# From RAM$^3$S to SPAF: Towards a Stream Processing Abstracting Framework*

Ilaria Bartolini[1,*,†], Marco Patella[1,†]

[1]Department of Computer Science and Engineering (DISI), Alma Mater Studiorum, University of Bologna, Italy

**Abstract**

We describe the evolution of RAM$^3$S, a software infrastructure for the integration of Big Data stream processing platforms, to SPAF, an abstraction framework able to provide programmers with a simple but powerful API to ease the development of stream processing applications. By using SPAF, the programmer can easily implement real-time complex analyses of massive streams on top of a distributed computing infrastructure, able to manage the volume and velocity of (multimedia) Big Data streams.

**Keywords**

stream processing, real-time analysis, big data, multimedia data streams

## 1. Introduction

Since at least the last three decades, multimedia (MM) data have been employed in a wide range of applications. The widespread accessibility of such data is made possible by the availability of inexpensive production (cameras, sensors, etc.) and storage technologies. Moreover, MM data frequently arrives in streams, or successions of the same sort of MM objects. Substantial value can be recovered from MM streams, but there are also significant demands placed on computational capacity and analytical ability [10]. Real-time analysis, in particular, necessitates the processing of data streams at high throughput and low latency in order to take advantage of data freshness to act and make judgments rapidly. A number of Big Data platforms exist [13, 1, 9] that provide services for the management and analysis of massive amounts of streaming data, enabling evidence-based decision making across a wide range of human activities. However, the usage of such platforms is complicated for analysts, because their primary attention is on issues of fault-tolerance, increased parallelism, and so forth, rather than offering an intuitive API.

This discussion paper shows how RAM$^3$S (Real-time Analysis of Massive MultiMedia Streams) [2, 3, 4], a framework we developed to integrate Big Data management platforms, has evolved int SPAF, a Stream Processing Abstracting Framework. The use of SPAF makes much easier, for researchers, to implement complex analyses of massive MM streams exploiting a distributed computing environment, without specific knowledge of the underlying infrastructure.

*SEBD 2023: 31st Symposium on Advanced Database System, July 02–05, 2023, Galzignano Terme, Padua, Italy*

*Discussion Paper

*Corresponding author.

†These authors contributed equally.

ilaria.bartolini@unibo.it (I. Bartolini); marco.patella@unibo.it (M. Patella)

0000-0002-8074-1129 (I. Bartolini); 0000-0003-2655-0759 (M. Patella)

### 1.1. Running Example: Face Recognition

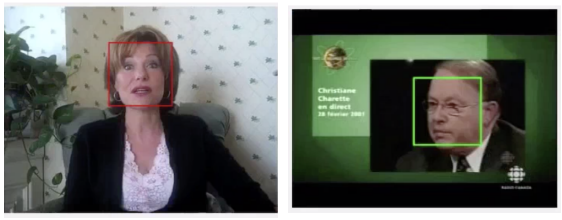

**Figure 1:** Face recognition use case: known (left) and unknown (right) face.

To exemplify the many RAM$^3$S/SPAF components throughout the paper, we will use the first RAM$^3$S application we created [2], i.e., automatic facial identification in videos, see Figure 1. For the purpose of face recognition, each incoming video is first streamed as a sequence of frames and each incoming frame is analyzed to: 1. verify if it contains a face (we use the well-known Viola–Jones algorithm [12]), 2. compare the (possibly) discovered face against a number of "known" faces, to retrieve the known face most similar to the input face (using a technique based on principal component analysis using eigenfaces [11]), 3. "recognizing" the face if there is a high enough similarity score between the newly discovered face and its most similar known face (otherwise, the face is marked as unknown).

## 2. Introducing RAM$^3$S

Several modern applications (e.g., face detection for the automatic identification of "suspect" people [11], recognition of suspicious behavior from videos [8] or audio events [7]) require a large amount of data to be analyzed as soon as these are available, so as to exploit their "freshness". In such Big Data *stream processing* paradigm, the efficiency of the system depends on the amount of data processed, keeping low latency, at the second, or even millisecond, level. For this, a number of Stream Processing Engines (SPEs) have been introduced, among which Storm, (http://storm.apache.org), Flink (http://flink.apache.org), Samza (http://samza.apache.org), and Spark Streaming (http://spark.apache.org). Abstracting by specificities of each SPE (see [13, 1, 9] for details on Spark, Flink, and Samza respectively), the following common characteristics can be discovered (see also [2] for a more detailed comparison of the SPEs considered in RAM$^3$S): some nodes in the architecture are in charge of receiving the input data stream, thus containing the data acquisition logic; other nodes perform the actual data processing; finally, data sources and data processing nodes are connected to realize the data processing architecture, which takes the form of a Directed Acyclic Graph (DAG), where arcs represent the data flow between nodes.

The initial challenge that led us to the introduction of RAM$^3$S was the implementation of a number of security-related MM stream processing applications on top of such SPEs, with the goal of comparing them by way of several performance KPI, such as throughput, scalability, latency, etc. This required to re-implement every MM stream analysis application on top of each SPE, leading to huge code replication and other inefficiencies. We were therefore led to realize a middleware software framework to allow users to avoid having to deal with details of each specific SPE (such as how fault-tolerance is achieved, how stream data are stored, etc.), and easily extend already available (centralized) software to a *scaled out* solution. This way, we strove to bridge the technological gap between facilities provided by SPEs and advanced applications (whose implementation to a distributed computing scenario could be daunting).

## 2.1. RAM$^3$S: *Almost* a Framework

Our original goal for RAM$^3$S was to create a *framework*, according to the definition in [6]: *"A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software... You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework."* A framework, therefore, determines the overall architecture of an application and focuses on the *reusability* of a solution architecture and its *extensibility*, so that it can evolve through future integrations.

The current version of RAM$^3$S allows to experiment with the various SPEs (namely, Spark Streaming, Storm, Flink, and Samza) by providing a separate "generic" application for each of them. Each of such applications allows the execution of a specific example on the respective SPE. RAM$^3$S interposes an abstraction layer based on interfaces between the generic and the example applications (see Figure 2). Such interfaces model aspects of both data streaming and data processing: The `Receiver` interface represents the external system from which the application receives data; its `receive` method accommodates the logic of receiving the single processable object from the external system. The `Analyzer` interface represents the container of all the processing logic of the application; its `analyze` methods takes a MM object as input, generating a single object as result. Finally, the `ApplicationFactory` interface is responsible for representing the application as a unit; in essence, it serves as a collector for the `Analyzer` implementation and for the `Receiver` implementation, by instantiating the concrete `Analyzer` and `Receiver` type classes defined in the application context.

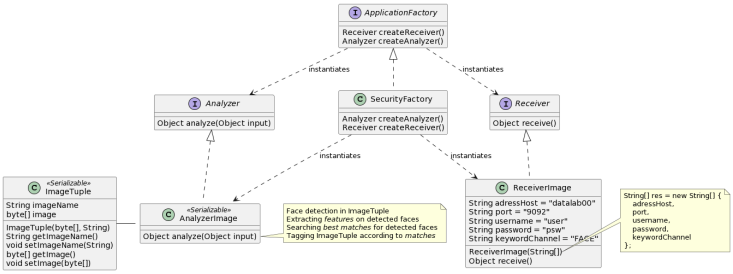


**Figure 2:** RAM$^3$S programming interface and its instantiation for face recognition.

Let us now consider how RAM$^3$S interfaces are used by a generic application. The purpose of such application is to map the application (defined in terms of the above interfaces) to the relevant SPE, thus executing its logic on the underlying runtime framework.

```
public static void main(String[] args) {
   ApplicationFactory factory =
         SecurityFactory.class.newInstance();
   Receiver receiver = factory.createReceiver();
   Analyzer analyzer = factory.createAnalyzer();
   /* Boilerplate Code specific for the SPE */
   ...
}
```

**Figure 3:** Code of a generic RAM$^3$S application.

Generic application code is always completely specified within the `main` method and has a recurring structure, presented in Figure 3. The concrete factory is used to create the `Receiver` and `Analyzer`; then such objects have to be translated into objects and procedures specific of the underlying SPE, as exposed by its APIs; the purpose of this step is to concretely establish the connection to the specific data processing infrastructure and to implement the application logic. Clearly, this is so-called boilerplate code, peculiar to the underlying SPE, that has to be repeated, almost verbatim, for each specific application.

A final component of RAM$^3$S is the one used to support different message brokers [4]. The abstraction layer devoted to message brokers consists of the `messageBroker` package shown in Figure 4. This package includes interfaces for the abstract representation of "readers" and "writers" (`Reader` and `Writer` interfaces) and concrete classes for the implementation of interfaces for a specific message broker (the figure reports only those for Apache Kafka, i.e., `KafkaReader` and `KafkaWriter`). For each stream processing framework (Figure 4 depicts the example of Flink), an additional layer is needed to map the `Reader` and `Writer` classes into, respectively, the source and destination of data processed by the application.

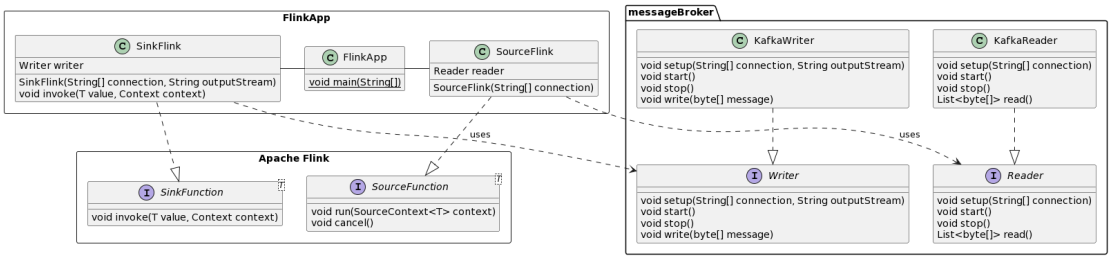


**Figure 4:** RAM$^3$S support for message brokers.

From the analysis we have presented, we can conclude that, at present, RAM$^3$S allows different stream processing applications to be executed in a facilitated manner, but is not yet able to allow the definition of a new application without having to write part of the code of RAM$^3$S itself. This drawback places RAM$^3$S in the role of a "quasi-framework". On the other hand, the message broker support is independent of the application code (thus satisfying the reusability requirement) and makes it possible to decouple the implementation of the read and write "adapters" of a certain message broker from the underlying stream processing engine (thus also offering extensibility). In the next section, we will introduce SPAF, whose main goal is to enable the use of RAM$^3$S according to the original intent, namely to facilitate the creation of new stream processing applications, while offering both reusability and extensibility.

## 3. From RAM$^3$S to SPAF

Before describing the concepts that have been used to define our Stream Processing Abstraction Framework, SPAF, it is useful to recollect the original requirements for RAM$^3$S: **Facilitate the creation of stream processing applications:** the SPAF API should explicitly expose the pivotal concepts of stream processing and, more importantly, allow the application to be defined by writing code as close as possible to a description in natural language. Checking type-safety at compile time would be also helpful. **Framework independence:** here, the target user is not the programmer of applications, rather the developer who wants to extend SPAF to work with a different SPE. Such programming interface is called Service Provider Interface (SPI) and is a programming pattern supported natively by Java. **Connector independence:** it should be possible to integrate, in a pluggable manner, new connector providers (e.g., message queues, file systems, databases), again exposing a SPI to be implemented by developers. SPAF will therefore expose a dual abstraction layer: one for SPEs and one for input/output supports. Connector

abstraction actually concerns the application programmer as well, since the SPAF API should relieve the programmer of the implementation details regarding the use of each connector's libraries and allow him to specify sources and destinations in a declarative manner.

In the first version of SPAF, a number of simplifying assumptions has been introduced to ease implementation of the previously described requisites: 1. Connectors will be limited to message queues, as provided by well-known message brokers such as Apache Kafka and RabbitMQ. 2. Streams will be composed exclusively by key-value pairs. 3. No support will be provided for storing intermediate computation results, i.e., stream processing will be stateless. 4. The processors operating on streams will accept a single input stream and a single output stream; essentially, it will be possible to define only "linear" topologies. 5. It will be possible to specify only the logical topology (definition of the transformation process from input to output) and not the physical topology, i.e., defining of how various computational elements of the logical topology (sources, transformation operators, destinations) are distributed on the physical nodes.

### 3.1. SPAF Architecture

Figure 5 shows the general architecture of SPAF. The application programmers will use the *user-facing* API to implement their stream processing application using Java code. Developers wanting to extend SPAF to include new SPEs (or connectors) will use the *provider-facing* SPI to write the *glue code* that allows bridging SPAF concepts to those peculiar to the SPE (see below).

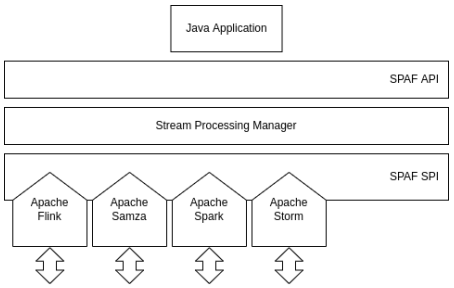


**Figure 5:** SPAF architecture.

The first step in the definition of SPAF was to provide a general model of the stream processing problem, thus defining a set of entities that are common to all SPEs (see Figure 6).

The `Context` entity represents the actual execution environment in which the abstractly defined application will be implemented. This therefore acts as a "bridge" between the abstract SPAF world and the concrete world made available by SPEs. The `run` method of the `Context` class is the "entry point" of the stream processing application, just like the `main` method of a Java class. Through the `run` method, we submit as a parameter the application, defined in terms the SPAF API, to the underlying SPE; the SPE will then "interpret" the application, "translating" it into an equivalent representation using its specific concepts. Implementation of the `Context` is thus totally peculiar to each SPE; for this, the class `Config` allows the pro-

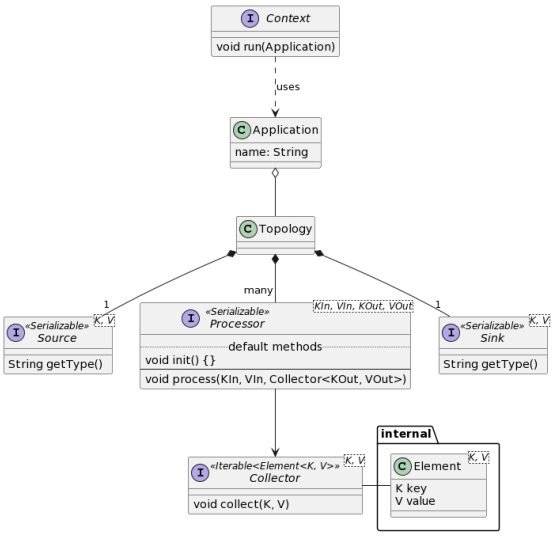


**Figure 6:** Main SPAF classes and interfaces.

grammer to configure the execution environment in an abstract way (on the API side) and to implement such configurations in a provider-specific way (on the SPI side).

A `Topology` defines the computational logic of a stream processing application, that is, how the input data are transformed into the output data. As said, in the first version of SPAF only linear topologies are supported. We conceived the framework to accept specifications of the logic of Processor nodes also via lambda-functions and present a *fluent* API to compose the topology. To achieve this, we exploited a "creational-type" design pattern called *Builder*, allowing complex objects (like a topology) to be constructed step by step; this could also re-used in future versions of SPAF, where it may be convenient to change the way the topology is represented (e.g., to support DAG topologies).

The concept of `Application` basically coincides with the one of `Topology`, essentially adding descriptive information only. Conceptually, however, a stream processing application could define more than one topology: this is why the two entities are separated, although, in this first version of SPAF, there is a 1-1 relationship between `Application` and `Topology`.

`Source` and `Sink` clearly denote the source and destination, respectively, for data in a stream processing application, while a `Processor` represents a node in the `Topology`, implementing a processing step that is used to transform data, thus realizing the actual data processing logic. `Processors` can be thought as "black boxes" with a single input and a single output, where data transformation can be defined arbitrarily through the `process` method. The additional `init` method can be defined in those cases needing a one-off initialization of the `Processor`.

Finally, the `Element` entity represents the only data type that can be streamed in a SPAF `Topology`. The `Element` class is somewhat hidden from the application programmer, while its use appears evident in the SPI layer, for both SPEs and connectors.

## 3.2. Developing an Application using SPAF

To create an application in SPAF, it is necessary to follow some steps, mostly independent of the specific streaming application logic to be created. In the following, we will exemplify the creation of an application for the use case of face recognition in videos (see Section 1.1): As it will be clear, the application-specific code can be easily distinguished from the generic SPAF-based code. The code needed for the complete specification of a SPAF application has actually a dual nature: some *declarative* code, included in a configuration file, needed to specify the customization of SPAF entities, like `Context`, `Source`, or `Sink`, and some *procedural* code, used to instantiate SPAF classes and to specify the actual stream processing application logic; this is included in the `main` method of the application and, again, is mainly boilerplate code.

Figure 7 shows, for the specific use case, (left) a possible config file, with specification of the context (using Flink as SPE), of the application, and of the input and output connectors (using Kafka), and (right) the code of the `main` method of the `FaceRecognition` application. It is clear that most of the code is indeed boilerplate, i.e., repeated with no variation across different applications. The only specific part is at step 3., where we declare that this application is composed of three main phases: 1) detection of faces in each image, 2) recognition of detected faces, and 3) (possible) marking of recognized faces (note that these correspond exactly with the three steps illustrated in Section 1.1). Obviously, the programmer should also write the code for each individual `Processor`, but this is absolutely independent of the underlying stream

```
context {
   flink {
      local = true
      web-ui = true
      web-ui-port = 10081
   }
}

application {
   name = "Face Recognition"
   dataset-path = "/tmp/training-faces/"
}

source {
   type = kafka
   bootstrap-servers = "kafka:9092"
   topic = FACES
}

sink {
   type = kafka
   bootstrap-servers = "kafka:9092"
   topic = RECOGNIZED_FACES
}
```

```
// 1. configuration and creation of the execution environment
Config config = ConfigFactory.load();
Context context = StreamProcessing.createContextFactory().createContext(config);
// 2. definition of input and output streams
Source<String, String> source = StreamProcessing.createSource(config);
Sink<String, String> sink = StreamProcessing.createSink(config);
// 3. definition of the logical topology (i.e., how data in transformed in processing
      nodes)
Topology topology = new Topology()
  .setSource("Source", source)
  .addProcessor("FaceDetector", new FaceDetectionProcessor(), "Source")
  .addProcessor("FaceRecognizer", new FaceRecognitionProcessor(config), "FaceDetector")
  .addProcessor("FaceMarker", new PersonFaceMarkingProcessor(), "FaceRecognizer")
  .setSink("Sink", sink);
// 4. creation of the application
Application application = new Application()
  .withName(config.getString("application.name"))
  .withTopology(topology);
// 5. launch of the application
context.run(application);
```

**Figure 7:** Configuration file (left) and code (right) for the SPAF face recognition application.

processing infrastructure (following the original goal of RAM$^3$S). Step 3. also demonstrates the use of the Builder pattern, where each `Processor` refers to its predecessor through its id.

A dimension of fundamental importance, which should be taken into account when choosing any development tool, is the so-called "learning curve", which relates the level of knowledge and the time invested in learning a new thing. SPAF plays the role of the "guide" that takes the programmer through the discovery of stream processing concepts, providing a logical path that facilitates their understanding and thus making the learning curve of stream processing less daunting. SPAF is therefore able to make life easier for the inexperienced programmer who must venture into the world of stream processing for the first time, and have that journey less treacherous (like Virgil, accompanying Dante through the "hell" of stream processing).

## 4. Discussion

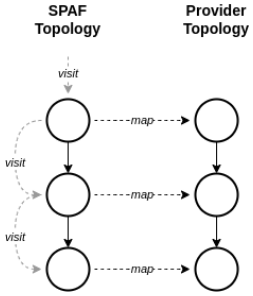


**Figure 8:** Mapping topologies.

In the previous section, where we described the main SPAF concepts and illustrated how they are used to implement a specific stream processing application, we deliberately omitted several issues that have been tackled in the design and implementation of SPAF, like the relationship between SPEs and connectors, or the one between type-safety and serialization. Additional details on the SPAF technicalities can be found in [5]. In this final section, we want to highlight two interesting concepts, both related to the constructions of DAG-shaped topologies, namely the SPAF representation of topologies and the so-called *super-topologies*.

When considering the representation of topologies in SPAF, we should remind that the key operation for a topology is how the SPAF stream processing layer (see Figure 5) is able to map the `Topology` entity in the corresponding topology of the underlying SPE (who will then autonomously map it to a physical topology on computing nodes). Since

this version of SPAF only accepts linear topologies, this mapping is extremely simple, being implemented as a loop visiting all topology nodes in an ordered way. Refactoring the `Source`, `Processor`, and `Sink` entities (Figure 6) currently used to build topologies, an abstract entity `TopologyNode`, exposing a common interface, could be created. In this way, `Sources`, `Processors`, and `Sinks` could be considered as `TopologyNode` instances, allowing to use polymorphism to implement simple (and elegant) algorithms for mapping topologies, e.g., by exploiting a *Visitor* pattern based on a topological sort of the DAG.

Finally, we consider the use of multiple SPAF applications, in a simultaneous and coordinated manner, with the aim of solving a stream processing problem in a "highly distributed" way (we will soon clarify what we mean by this adjective). The main idea is based on the decomposition of the stream processing problem into sub-problems, and in solving them through the use of multiple, independent but cooperative, SPAF projects. Each SPAF application will define a certain logical topology able of solve a certain sub-problem; each topology will receive data via the connectors provided by SPAF, process them, and send the processed data back to the outside world. In this scenario, message queues are used as means of communication between the topologies of individual projects. In other words, we can implement a topology of topologies (see Figure 9). Individual topologies, in fact, can be thought as "black boxes", processing nodes of a DAG, receiving and sending data via the arcs connecting them, the latter realized by different message queues. What we just described corresponds precisely to the definition of topology given in Section 3.1, but at a higher level, thus the name of super-topology, where the prefix super- is to be understood in the Latin sense of "that stands above".

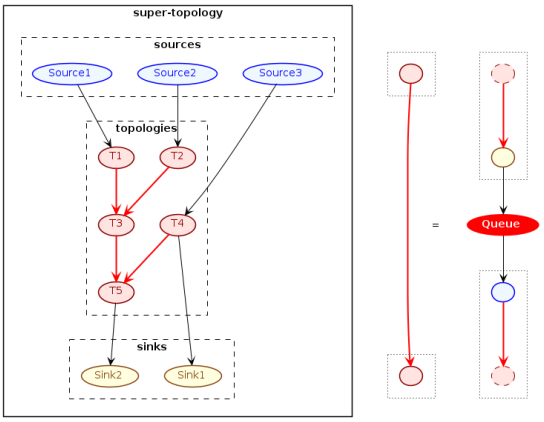


**Figure 9:** Super-topologies in SPAF.

In the diagram of Figure 9, it is shown how to conceptually realize a SPAF super-topology. It will be necessary to provide "border" sources and sinks (shown in blue and yellow, respectively), functioning as the input and outputs of the entire super-topology. The processing (red) nodes of the super-topology will instead correspond to a single SPAF application each, defining its own `Source` and `Sink`, and will consist of a (currently, only linear) `Topology` of `Processors`. Communication between the nodes of the super-topology, i.e., between SPAF applications, is realized by way of appropriate message queues between node pairs (these are represented by red arcs in the figure and exemplified on the right). Obviously, each topology will have to be configured to receive and send data to the right message queues, whether they are "border" (in the example, topologies T1, T2, T4 and T5) or "internal" (T3). The use of super-topologies in SPAF opens up some interesting scenarios: 1. each SPAF application, i.e., each node in the super-topology, can be executed by a potentially different SPE, since each application can in fact specify the desired SPAF provider independently from the others; 2. it follows that each SPAF application, i.e., each node of the super-topology, can potentially be executed on a cluster of nodes by itself; for this, we previously used the term "highly distributed" execution.

## Acknowledgments

The authors thank Nicolò Scarpa for implementing SPAF and maintaining the GitHub repository.[1] Most of the figures, as well as the analogy between SPAF and Virgil, have been created by Nicolò, and elaborated by the authors.

## References

[1] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere Platform for Big Data Analytics. VLDB Journal 23 (2014), 939–964.

[2] I. Bartolini and M. Patella. A General Framework for Real-time Analysis of Massive Multimedia Streams. Multimedia Systems 24 (2018), 391–406.

[3] I. Bartolini and M. Patella. Real-Time Stream Processing in Social Networks with RAM$^3$S. Future Internet 11 (2019), 249.

[4] I. Bartolini and M. Patella. The Metamorphosis (of RAM$^3$S). Applied Sciences 11(24) (2021), 11584.

[5] I. Bartolini and M. Patella. A Stream Processing Abstraction Framework. Submitted for publication.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[7] K. Łopatka, J.Kotus, and A. Czyżewski. Detection, Classification and Localization of Acoustic Events in the Presence of Background Noise for Acoustic Surveillance of Hazardous Situations. Multimedia Tools and Applications 75 (2016), 10407–10439.

[8] C. Mu, J. Xie, W. Yan, T. Liu, and P. Li. A Fast Recognition Algorithm for Suspicious Behavior in High Definition Videos. Multimedia Systems 22 (2016), 275–285.

[9] S.A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R.H. Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. Proceedings of the VLDB Endowment 10 (2017), 1634–1645.

[10] M. Tang, S. Pongpaichet, and R. Jain. Research Challenges in Developing Multimedia Systems for Managing Emergency Situations. Proceedings of the 24th ACM international conference on Multimedia (ACM MM 2016), Amsterdam, The Netherlands, 15–19 October 2016; pp. 938–947.

[11] M. Turk and A.P. Pentland. Face Recognition Using Eigenfaces. In Proceedings of the 1991 Conference on Computer Vision and Pattern Recognition (CVPR 1991), Lahaina, HI, USA, 3–6 June 1991; pp. 586–591.

[12] P. Viola and M. Jones. Rapid Object Detection Using a Boosted Cascade of Simple Features. In Proceedings of the 2001 Conference on Computer Vision and Pattern Recognition (CVPR 2001), Kauai, HI, USA, 8–14 December 2001; pp. 511–518.

[13] 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), Boston, MA, USA, 22 June 2010; p. 10.

[1] SPAF is distributed under the Apache 2.0 License.