apache spark: a unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65.

[13] M. Armbrust, D. Bateman, R. Xin, M. Zaharia, Introduction to spark 2.0 for database researchers, in: Proceedings of the 2016 International Conference on Management of Data, ACM, 2016, pp. 2193–2194.

[14] E. Friedman, K. Tzoumas, Introduction to Apache Flink: Stream Processing for Real Time and Beyond, O'Reilly Media, Inc., 2016.

[15] J. Scott, Avro–more than just a serialization framework, Chicago Hadoop Users Group (2012).

[16] D. Vohra, Apache avro, in: Practical Hadoop Ecosystem, Springer, 2016, pp. 303–323.

[17] A.J. Nebro, A.B. Ruiz, C. Barba-González, J. García-Nieto, M. Luque, J.F. Aldana-Montes, InDM2: Interactive Dynamic multi-objective decision making using evolutionary algorithms, Swarm Evol. Comput. 40 (2018) 184–195.

[18] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.

[19] Y. Yuan, H. Xu, B. Wang, An improved NSGA-III procedure for evolutionary many-objective optimization, in: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, ACM, 2014, pp. 661–668.

[20] K. Deb, J. Sundar, Reference point based multi-objective optimization using evolutionary algorithms, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, ACM, 2006, pp. 635–642.

[21] A.J. Nebro, J.J. Durillo, J. Garcia-Nieto, C.C. Coello, F. Luna, E. Alba, SMPSO: A new PSO-based metaheuristic for multi-objective optimization, in: 2009 IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making (MCDM), IEEE, 2009, pp. 66–73.
[22] A.J. Nebro, J.J. Durillo, J. García-Nieto, C. Barba-González, J. Del Ser, C.A.C. Coello, A. Benítez-Hidalgo, J.F. Aldana-Montes, Extending the speed-constrained multi-objective PSO (SMPSO) with reference point based preference articulation, in: International Conference on Parallel Problem Solving from Nature, Springer, 2018, pp. 298–310.
[23] A.B. Ruiz, M. Luque, K. Miettinen, R. Saborido, An interactive evolutionary multiobjective optimization method: interactive WASF-GA, in: International Conference on Evolutionary Multi-Criterion Optimization, Springer, 2015, pp. 249–263.
[24] T. Weise, M. Zapf, R. Chiong, A.J. Nebro, Why is optimization difficult? in: R. Chiong (Ed.), Nature-Inspired Algorithms for Optimisation, Springer, Berlin, 2009, pp. 1–50, ISBN 978-3-642-00266-3.
[25] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptual comparison, ACM Comput. Surv. 35 (3) (2003) 268–308.
[26] C.A.C. Coello, G.B. Lamont, D.A. Van Veldhuizen, et al., Evolutionary Algorithms for Solving Multi-Objective Problems, vol. 5, Springer, 2007.
[27] J. Kennedy, Particle swarm optimization, Encyclopedia Mach. Learn. (2010) 760–766.
[28] M. Dorigo, M. Birattari, Ant Colony Optimization, Springer, 2010.
[29] C. Coello Coello, G. Lamont, D. van Veldhuizen, Multi-Objective Optimization Using Evolutionary Algorithms, second ed., John Wiley & Sons, Inc., NY, USA, 2007.
[30] M. Farina, K. Deb, P. Amato, Dynamic multiobjective optimization problems: test cases, approximations, and applications, IEEE Trans. Evol. Comput. 8 (5) (2004) 425–442, http://dx.doi.org/10.1109/TEVC.2004.831456.
[31] C. Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: A survey on big data, Inform. Sci. 275 (2014) 314–347.
[32] S. Owen, S. Owen, Mahout in Action, Manning Shelter Island, 2012.
[33] A. Bhardwaj, A. Kumar, Y. Narayan, P. Kumar, et al., Big data emerging technologies: A Casestudy with analyzing twitter data using apache hive, in: 2015 2nd International Conference on Recent Advances in Engineering & Computational Sciences, RAECS, IEEE, 2015, pp. 1–6.
[34] J. Heer, S. Kandel, Interactive analysis of big data, XRDS: Crossroads ACM Mag. Stud. 19 (1) (2012) 50–54.
[35] C. Lam, Hadoop in action, Manning Publications Co., 2010.
[36] M. Bhandarkar, Mapreduce programming with apache hadoop, in: 2010 IEEE International Symposium on Parallel & Distributed Processing, IPDPS, IEEE, 2010, p. 1.
[37] S. Kandel, A. Paepcke, J. Hellerstein, J. Heer, Wrangler: Interactive visual specification of data transformation scripts, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2011, pp. 3363–3372.
[38] S. Kandel, R. Parikh, A. Paepcke, J.M. Hellerstein, J. Heer, Profiler: Integrated statistical analysis and visualization for data quality assessment, in: Proceedings of the International Working Conference on Advanced Visual Interfaces, ACM, 2012, pp. 547–554.
[39] A. Mohamed, M.K. Najafabadi, Y.B. Wah, E.A.K. Zaman, R. Maskat, The state of the art and taxonomy of big data analytics: view from new big data framework, Artif. Intell. Rev. (2019) 1–49.
[40] A.I. Stojnev, D.H. Stojanović, Software systems for processing and analysis of big data and event streams, in: 2017 13th International Conference on Advanced Technologies, Systems and Services in Telecommunications, TELSIKS, 2017, pp. 128–131, http://dx.doi.org/10.1109/TELSKS.2017.8246245.
[41] X. Liao, Z. Gao, W. Ji, Y. Wang, An enforcement of real time scheduling in spark streaming, in: 2015 Sixth International Green and Sustainable Computing Conference, IGSC, IEEE, 2015, pp. 1–6.
[42] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: Fault-tolerant streaming computation at scale, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 423–438.
[43] D. Kakadia, Apache Mesos Essentials, Packt Publishing Ltd, 2015.
[44] V.K. Vavilapalli, A.C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache hadoop yarn: Yet another resource negotiator, in: Proceedings of the 4th Annual Symposium on Cloud Computing, ACM, 2013, p. 5.
[45] J.A. Blakeley, P.-A. Larson, F.W. Tompa, Efficiently updating materialized views, in: ACM SIGMOD Record, vol. 15(2), ACM, 1986, pp. 61–71.
[46] X. Qian, G. Wiederhold, Incremental recomputation of active relational expressions, IEEE Trans. Knowl. Data Eng. 3 (3) (1991) 337–341.
[47] S. Salloum, R. Dautov, X. Chen, P.X. Peng, J.Z. Huang, Big data analytics on apache spark, Int. J. Data Sci. Anal. 1 (3–4) (2016) 145–164.
[48] K. Thein, Apache kafka: Next generation distributed messaging system, Int. J. Sci. Eng. Technol. Res. 3 (47) (2014) 9478–9483.
[49] J. Kreps, N. Narkhede, J. Rao, et al., Kafka: A distributed messaging system for log processing, in: Proceedings of the NetDB, 2011, pp. 1–7.
[50] A.I. Stojnev, D.H. Stojanović, Software systems for processing and analysis of big data and event streams, in: Advanced Technologies, Systems and Services in Telecommunications (TELSIKS), 2017 13th International Conference on, IEEE, 2017, pp. 128–131.
[51] A. Nebro, J.J. Durillo, M. Vergne, Redesigning the jmetal multi-objective optimization framework, in: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO Companion '15, ACM, 2015, pp. 1093–1100.
[52] A.J. Nebro, C. Barba-González, J.G. Nieto, J.A. Cordero, J.F.A. Montes, Design and architecture of the jMetalSP framework, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, ACM, 2017, pp. 1239–1246.
[53] G. Reinelt, TSPLIB—A Traveling salesman problem library, ORSA J. Comput. 3 (4) (1991) 376–384.