Towards Flexible Event Processing in Distributed Data Streams *

Sebastian Bothe
Fraunhofer IAIS
sebastian.bothe@ iais.fraunhofer.de

Vasiliki Manikaki
Technical University of Crete
manikaki@softnet.tuc.gr

Antonios Deligiannakis
Technical University of Crete
adeli@softnet.tuc.gr

Michael Mock
Fraunhofer IAIS
michael.mock@ iais.fraunhofer.de

ABSTRACT

The FERARI project aims to develop a highly scalable distributed streaming architecture supporting complex event processing in a communication-efficient manner. Two key requirements for our system are that its architecture is not tied to the underlying streaming platform used in its implementation and that it allows the easy definition of communication-efficient methods for monitoring a global condition over a distributed set of states. In this paper we present the architecture of our system and explain how these key requirements are met. Concerning the actual implementation of our system in a scalable distributed streaming platform, it is reasonable not to re-invent the wheel but to use one of the actual Big Data Streaming platforms as a starting point. For this reason, we evaluate some popular platforms and discuss whether they meet our requirements.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

1. INTRODUCTION

In recent years, an area with great future potential for Big Data is machine-to-machine interaction (M2M), and the Internet of Things. Examples of relevant applications include smart energy grids, car-to-car communication, mobile network quality monitoring, optimizing operation of large and complex systems, fault detection in clouds, automated negotiation systems – all these have been identified as important hot use cases for Big Data.

Current Big Data technologies, developed for systems that process and analyze human generated data such streams, managing social networks at Facebook, or indexing web pages at Google, seem inadequate for processing of such M2M applications. In order to understand this, note that the data volumes generated from M2M interaction surpass by far the amount of data generated by humans. M2M data is typically required to be processed in real-time as it is produced, it is predominantly transient (does not need to be and is too large to be stored for future reference), and is typically much more structured in nature than human-generated data.

Due to the sheer size of M2M data, approaches that seek to centralize this data are not an option, as they (i) would require enormous infrastructures both for storage, as well as for the required bandwidth for transmitting this data, (ii) would impose unnecessary latency due to the data shuffling possible, and (iii) do not consider the characteristics and limitations of the data sources of M2M data - sensors are often the sources of M2M data and constant communication of sensor readings would quickly drain the energy of sensor nodes. It is thus important to be able to process M2M data and to detect important events without centralizing the collected data, but rather doing as much processing and filtering of the data at the nodes that produce it.

The project *Flexible Event pRocessing for big dAta aRchItectures* (FERARI) aims at developing a highly scalable distributed streaming architecture supporting complex event processing (CEP) in a communication-efficient manner. While most CEP systems are built on the premise that primitive events are obtained and transmitted by the remote data sources based on their own data, a key element of the FERARI architecture will include the development of communication efficient distributed methods for also detecting events, expressed over the data of multiple nodes, in a distributed manner. We consider the important case of complex events that can be expressed as a monitoring task that alerts whenever a complex function, expressed over the data of multiple nodes, has exceeded a threshold. In order to make the complex event processing feasible, a key component is to perform in-situ processing at the nodes generating the data, thus avoiding continuously pushing related data or events to our CEP engine. A key component that we will utilize for such distributed monitoring tasks is the recently developed [10, 11, 12, 13, 8, 7, 9] geometric approach. The details of this geometric approach are presented in Section 2.

We present the general architecture of FERARI and argue that a flexible CEP system for M2M data should not be tied to a specific implementation using existing stream processing systems as its infrastructure. To develop a generic architecture, in Section 3 we specify the essential building blocks that it must contain and then consider which of some existing big data streaming platforms seems more appropriate for our actual implementation. Given our architecture, in Section 4 we explain how an important part of this architecture, namely the distributed detection of events using the geometric approach, can be developed in an existing open source platform, such as Apache Storm, and explain how some distributed monitoring tasks (that may use the geometric approach, or not) fit within our architecture.

2. BASICS - THE GEOMETRIC APPROACH

We now describe in more detail the geometric approach for function monitoring over a distributed system of n sites. Figure 1 demonstrates the basic ideas of the geometric approach that we discuss in this section.

Each site S_i maintains a local d-dimensional vector, termed as the *local statistics vector*, with the j-th ($j = 1 \dots d$) element of the local statistics vector of S_i denoted as $\vec{v}_{j,i}$. All sites contain a vector of the same dimensionality (i.e., number of elements). The *global statistics vector* $\vec{v}$ is computed as the average[1] amongst all local statistics vectors. Thus, the j-th component of the global statistics vector, denoted as $\vec{v}_j$ is computed as: $\vec{v}_j = \frac{1}{n} \sum_{i=1}^{n} \vec{v}_{j,i}$.

For the framework to be applicable, any supported monitoring function $f : \mathcal{R}^d \rightarrow \mathcal{R}$ must be expressed over the global statistics vector $\vec{v}$ (thus, over the average of all local statistics vectors). An important feature is the wide applicability of the geometric approach, as the threshold function can in general be non-linear. Given a threshold T, the framework in [10, 11, 12, 13, 8, 7, 9] can safely determine whether $f(\vec{v}) > T$.

The geometric approach decomposes the monitoring task into a set of constraints (one per site) that each site can monitor *locally*. To achieve this, during the operation of the algorithm, each site S_i maintains (i) the estimate vector $\vec{e}$, which is equal to the global statistics vector $\vec{v}$ computed by the local statistics vectors transmitted by sites at certain times, and (ii) a delta vector $\Delta\vec{v}_i$, denoting the difference of the current local statistics vector from the last local statistic vector that S_i has transmitted. Based on these two quantities, S_i calculates its drift vector $\vec{u}_i = \vec{e} + \Delta\vec{v}_i$. Additional optimization have been developed in the framework, such as the ability to *balance* only a portion of the network in case of violations. In that case, an additional *slack* vector needs to be maintained and added in the calculation of the drift vector.

The domain space $\mathcal{R}^d$ represents the potential locations of the global statistics vector at any time. Let all points in $\mathcal{R}^d$ where $f(\vec{v}) <= T$ be colored by the same color (i.e., white in Figure 1), while the remaining points be colored by a different color (i.e., green in Figure 1). Because the sites do not perform transmissions at each time period, the current global statistics vector $\vec{v}$ is not known to the sites. However, what is guaranteed is that $\vec{v}$ will always lie within the convex hull $Conv(\vec{u}_1, \dots, \vec{u}_n)$ of the drift vectors and, thus, within the convex hull $Conv(\vec{e}, \vec{u}_1, \dots, \vec{u}_n)$ of the drift vectors and the estimate vector. Thus, if $Conv(\vec{e}, \vec{u}_1, \dots, \vec{u}_n)$ is *monochromatic* (i.e., either entirely below/equal to the threshold, or entirely above to the threshold), then all sites are certain about the color of the function $f()$, since this will coincide with the color of $f(\vec{e})$. Of course, each node cannot compute $Conv(\vec{e}, \vec{u}_1, \dots, \vec{u}_n)$, since it is not aware of the current drift vectors of other sites. However, an important observation [11] is that if each site monitors the sphere $B(\vec{e}, \vec{u}_i)$ constructed with diameter the estimate vector and its own drift vector, then the union of these spheres covers the convex hull. Thus, it suffices for each node to simply monitor whether its sphere is monochromatic. If all the spheres are monochromatic, then the convex hull is also monochromatic and, thus, $f(\vec{v})$ has the same color as $f(\vec{e})$. Otherwise, nodes transmit their local statistics vectors, and a new estimate vector is computed and made known to all nodes.

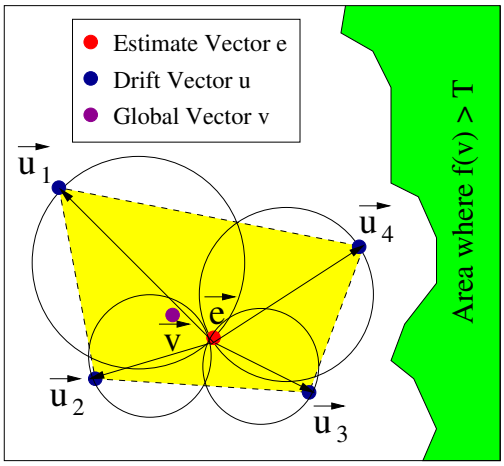

Figure 1: Local constraints using the Geometric Approach. Each node constructs a sphere with diameter the drift vector $\vec{u}$ of the node and the estimate vector $\vec{e}$. The global statistics vector $\vec{v}$ is guaranteed to lie in the convex hull of $\vec{e}$, $\vec{u}_1$, $\vec{u}_2$, $\vec{u}_3$, $\vec{u}_4$. The union of the local spheres covers the convex hull.

Using Safe-Zones. The more recent work of [8, 7, 9] simplifies the local tests performed by nodes by having each node test whether its drift vector [8, 9] or its local statistics vector [7] lies within a convex region. This test is very efficient and only depends on the complexity of the bounding convex region. For example, the work in [9] demonstrates how this convex region can be determined by the intersection of hyperplanes. In that case, the local test of each node simply checks that a tested vector lies on the "correct" side of these hyperplanes.

3. PROPOSED ARCHITECTURE

To build a flexible event processing application, we assemble an appropriate algorithmic approach and a stream processing platform. The approach we use to create this assembly is

[1]The same framework also applies when the global statistics vector is calculated as a weighted average of the local statistics vectors.

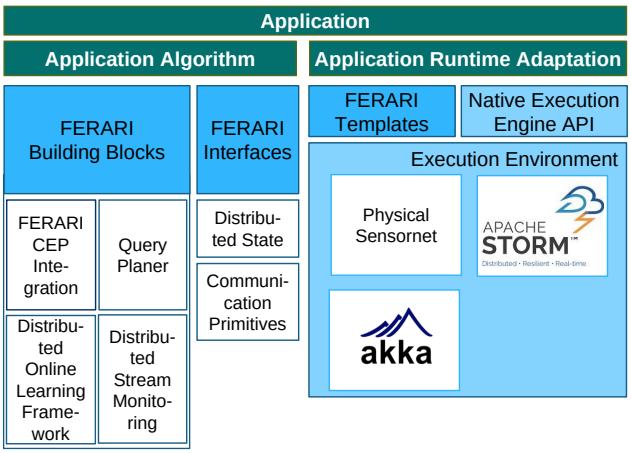

Figure 2: FERARI architecture overview.

described in Section 3.1. We show how the application can be organized such that the application core can be made independent from the execution environment. Additionally, several open source projects are working on scalable distributed streaming platforms. It is therefore reasonable not to start from the beginning. To identify a good starting point, we compare the current systems in Section 3.2.

3.1 Architecture Components

We design the FERARI architecture to allow the fast development of distributed flexible event processing applications. To achieve this, it is required that we can create new functionality on the basis of existing ones, and that we can exchange parts of the system without affecting other parts. A common approach in software development for this requirement is to decompose the complex system into smaller units, called components or modules. Each of the components has clearly defined bounds and a dedicated functionality. Additionally, the interaction between the components is defined in terms of interfaces. We use this approach and derive appropriate components and interfaces. An overview of the proposed FERARI architecture is depicted in Figure 2.

We now describe the components and the decisions, that lead to our choices. It is important to keep the architecture as generic as possible, as we need to execute applications in different runtime environments. As a runtime we consider a distributed system, that provides the actual execution context for the application. This may be a simulator, a distributed stream processing framework or even a distributed physical sensor network. Especially for in-situ processing there needs to be an adaptation optimized for the application's concrete environment. For that reason, we start the decomposition of the application by isolating its algorithm from the application's part that depends on a specific execution environment. The first part we call *application algorithm* and the latter *application runtime adaptation*. The division now makes the application algorithm independent from the selected runtime platform and can, therefore, be reused in different execution environments. This decoupling from the runtime now raises the demand to define abstract versions of mechanisms used by our distributed algorithms.

The distributed algorithms in our current set of example applications need access to communication primitives and a distributed state. The concrete mechanisms for both depend on the selected runtime adaptation, which is not accessible by the application algorithm. To enable this access, we introduce a set of *FERARI interfaces*. We develop the FERARI system using an agile process, therefore we created the interfaces that we need for the geometric approach. We want to monitor a global condition over a function, using the derived local conditions (c.f. Section 2). The local and global condition are based on a local and a global state respectively. These two parts of the state are represented by the interface for distributed state. Additionally, the coordinator needs to send information to the local node and vice versa. Primitives for sending messages in each of theses directions are provided by the other interface. The application algorithm can now be created using this interfaces. On side of the application runtime adaptation, we need to provide the concrete implementation for the mechanism exposed by the interfaces. All of the execution environments use different types of abstractions, interfaces and nomenclature. Additionally, it is a very demanding task to create a efficient runtime adaptation for an arbitrary set of possible applications. Following an agile development principle, we instead solve the more viable task of creating templates for common application types. These FERARI templates allow the re-use of runtime adaptations for different applications. They can provide a mapping of common patterns onto the runtime implementation. Beyond this, some applications may require access to special features of a runtime system. This may for instance be optimized network operations. Especially in the case of in situ processing, access to these features is important to create an efficient runtime adaptation. Therefore we provide access to these features by exposing the native API of the execution engine.

It is possible to create applications using just the FERARI interfaces and a runtime adaptation, and we give an example for this in Section 4.1. Nevertheless, writing the application code using these interfaces requires skills in distributed programming and especially knowledge about communication efficient algorithms. Therefore, we plan to provide a set of ready to use blocks that can be called from the application code directly, the *FERARI Building Blocks*. As a first example of these blocks, we describe the current status of the distributed stream monitoring block in Section 4. An outlook for the other planned blocks is given in our concluding remarks (Section 5).

3.2 Candidate Streaming Platforms

As we identified in Section 3.1, it is important to decouple the application algorithm and the execution runtime. We will now focus on candidate platforms within the existing open source streaming platform implementations. The FERARI architecture itself will be available under a liberate open source license, the Apache license. For that reason, we consider only platform candidates that are compatible with this license. Since there are a lot of streaming platform candidates, we focus on some popular ones, Storm, Trident, Spark Streaming and Akka. We require a system that can be scaled to handle the streams of data for huge volume and high velocity. In current stream processing platforms, this is achieved by horizontally scaling out, which means

additional processing nodes are added to increase the capacity. An effect of horizontal scaling is that the increased amount of nodes also increases the probability of nodes failures. Therefore, the platform needs to provide mechanisms to deal with system components failing. In Section 2 we explained how the geometric approach divides the monitoring task to a local (checking for local violations) and a global part (synchronization and determining if there is a global violation/event). These two parts interact with each other in adaptation cycles. The ability to support such cycles is a key requirement for the platform that we choose. Additionally, our application scenarios include monitoring tasks. In this application area it is important to create immediate reaction, for instance raise an alarm as soon as possible. The requirement for the platform is, therefore, that it allows processing with low latency. Another important aspect of this application scenario is that potential alarms must not be omitted. The processing of an input event may either be guaranteed by the underlying platform (i.e., by ensuring that messages are not lost), or it may be a concern of the application. We now evaluate our candidates with respect to the properties, possibility of adaptation cycles, latency of processing and guarantees for processing provided by the platform.

Apache Storm [3] initially was developed at Twitter, got open-sourced in 2011 and is now an Apache top level project. Processing in Storm is organized by a graph, the Storm topology. Input stream data items enter the topology by spouts and are called tuples in Storm terminology. Each of these tuples is processed by Storm bolt, one at a time. The approach of processing a single tuple at a time allows for low latency processing. A storm bolt can execute arbitrary Java code, and it emits new tuples as processing results. These result tuples are then processed by the next bolts in the topology. A storm topology can contain cycles, and therefore allows for the adaptation cycles that we need. The Storm system recovers from node failures, by restarting the broken processing task at a different place. In case of failures, all state associated with a crashed tasks is lost. Storm provides two types of processing guarantees, best effort and at least once processing semantics. More on Storm's implementation can be found in [14].

Trident is not an independent system - it is an abstraction layer on top of Storm. It extends Storm by introducing exactly-once execution semantics and a model for consistent states. To achieve this, it switches from processing each tuple individually to processing small amounts of tuples, the mini batches, together. In general, mini batching increases the time that elapses between the entering of a new tuple in the system and the result being available. Further, the processing topology in Trident is required to be a cycle-free graph, which conflicts with the adaptation loop requirement that we have.

Another development independent from Storm is the distributed processing framework Akka [1]. The organization of the processing units follows the actor model. The actors interact with each other by message passing and especially cyclic connections can be constructed. There are no guarantees on delivery or processing of messages - they are handled at best effort. Each single message is processed individually, therefore achieving low latency.

	Storm	Trident	Akka	Spark
Cycles	yes	no	yes	no
Processing Guarantees	best effort and at least once	exactly once	best effort	exactly once
Processing Granularity	single tuple	mini batch	single message	mini batch

Figure 3: Properties of candidate distributed stream porcessing platforms

Apache Spark Streaming [2] is an extension of the recently developed Spark processing system. Spark is a batch processing system designed for caching intermediate results. The streaming variant organizes the processing of the stream in mini batches. Each operation on these batches is guaranteed to be performed exactly once. There is no way known to the authors for constructing loops in the processing operations.

A summary of the supported features of the evaluated platforms is provided in Table 3. Our analysis reveals that Storm provides most of the features we need for our system. Akka remains to be an interesting candidate, if the targeted application does not require at least once processing guarantees.

4. DISTRIBUTED STREAM MONITORING

We now focus on the important *Distributed Stream Monitoring* building block and explain how distributed monitoring functions can be incorporated in our framework. As we have mentioned, we are interested in detecting events, which are emitted when a function, computed over the data of different distributed nodes, has crossed a specific global threshold. We give a basic example, counting the number of distinct items in streams, to show the usage of the FERARI interfaces in Section 4.1. This basic example does not yet make use of the geometric approach. Since it is desirable to allow code reusability for different monitoring functions, we then present how declared distributed monitored functions can be implemented using a hierarchy that we define. Our function hierarchy alleviates the development of many important details of the geometric method, requiring minimal new code for each new monitoring function, while at the same time providing support for both the original geometric approach, as presented in [10, 11, 12, 13], as well as the more recent Safe-Zone [8, 7, 9] approach, which improves upon the original approach. In Section 4.2 we present our function hierarchy that allows for easily defining and incorporating new functions for distributed monitoring. Please note that both methods presented in Sections 4.1.1. and 4.2.1 do not depend on the underlying execution environment (i.e., Storm, SPARK etc). We show in Sections 4.1.2 and 4.1.3 respectively, how they can be be mapped to the FERARI architecture.

4.1 Monitoring Global Threshold with Count Distinct

4.1.1 Application Algorithm

Counting the number of distinct elements in streams of data is a common pattern in various types of applications. For

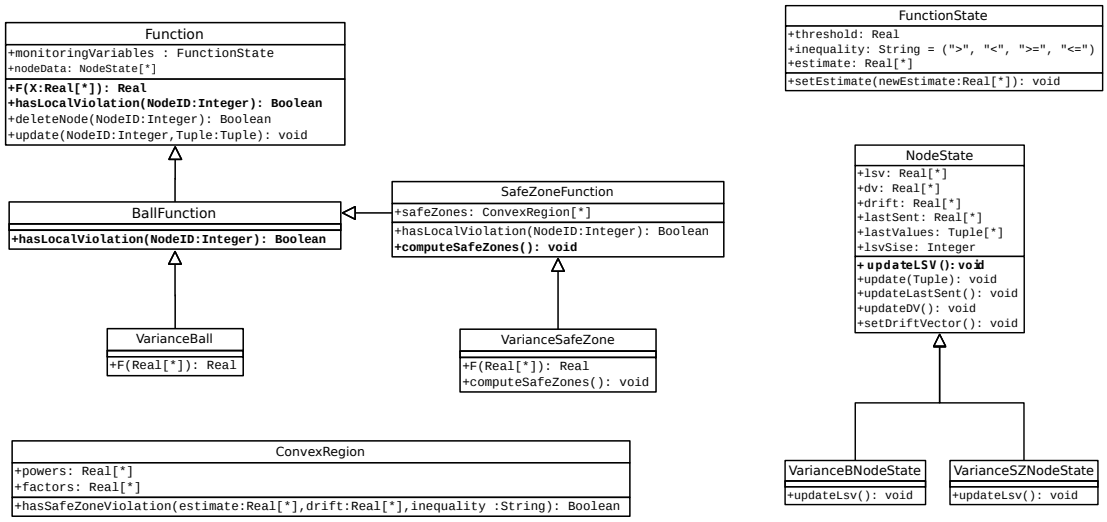

Figure 4: Function Class Hierarchy.

instance, in a mobile fraud detection scenario, it is important to keep track of the number of different locations a mobile device is used within a certain period of time. If a device changes too many times, this could be an indication of fraudulent usage. Another example for the same counting pattern, originating from a different scenario is to determine the popularity of an artist. For instance a service like last.fm[2] may monitor streams of events, in which each of the events in a stream corresponds to a user listening to a song. The popularity of an artist is measured by the number of distinct users listening to songs of the artist. To discover artists getting popular, it is relevant to know when an artists exceeded a certain global threshold, the count of distinct listeners.

An data sample of such events was collected by the author of [4] and is now publicly available at the webpage[3]. We will use these data and describe, how the pattern of counting distinct elements in streams of data can be instantiated using the proposed FERARI architecture. There are two application dependent blocks we need to fill, the application algorithm and the runtime adaption. Concerning the application algorithm, it is common practice in the streaming scenario to approximate the count of distinct items by applying sketching techniques. One type of such sketches are linear sketches, which fit our needs in this example. The details of the sketching algorithm can be found for instance in [5]. The sketch is a compact synopsis of our data that can be communicated more efficiently between the local nodes and the coordinator. For each artist we maintain a sketch to count the distinct at each of the distributed processing units. Such a sketch is communicated with the global coordinator only if it is required, i.e. if the count of distinct users has increased by some percentage or by a fixed amount. The coordinator, than also updates it's global sketch for this

[2] `http://www.last.fm/`

[3] `http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-1K.html`

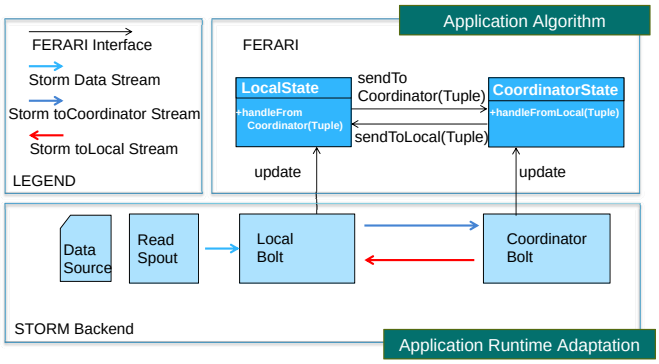

Figure 5: Interface for application algorithm.

artist appropriately. Additionally, the coordinator detects whether the global threshold has been crossed. In the case of a global threshold violation, the coordinator takes appropriate actions, for instance raises the information about the new popular artist and interacts with the local processing units to make them reset the counters.

4.1.2 Mapping to the FERARI Architecture

This algorithm can now be created using the FERARI interfaces. The local part implements the `LocalState` interface, with two methods `update` and `handleFromCoordinator`. The `update` method is called on incoming data and provides new the listen events. `HandleFromCoordinatorl` is used to reset the local counters as indicated by the coordinator. Further, increased counts are reported to the coordinator, via the `sendToCoordinator` method. For the global part, the coordinator, the equally named interface `CoordinatorState` is implemented. Here, we use the update method to receive the notifications of increased counts. As already mentioned, the coordinator resets counters in regular intervals and notifies the local units by the `send-`

ToLocal method. Note that the application algorithm now is formulated using FERARI interfaces and does not dependend on the runtime. The application algorithm can now be executed using an appropriate runtime system as proposed in Section 3. We choose the Storm runtime for our example and setup the topology as follows:

- A spout inserts the listen events as Storm tuples to the topology.
- At random choice the tuples reach one of the `LocalBolts`.
- The `LocalBolts` are connected with the `CoordinatorBolt` by a named channel, a Storm stream. This connection is achieved using a Storm `global grouping`. Vice versa, the Coordinator uses another stream to the `LocalBolts`, which is translated to a Storm `all grouping` in the topology.

The Storm runtime adaption for this example, now maps `sendToLocal` and `sendToGlobal` to emitting tuples on the dedicated channels for each of the operations. The processing in the `LocalBolt` decides on basis of the channel name, if the `update` method or the `handleFromCoordinator` is invoked. In this example the coordinator only receives tuples containing the updated sketches from the `LocalBolts`, therefore the mapping to the processing hook is unique. The complete view on the FERARI interfaces and the link to the runtime adaption is given in Figure 5.

We now describe our more general solution to define monitoring tasks exploiting the geometric approach.

4.2 Monitoring Global Function Thresholds

4.2.1 Application Algorithm

In Figure 4 we present the function hierarchy that we have developed in order to facilitate writing of applications that make use of the geometric approach. The abstract class `Function` represents the core elements that the geometric approach contains. In this class, the abstract method `F` must be provided for all developed functions and simply returns the value of the function, computed over a multidimensional point in the input domain. When an instance of a function is created, this is done by also specifying two important parameters: a `threshold` value and a parameter `inequality`, which may obtain one of four possible values ">", "<", "<=", ">=". A distributed event is then detected when the condition $f(v)$ *inequality threshold* becomes true. For example, when `inequality` = " >", an event is detected when $f(v) > threshold$. Once an event is detected, it remains valid for the entire time until another global violation occurs, meaning that the monitored condition has stopped being true.

In order to minimize the implementation overhead when adding new monitoring functions, a significant part of the functionality of the geometric approach has been implemented in our architecture, either at the most general `Function` class, or at the two abstract classes `BallFunction` and `SafeZoneFunction`. Starting from most general `Function` class, we notice that it contains contains two types of variables. The `monitoringVariables` parameter contains information related to the function input parameters (`threshold, inequality`) and the estimate vector `estimate`. The `nodeData` parameter contains information for one or more nodes. This is general enough to accommodate implementations over distributed systems such as Storm, where each processing node (i.e., bolts in Storm) may receive and process data for multiple nodes. For each node, we maintain:

- The most recently received data (`lastValues` variable in the `NodeState` class). The addition of this data is done through the `update` method. All stored tuples are accompanied by the corresponding timestamp that specifies when they were produced.
- The current local statistics vector (`lsv`) of the node. This is calculated, if recent data arrives by the FERARI interface `update` method of the node, through the abstract method updateLSV. For each new declared function, this method must be defined in a subclass of `NodeState`. The parameter `lsvSize` specifies the dimension of the `lsv` vector.
- Parameters relevant to the geometric approach, such as the drift vector `drift`, the delta vector `dv` from the last transmission of this node, a vector `lastSent` containing the last transmitted `lsv` vector, and the corresponding methods that update the values of these parameters. The transmission is achieved by using the FERARI interface `sendToCoordinator`.

Besides the abstract `F` method that has already been mentioned, the class `Function` also contains some additional methods. The abstract `hasLocalViolation` method answers whether a local violation has occurred using the geometric approach. In case a violation has occurred, it communicates using the FERARI interface method `sendToCoordinator`. The `hasLocalViolation` method is defined in a different way for the two subclasses of `Function`.

The `BallFunction` and `SafeZoneFunction` classes contain important functionality regarding the detection of events. The `hasLocalViolation` method is implemented in both the `BallFunction`, as well in the `SafeZoneFunction` subclasses. In `BallFunction`, the way to check for a local violation in a generic way is performed using a grid of points within the sphere that each node constructs in order to check for a local violation. Any function that wants to use the original technique with the spheres (c.f. Section 2) simply needs to: (i) Create a subclass of `BallFunction` that provides the code for the `F` method, (ii) Create a subclass of `NodeState` that provides the code for the `updateLSV` method, and (iii) optionally provide a better method for `hasLocalViolation` if for the specific function it is simple to compute its maximum and minimum values within a sphere. If the third step is omitted, the development of a new function literally requires just a few lines of code and very limited knowledge on the internals of the geometric approach.

The `SafeZoneFunction` class inherits the `BallFunction` class because we may want to define a function that uses a safe zone only when the estimate vector lies on one side of the threshold, while checking for a local violation using the spheres

in the other case. In case we want to use a safe zone, this is determined by the intersection of one or more convex regions (class `ConvexRegion`). Given the value of `lsvSize`, and two vectors `factors` and `powers` (having a dimensionality of `lsvSize+1` and `lsvSize`, respectively) each convex region is defined as the set of points P that satisfy a multivariate polynomial of the form: $\sum_{i=1}^{lsvSize} factors[i] * P[i]^{factors[i]} = factors[lsvSize]$. When developing the code for a function that uses safe zones, one simply needs to: (i) Create a subclass of `SafeZoneFunction` and provide the code for the `F` method, (ii) Create a subclass of `NodeState` that provides the code for the `updateLSV` method, and (iii) Provide the method `computeSafeZones` that computes the safe zone to use whenever the estimate vector is updated. With this hierarchy, it is now possible to implement monitored conditions over functions with little implementation effort.

4.2.2 Mapping to the FERARI Architecture

The Storm topology we derived in our basic example in Section 4.1, can also be used to instantiate the more powerful approach of our function hierarchy. As the FERARI interfaces are used to express the algorithm, we can map the required runtime adaption to our Storm topology. New input data from the monitored streams reach the local nodes by the `update` method and are directed to the function we monitor accordingly. Local violations are communicated with the coordinator by the `sendToCoordinator` method, which is being mapped to a `global grouping` between the `LocalBolt` and `CoordinatorBolt`. In the other direction, the coordinator provides the local nodes with an updated estimate vector by the method `sendToLocal`. This is translated on the Storm topology to an `all grouping` between the `CoordinatorBolt` and the `LocalBolts`. The appropriate calculations required by the geometric approach, update of `lsv`, `dv` and `estimate`. are invoked by the `handleFromCoordinator` and `handleFromLocal` respectively. The same Storm, topology with `LocalBolt`, `CoordinatorBolt` and the connecting streams can be used, as runitme for the geometric monitoring hierarchy. Therefore, this topology can act as FERARI runtime adaption template.

5. CONCUSIONS AND FUTURE DIRECTIONS

In this paper we proposed an architecture for distributed detection and processing of events that allows the separation of application specific code from runtime dependent code. This is achieved by the introduction of the FERARI interfaces, which allow us to create applications for different execution environments. We evaluated established open source distributed streaming platforms, Akka, Spark, Storm and Trident and found that Storm best suits our requirements for the distributed monitoring scenario. We demonstrated how the proposed architecture can be used to implement an interesting example, monitoring when artists get popular by analyzing a stream of listen events. Additionally, we described how the powerful geometric monitoring approach can be implemented using our architecture while requiring tiny programming development efforts. Our code, in advance is public and available on-line [4].

Our future direction is to provide additional FERARI building blocks. Especially, we are interested in including the distributed online learning framework recently published in [6]. Another important block we are working on within the consortium is in providing the capabilities of the PROTON engine, that is open sourced by our partner IBM [5]. Finally, we work on creating a query planer, that will allow to formulate and dynamically optimize the monitoring task in a very convenient way.

6. ACKNOWLEDGMENTS

This research has been supported by the EU FP7-ICT-2013-11 under grant 619491 (FERARI).

7. REFERENCES

[1] Akka, `http://akka.io/`.
[2] Apache spark, `https://spark.apache.org/streaming/`.
[3] Apache storm, `http://storm.apache.org/`.
[4] O. Celma. *Music Recommendation and Discovery in the Long Tail*. Springer, 2010.
[5] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends databases*, 4(1–3):1–294, Jan. 2012.
[6] M. Kamp, M. Boley, D. Keren, A. Schuster, and I. Sharfman. Communication–efficient distributed online prediction by dynamic model synchronization. In *ECML PKDD*, 2014.
[7] D. Keren, G. Sagy, A. Abboud, D. Ben-David, A. Schuster, I. Sharfman, and A. Deligiannakis. Geometric monitoring of heterogeneous streams. *IEEE TKDE*, 26(8):1890–1903, 2014.
[8] D. Keren, I. Sharfman, A. Schuster, and A. Livne. Shape sensitive geometric monitoring. *IEEE TKDE*, 24(8):1520–1535, 2012.
[9] A. Lazerson, I. Sharfman, D. Keren, A. Schuster, M. Garofalakis, and V. Samoladas. Monitoring Distributed Streams using Convex Decomposition. *VLDB*, 2015 (to appear).
[10] G. Sagy, D. Keren, I. Sharfman, and A. Schuster. Distributed threshold querying of general functions by a difference of monotonic representation. *Proc. VLDB Endow.*, 4, November 2010.
[11] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. In *SIGMOD*, 2006.
[12] I. Sharfman, A. Schuster, and D. Keren. Aggregate threshold queries in sensor networks. In *IPDPS*, 2007.
[13] I. Sharfman, A. Schuster, and D. Keren. Shape sensitive geometric monitoring. In *PODS*, 2008.
[14] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD*, 2014.

[4] `https://bitbucket.org/sbothe-iais/ferari`

[5] `https://github.com/ishkin/Proton`